feature_engine Documentation

Release 1.7.0

Feature-engine Developers

Apr 29, 2024

CONTENTS

CHAPTER

ONE

A PYTHON LIBRARY FOR FEATURE ENGINEERING AND SELECTION

Fig. 1: **Feature-engine rocks!**

Feature-engine is a Python library with multiple transformers to engineer and select features to use in machine learning models. Feature-engine preserves Scikit-learn functionality with methods `fit()` and `transform()` to learn parameters from and then transform the data.

Feature-engine includes transformers for:

- Missing data imputation
- Categorical encoding
- Discretisation
- Outlier capping or removal
- Variable transformation
- Variable creation
- Variable selection
- Datetime features
- Time series
- Preprocessing

Feature-engine allows you to select the variables you want to transform **within** each transformer. This way, different engineering procedures can be easily applied to different feature subsets.

Feature-engine transformers can be assembled within the Scikit-learn pipeline, therefore making it possible to save and deploy one single object (.pkl) with the entire machine learning pipeline. Check ***Quick Start*** for an example.

1.1 Pst! How did you find us?

We want to share Feature-engine with more people. It'd help us loads if you tell us how you discovered us.

Then we'd know what we are doing right and which channels to use to share the love.

Please share your story by answering 1 quick question at this link .

CHAPTER

TWO

WHAT IS UNIQUE ABOUT FEATURE-ENGINE?

The following characteristics make Feature-engine unique:

- Feature-engine contains the most exhaustive collection of feature engineering transformations.
- Feature-engine can transform a specific group of variables in the dataframe.
- Feature-engine returns dataframes, hence suitable for data exploration and model deployment.
- Feature-engine is compatible with the Scikit-learn pipeline, Grid and Random search and cross validation.
- Feature-engine automatically recognizes numerical, categorical and datetime variables.
- Feature-engine alerts you if a transformation is not possible, e.g., if applying logarithm to negative variables or divisions by 0.

If you want to know more about what makes Feature-engine unique, check this article.

CHAPTER

THREE

INSTALLATION

Feature-engine is a Python 3 package and works well with 3.7 or later. Earlier versions are not compatible with the latest versions of Python numerical computing libraries.

The simplest way to install Feature-engine is from PyPI with pip:

```
$ pip install feature-engine
```

Note, you can also install it with a _ as follows:

```
$ pip install feature_engine
```

Feature-engine is an active project and routinely publishes new releases. To upgrade Feature-engine to the latest version, use pip like this:

```
$ pip install -U feature-engine
```

If you're using Anaconda, you can install the Anaconda Feature-engine package:

```
$ conda install -c conda-forge feature_engine
```

CHAPTER

FOUR

FEATURE-ENGINE FEATURES IN THE FOLLOWING RESOURCES

- Feature Engineering for Machine Learning, Online Course.
- Feature Selection for Machine Learning, Online Course.
- Feature Engineering for Time Series Forecasting, Online Course.
- Python Feature Engineering Cookbook, book.
- Feature Selection in Machine Learning with Python, book.

More learning resources in the ***Learning Resources***.

CHAPTER

FIVE

FEATURE-ENGINE'S TRANSFORMERS

Feature-engine hosts the following groups of transformers:

5.1 Missing Data Imputation: Imputers

- *MeanMedianImputer*: replaces missing data in numerical variables by the mean or median
- *ArbitraryNumberImputer*: replaces missing data in numerical variables by an arbitrary number
- *EndTailImputer*: replaces missing data in numerical variables by numbers at the distribution tails
- *CategoricalImputer*: replaces missing data with an arbitrary string or by the most frequent category
- *RandomSampleImputer*: replaces missing data by random sampling observations from the variable
- *AddMissingIndicator*: adds a binary missing indicator to flag observations with missing data
- *DropMissingData*: removes observations (rows) containing missing values from dataframe

5.2 Categorical Encoders: Encoders

- *OneHotEncoder*: performs one hot encoding, optional: of popular categories
- *CountFrequencyEncoder*: replaces categories by the observation count or percentage
- *OrdinalEncoder*: replaces categories by numbers arbitrarily or ordered by target
- *MeanEncoder*: replaces categories by the target mean
- *WoEEncoder*: replaces categories by the weight of evidence
- *DecisionTreeEncoder*: replaces categories by predictions of a decision tree
- *RareLabelEncoder*: groups infrequent categories
- *StringSimilarityEncoder*: encodes categories based on string similarity

5.3 Variable Discretisation: Discretisers

- *ArbitraryDiscretiser*: sorts variable into intervals defined by the user
- *EqualFrequencyDiscretiser*: sorts variable into equal frequency intervals
- *EqualWidthDiscretiser*: sorts variable into equal width intervals
- *DecisionTreeDiscretiser*: uses decision trees to create finite variables
- *GeometricWidthDiscretiser*: sorts variable into geometrical intervals

5.4 Outlier Capping or Removal

- *ArbitraryOutlierCapper*: caps maximum and minimum values at user defined values
- *Winsorizer*: caps maximum or minimum values using statistical parameters
- *OutlierTrimmer*: removes outliers from the dataset

5.5 Numerical Transformation: Transformers

- *LogTransformer*: performs logarithmic transformation of numerical variables
- *LogCpTransformer*: performs logarithmic transformation after adding a constant value
- *ReciprocalTransformer*: performs reciprocal transformation of numerical variables
- *PowerTransformer*: performs power transformation of numerical variables
- *BoxCoxTransformer*: performs Box-Cox transformation of numerical variables
- *YeoJohnsonTransformer*: performs Yeo-Johnson transformation of numerical variables
- *ArcsinTransformer*: performs arcsin transformation of numerical variables

5.6 Feature Creation:

- *MathFeatures*: creates new variables by combining features with mathematical operations
- *RelativeFeatures*: combines variables with reference features
- *CyclicalFeatures*: creates variables using sine and cosine, suitable for cyclical features

5.7 Datetime:

- *DatetimeFeatures*: extract features from datetime variables
- *DatetimeSubtraction*: computes subtractions between datetime variables

5.8 Feature Selection:

- *DropFeatures*: drops an arbitrary subset of variables from a dataframe
- *DropConstantFeatures*: drops constant and quasi-constant variables from a dataframe
- *DropDuplicateFeatures*: drops duplicated variables from a dataframe
- *DropCorrelatedFeatures*: drops correlated variables from a dataframe
- *SmartCorrelatedSelection*: selects best features from correlated groups
- *DropHighPSIFeatures*: selects features based on the Population Stability Index (PSI)
- *SelectByInformationValue*: selects features based on their information value
- *SelectByShuffling*: selects features by evaluating model performance after feature shuffling
- *SelectBySingleFeaturePerformance*: selects features based on their performance on univariate estimators
- *SelectByTargetMeanPerformance*: selects features based on target mean encoding performance
- *RecursiveFeatureElimination*: selects features recursively, by evaluating model performance
- *RecursiveFeatureAddition*: selects features recursively, by evaluating model performance
- *ProbeFeatureSelection*: selects features whose importance is greater than those of random variables

5.9 Forecasting:

- *LagFeatures*: extract lag features
- *WindowFeatures*: create window features
- *ExpandingWindowFeatures*: create expanding window features

5.10 Preprocessing:

- *MatchCategories*: ensures categorical variables are of type 'category'
- *MatchVariables*: ensures that columns in test set match those in train set

5.11 Scikit-learn Wrapper:

- *SklearnTransformerWrapper*: applies Scikit-learn transformers to a selected subset of features

CHAPTER

SIX

GETTING HELP

Can't get something to work? Here are places where you can find help.

1. The ***User Guide*** in the docs.
2. Stack Overflow. If you ask a question, please mention "feature_engine" in it.
3. If you are enrolled in the Feature Engineering for Machine Learning course , post a question in a relevant section.
4. If you are enrolled in the Feature Selection for Machine Learning course , post a question in a relevant section.
5. Join our gitter community. You an ask questions here as well.
6. Ask a question in the repo by filing an issue (check before if there is already a similar issue created :)).

CHAPTER

SEVEN

CONTRIBUTING

Interested in contributing to Feature-engine? That is great news!

Feature-engine is a welcoming and inclusive project and we would be delighted to have you on board. We follow the Python Software Foundation Code of Conduct.

Regardless of your skill level you can help us. We appreciate bug reports, user testing, feature requests, bug fixes, addition of tests, product enhancements, and documentation improvements. We also appreciate blogs about Feature-engine. If you happen to have one, let us know!

For more details on how to contribute check the contributing page. Click on the ***Contribute*** guide.

CHAPTER

EIGHT

SPONSOR US

Empower Sole, the main developer of Feature-engine, to assemble a team of paid contributors to accelerate the development of Feature-engine.

Currently, Sole and our contributors dedicate their free time voluntarily to advancing the project. You can help us reach a funding milestone, so that we can gather on a group of 2-3 contributors who will commit regular hours each week to enhance documentation and expand Feature-engine's functionality at a faster pace.

Your contribution will play a vital role in propelling Feature-engine to new heights, ensuring it remains a valuable resource for the data science community.

If you don't have a Github account, you can also sponsor us here.

CHAPTER

NINE

OPEN SOURCE

Feature-engine's license is an open source BSD 3-Clause.

Feature-engine is hosted on GitHub. The issues and pull requests are tracked there.

CHAPTER

TEN

TABLE OF CONTENTS

10.1 Quick Start

If you're new to Feature-engine this guide will get you started. Feature-engine transformers have the methods `fit()` and `transform()` to learn parameters from the data and then modify the data. They work just like any Scikit-learn transformer.

10.1.1 Installation

Feature-engine is a Python 3 package and works well with 3.7 or later. Earlier versions are not compatible with the latest versions of Python numerical computing libraries.

```
$ pip install feature-engine
```

Note, you can also install it with a _ as follows:

```
$ pip install feature_engine
```

Note that Feature-engine is an active project and routinely publishes new releases. In order to upgrade Feature-engine to the latest version, use `pip` as follows.

```
$ pip install -U feature-engine
```

If you're using Anaconda, you can install the Anaconda Feature-engine package:

```
$ conda install -c conda-forge feature_engine
```

Once installed, you should be able to import Feature-engine without an error, both in Python and in Jupyter notebooks.

10.1.2 Example Use

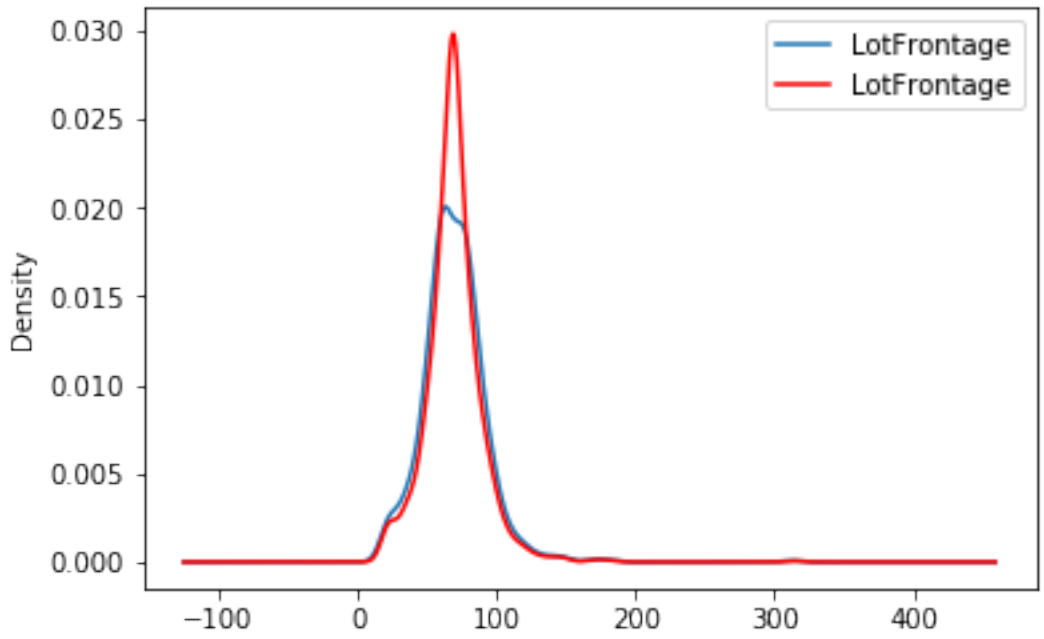
This is an example of how to use Feature-engine's transformers to perform missing data imputation.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

from feature_engine.imputation import MeanMedianImputer

```

(continues on next page)

(continued from previous page)

```
# Load dataset
data = pd.read_csv('houseprice.csv')

# Separate into train and test sets
X_train, X_test, y_train, y_test = train_test_split(
    data.drop(['Id', 'SalePrice'], axis=1),
    data['SalePrice'],
    test_size=0.3,
    random_state=0
)

# set up the imputer
median_imputer = MeanMedianImputer(
    imputation_method='median', variables=['LotFrontage', 'MasVnrArea']
    )

# fit the imputer
median_imputer.fit(X_train)

# transform the data
train_t = median_imputer.transform(X_train)
test_t = median_imputer.transform(X_test)

fig = plt.figure()
ax = fig.add_subplot(111)
X_train['LotFrontage'].plot(kind='kde', ax=ax)
train_t['LotFrontage'].plot(kind='kde', ax=ax, color='red')
lines, labels = ax.get_legend_handles_labels()
ax.legend(lines, labels, loc='best')
```



10.1.3 Feature-engine with the Scikit-learn's pipeline

Feature-engine's transformers can be assembled within a Scikit-learn pipeline. This way, we can store our entire feature engineering pipeline in one single object or pickle (.pkl). Here is an example of how to do it:

```
from math import sqrt
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

from sklearn.linear_model import Lasso
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
from sklearn.pipeline import Pipeline as pipe
from sklearn.preprocessing import MinMaxScaler

from feature_engine.encoding import RareLabelEncoder, MeanEncoder
from feature_engine.discretisation import DecisionTreeDiscretiser
from feature_engine.imputation import (
    AddMissingIndicator,
    MeanMedianImputer,
    CategoricalImputer,
)

# load dataset
data = pd.read_csv('houseprice.csv')

# drop some variables
data.drop(
    labels=['YearBuilt', 'YearRemodAdd', 'GarageYrBlt', 'Id'],
    axis=1,
    inplace=True
)

# make a list of categorical variables
categorical = [var for var in data.columns if data[var].dtype == 'O']

# make a list of numerical variables
numerical = [var for var in data.columns if data[var].dtype != 'O']

# make a list of discrete variables
discrete = [ var for var in numerical if len(data[var].unique()) < 20]

# categorical encoders work only with object type variables
# to treat numerical variables as categorical, we need to re-cast them
data[discrete]= data[discrete].astype('O')

# continuous variables
numerical = [
    var for var in numerical if var not in discrete
    and var not in ['Id', 'SalePrice']
    ]
```

(continues on next page)

(continued from previous page)

```
# separate into train and test sets
X_train, X_test, y_train, y_test = train_test_split(
                                        data.drop(labels=['SalePrice'], axis=1),
                                        data.SalePrice,
                                        test_size=0.1,
                                        random_state=0
                                        )

# set up the pipeline
price_pipe = pipe([
    # add a binary variable to indicate missing information for the 2 variables below
    ('continuous_var_imputer', AddMissingIndicator(variables=['LotFrontage'])),

    # replace NA by the median in the 2 variables below, they are numerical
    ('continuous_var_median_imputer', MeanMedianImputer(
        imputation_method='median', variables=['LotFrontage', 'MasVnrArea']
    )),

    # replace NA by adding the label "Missing" in categorical variables
    ('categorical_imputer', CategoricalImputer(variables=categorical)),

    # disretise continuous variables using trees
    ('numerical_tree_discretiser', DecisionTreeDiscretiser(
        cv=3,
        scoring='neg_mean_squared_error',
        variables=numerical,
        regression=True)),

    # remove rare labels in categorical and discrete variables
    ('rare_label_encoder', RareLabelEncoder(
        tol=0.03, n_categories=1, variables=categorical+discrete
    )),

    # encode categorical and discrete variables using the target mean
    ('categorical_encoder', MeanEncoder(variables=categorical+discrete)),

    # scale features
    ('scaler', MinMaxScaler()),

    # Lasso
    ('lasso', Lasso(random_state=2909, alpha=0.005))

])

# train feature engineering transformers and Lasso
price_pipe.fit(X_train, np.log(y_train))

# predict
pred_train = price_pipe.predict(X_train)
pred_test = price_pipe.predict(X_test)

# Evaluate
```

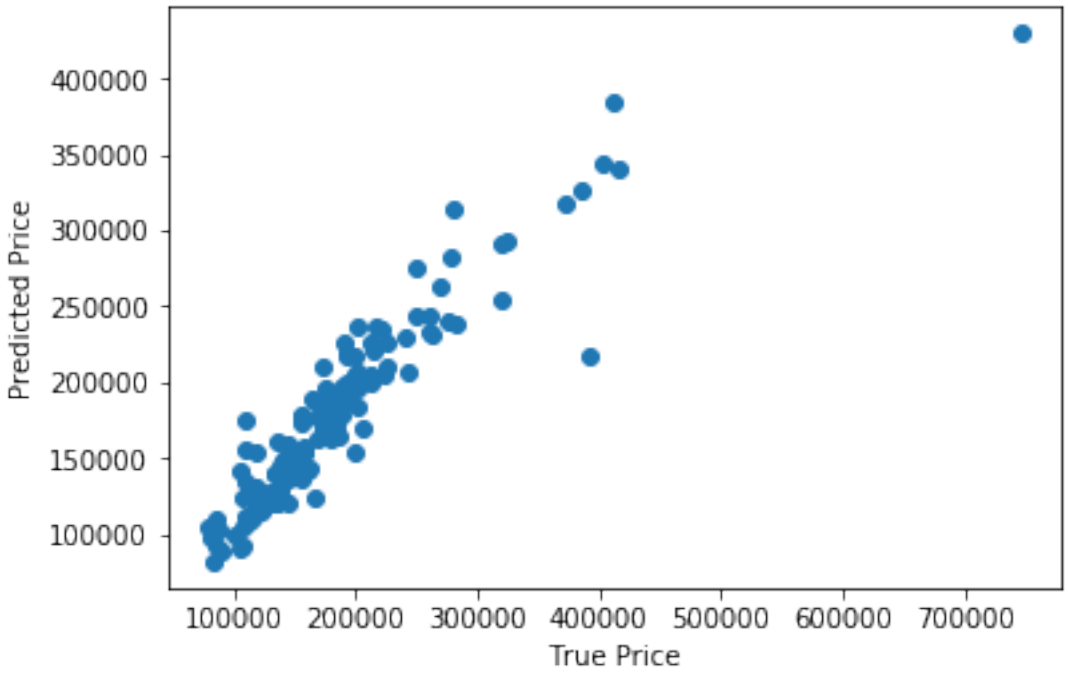
(continues on next page)

(continued from previous page)

```
print('Lasso Linear Model train mse: {}'.format(
    mean_squared_error(y_train, np.exp(pred_train))))
print('Lasso Linear Model train rmse: {}'.format(
    sqrt(mean_squared_error(y_train, np.exp(pred_train)))))
print()
print('Lasso Linear Model test mse: {}'.format(
    mean_squared_error(y_test, np.exp(pred_test))))
print('Lasso Linear Model test rmse: {}'.format(
    sqrt(mean_squared_error(y_test, np.exp(pred_test)))))
```

```
Lasso Linear Model train mse: 949189263.8948538
Lasso Linear Model train rmse: 30808.9153313591

Lasso Linear Model test mse: 1344649485.0641894
Lasso Linear Model train rmse: 36669.46256852136
```

```
plt.scatter(y_test, np.exp(pred_test))
plt.xlabel('True Price')
plt.ylabel('Predicted Price')
plt.show()
```



More examples

More examples can be found in:

- *User Guide*
- *Learning Resources*
- Jupyter notebooks

Datasets

The user guide and examples included in Feature-engine's documentation are based on these 3 datasets:

Titanic dataset

We use the dataset available in openML which can be downloaded from here.

Ames House Prices dataset

We use the data set created by Professor Dean De Cock: * Dean De Cock (2011) Ames, Iowa: Alternative to the Boston Housing * Data as an End of Semester Regression Project, Journal of Statistics Education, Vol.19, No. 3.

The examples are based on a copy of the dataset available on Kaggle.

The original data and documentation can be found here:

- Documentation
- Data

Credit Approval dataset

We use the Credit Approval dataset from the UCI Machine Learning Repository:

Dua, D. and Graff, C. (2019). UCI Machine Learning Repository. Irvine, CA: University of California, School of Information and Computer Science.

To download the dataset visit this website and click on "crx.data" to download the data set.

To prepare the data for the examples:

```
import random
import pandas as pd
import numpy as np

# load data
data = pd.read_csv('crx.data', header=None)

# create variable names according to UCI Machine Learning information
varnames = ['A'+str(s) for s in range(1,17)]
data.columns = varnames

# replace ? by np.nan
```

(continues on next page)

(continued from previous page)

```
data = data.replace('?', np.nan)

# re-cast some variables to the correct types
data['A2'] = data['A2'].astype('float')
data['A14'] = data['A14'].astype('float')

# encode target to binary
data['A16'] = data['A16'].map({'+':1, '-':0})

# save the data
data.to_csv('creditApprovalUCI.csv', index=False)
```

10.2 User Guide

In this section you will find additional information about Feature-engine's transformers and feature engineering transformations in general, as well as additional examples.

10.2.1 Transformation

Missing Data Imputation

Feature-engine's missing data imputers replace missing data by parameters estimated from data or arbitrary values pre-defined by the user. The following image summarizes the main imputer's functionality.

	Numerical variables	Categorical variables	Description
MeanMedianImputer	√	×	Replaces missing values by the mean or median
ArbitraryNumberImputer	√		Replaces missing values by an arbitrary value
EndTailImputer	√	×	Replaces missing values by a value at the end of the distribution
CategoricalImputer	√	√	Replaces missing values by the most frequent category or by an arbitrary value
RandomSampleImputer	√	√	Replaces missing values by random value extractions from the variable
AddMissingIndicator	√	√	Adds a binary variable to flag missing observations
DropMissingData	√	√	Removes observations with missing data from the dataset

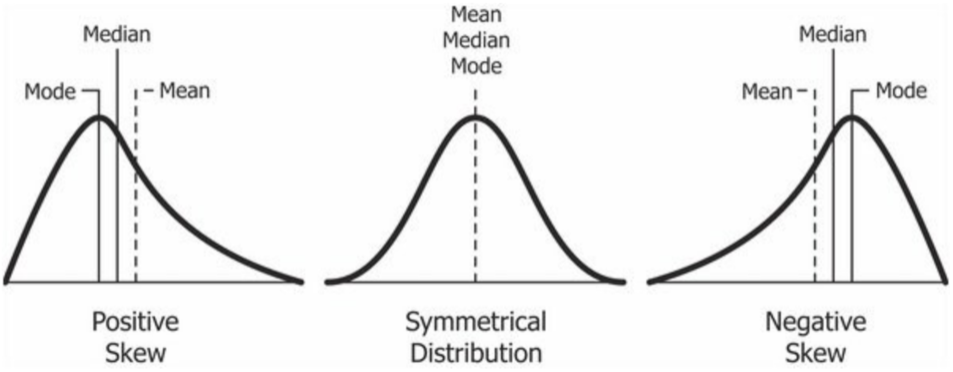
In this guide, you will find code snippets to quickly be able to apply the imputers to your datasets, as well as general knowledge and guidance on the imputation techniques.

Imputers

MeanMedianImputer

The *MeanMedianImputer()* replaces missing data with the mean or median of the variable. It works only with numerical variables. You can pass the list of variables you want to impute, or alternatively, the imputer will automatically select all numerical variables in the train set.

Note that in symetrical distributions, the mean and the median are very similar. But in skewed distributions, the median is a better representation of the majority, as the mean is biased to extreme values. The following image was taken from Wikipedia. The image links to the use license.



With the `fit()` method, the transformer learns and stores the mean or median values per variable. Then it uses these values in the `transform()` method to transform the data.

Below a code example using the House Prices Dataset (more details about the dataset *here*).

First, let's load the data and separate it into train and test:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

from feature_engine.imputation import MeanMedianImputer

# Load dataset
data = pd.read_csv('houseprice.csv')

# Separate into train and test sets
X_train, X_test, y_train, y_test = train_test_split(
                                    data.drop(['Id', 'SalePrice'], axis=1),
                                    data['SalePrice'],
                                    test_size=0.3,
                                    random_state=0,
                                    )
```

Now we set up the *MeanMedianImputer()* to impute in this case with the median and only 2 variables from the dataset.

```
# set up the imputer
median_imputer = MeanMedianImputer(
```

(continues on next page)

(continued from previous page)

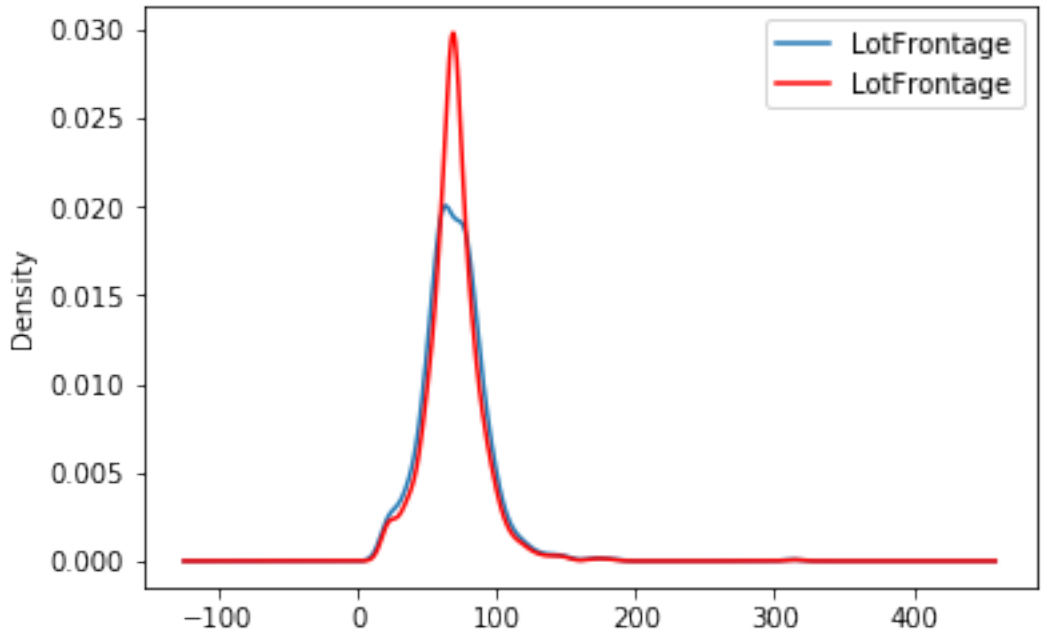
```
                    imputation_method='median',
                    variables=['LotFrontage', 'MasVnrArea']
                    )

# fit the imputer
median_imputer.fit(X_train)
```

With fit, the *MeanMedianImputer()* learned the median values for the indicated variables and stored it in one of its attributes. We can now go ahead and impute both the train and the test sets.

```
# transform the data
train_t= median_imputer.transform(X_train)
test_t= median_imputer.transform(X_test)
```

Note that after the imputation, if the percentage of missing values is relatively big, the variable distribution will differ from the original one (in red the imputed variable):

```
fig = plt.figure()
ax = fig.add_subplot(111)
X_train['LotFrontage'].plot(kind='kde', ax=ax)
train_t['LotFrontage'].plot(kind='kde', ax=ax, color='red')
lines, labels = ax.get_legend_handles_labels()
ax.legend(lines, labels, loc='best')
```



Additional resources

In the following Jupyter notebook you will find more details on the functionality of the `MeanMedianImputer()`, including how to select numerical variables automatically. You will also see how to navigate the different attributes of the transformer to find the mean or median values of the variables.

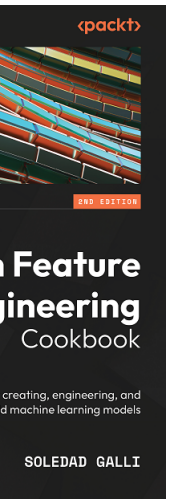
- Jupyter notebook

For more details about this and other feature engineering methods check out these resources:

Or read our book:

Fig. 1: Feature Engineering for Machine Learning

Both our book and course are suitable for beginners and more advanced data scientists alike. By purchasing them you are supporting Sole, the main developer of Feature-engine.



re Engineering

ArbitraryNumberImputer

The *ArbitraryNumberImputer()* replaces missing data with an arbitrary numerical value determined by the user. It works only with numerical variables.

The *ArbitraryNumberImputer()* can find and impute all numerical variables automatically. Alternatively, you can pass a list of the variables you want to impute to the `variables` parameter.

You can impute all variables with the same number, in which case you need to define the variables to impute in the `variables` parameter and the imputation number in `arbitrary_number` parameter. For example, you can impute varA and varB with 99 like this:

```
transformer = ArbitraryNumberImputer(
        variables = ['varA', 'varB'],
        arbitrary_number = 99
        )

Xt = transformer.fit_transform(X)
```

You can also impute different variables with different numbers. To do this, you need to pass a dictionary with the variable names and the numbers to use for their imputation to the `imputer_dict` parameter. For example, you can impute varA with 1 and varB with 99 like this:

```
transformer = ArbitraryNumberImputer(
        imputer_dict = {'varA' : 1, 'varB': 99}
        )

Xt = transformer.fit_transform(X)
```

Below a code example using the House Prices Dataset (more details about the dataset *here*).

First, let's load the data and separate it into train and test:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

from␣
↪feature_engine.imputation import ArbitraryNumberImputer

# Load dataset
data = pd.read_csv('houseprice.csv')

# Separate into train and test sets
X_train, X_test, y_train, y_test = train_test_split(
          ␣
↪                 data.drop(['Id', 'SalePrice'], axis=1),
                  data['SalePrice'],
                  test_size=0.3,
```

(continues on next page)

(continued from previous page)

```
                                random_state=0,
                                )
```

Now we set up the *ArbitraryNumberImputer()* to impute 2 variables from the dataset with the number -999:

```
# set up the imputer
arbitrary_imputer = ArbitraryNumberImputer(
    arbitrary_number=-999,
    variables=['LotFrontage', 'MasVnrArea'],
    )

# fit the imputer
arbitrary_imputer.fit(X_train)
```

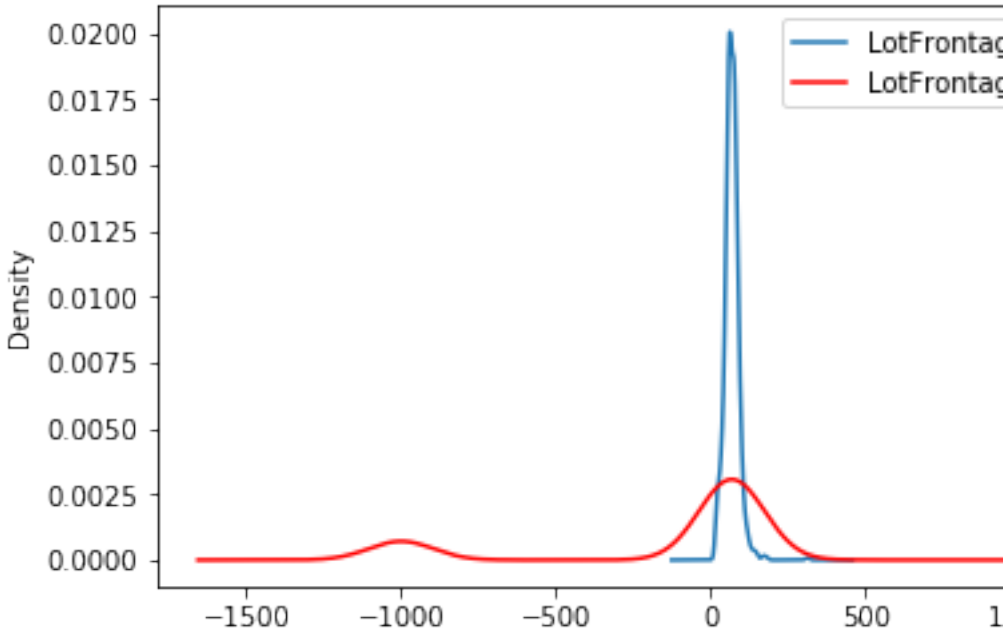
With `fit()`, the transformer does not learn any parameter. It just assigns the imputation values to each variable, which can be found in the attribute `imputer_dict_`.

With transform, we replace the missing data with the arbitrary values both in train and test sets:

```
# transform the data
train_t= arbitrary_imputer.transform(X_train)
test_t= arbitrary_imputer.transform(X_test)
```

Note that after the imputation, if the percentage of missing values is relatively big, the variable distribution will differ from the original one (in red the imputed variable):

```
fig = plt.figure()
ax = fig.add_subplot(111)
X_train['LotFrontage'].plot(kind='kde', ax=ax)
train_
↪t['LotFrontage'].plot(kind='kde', ax=ax, color='red')
lines, labels = ax.get_legend_handles_labels()
ax.legend(lines, labels, loc='best')
```



Additional resources

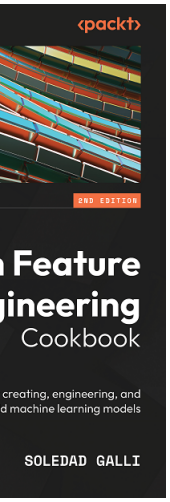
In the following Jupyter notebook you will find more details on the functionality of the *ArbitraryNumberImputer()*, including how to select numerical variables automatically. You will also see how to navigate the different attributes of the transformer.

- Jupyter notebook

For more details about this and other feature engineering methods check out these resources:

Or read our book:

Both our book and course are suitable for beginners and more advanced data scientists alike. By purchasing them you are supporting Sole, the main developer of Feature-engine.



ure Engineering

EndTailImputer

The *EndTailImputer()* replaces missing data with a value at the end of the distribution. The value can be determined using the mean plus or minus a number of times the standard deviation, or using the inter-quartile range proximity rule. The value can also be determined as a factor of the maximum value.

You decide whether the missing data should be placed at the right or left tail of the variable distribution.

In a sense, the *EndTailImputer()* **"automates"** the work of the *ArbitraryNumberImputer()* because it will find automatically "arbitrary values" far out at the end of the variable distributions.

EndTailImputer() works only with numerical variables. You can impute only a subset of the variables in the data by passing the variable names in a list. Alternatively, the imputer will automatically select all numerical variables in the train set.

Below a code example using the House Prices Dataset (more details about the dataset *here*).

First, let's load the data and separate it into train and test:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

from feature_engine.imputation import EndTailImputer

# Load dataset
data = pd.read_csv('houseprice.csv')

# Separate into train and test sets
X_train, X_test, y_train, y_test = train_test_split(
```

(continues on next page)

(continued from previous page)

```
↪                 data.drop(['Id', 'SalePrice'], axis=1),
                                  data['SalePrice'],
                                  test_size=0.3,
                                  random_state=0,
                                  )
```

Now we set up the *EndTailImputer()* to impute in this case only 2 variables from the dataset. We instruct the imputer to find the imputation values using the mean plus 3 times the standard deviation as follows:

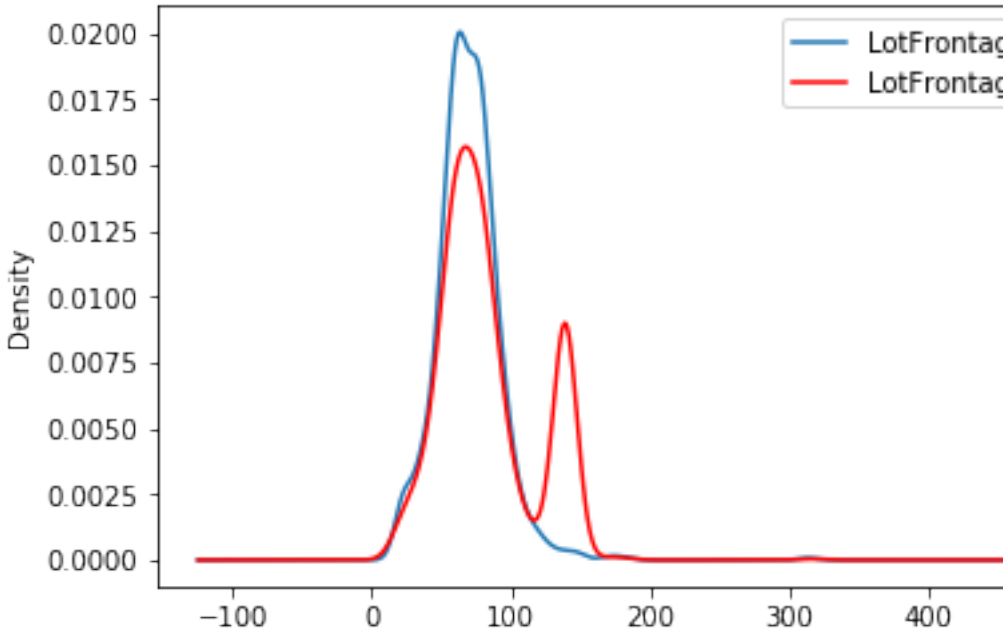
```
# set up the imputer
tail_
↪imputer = EndTailImputer(imputation_method='gaussian',
                          tail='right',
                          fold=3,

↪                 variables=['LotFrontage', 'MasVnrArea'])
# fit the imputer
tail_imputer.fit(X_train)
```

With fit, the *EndTailImputer()* learned the imputation values for the indicated variables and stored it in one of its attributes. We can now go ahead and impute both the train and the test sets.

```
# transform the data
train_t= tail_imputer.transform(X_train)
test_t= tail_imputer.transform(X_test)
```

Note that after the imputation, if the percentage of missing values is relatively big, the variable distribution will differ from the original one (in red the imputed variable):

```
fig = plt.figure()
ax = fig.add_subplot(111)
X_train['LotFrontage'].plot(kind='kde', ax=ax)
train_
↪t['LotFrontage'].plot(kind='kde', ax=ax, color='red')
lines, labels = ax.get_legend_handles_labels()
ax.legend(lines, labels, loc='best')
```



Additional resources

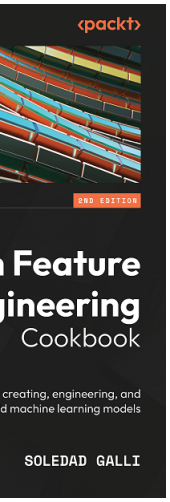
In the following Jupyter notebook you will find more details on the functionality of the `CategoricalImputer()`, including how to select numerical variables automatically, how to impute with the most frequent category, and how to impute with a used defined string.

- Jupyter notebook

For more details about this and other feature engineering methods check out these resources:

Or read our book:

Both our book and course are suitable for beginners and more advanced data scientists alike. By purchasing them you are supporting Sole, the main developer of Feature-engine.



ure Engineering

CategoricalImputer

Categorical data are common in most data science projects and can also show missing values. There are **2 main imputation methods** that are used to replace missing data in categorical variables. One method consists of replacing the missing values with the most frequent category. The second method consists of replacing missing values with a dedicated string, for example, "Missing."

Scikit-learn's machine learning algorithms can neither handle missing data nor categorical variables out of the box. Hence, during data preprocessing, we need to use imputation techniques to replace the nan values by any permitted value and then proceed with categorical encoding, before training classification or regression models.

Handling missing values

Feature-engine's *CategoricalImputer()* can replace missing data in categorical variables with an arbitrary value, like the string 'Missing', or with the most frequent category.

You can impute a subset of the categorical variables by passing their names to *CategoricalImputer()* in a list. Alternatively, the categorical imputer automatically finds and imputes all variables of type object and categorical found in the training dataframe.

Originally, we designed this imputer to work only with categorical variables. In version 1.1.0, we introduced the parameter `ignore_format` to allow the imputer to also impute numerical variables with this functionality. This is because, in some cases, variables that are by nature categorical have numerical values.

Python implementation

We'll show the *CategoricalImputer()*'s data imputation functionality using the Ames house prices dataset. We'll start by loading the necessary libraries, functions and classes, loading the dataset, and separating it into a training and a test set.

```
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split

from feature_engine.imputation import CategoricalImputer

data = fetch_openml(name='house_prices', as_frame=True)
data = data.frame

X = data.drop(['SalePrice', 'Id'], axis=1)
y = data['SalePrice']

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42)

print(X_train.head())
```

In the following output we see the predictor variables of the house prices dataset:

```
      MSSubClass␣
↪MSZoning  LotFrontage  LotArea Street Alley LotShape  \
254          ␣
↪ 20       RL         70.0     8400   Pave   NaN      Reg
1066         ␣
↪ 60       RL         59.0     7837   Pave   NaN      IR1
638          ␣
↪ 30       RL         67.0     8777   Pave   NaN      Reg
799          ␣
↪ 50       RL         60.0     7200   Pave   NaN      Reg
380          ␣
↪ 50       RL         50.0     5000   Pave  Pave      Reg

     LandContour Utilities␣
↪LotConfig  ... ScreenPorch PoolArea PoolQC  Fence  \
254          Lvl    ␣
↪AllPub    Inside  ...           0        0    NaN    NaN
1066         Lvl    ␣
↪AllPub    Inside  ...           0        0    NaN    NaN
638          Lvl    ␣
↪AllPub    Inside  ...           0        0    NaN  MnPrv
799          Lvl    ␣
↪AllPub    Corner  ...           0        0    NaN  MnPrv
380          Lvl    ␣
↪AllPub    Inside  ...           0        0    NaN    NaN
```

(continues on next page)

(continued from previous page)

```
     MiscFeature␣
↪MiscVal  MoSold  YrSold  SaleType  SaleCondition
254      ␣
↪     NaN       0       6    2010        WD         Normal
1066     ␣
↪     NaN       0       5    2009        WD         Normal
638      ␣
↪     NaN       0       5    2008        WD         Normal
799      ␣
↪     NaN       0       6    2007        WD         Normal
380      ␣
↪     NaN       0       5    2010        WD         Normal

[5 rows x 79 columns]
```

These 2 variables show null values, let's check that out:

```
X_train[['Alley', 'MasVnrType']].isnull().sum()
```

We see the null values in the following output:

```
Alley         1094
MasVnrType       6
dtype: int64
```

Imputation with an arbitrary string

Let's set up the categorical imputer to impute these 2 variables with the arbitrary string 'missing':

```
imputer = CategoricalImputer(
    variables=['Alley', 'MasVnrType'],
    fill_value="missing",
)

imputer.fit(X_train)
```

During fit, the transformer corroborates that the 2 variables are of type object or categorical and creates a dictionary of variable to replacement value.

We can check the value that will be use to "fillna" as follows:

```
imputer.fill_value
```

We can check the dictionary with the replacement values per variable like this:

```
imputer.imputer_dict_
```

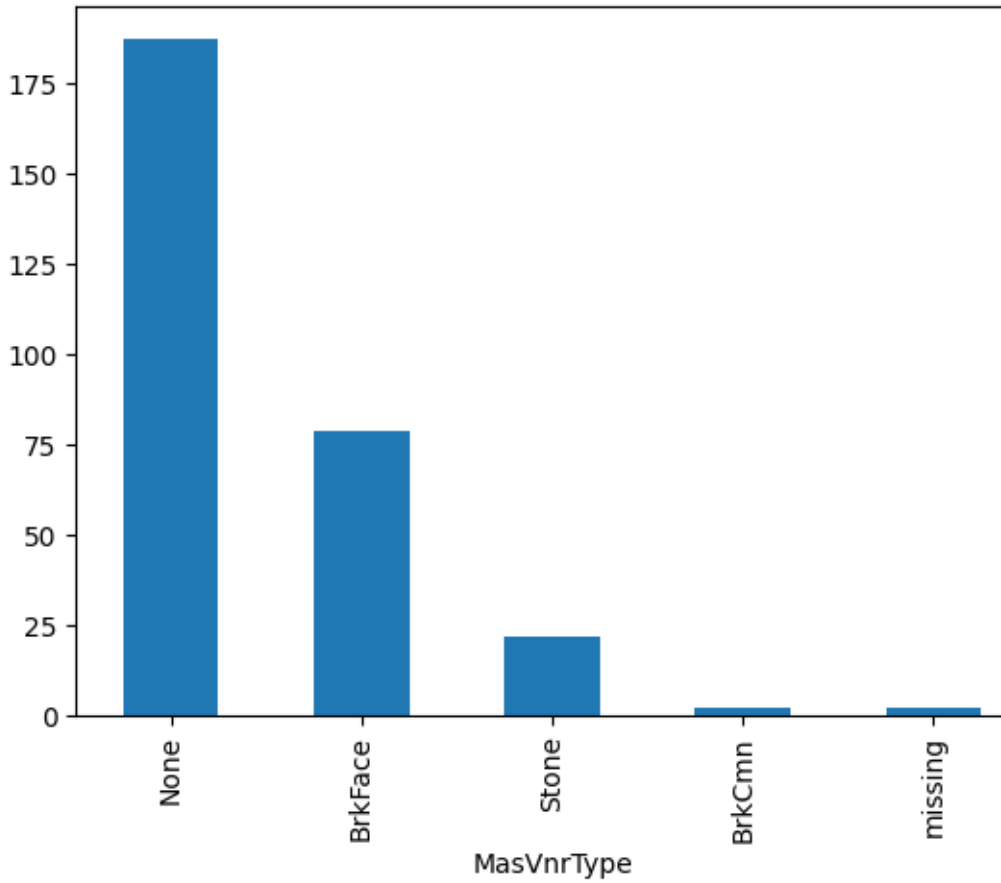
The dictionary contains the names of the variables in its keys and the imputation value among its values. In this case, the result is not super exciting because we are replacing nan values in all variables with the same value:

```
{'Alley': 'missing', 'MasVnrType': 'missing'}
```

We can now go ahead and impute the missing data and then plot the categories in the resulting variable after the imputation:

```
train_t = imputer.transform(X_train)
test_t = imputer.transform(X_test)

test_t['MasVnrType'].value_counts().plot.bar()
plt.ylabel("Number of observations")
plt.show()
```

In the following plot, we see the presence of the category "missing", corresponding to the imputed values:



Imputation with the most frequent category

Let's now impute the variables with the most frequent category instead:

```
imputer = CategoricalImputer(
    variables=['Alley', 'MasVnrType'],
    imputation_method="frequent"
)

imputer.fit(X_train)
```

We can find the most frequent category per variable in the imputer dictionary:

```
imputer.imputer_dict_
```

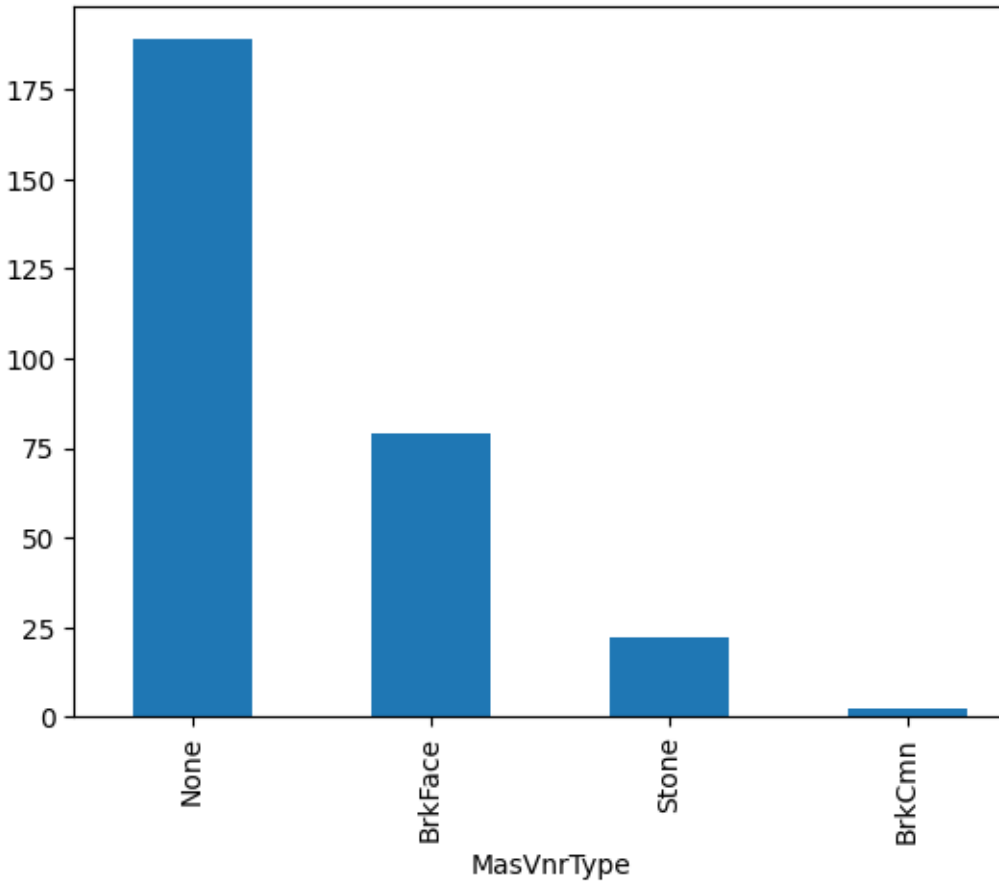
In the following output, we see that the most frequent category for `Alley` is `'Grvl'` and the most frequent value for `MasVnrType` is `'None'`.

```
{'Alley': 'Grvl', 'MasVnrType': 'None'}
```

We can now go ahead and impute the missing data to obtain a complete dataset, at least for these 2 variables, and then plot the distribution of values after the imputation:

```
train_t = imputer.transform(X_train)
test_t = imputer.transform(X_test)

test_t['MasVnrType'].value_counts().plot.bar()
plt.ylabel("Number of observations")
plt.show()
```

In the following image we see the resulting variable distribution:



Automatically impute all categorical variables

CategoricalImputer() can automatically find and impute all categorical features in the training dataset when we set the parameter `variables` to None:

```
imputer = CategoricalImputer(
    variables=None,
)

train_t = imputer.fit_transform(X_train)
test_t = imputer.transform(X_test)
```

We can find the categorical variables in the `variables_` attribute:

```
imputer.variables_
```

Below, we see the list of categorical variables that were found in the training dataframe:

```
['MSZoning',
 'Street',
 'Alley',
 'LotShape',
 'LandContour',
 ...
 'SaleType',
 'SaleCondition']
```

Categorical features with 2 modes

It is possible that one variable has more than one mode. In that case, the transformer will raise an error. For example, when you set the transformer to impute the variable 'PoolQC` with the most frequent value:

```
imputer = CategoricalImputer(
    variables=['PoolQC'],
    imputation_method="frequent"
)

imputer.fit(X_train)
```

'PoolQC` has more than 1 mode, so the transformer raises the following error:

```
    196     self.imputer_dict_ = {var: mode_vals[0]}
    198 # imputing multiple variables:
    199 else:
    200     #␣
↪Returns a dataframe with 1 row if there is one mode per
    201␣
↪    # variable, or more rows if there are more modes:

ValueError: The␣
↪variable PoolQC contains multiple frequent categories.
```

We can check that the variable has various modes like this:

```
X_train['PoolQC'].mode()
```

We see that this variable has 3 categories with similar maximum number of observations:

```
0    Ex
1    Fa
2    Gd
Name: PoolQC, dtype: object
```

Considerations

Replacing missing values in categorical features with a bespoke category is standard practice and perhaps the more natural thing to do. We'll probably want to impute with the most frequent category when the percentage of missing values is small and the cardinality of the variable is low, not to introduce unnecessary noise.

Combining imputation with data analysis is useful to decide the most convenient imputation method as well as the impact of the imputation on the variable distribution. Note that the variable distribution and its cardinality will affect the performance and workings of machine learning models.

Imputation with the most frequent category will blend the missing values with the most common values of the variable. Hence, it is common practice to add dummy variables to indicate that the values were originally missing. See *AddMissingIndicator*.

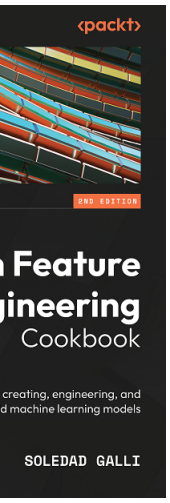
Additional resources

For more details about this and other feature engineering methods check out these resources:

Or read our book:

Fig. 7: Feature Engineering for Machine Learning

RandomSampleImputer



The *RandomSampleImputer()* replaces missing data with a random sample extracted from the variable. It works with both numerical and categorical variables. A list of variables can be indicated, or the imputer will automatically select all variables in the train set.

Note

The random samples used to replace missing values may vary from execution to execution. This may affect the results of your work. Thus, it is advisable to set a seed.

Setting the seed

There are 2 ways in which the seed can be set in the *RandomSampleImputer()*:

If `seed = 'general'` then the random_state can be either `None` or an integer. The `random_state` then provides the seed to use in the imputation. All observations will be imputed in one go with a single seed. This is equivalent to `pandas.sample(n, random_state=seed)` where `n` is the number of observations with missing data and `seed` is the number you entered in the `random_state`.

If `seed = 'observation'`, then the random_state should be a variable name or a list of variable names. The seed will be calculated observation per observation, either by adding or multiplying the values of the variables indicated in the `random_state`. Then, a value will be extracted from the train set using that seed and used to replace the NAN in that particular observation. This is the equivalent of `pandas.sample(1, random_state=var1+var2)` if the `seeding_method` is set to `add` or `pandas.sample(1, random_state=var1*var2)` if the `seeding_method` is set to `multiply`.

For example, if the observation shows variables color: np.nan, height: 152, weight:52, and we set the imputer as:

```
RandomSampleImputer(random_state=['height', 'weight'],
                            seed='observation',
                            seeding_method='add'))
```

the np.nan in the variable colour will be replaced using pandas sample as follows:

```
observation.sample(1, random_state=int(152+52))
```

For more details on why this functionality is important refer to the course Feature Engineering for Machine Learning.

You can also find more details about this imputation in the following notebook.

Note, if the variables indicated in the `random_state` list are not numerical the imputer will return an error. In addition, the variables indicated as seed should not contain missing values themselves.

Important for GDPR

This estimator stores a copy of the training set when the `fit()` method is called. Therefore, the object can become quite heavy. Also, it may not be GDPR compliant if your training data set contains Personal Information. Please check if this behaviour is allowed within your organisation.

Below a code example using the House Prices Dataset (more details about the dataset *here*).

First, let's load the data and separate it into train and test:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

from feature_engine.imputation import RandomSampleImputer

# Load dataset
data = pd.read_csv('houseprice.csv')

# Separate into train and test sets
X_train, X_test, y_train, y_test = train_test_split(
    data.drop(['Id', 'SalePrice'], axis=1),
    data['SalePrice'],
    test_size=0.3,
    random_state=0
)
```

In this example, we sample values at random, observation per observation, using as seed the value of the variable 'MSSubClass' plus the value of the variable 'YrSold'. Note that this value might be different for each observation.

The *RandomSampleImputer()* will impute all variables in the data, as we left the default value of the parameter `variables` to `None`.

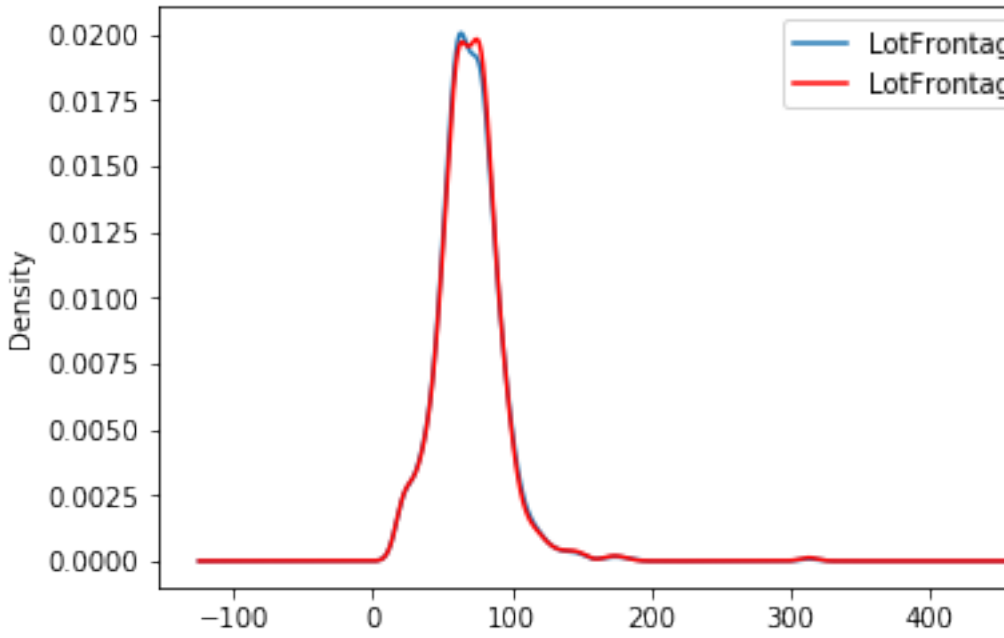
```
# set up the imputer
imputer = RandomSampleImputer(
        random_state=['MSSubClass', 'YrSold'],
        seed='observation',
        seeding_method='add'
    )

# fit the imputer
imputer.fit(X_train)
```

With `fit()` the imputer stored a copy of the X_train. And with transform, it will extract values at random from this X_train to replace NA in the datasets indicated in the `transform()` methods.

```
# transform the data
train_t = imputer.transform(X_train)
test_t = imputer.transform(X_test)
```

The beauty of the random sampler is that it preserves the original variable distribution:

```
fig = plt.figure()
ax = fig.add_subplot(111)
X_train['LotFrontage'].plot(kind='kde', ax=ax)
train_
↪t['LotFrontage'].plot(kind='kde', ax=ax, color='red')
lines, labels = ax.get_legend_handles_labels()
ax.legend(lines, labels, loc='best')
```



Additional resources

In the following Jupyter notebook you will find more details on the functionality of the *RandomSampleImputer()*, including how to set the different types of seeds.

- Jupyter notebook

All Feature-engine notebooks can be found in a dedicated repository.

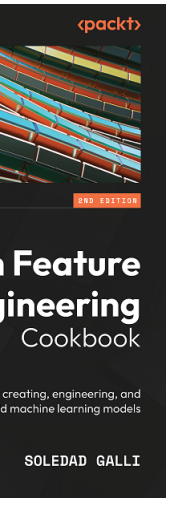
And finally, there is also a lot of information about this and other imputation techniques in this online course:

Or read our book:

Fig. 9: Feature Engineering for Machine Learning

Both our book and course are suitable for beginners and more advanced data scientists alike. By purchasing them you are supporting Sole, the main developer of Feature-engine.

AddMissingIndicator



The *AddMissingIndicator()* adds a binary variable indicating if observations are missing (missing indicator). It adds missing indicators to both categorical and numerical variables.

You can select the variables for which the missing indicators should be created passing a variable list to the `variables` parameter. Alternatively, the imputer will automatically select all variables.

The imputer has the option to add missing indicators to all variables or only to those that have missing data in the train set. You can change the behaviour using the parameter `missing_only`.

If `missing_only=True`, missing indicators will be added only to those variables with missing data in the train set. This means that if you passed a variable list to `variables` and some of those variables did not have missing data, no

missing indicators will be added to them. If it is paramount that all variables in your list get their missing indicators, make sure to set `missing_only=False`.

It is recommended to use `missing_only=True` when not passing a list of variables to impute.

Below a code example using the House Prices Dataset (more details about the dataset *here*).

First, let's load the data and separate it into train and test:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

from feature_engine.imputation import AddMissingIndicator

# Load dataset
data = pd.read_csv('houseprice.csv')


# Separate into train and test sets
X_train, X_test, y_train, y_test = train_test_split(
data.drop(['Id', 'SalePrice'], axis=1),
↪ data['SalePrice'], test_size=0.3, random_state=0)
```

Now we set up the imputer to add missing indicators to the 4 indicated variables:

```
# set up the imputer
addBinary_imputer = AddMissingIndicator(
variables=[
↪'Alley', 'MasVnrType', 'LotFrontage', 'MasVnrArea'],
)

# fit the imputer
addBinary_imputer.fit(X_train)
```

Because we left the default value for `missing_only`, the *AddMissingIndicator()* will check if the variables indicated above have missing data in X_train. If they do, missing indicators will be added for all 4 variables looking forward. If one of them had not had missing data in X_train, missing indicators would have been added to the remaining 3 variables only.

We can know which variables will have missing indicators by looking at the variable list in the *AddMissingIndicator()*'s attribute `variables_`.

Now, we can go ahead and add the missing indicators:

```
# transform the data
train_t = addBinary_imputer.transform(X_train)
test_t = addBinary_imputer.transform(X_test)

train_t[['Alley_na', 'MasVnrType_
↪na', 'LotFrontage_na', 'MasVnrArea_na']].head()
```

	Alley_na	MasVnrType_na	LotFrontage_na	MasVnrArea_na
64	1	0	1	0
682	1	0	1	0
960	1	0	0	0
1384	1	0	0	0
1100	1	0	0	0

Note that after adding missing indicators, we still need to replace NA in the original variables if we plan to use them to train machine learning models.

Tip

Missing indicators are commonly used together with random sampling, mean or median imputation, or frequent category imputation.

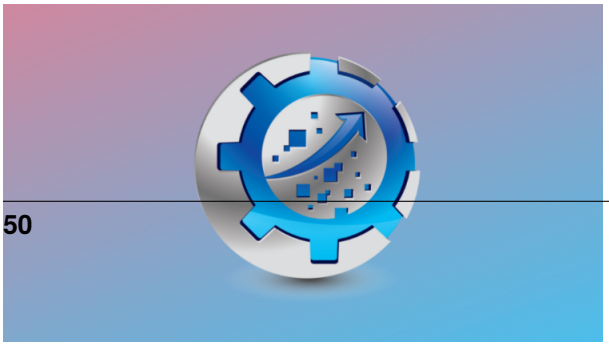
Additional resources

In the following Jupyter notebook you will find more details on the functionality of the *AddMissingIndicator()*, including how to use the parameter `missing_indicator` and how to select the variables automatically.

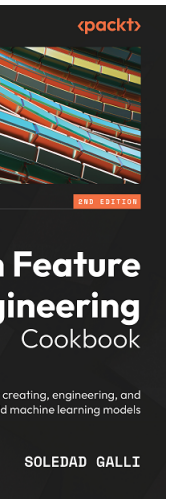
- Jupyter notebook

For more details about this and other feature engineering methods check out these resources:

Or read our book:



Both our book and course are suitable for beginners and more advanced data scientists alike. By purchasing them you are supporting Sole, the main developer of Feature-engine.



ure Engineering

DropMissingData

Removing rows with nan values from a dataset is a common practice in data science and machine learning projects.

You are probably familiar with the use of pandas dropna. You basically take a pandas dataframe or a pandas series, apply dropna, and eliminate those rows that contain nan values in one or more columns.

Here, we have an example of that syntax:

```
import numpy as np
import pandas as pd

X = pd.DataFrame(dict(
       x1 = [np.nan,1,1,0,np.nan],
       x2 = ["a", np.nan, "b", np.nan, "a"],
       ))

X.dropna(inplace=True)
print(X)
```

The previous code returns a dataframe without missing values:

```
    x1 x2
2  1.0  b
```

Feature-engine's *DropMissingData()* wraps pandas dropna in a transformer that will remove rows with na values while adhering to scikit-learn's `fit` and `transform` functionality.

Here we have a snapshot of *DropMissingData()*'s syntax:

```
import pandas as pd
import numpy as np
from feature_engine.imputation import DropMissingData
```

(continues on next page)

(continued from previous page)

```
X = pd.DataFrame(dict(
       x1 = [np.nan,1,1,0,np.nan],
       x2 = ["a", np.nan, "b", np.nan, "a"],
       ))

dmd = DropMissingData()
dmd.fit(X)
dmd.transform(X)
```

The previous code returns a dataframe without missing values:

```
    x1 x2
2  1.0  b
```

DropMissingData() allows you therefore to remove null values as part of any scikit-learn feature engineering workflow.

DropMissingData

DropMissingData() has some advantages over pandas:

- It learns and stores the variables for which rows with nan values should be deleted.
- It can be used within a Scikit-learn like pipeline.

With *DropMissingData()*, you can drop nan values from numerical and categorical variables. In other words, you can remove null values from numerical, categorical or object datatypes.

You have the option to remove nan values from all columns or only from a subset of them. Alternatively, you can remove rows if they have more than a certain percentage of nan values.

Let's better illustrate *DropMissingData()*'s functionality through code examples.

Dropna

Let's start by importing pandas and numpy, and creating a toy dataframe with nan values in 2 columns:

```
import numpy as np
import pandas as pd

from feature_engine.imputation import DropMissingData

X = pd.DataFrame(
    dict(
        x1=[2, 1, 1, 0, np.nan],
        x2=["a", np.nan, "b", np.nan, "a"],
        x3=[2, 3, 4, 5, 5],
```

(continues on next page)

(continued from previous page)

```
    )
)
y = pd.Series([1, 2, 3, 4, 5])

print(X.head())
```

Below we see the new dataframe:

```
    x1   x2  x3
0  2.0    a   2
1  1.0  NaN   3
2  1.0    b   4
3  0.0  NaN   5
4  NaN    a   5
```

We can drop nan values across all columns as follows:

```
dmd =  DropMissingData()
Xt = dmd.fit_transform(X)
Xt.head()
```

We see the transformed dataframe without null values:

```
    x1 x2  x3
0  2.0  a   2
2  1.0  b   4
```

By default, *DropMissingData()* will find and store the columns that had missing data during fit, that is, in the training set. They are stored here:

```
dmd.variables_
```

```
['x1', 'x2']
```

That means that every time that we apply `transform()` to a new dataframe, the transformer will remove rows with nan values only in those columns.

If we want to force *DropMissingData()* to drop na across all columns, regardless of whether they had nan values during fit, we need to set up the class like this:

```
dmd =  DropMissingData(missing_only=False)
Xt = dmd.fit_transform(X)
```

Now, when we explore the paramter `variables_`, we see that all the variables in the train set are stored, and hence, will be used to remove nan values:

```
dmd.variables_
```

```
['x1', 'x2', 'x3']
```

Adjust target after dropna

DropMissingData() has the option to remove rows with nan from both training set and target variable. Like this, we can obtain a target that is aligned with the resulting dataframe after the transformation.

The method `transform_x_y` removes rows with null values from the train set, and then realigns the target. Let's take a look:

```
Xt, yt = dmd.transform_x_y(X, y)
Xt
```

Below we see the dataframe without nan:

```
    x1 x2  x3
0  2.0  a   2
2  1.0  b   4
```

```
yt
```

And here we see the target with those rows corresponing to the remaining rows in the transformed dataframe:

```
0    1
2    3
dtype: int64
```

Let's check that the shape of the transformed dataframe and target are the same:

```
Xt.shape, yt.shape
```

We see that the resulting training set and target have each 2 rows, instead of the 5 original rows.

```
((2, 3), (2,))
```

Return the rows with nan

When we have a model in production, it might be useful to know which rows are being removed by the transformer. We can obtain that information as follows:

```
dmd.return_na_data(X)
```

The previous command returns the rows with nan. In other words, it does the opposite of `transform()`, or pandas.dropna.

```
    x1   x2  x3
1  1.0  NaN   3
3  0.0  NaN   5
4  NaN    a   5
```

Dropna from subset of variables

We can choose to remove missing data only from a specific column or group of columns. We just need to pass the column name or names to the `variables` parameter:

Here, we'll dropna from the variables "x1", "x3".

```
dmd = DropMissingData(variables=[
↪"x1", "x3"], missing_only=False)
Xt = dmd.fit_transform(X)
Xt.head()
```

Below, we see the transformed dataframe. It removed the rows with nan in "x1", and we see that those rows with nan in "x2" are still in the dataframe:

```
    x1   x2  x3
0  2.0    a   2
1  1.0  NaN   3
2  1.0    b   4
3  0.0  NaN   5
```

Only rows with nan in "x1" and "x3" are removed. We can corroborate that by examining the `variables_` parameter:

Important

When you indicate which variables should be examined to remove rows with nan, make sure you set the parameter `missing_only` to the boolean `False`. Otherwise, *DropMissingData()* will select from your list only those variables that showed nan values in the train set.

See for example what happens when we set up the class like this:

```
dmd = DropMissingData(variables=[
↪"x1", "x3"], missing_only=True)
Xt = dmd.fit_transform(X)
dmd.variables_
```

Note, that we indicated that we wanted to remove nan from "x1", "x3". Yet, only "x1" has nan in X. So the transformer learns that nan should be only dropped from "x1":

```
['x1']
```

DropMissingData() took the 2 variables indicated in the list, and stored only the one that showed nan in during fit. That means that when transforming future dataframes, it will only remove rows with nan in "x1".

In other words, if you pass a list of variables to impute and set `missing_only=True`, and some of the variables in your list do not have missing data in the train set, missing data will not be removed during transform for those particular variables.

When `missing_only=True`, the transformer "double checks" that the entered variables have missing data in the train set. If not, it ignores them during `transform()`.

It is recommended to use `missing_only=True` when not passing a list of variables to impute.

Dropna based on percentage of non-nan values

We can set *DropMissingData()* to require a percentage of non-NA values in a row to keep it. We can control this behaviour through the `threshold` parameter, which is equivalent to pandas.dropna's `thresh` parameter.

If `threshold=1`, all variables need to have data to keep a row. If `threshold=0.5`, 50% of the variables need to have data to keep a row. If `threshold=0.01`, 10% of the variables need to have data to keep the row. If `threshold=None`, rows with NA in any of the variables will be dropped.

Let's see this with an example. We create a new dataframe that has different proportion of non-nan values in every row.

```
X = pd.DataFrame(
    dict(
        x1=[2, 1, 1, np.nan, np.nan],
        x2=["a", np.nan, "b", np.nan, np.nan],
        x3=[2, 3, 4, 5, np.nan],
    )
)
X
```

We see that the bottom row has nan in all columns, row 3 has nan in 2 of 3 columns, and row 1 has nan in 1 variable:

```
    x1   x2   x3
0  2.0    a  2.0
1  1.0  NaN  3.0
2  1.0    b  4.0
3  NaN  NaN  5.0
4  NaN  NaN  NaN
```

Now, we can set *DropMissingData()* to drop rows if >50% of its values are nan:

```
dmd = DropMissingData(threshold=.5)
dmd.fit(X)
dmd.transform(X)
```

We see that the last 2 rows are dropped, because they have more than 50% nan values.

```
    x1   x2   x3
0  2.0    a  2.0
1  1.0  NaN  3.0
2  1.0    b  4.0
```

Instead, we can set class:`DropMissingData()` to drop rows if >70% of its values are nan as follows:

```
dmd = DropMissingData(threshold=.3)
dmd.fit(X)
dmd.transform(X)
```

Now we see that only the last row was removed.

```
    x1   x2   x3
0  2.0    a  2.0
1  1.0  NaN  3.0
2  1.0    b  4.0
3  NaN  NaN  5.0
```

Scikit-learn compatible

DropMissingData() is fully compatible with the Scikit-learn API, so you will find common methods that you also find in Scikit-learn transformers, like, for example, the `get_feature_names_out()` method to obtain the variable names in the transformed dataframe.

Pipeline

When we dropna from a dataframe, we then need to realign the target. We saw previously that we can do that by using the method `transform_x_y`.

We can align the target with the resulting dataframe automatically from within a pipeline as well, by utilizing Feature-engine's pipeline.

Let's start by importing the necessary libraries:

```
import numpy as np
import pandas as pd

from feature_engine.imputation import DropMissingData
from feature_engine.encoding import OrdinalEncoder
from feature_engine.pipeline import Pipeline
```

Let's create a new dataframe with nan values in some rows, two numerical and one categorical variable, and its corresponding target variable:

```
X = pd.DataFrame(
    dict(
        x1=[2, 1, 1, 0, np.nan],
        x2=["a", np.nan, "b", np.nan, "a"],
        x3=[2, 3, 4, 5, 5],
    )
)
y = pd.Series([1, 2, 3, 4, 5])

X.head()
```

Below, we see the resulting dataframe:

```
    x1   x2  x3
0  2.0    a   2
1  1.0  NaN   3
2  1.0    b   4
3  0.0  NaN   5
4  NaN    a   5
```

Let's now set up a pipeline to dropna first, and then encode the categorical variable by using ordinal encoding:

```
pipe = Pipeline(
    [
        ("drop", DropMissingData()),
    ␣
↪    ("enc", OrdinalEncoder(encoding_method="arbitrary")),
    ]
)

pipe.fit_transform(X, y)
```

When we apply `fit` and `transform` or `fit_transform`, we will obtain the transformed training set only:

```
    x1  x2  x3
0  2.0   0   2
2  1.0   1   4
```

To obtain the transform training set and target, we use `transform_x_y`:

```
pipe.fit(X,y)
Xt, yt = pipe.transform_x_y(X, y)
Xt
```

Here we see the transformed training set:

```
    x1  x2  x3
0  2.0   0   2
2  1.0   1   4
```

```
yt
```

And here we see the re-aligned target variable:

```
0    1
2    3
```

And to wrap up, let's add an estimator to the pipeline:

```
import numpy as np
import pandas as pd

from sklearn.linear_model import Lasso

from feature_engine.imputation import DropMissingData
```

(continues on next page)

(continued from previous page)

```
from feature_engine.encoding import OrdinalEncoder
from feature_engine.pipeline import Pipeline

df = pd.DataFrame(
    dict(
        x1=[2, 1, 1, 0, np.nan],
        x2=["a", np.nan, "b", np.nan, "a"],
        x3=[2, 3, 4, 5, 5],
    )
)
y = pd.Series([1, 2, 3, 4, 5])

pipe = Pipeline(
    [
        ("drop", DropMissingData()),
        ("enc", OrdinalEncoder(encoding_method="arbitrary")),
        ("lasso", Lasso(random_state=2))
    ]
)

pipe.fit(df, y)
pipe.predict(df)
```

```
array([2., 2.])
```

Dropna or fillna?

DropMissingData() has the same functionality than `pandas.series.dropna` or `pandas.dataframe.dropna`. If you want functionality compatible with `pandas.fillna` instead, check our other imputation transformers.

Drop columns with nan

At the moment, Feature-engine does not have transformers that will find columns with a certain percentage of missing values and drop them. Instead, you can find those columns manually, and then drop them with the help of `DropFeatures` from the selection module.

See also

Check out our tutorials on `LagFeatures` and `WindowFeatures` to see how to combine *DropMissingData()* with lags or rolling windows, to create features for forecasting.

Tutorials, books and courses

In the following Jupyter notebook, in our accompanying Github repository, you will find more examples using `DropMissingData()`.

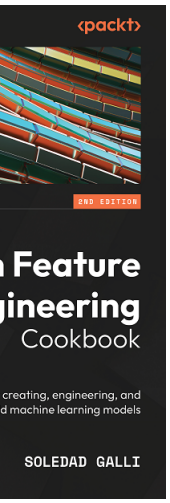
- Jupyter notebook

For tutorials about this and other feature engineering methods check out our online course:

Or read our book:

Fig. 13: Feature Engineering for Machine Learning

Both our book and course are suitable for beginners and more advanced data scientists alike. By purchasing them you are supporting Sole, the main developer of Feature-engine.



ure Engineering

Categorical Encoding

Feature-engine's categorical encoders replace variable strings by estimated or arbitrary numbers. The following image summarizes the main encoder's functionality.

Summary of Feature-engine's encoders characteristics

Transformer	Regression	Classification	Multi-class	Description
OneHotEncoder()				Adds dummy variables to represent each category
OrdinalEncoder()				Replaces categories with an integer
CountFreuencyEncoder()				Replaces categories with their count or frequency
MeanEncoder()			x	Replaces categories with the targe mean value
WoEEncoder()	x		x	Replaces categories with the weight of the evidence
DecisionTreeEncoder()				Replaces categories with the predictions of a decision tree
RareLabelEncoder()				Groups infrequent categories into a single one

Feature-engine's categorical encoders work only with categorical variables by default. From version 1.1.0, you have the option to set the parameter ignore_format to False, and make the transformers also accept numerical variables as input.

Monotonicity

Most Feature-engine's encoders will return, or attempt to return monotonic relationships between the encoded variable and the target. A monotonic relationship is one in which the variable value increases as the values in the other variable increase, or decrease. See the following illustration as examples:

Monotonic relationships tend to help improve the performance of linear models and build shallower decision trees.

Regression vs Classification

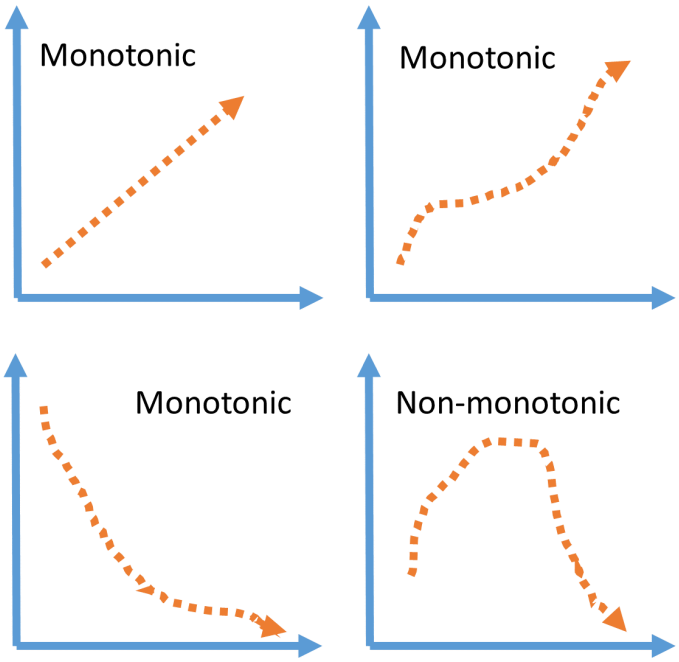
Most Feature-engine's encoders are suitable for both regression and classification, with the exception of the *WoEEncoder()* and the PRatioEncoder() which are designed solely for **binary** classification.

Multi-class classification

Finally, some Feature-engine's encoders can handle multi-class targets off-the-shelf for example the *OneHotEncoder()*, the :class:CountFrequencyEncoder()` and the *DecisionTreeEncoder()*.

Note that while the *MeanEncoder()* and the *OrdinalEncoder()* will operate with multi-class targets, but the mean of the classes may not be significant and this will defeat the purpose of these encoding techniques.

Encoders



OneHotEncoder

One-hot encoding is a method used to represent categorical data, where each category is represented by a binary variable. The binary variable takes the value 1 if the category is present and 0 otherwise. The binary variables are also known as dummy variables.

To represent the categorical feature "is-smoker" with categories "Smoker" and "Non-smoker", we can generate the dummy variable "Smoker", which takes 1 if the person smokes and 0 otherwise. We can also generate the variable "Non-smoker", which takes 1 if the person does not smoke and 0 otherwise.

The following table shows a possible one hot encoded representation of the variable "is smoker":

is smoker	smoker	non-smoker
smoker	1	0
non-smoker	0	1
non-smoker	0	1
smoker	1	0
non-smoker	0	1

For the categorical variable **Country** with values **England**, **Argentina**, and **Germany**, we can create three variables called `England`, `Argentina`, and `Germany`. These variables will take the value of 1 if the observation is England, Argentina, or Germany, respectively, and 0 otherwise.

Encoding into k vs k-1 variables

A categorical feature with k unique categories can be encoded using k-1 binary variables. For `Smoker`, k is 2 as it contains two labels (Smoker and Non-Smoker), so we only need one binary variable (k - 1 = 1) to capture all of the information.

In the following table we see that the dummy variable `Smoker` fully represents the original categorical values:

is smoker	smoker
smoker	1
non-smoker	0
non-smoker	0
smoker	1
non-smoker	0

For the **Country** variable, which has three categories (k=3; England, Argentina, and Germany), we need two (k - 1 = 2) binary variables to capture all the information. The variable will be fully represented like this:

Country	England	Argentina
England	1	0
Argentina	0	1
Germany	0	0

As we see in the previous table, if the observation is England, it will show the value 1 in the `England` variable; if the observation is Argentina, it will show the value 1 in the `Argentina` variable; and if the observation is Germany, it will show zeroes in both dummy variables.

Like these, by looking at the values of the k-1 dummies, we can infer the original categorical value of each observation.

Encoding into k-1 binary variables is well-suited for linear regression models. Linear models evaluate all features during fit, thus, with k-1 they have all the information about the original categorical variable.

There are a few occasions in which we may prefer to encode the categorical variables with k binary variables.

Encode into k dummy variables if training decision trees based models or performing feature selection. Decision tree based models and many feature selection algorithms evaluate variables or groups of variables separately. Thus, if encoding into k-1, the last category will not be examined. In other words, we lose the information contained in that category.

Binary variables

When a categorical variable has only 2 categories, like "Smoker" in our previous example, then encoding into k-1 suits all purposes, because the second dummy variable created by one hot encoding is completely redundant.

Encoding popular categories

One hot encoding can increase the feature space dramatically, particularly if we have many categorical features, or the features have high cardinality. To control the feature space, it is common practice to encode only the most frequent categories in each categorical variable.

When we encode the most frequent categories, we will create binary variables for each of these frequent categories, and when the observation has a different, less popular category, it will have a 0 in all binary variables. See the following example:

var	popular1	popular2
popular1	1	0
popular2	0	1
popular1	1	0
non-popular	0	0
popular2	0	1
less popular	0	0
unpopular	0	0
lonely	0	0

As we see in the previous table, less popular categories are represented as a group by showing zeroes in all binary variables.

OneHotEncoder

Feature-engine's *OneHotEncoder()* encodes categorical data as a one-hot numeric dataframe.

OneHotEncoder() can encode into k or k-1 dummy variables. The behaviour is specified through the `drop_last` parameter, which can be set to `False` for k, or to `True` for k-1 dummy variables.

OneHotEncoder() can specifically encode binary variables into k-1 variables (that is, 1 dummy) while encoding categorical features of higher cardinality into k dummies. This behaviour is specified by setting the parameter `drop_last_binary=True`. This will ensure that for every binary variable in the dataset, that is, for every categorical variable with ONLY 2 categories, only 1 dummy is created. This is recommended, unless you suspect that the variable could, in principle, take more than 2 values.

OneHotEncoder() can also create binary variables for the **n** most popular categories, n being determined by the user. For example, if we encode only the 6 more popular categories, by setting the parameter `top_categories=6`, the transformer will add binary variables only for the 6 most frequent categories.

The most frequent categories are those with the greatest number of observations. The remaining categories will show zeroes in each one of the derived dummies. This behaviour is useful when the categorical variables are highly cardinal to control the expansion of the feature space.

Note

The parameter `drop_last` is ignored when encoding the most popular categories.

Python implementation

Let's look at an example of one hot encoding, using Feature-engine's *OneHotEncoder()* utilizing the Titanic Dataset.

We'll start by importing the libraries, functions and classes, and loading the data into a pandas dataframe and dividing it into a training and a testing set:

```
from sklearn.model_selection import train_test_split
from feature_engine.datasets import load_titanic
from feature_engine.encoding import OneHotEncoder

X, y = load_titanic(
    return_X_y_frame=True,
    handle_missing=True,
    predictors_only=True,
    cabin="letter_only",
)

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=0,
)

print(X_train.head())
```

We see the first 5 rows of the training data below:

```
      pclass␣
↪     sex        age  sibsp  parch     fare cabin embarked
501        2␣
↪ female  13.000000      0      1  19.5000     M        S
588        2␣
↪ female   4.000000      1      1  23.0000     M        S
402        2␣
↪ female  30.000000      1      0  13.8583     M        C
1193       3␣
↪   male  29.881135      0      0   7.7250     M        Q
686        3␣
↪ female  22.000000      0      0   7.7250     M        Q
```

Let's explore the cardinality of 4 of the categorical features:

```
X_train[['sex', 'pclass', 'cabin', 'embarked']].nunique()
```

```
sex         2
pclass      3
cabin       9
embarked    4
dtype: int64
```

We see that the variable sex has 2 categories, pclass has 3 categories, the variable cabin has 9 categories, and the variable embarked has 4 categories.

Let's now set up the OneHotEncoder to encode 2 of the categorical variables into k-1 dummy variables:

```
encoder = OneHotEncoder(
    variables=['cabin', 'embarked'],
    drop_last=True,
    )

encoder.fit(X_train)
```

With `fit()` the encoder learns the categories of the variables, which are stored in the attribute `encoder_dict_`.

```
encoder.encoder_dict_
```

```
{'cabin': ['M', 'E', 'C', 'D', 'B', 'A', 'F', 'T'],
 'embarked': ['S', 'C', 'Q']}
```

The `encoder_dict_` contains the categories that will be represented by dummy variables for each categorical variable.

With transform, we go ahead and encode the variables. Note that by default, the *OneHotEncoder()* drops the original categorical variables, which are now represented by the one-hot array.

```
train_t = encoder.transform(X_train)
test_t = encoder.transform(X_test)

print(train_t.head())
```

Below we see the one hot dummy variables added to the dataset and the original variables are no longer in the dataframe:

```
      pclass     sex␣
↪       age  sibsp  parch     fare  cabin_M  cabin_E  \
501        2  female␣
↪ 13.000000      0      1  19.5000        1        0
588        2  female␣
↪  4.000000      1      1  23.0000        1        0
402        2  female␣
↪ 30.000000      1      0  13.8583        1        0
1193       3    ␣
↪male  29.881135      0      0   7.7250        1        0
686        3  female␣
↪ 22.000000      0      0   7.7250        1        0
```

(continues on next page)

(continued from previous page)

```
      cabin_C  cabin_
↪D  cabin_B  cabin_A  cabin_F  cabin_T  embarked_S  \
501         0␣
↪       0         0        0        0         0           1
588         0␣
↪       0         0        0        0         0           1
402         0␣
↪       0         0        0        0         0           0
1193        0␣
↪       0         0        0        0         0           0
686         0␣
↪       0         0        0        0         0           0

      embarked_C  embarked_Q
501            0           0
588            0           0
402            1           0
1193           0           1
686            0           1
```

Finding categorical variables automatically

Feature-engine's *OneHotEncoder()* can automatically find and encode all categorical features in the pandas dataframe. Let's show that with an example.

Let's set up the OneHotEncoder to find and encode all categorical features:

```
encoder = OneHotEncoder(
    variables=None,
    drop_last=True,
    )

encoder.fit(X_train)
```

With fit, the encoder finds the categorical features and identifies it's unique categories. We can find the categorical variables like this:

```
encoder.variables_
```

```
['sex', 'cabin', 'embarked']
```

And we can identify the unique categories for each variables like this:

```
encoder.encoder_dict_
```

```
{'sex': ['female'],
 'cabin': ['M', 'E', 'C', 'D', 'B', 'A', 'F', 'T'],
 'embarked': ['S', 'C', 'Q']}
```

We can now encode the categorical variables:

```
train_t = encoder.transform(X_train)
test_t = encoder.transform(X_test)

print(train_t.head())
```

And here we see the resulting dataframe:

```
      pclass        age␣
↪ sibsp  parch     fare  sex_female  cabin_M  cabin_E  \
501        2  13.000000␣
↪     0      1  19.5000           1        1        0
588        2   4.000000␣
↪     1      1  23.0000           1        1        0
402        2  30.000000␣
↪     1      0  13.8583           1        1        0
1193       3  29.881135␣
↪     0      0   7.7250           0        1        0
686        3  22.000000␣
↪     0      0   7.7250           1        1        0

      cabin_C  cabin_
↪D  cabin_B  cabin_A  cabin_F  cabin_T  embarked_S  \
501         0␣
↪       0        0        0        0        0           1
588         0␣
↪       0        0        0        0        0           1
402         0␣
↪       0        0        0        0        0           0
1193        0␣
↪       0        0        0        0        0           0
686         0␣
↪       0        0        0        0        0           0

      embarked_C  embarked_Q
501            0           0
588            0           0
402            1           0
1193           0           1
686            0           1
```

Encoding variables of type numeric

By default, Feature-engine's *OneHotEncoder()* will only encode categorical features. If you attempt to encode a variable of numeric dtype, it will raise an error. To avoid this error, you can instruct the encoder to ignore the data type format as follows:

```
enc = OneHotEncoder(
    variables=['pclass'],
    drop_last=True,
    ignore_format=True,
    )
```

(continues on next page)

(continued from previous page)

```
enc.fit(X_train)

train_t = enc.transform(X_train)
test_t = enc.transform(X_test)

print(train_t.head())
```

Note that pclass had numeric values instead of strings, and it was one hot encoded by the transformer into 2 dummies:

```
        sex
↪  age  sibsp  parch     fare cabin embarked  pclass_2  \
501   female  13.
↪000000      0      1  19.5000     M        S         1
588   female
↪4.000000      1      1  23.0000     M        S         1
402   female  30.
↪000000      1      0  13.8583     M        C         1
1193    male  29.
↪881135      0      0   7.7250     M        Q         0
686   female  22.
↪000000      0      0   7.7250     M        Q         0

      pclass_3
501          0
588          0
402          0
1193         1
686          1
```

Encoding binary variables into 1 dummy

With Feature-engine's *OneHotEncoder()* we can encode all categorical variables into k dummies and the binary variables into k-1 by setting the encoder as follows:

```
ohe = OneHotEncoder(
    variables=['sex', 'cabin','embarked'],
    drop_last=False,
    drop_last_binary=True,
    )

train_t = ohe.fit_transform(X_train)
test_t = ohe.transform(X_test)

print(train_t.head())
```

As we see in the following input, for the variable sex, we have only have 1 dummy, and for all the rest we have k dummies:

```
      pclass        age␣
↪ sibsp  parch     fare  sex_female  cabin_M  cabin_E  \
501        2  13.000000␣
↪     0      1  19.5000           1        1        0
588        2   4.000000␣
↪     1      1  23.0000           1        1        0
402        2  30.000000␣
↪     1      0  13.8583           1        1        0
1193       3  29.881135␣
↪     0      0   7.7250           0        1        0
686        3  22.000000␣
↪     0      0   7.7250           1        1        0

      cabin_C␣
↪ cabin_D  cabin_B  cabin_A  cabin_F  cabin_T  cabin_G  \
501       ␣
↪ 0        0        0        0        0        0        0
588       ␣
↪ 0        0        0        0        0        0        0
402       ␣
↪ 0        0        0        0        0        0        0
1193      ␣
↪ 0        0        0        0        0        0        0
686       ␣
↪ 0        0        0        0        0        0        0

      embarked_S  embarked_C  embarked_Q  embarked_Missing
501            1           0           0                 0
588            1           0           0                 0
402            0           1           0                 0
1193           0           0           1                 0
```

Encoding frequent categories

If the categorical variables are highly cardinal, we may end up with very big datasets after one hot encoding. In addition, if some of these variables are fairly constant or fairly similar, we may end up with one hot encoded features that are highly correlated, if not identical. To avoid this behaviour, we can encode only the most frequent categories.

To encode the 2 most frequent categories of each categorical column, we set up the transformer as follows:

```
ohe = OneHotEncoder(
    top_categories=2,
    variables=['pclass', 'cabin', 'embarked'],
    ignore_format=True,
    )

train_t = ohe.fit_transform(X_train)
test_t = ohe.transform(X_test)
```

(continues on next page)

(continued from previous page)

```
print(train_t.head())
```

As we see in the resulting dataframe, we created only 2 dummies per variable:

```
        sex        age␣
↪ sibsp  parch     fare  pclass_3  pclass_1  cabin_M  \
501   female  13.000000␣
↪     0      1  19.5000         0         0        1
588   female   4.000000␣
↪     1      1  23.0000         0         0        1
402   female  30.000000␣
↪     1      0  13.8583         0         0        1
1193    male  29.881135␣
↪     0      0   7.7250         1         0        1
686   female  22.000000␣
↪     0      0   7.7250         1         0        1

      cabin_C  embarked_S  embarked_C
501         0           1           0
588         0           1           0
402         0           0           1
1193        0           0           0
686         0           0           0
```

Finally, if we want to obtain the column names in the resulting dataframe we can do the following:

```
encoder.get_feature_names_out()
```

We see the names of the columns below:

```
['sex',
 'age',
 'sibsp',
 'parch',
 'fare',
 'pclass_3',
 'pclass_1',
 'cabin_M',
 'cabin_C',
 'embarked_S',
 'embarked_C']
```

Considerations

Encoding categorical variables into k dummies, will handle unknown categories automatically. Those features not seen during training will show zeroes in all dummies.

Encoding categorical features into k-1 dummies, will cause unseen data to be treated as the category that is dropped.

Encoding the top categories will make unseen categories part of the group of less popular categories.

If you add a big number of dummy variables to your data, many may be identical or highly correlated. Consider dropping identical and correlated features with the transformers from the *selection module*.

For alternative encoding methods used in data science check the `OrdinalEncoder()` and other encoders included in the *encoding module*.

Tutorials, books and courses

For more details into `OneHotEncoder()`'s functionality visit:

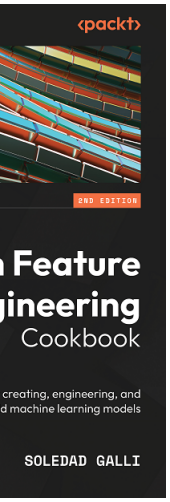
- Jupyter notebook

For tutorials about this and other data preprocessing methods check out our online course:

Or read our book:

Fig. 15: Feature Engineering for Machine Learning

Both our book and course are suitable for beginners and more advanced data scientists alike. By purchasing them you are supporting Sole, the main developer of Feature-engine.



ure Engineering

CountFrequencyEncoder

Count encoding and frequency encoding are 2 categorical encoding techniques that were commonly used during data preprocessing in Kaggle's data science competitions, even when their predictive value is not immediately obvious.

Count encoding consists of replacing the categories of categorical features by their counts, which are estimated from the training set. For example, in the variable color, if 10 observations are blue and 5 observations are red, blue will be replaced by 10 and red by 5.

Frequency encoding consists of replacing the labels of categorical data with their frequency, which is also estimated from the training set. Then, in the variable City, if London appears in 10% of the observations and Bristol in 1%, London will be replaced by 0.1 and Bristol with 0.01.

Count and frequency encoding in machine learning

We'd use count encoding or frequency encoding when we think that the representation of the categories in the dataset has some sort of predictive value. To be honest, the only example that I can think of where count encoding could be useful is in sales forecasting or sales data analysis scenarios, where the count of a product or an item represents its popularity. In other words, we may be more likely to sell a product with a high count.

Count encoding and frequency encoding can be suitable for categorical variables with high cardinality because these types of categorical encoding will cause what is called collisions: categories that are present in a similar number of observations will be replaced with similar, if not the same values, which reduces the variability.

This, of course, can result in the loss of information by placing two categories that are otherwise different in the same pot. But on the other hand, if we are using count encoding or frequency encoding, we have reasons to believe that the count or the frequency are a good indicator of predictive performance or somehow capture data insight, so that categories with similar counts would show similar patterns or behaviors.

Count and Frequency encoding with Feature-engine

The *CountFrequencyEncoder()* replaces categories of categorical features by either the count or the percentage of observations each category shows in the training set.

With *CountFrequencyEncoder()* we can automatically encode all categorical features in the dataset, or only a subset of them, by passing the variable names in a list to the `variables` parameter when we set up the encoder.

By default, *CountFrequencyEncoder()* will encode only categorical data. If we want to encode numerical values, we need to explicitly say so by setting the parameter `ignore_format` to True.

Count and frequency encoding with unseen categories

When we learn mappings from strings to numbers, either with count encoding or other encoding techniques like ordinal encoding or target encoding, we do so by observing the categories in the training set. Hence, we won't have mappings for categories that appear only in the test set. These are the so-called "unseen categories."

When encountering unseen categories, *CountFrequencyEncoder()* will ignore them by default, which means that unseen categories will be replaced with missing values. We can instruct the encoder to raise an error when a new category is encountered, or alternatively, to encode unseen categories with zero.

Count encoding vs other encoding methods

Count and frequency encoding, similar to ordinal encoding and contrarily to one-hot encoding, feature hashing, or binary encoding, does not increase the dataset dimensionality. From one categorical variable, we obtain one numerical feature.

Python example

Let's examine the functionality of *CountFrequencyEncoder()* by using the Titanic dataset. We'll start by loading the libraries and functions, loading the dataset, and then splitting it into a training and a testing set.

```
from sklearn.model_selection import train_test_split
from feature_engine.datasets import load_titanic
from feature_engine.encoding import CountFrequencyEncoder

X, y = load_titanic(
    return_X_y_frame=True,
    handle_missing=True,
    predictors_only=True,
    cabin="letter_only",
)

X_train, X_test, y_train, y_test = train_test_split(
```

(continues on next page)

(continued from previous page)

```
    X, y, test_size=0.3, random_state=0,
)

print(X_train.head())
```

We see the resulting dataframe with the predictor variables below:

```
      pclass␣
↪     sex        age  sibsp  parch     fare cabin embarked
501        2␣
↪ female  13.000000      0      1  19.5000     M        S
588        2␣
↪ female   4.000000      1      1  23.0000     M        S
402        2␣
↪ female  30.000000      1      0  13.8583     M        C
1193       3␣
↪   male  29.881135      0      0   7.7250     M        Q
686        3␣
↪ female  22.000000      0      0   7.7250     M        Q
```

This dataset has three obvious categorical features: cabin, embarked, and sex, and in addition, pclass could also be handled as a categorical.

Count encoding

We'll start by encoding the three categorical variables using their counts, that is, replacing the strings with the number of times each category is present in the training dataset.

```
encoder = CountFrequencyEncoder(
encoding_method='count',
variables=['cabin', 'sex', 'embarked'],
)

encoder.fit(X_train)
```

With `fit()`, the count encoder learns the counts of each category. We can inspect the counts as follows:

```
encoder.encoder_dict_
```

We see the counts of each category for each of the three variables in the following output:

```
{'cabin': {'M': 702,
  'C': 71,
  'B': 42,
  'E': 32,
  'D': 32,
  'A': 17,
  'F': 15,
  'G': 4,
```

(continues on next page)

(continued from previous page)

```
 'T': 1},
'sex': {'male': 581, 'female': 335},
'embarked': {'S': 652, 'C': 179, 'Q': 83, 'Missing': 2}}
```

Now, we can go ahead and encode the variables:

```
train_t = encoder.transform(X_train)
test_t = encoder.transform(X_test)

print(train_t.head())
```

We see the resulting dataframe where the categorical features are now replaced with integer values corresponding to the category counts:

```
      pclass␣
↪ sex        age  sibsp  parch     fare  cabin  embarked
501         2␣
↪ 335  13.000000      0      1  19.5000    702       652
588         2␣
↪ 335   4.000000      1      1  23.0000    702       652
402         2␣
↪ 335  30.000000      1      0  13.8583    702       179
1193        3␣
↪ 581  29.881135      0      0   7.7250    702        83
686         3␣
↪ 335  22.000000      0      0   7.7250    702        83
```

We can now use the encoded dataframes to train machine learning models.

Frequency encoding

Let's now perform frequency encoding. We'll encode 2 categorical and 1 numerical variable, hence, we need to set the encoder to ignore the variable's type:

```
encoder = CountFrequencyEncoder(
encoding_method='frequency',
variables=['cabin', 'pclass', 'embarked'],
ignore_format=True,
)
```

Now, we fit the frequency encoder to the train set and transform it straightaway, and then we transform the test set:

```
t_train = encoder.fit_transform(X_train)
t_test = encoder.transform(X_test)

test.head()
```

In the following output we see the transformed dataframe, where the categorical features are now encoded into their frequencies:

```
       pclass     sex␣
↪       age  sibsp  parch     fare     cabin  embarked
1139  0.543668    male␣
↪ 38.000000      0      0   7.8958  0.766376   0.71179
533   0.205240  female␣
↪ 21.000000      0      1  21.0000  0.766376   0.71179
459   0.205240    male␣
↪ 42.000000      1      0  27.0000  0.766376   0.71179
1150  0.543668    male␣
↪ 29.881135      0      0  14.5000  0.766376   0.71179
393   0.205240    male␣
↪ 25.000000      0      0  31.5000  0.766376   0.71179
```

With `fit()` the encoder learns the frequencies of each category, which are stored in its `encoder_dict_` parameter. We can inspect them like this:

```
encoder.encoder_dict_
```

In the `encoder_dict_` we find the frequencies for each one of the unique categories of each variable to encode. This way, we can map the original value to the new value.

```
{'cabin': {'M': 0.7663755458515283,
   'C': 0.07751091703056769,
   'B': 0.04585152838427948,
   'E': 0.034934497816593885,
   'D': 0.034934497816593885,
   'A': 0.018558951965065504,
   'F': 0.016375545851528384,
   'G': 0.004366812227074236,
   'T': 0.001091703056768559},
'pclass': {3: 0.5436681222707423,
   1: 0.25109170305676853,
   2: 0.2052401746724891},
'embarked': {'S': 0.7117903930131004,
   'C': 0.19541484716157206,
   'Q': 0.0906113537117904,
   'Missing': 0.002183406113537118}}
```

We can now use these dataframes to train machine learning algorithms.

With the method `inverse_transform`, we can transform the encoded dataframes back to their original representation, that is, we can replace the encoding with the original categorical values.

Additional resources

In the following notebook, you can find more details into the `CountFrequencyEncoder()` functionality and example plots with the encoded variables:

- Jupyter notebook

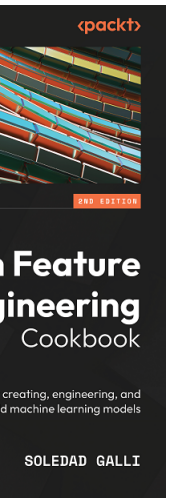
For more details about this and other feature engineering methods check out these resources:

Fig. 17: Feature Engineering for Machine Learning

Our book:

Fig. 18: Feature Engineering for Time Series Forecasting

Both our book and courses are suitable for beginners and
more advanced data scientists alike. By purchasing them you are supporting Sole, the main developer of Feature-engine.



ure Engineering

OrdinalEncoder

The *OrdinalEncoder()* replaces the categories by digits, starting from 0 to k-1, where k is the number of different categories. If you select **"arbitrary"** in the `encoding_method`, then the encoder will assign numbers as the labels appear in the variable (first come first served). If you select **"ordered"**, the encoder will assign numbers following the mean of the target value for that label. So labels for which the mean of the target is higher will get the number 0, and those where the mean of the target is smallest will get the number k-1. This way, we create a monotonic relationship between the encoded variable and the target.

Arbitrary vs ordered encoding

Ordered ordinal encoding: for the variable colour, if the mean of the target for blue, red and grey is 0.5, 0.8 and 0.1 respectively, blue is replaced by 1, red by 2 and grey by 0.

The motivation is to try and create a monotonic relationship between the target and the encoded categories. This tends to help improve performance of linear models.

Arbitrary ordinal encoding: the numbers will be assigned arbitrarily to the categories, on a first seen first served basis.

Let's look at an example using the Titanic Dataset.

First, let's load the data and separate it into train and test:

```
from sklearn.model_selection import train_test_split
from feature_engine.datasets import load_titanic
from feature_engine.encoding import OrdinalEncoder

X, y = load_titanic(
    return_X_y_frame=True,
    handle_missing=True,
    predictors_only=True,
    cabin="letter_only",
)

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=0,
)

print(X_train.head())
```

We see the resulting data below:

```
      pclass␣
↪     sex        age  sibsp  parch     fare cabin embarked
501         2␣
↪ female  13.000000      0      1  19.5000     M        S
588         2␣
↪ female   4.000000      1      1  23.0000     M        S
402         2␣
↪ female  30.000000      1      0  13.8583     M        C
1193        3␣
↪   male  29.881135      0      0   7.7250     M        Q
686         3␣
↪ female  22.000000      0      0   7.7250     M        Q
```

Now, we set up the *OrdinalEncoder()* to replace the categories by strings based on the target mean value and only in the 3 indicated variables:

```
encoder = OrdinalEncoder(
    encoding_method='ordered',
    variables=['pclass', 'cabin', 'embarked'],
    ignore_format=True)

encoder.fit(X_train, y_train)
```

With `fit()` the encoder learns the mappings for each category, which are stored in its `encoder_dict_` parameter:

```
encoder.encoder_dict_
```

In the `encoder_dict_` we find the integers that will replace each one of the categories of each variable that we want to encode. This way, we can map the original value of the variable to the new value.

```
{'pclass': {3: 0, 2: 1, 1: 2},
'cabin': {'T': 0,
        'M': 1,
        'G': 2,
        'A': 3,
        'C': 4,
        'F': 5,
        'D': 6,
        'E': 7,
        'B': 8},
'embarked': {'S': 0, 'Q': 1, 'C': 2, 'Missing': 3}}
```

We can now go ahead and replace the original strings with the numbers:

```
train_t = encoder.transform(X_train)
test_t = encoder.transform(X_test)

print(train_t.head())
```

Below we see the resulting dataframe, where the original variable values are now replaced with integers:

```
      pclass  ␣
↪  sex        age  sibsp  parch     fare  cabin  embarked
501         1  female␣
↪ 13.000000      0      1  19.5000      1         0
588         1  female␣
↪  4.000000      1      1  23.0000      1         0
402         1  female␣
↪ 30.000000      1      0  13.8583      1         2
1193        0  ␣
↪ male  29.881135      0      0   7.7250      1         1
686         0  female␣
↪ 22.000000      0      0   7.7250      1         1
```

Additional resources

In the following notebook, you can find more details into the *OrdinalEncoder()*'s functionality and example plots with the encoded variables:

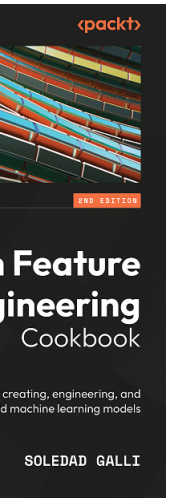
- Jupyter notebook

For more details about this and other feature engineering methods check out these resources:

Or read our book:

Fig. 20: Feature Engineering for Machine Learning

Both our book and course are suitable for beginners and more advanced data scientists alike. By purchasing them you are supporting Sole, the main developer of Feature-engine.



ure Engineering

MeanEncoder

Mean encoding is the process of replacing the categories in categorical features by the mean value of the target variable shown by each category. For example, if we are trying to predict the default rate (that's the target variable), and our dataset has the categorical variable **City**, with the categories of **London**, **Manchester**, and **Bristol**, and the default rate per city is 0.1, 0.5, and 0.3, respectively, with mean encoding, we would replace London by 0.1, Manchester by 0.5, and Bristol by 0.3.

Mean encoding, together with one hot encoding and ordinal encoding, belongs to the most commonly used categorical encoding techniques in data science.

It is said that mean encoding can easily cause overfitting. That's because we are capturing some information about the target into the predictive features during the encoding. More importantly, the overfitting can be caused by encoding categories with low frequencies with mean target values that are unreliable. In short, the mean target values seen for those categories in the training set do not hold for test data or new observations.

Overfitting

When the categories in the categorical features have a good representation, or, in other words, when there are enough observations in our dataset that show the categories that we want to encode, then taking the simple average of the target variable per category is a good approximation. We can trust that a new data point, say from the test data, that shows that category will also have a target value that is similar to the target mean value that we calculated for said category during training.

However, if there are only a few observations that show some of the categories, then the mean target value for those categories will be unreliable. In other words, the certainty that we have that a new observation that shows this category will have a mean target value close to the one we estimated decreases.

To account for the uncertainty of the encoding values for rare categories, what we normally do is **"blend"** the mean target variable per category with the general mean of the target, calculated over the entire training dataset. And this blending is proportional to the variability of the target within that category and the category frequency.

Smoothing

To avoid overfitting, we can determine the mean target value estimates as a mixture of two values: the mean target value per category (known as the posterior) and the mean target value in the entire dataset (known as the prior).

The following formula shows the estimation of the mean target value with smoothing:

$$mapping = (w_i)posterior + (1 - w_i)prior$$

The prior and posterior values are "blended" using a weighting factor (`wi`). This weighting factor is a function of the category group size (`n_i`) and the variance of the target in the data (`t`) and within the category (`s`):

$$w_i = n_i t/(s + n_i t)$$

When the category group is large, the weighing factor is close to 1, and therefore more weight is given to the posterior (the mean of the target per category). When the category group size is small, then the weight gets closer to 0, and more weight is given to the prior (the mean of the target in the entire dataset).

In addition, if the variability of the target within that category is large, we also give more weight to the prior, whereas if it is small, then we give more weight to the posterior.

In short, adding smoothing can help prevent overfitting in those cases where categorical data have many infrequent categories or show high cardinality.

High cardinality

High cardinality refers to a high number of unique categories in the categorical features. Mean encoding was specifically designed to tackle highly cardinal variables by taking advantage of this smoothing function, which will essentially blend infrequent categories together by replacing them with values very close to the overall target mean calculated over the training data.

Another encoding method that tackles cardinality out of the box is count encoding. See for example *CountFrequencyEncoder*.

To account for highly cardinal variables in alternative encoding methods, you can group rare categories together by using the *RareLabelEncoder*.

Alternative Python implementations of mean encoding

In Feature-engine, we blend the probabilities considering the target variability and the category frequency. In the original paper, there are alternative formulations to determine the blending. If you want to check those out, use the transformers from the Python library Category encoders:

- M-estimate
- Target Encoder

Mean encoder

Feature-engine's *MeanEncoder()* replaces categories with the mean of the target per category. By default, it does not implement smoothing. That means that it will replace categories by the mean target value as determined during training over the training data set (just the posterior).

To apply smoothing using the formulation that we described earlier, set the parameter `smoothing` to `"auto"`. That would be our recommended solution. Alternatively, you can set the parameter `smoothing` to any value that you want, in which case the weighting factor `wi` will be calculated like this:

$$w_i = n_i/(s + n_i)$$

where s is the value your pass to `smoothing`.

Unseen categories

Unseen categories are those labels that were not seen during training. Or in other words, categories that were not present in the training data.

With the *MeanEncoder()*, we can take care of unseen categories in 1 of 3 ways:

- We can set the mean encoder to ignore unseen categories, in which case those categories will be replaced by nan.
- We can set the mean encoder to raise an error when it encounters unseen categories. This is useful when we don't expect new categories for those categorical variables.
- We can instruct the mean encoder to replace unseen or new categories with the mean of the target shown in the training data, that is, the prior.

Mean encoding and machine learning

Feature-engine's *MeanEncoder()* can perform mean encoding for regression and binary classification datasets. At the moment, we do not support multiclass targets.

Python examples

In the following sections, we'll show the functionality of *MeanEncoder()* using the Titanic Dataset.

First, let's load the libraries, functions and classes:

```
from sklearn.model_selection import train_test_split
from feature_engine.datasets import load_titanic
from feature_engine.encoding import MeanEncoder
```

To avoid data leakage, it is important to separate the data into training and test sets. The mean target values, with or without smoothing, will be determined using the training data only.

Let's load and split the data:

```
X, y = load_titanic(
    return_X_y_frame=True,
    handle_missing=True,
    predictors_only=True,
    cabin="letter_only",
)

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=0,
)

print(X_train.head())
```

We see the resulting dataframe containing 3 categorical columns: sex, cabin and embarked:

```
      pclass␣
↪     sex        age  sibsp  parch     fare cabin embarked
501         2␣
↪ female  13.000000      0      1  19.5000     M        S
588         2␣
↪ female   4.000000      1      1  23.0000     M        S
402         2␣
↪ female  30.000000      1      0  13.8583     M        C
1193        3␣
↪   male  29.881135      0      0   7.7250     M        Q
686         3␣
↪ female  22.000000      0      0   7.7250     M        Q
```

Simple mean encoding

Let's set up the *MeanEncoder()* to replace the categories in the categorical features with the target mean, without smoothing:

```
encoder = MeanEncoder(
    variables=['cabin', 'sex', 'embarked'],
)

encoder.fit(X_train, y_train)
```

With `fit()` the encoder learns the target mean value for each category and stores those values in the `encoder_dict_` attribute:

```
encoder.encoder_dict_
```

The `encoder_dict_` contains the mean value of the target per category, per variable. We can use this dictionary to map the numbers in the encoded features to the original categorical values.

```
{'cabin': {'A': 0.5294117647058824,
  'B': 0.7619047619047619,
```

(continues on next page)

(continued from previous page)

```
  'C': 0.5633802816901409,
  'D': 0.71875,
  'E': 0.71875,
  'F': 0.6666666666666666,
  'G': 0.5,
  'M': 0.30484330484330485,
  'T': 0.0},
 'sex': {'female
↪': 0.7283582089552239, 'male': 0.18760757314974183},
 'embarked': {'C': 0.553072625698324,
  'Missing': 1.0,
  'Q': 0.37349397590361444,
  'S': 0.3389570552147239}}
```

We can now go ahead and replace the categorical values with the numerical values:

```
train_t = encoder.transform(X_train)
test_t = encoder.transform(X_test)

print(train_t.head())
```

Below we see the resulting dataframe, where the categorical values are now replaced with the target mean values:

```
      pclass       sex␣
↪       age  sibsp  parch     fare     cabin  embarked
501        2  0.728358␣
↪ 13.000000      0      1  19.5000  0.304843  0.338957
588        2  0.728358␣
↪  4.000000      1      1  23.0000  0.304843  0.338957
402        2  0.728358␣
↪ 30.000000      1      0  13.8583  0.304843  0.553073
1193       3  0.187608␣
↪ 29.881135      0      0   7.7250  0.304843  0.373494
686        3  0.728358␣
↪ 22.000000      0      0   7.7250  0.304843  0.373494
```

Mean encoding with smoothing

By default, *MeanEncoder()* determines the mean target values without blending. If we want to apply smoothing to control the cardinality of the variable and avoid overfitting, we set up the transformer as follows:

```
encoder = MeanEncoder(
    variables=None,
    smoothing="auto"
)

encoder.fit(X_train, y_train)
```

In this example, we did not indicate which variables to encode.

MeanEncoder() can automatically find the categorical variables, which are stored in one of its attributes:

```
encoder.variables_
```

Below we see the categorical features found by *MeanEncoder()*:

```
['sex', 'cabin', 'embarked']
```

We can find the categorical mappings calculated by the mean encoder:

```
encoder.encoder_dict_
```

Note that these values are different to those determined without smoothing:

```
{'sex': {'female
↪': 0.7275051072923914, 'male': 0.18782635616273297},
 'cabin': {'A': 0.5210189753697639,
  'B': 0.755161569137655,
  'C': 0.5608140829162441,
  'D': 0.7100896537503179,
  'E': 0.7100896537503179,
  'F': 0.6501082490288561,
  'G': 0.47606795923242295,
  'M': 0.3049458046855866,
  'T': 0.0},
 'embarked': {'C': 0.552100581239763,
  'Missing': 1.0,
  'Q': 0.3736336816011083,
  'S': 0.3390242994568531}}
```

We can now go ahead and replace the categorical values with the numerical values:

```
train_t = encoder.transform(X_train)
test_t = encoder.transform(X_test)

print(train_t.head())
```

Below we see the resulting dataframe with the encoded features:

```
      pclass       sex␣
↪       age  sibsp  parch     fare     cabin  embarked
501        2  0.727505␣
↪ 13.000000      0      1  19.5000  0.304946  0.339024
588        2  0.727505␣
↪  4.000000      1      1  23.0000  0.304946  0.339024
402        2  0.727505␣
↪ 30.000000      1      0  13.8583  0.304946  0.552101
1193       3  0.187826␣
↪ 29.881135      0      0   7.7250  0.304946  0.373634
686        3  0.727505␣
↪ 22.000000      0      0   7.7250  0.304946  0.373634
```

We can now use this dataframes to train machine learning models for regression or classification.

Mean encoding variables with numerical values

MeanEncoder(), and all Feature-engine encoders, have been designed to work with variables of type object or categorical by default. If you want to encode variables that are numeric, you need to instruct the transformer to ignore the data type:

```
encoder = MeanEncoder(
    variables=['cabin', 'pclass'],
    ignore_format=True,
)

t_train = encoder.fit_transform(X_train, y_train)
t_test = encoder.transform(X_test)
```

After encoding the features we can use the data sets to train machine learning algorithms.

Last thing to note before closing in is that mean encoding does not increase the dimensionality of the resulting dataframes: from 1 categorical feature, we obtain 1 encoded variable. Hence, this encoding method is suitable for predictive modeling that uses models that are sensitive to the size of the feature space.

Additional resources

In the following notebook, you can find more details into the *MeanEncoder()* functionality and example plots with the encoded variables:

- Jupyter notebook

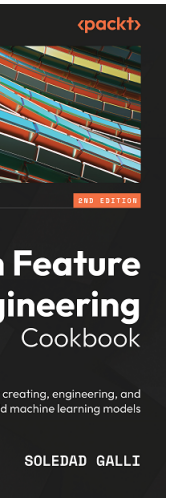
For tutorials about this and other feature engineering methods check out these resources:

Fig. 22: Feature Engineering for Machine Learning

Or read our book:

Fig. 23: Feature Engineering for Time Series Forecasting

Both our book and courses are suitable for beginners and
more advanced data scientists alike. By purchasing them you are supporting Sole, the main developer of Feature-engine.



ure Engineering

WoEEncoder

The *WoEEncoder()* replaces categories by the weight of evidence (WoE). The WoE was used primarily in the financial sector to create credit risk scorecards.

The weight of evidence is given by:

$$log(p(X = xj|Y = 1)/p(X = xj|Y = 0))$$

The WoE is determined as follows:

We calculate the percentage positive cases in each category of the total of all positive cases. For example 20 positive cases in category A out of 100 total positive cases equals 20 %. Next, we calculate the percentage of negative cases in each category respect to the total negative cases, for example 5 negative cases in category A out of a total of 50 negative cases equals 10%. Then we calculate the WoE by dividing the category percentages of positive cases by the category percentage of negative cases, and take the logarithm, so for category A in our example WoE = log(20/10).

Note

- If WoE values are negative, negative cases supersede the positive cases.
- If WoE values are positive, positive cases supersede the negative cases.
- And if WoE is 0, then there are equal number of positive and negative examples.

Encoding into WoE:

- Creates a monotonic relationship between the encoded variable and the target
- Returns variables in a similar scale

Note

This categorical encoding is exclusive for binary classification.

Let's look at an example using the Titanic Dataset.

First, let's load the data and separate it into train and test:

```
from sklearn.model_selection import train_test_split
from feature_engine.datasets import load_titanic
from feature_
↪engine.encoding import WoEEncoder, RareLabelEncoder

X, y = load_titanic(
    return_X_y_frame=True,
    handle_missing=True,
    predictors_only=True,
    cabin="letter_only",
)

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=0,
)

print(X_train.head())
```

We see the resulting data below:

```
      pclass␣
↪     sex        age  sibsp  parch     fare cabin embarked
501        2␣
↪ female  13.000000      0      1  19.5000     M        S
588        2␣
↪ female   4.000000      1      1  23.0000     M        S
402        2␣
↪ female  30.000000      1      0  13.8583     M        C
1193       3␣
↪   male  29.881135      0      0   7.7250     M        Q
686        3␣
↪ female  22.000000      0      0   7.7250     M        Q
```

Before we encode the variables, I would like to group infrequent categories into one category, called 'Rare'. For this, I will use the *RareLabelEncoder()* as follows:

```
# set up a rare label encoder
rare_encoder = RareLabelEncoder(
    tol=0.1,
    n_categories=2,
    variables=['cabin', 'pclass', 'embarked'],
    ignore_format=True,
)

# fit and transform data
train_t = rare_encoder.fit_transform(X_train)
test_t = rare_encoder.transform(X_train)
```

Now, we set up the *WoEEncoder()* to replace the categories by the weight of the evidence, only in the 3 indicated variables:

```
# set up a weight of evidence encoder
woe_encoder = WoEEncoder(
    variables=['cabin', 'pclass', 'embarked'],
    ignore_format=True,
)

# fit the encoder
woe_encoder.fit(train_t, y_train)
```

With `fit()` the encoder learns the weight of the evidence for each category, which are stored in its `encoder_dict_` parameter:

```
woe_encoder.encoder_dict_
```

In the `encoder_dict_` we find the WoE for each one of the categories of the variables to encode. This way, we can map the original values to the new value.

```
{'cabin':␣
↪{'M': -0.35752781962490193, 'Rare': 1.083797390800775},
 'pclass': {'1': 0.9453018143294478,
  '2': 0.21009172435857942,
  '3': -0.5841726684724614},
 'embarked': {'C': 0.679904786667102,
  'Rare': 0.012075414091446468,
  'S': -0.20113381737960143}}
```

Now, we can go ahead and encode the variables:

```
train_t = woe_encoder.transform(train_t)
test_t = woe_encoder.transform(test_t)

print(train_t.head())
```

Below we see the resulting dataset with the weight of the evidence:

```
        pclass     sex␣
↪       age  sibsp  parch     fare     cabin  embarked
501   0.210092  female␣
↪ 13.000000      0      1  19.5000 -0.357528 -0.201134
```

(continues on next page)

(continued from previous page)

```
588   0.210092  female␣
↪  4.000000      1      1  23.0000 -0.357528 -0.201134
402   0.210092  female␣
↪ 30.000000      1      0  13.8583 -0.357528  0.679905
1193 -0.584173    male␣
↪ 29.881135      0      0   7.7250 -0.357528  0.012075
686  -0.584173  female␣
↪ 22.000000      0      0   7.7250 -0.357528  0.012075
```

WoE for continuous variables

In credit scoring, continuous variables are also transformed using the WoE. To do this, first variables are sorted into a discrete number of bins, and then these bins are encoded with the WoE as explained here for categorical variables. You can do this by combining the use of the equal width, equal frequency or arbitrary discretisers.

Additional resources

In the following notebooks, you can find more details into the *WoEEncoder()* functionality and example plots with the encoded variables:

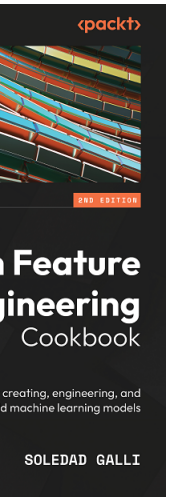
- WoE in categorical variables
- WoE in numerical variables

For more details about this and other feature engineering methods check out these resources:

Or read our book:

Fig. 25: Feature Engineering for Machine Learning

Both our book and course are suitable for beginners and more advanced data scientists alike. By purchasing them you are supporting Sole, the main developer of Feature-engine.



ure Engineering

DecisionTreeEncoder

The *DecisionTreeEncoder()* replaces categories in the variable with the predictions of a decision tree.

The transformer first encodes categorical variables into numerical variables using *OrdinalEncoder()*. You have the option to have the integers assigned to the categories as they appear in the variable, or ordered by the mean value of the target per category. You can regulate this behaviour with the parameter `encoding_method`. As decision trees are able to pick non-linear relationships, replacing categories by arbitrary numbers should be enough in practice.

After this, the transformer fits with this numerical variable a decision tree to predict the target variable. Finally, the original categorical variable is replaced by the predictions of the decision tree.

The motivation of the *DecisionTreeEncoder()* is to try and create monotonic relationships between the categorical variables and the target.

Let's look at an example using the Titanic Dataset.

First, let's load the data and separate it into train and test:

```
from sklearn.model_selection import train_test_split
from feature_engine.datasets import load_titanic
from feature_engine.encoding import DecisionTreeEncoder

X, y = load_titanic(
    return_X_y_frame=True,
    handle_missing=True,
    predictors_only=True,
    cabin="letter_only",
)

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=0,
)

print(X_train[['cabin', 'pclass', 'embarked']].head(10))
```

We will encode the following categorical variables:

```
    cabin  pclass embarked
501      M       2        S
588      M       2        S
402      M       2        C
1193     M       3        Q
686      M       3        Q
971      M       3        Q
117      E       1        C
540      M       2        S
294      C       1        C
261      E       1        S
```

We set up the encoder to encode the variables above with 3 fold cross-validation, using a grid search to find the optimal depth of the decision tree (this is the default behaviour of the *DecisionTreeEncoder()*). In this example, we optimize the tree using the roc-auc metric.

```
encoder = DecisionTreeEncoder(
    variables=['cabin', 'pclass', 'embarked'],
    regression=False,
    scoring='roc_auc',
    cv=3,
    random_state=0,
    ignore_format=True)

encoder.fit(X_train, y_train)
```

With `fit()` the *DecisionTreeEncoder()* fits 1 decision tree per variable. Now we can go ahead and transform the categorical variables into numbers, using the predictions of these trees:

```
train_t = encoder.transform(X_train)
test_t = encoder.transform(X_test)

train_t[['cabin', 'pclass', 'embarked']].head(10)
```

We can see the encoded variables below:

```
        cabin    pclass  embarked
501   0.304843  0.436170  0.338957
588   0.304843  0.436170  0.338957
402   0.304843  0.436170  0.553073
1193  0.304843  0.259036  0.373494
686   0.304843  0.259036  0.373494
971   0.304843  0.259036  0.373494
117   0.611650  0.617391  0.553073
540   0.304843  0.436170  0.338957
294   0.611650  0.617391  0.553073
261   0.611650  0.617391  0.338957
```

Additional resources

In the following notebook, you can find more details into the `DecisionTreeEncoder()` functionality and example plots with the encoded variables:

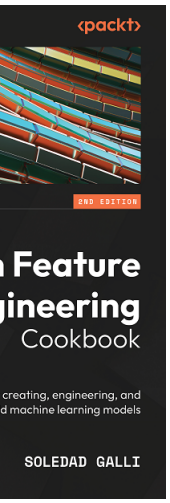
- Jupyter notebook

For more details about this and other feature engineering methods check out these resources:

Or read our book:

Fig. 27: Feature Engineering for Machine Learning

Both our book and course are suitable for beginners and more advanced data scientists alike. By purchasing them you are supporting Sole, the main developer of Feature-engine.



ure Engineering

RareLabelEncoder

The *RareLabelEncoder()* groups infrequent categories into one new category called 'Rare' or a different string indicated by the user. We need to specify the minimum percentage of observations a category should have to be preserved and the minimum number of unique categories a variable should have to be re-grouped.

tol

In the parameter `tol` we indicate the minimum proportion of observations a category should have, not to be grouped. In other words, categories which frequency, or proportion of observations is <= `tol` will be grouped into a unique term.

n_categories

In the parameter `n_categories` we indicate the minimum cardinality of the categorical variable in order to group infrequent categories. For example, if `n_categories=5`, categories will be grouped only in those categorical variables with more than 5 unique categories. The rest of the variables will be ignored.

This parameter is useful when we have big datasets and do not have time to examine all categorical variables individually. This way, we ensure that variables with low cardinality are not reduced any further.

max_n_categories

In the parameter `max_n_categories` we indicate the maximum number of unique categories that we want in the encoded variable. If `max_n_categories=5`, then the most popular 5 categories will remain in the variable after the encoding, all other will be grouped into a single category.

This parameter is useful if we are going to perform one hot encoding at the back of it, to control the expansion of the feature space.

Example

Let's look at an example using the Titanic Dataset.

First, let's load the data and separate it into train and test:

```
from sklearn.model_selection import train_test_split
from feature_engine.datasets import load_titanic
from feature_engine.encoding import RareLabelEncoder

X, y = load_titanic(
    return_X_y_frame=True,
    handle_missing=True,
    predictors_only=True,
    cabin="letter_only",
)
X["pclass"] = X["pclass"].astype("O")

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=0,
)
```

(continues on next page)

(continued from previous page)

```
print(X_train.head())
```

We see the resulting data below:

```
      pclass␣
↪     sex         age  sibsp  parch     fare cabin embarked
501         2␣
↪ female  13.000000      0      1  19.5000     M        S
588         2␣
↪ female   4.000000      1      1  23.0000     M        S
402         2␣
↪ female  30.000000      1      0  13.8583     M        C
1193        3␣
↪   male  29.881135      0      0   7.7250     M        Q
686         3␣
↪ female  22.000000      0      0   7.7250     M        Q
```

Let's explore the number of uniue categories in the variable "cabin".

```
X_train["cabin"].unique()
```

We see the number of unique categories in the output below:

```
array(['M',
↪ 'E', 'C', 'D', 'B', 'A', 'F', 'T', 'G'], dtype=object)
```

Now, we set up the *RareLabelEncoder()* to group categories shown by less than 3% of the observations into a new group or category called 'Rare'. We will group the categories in the indicated variables if they have more than 2 unique categories each.

```
encoder = RareLabelEncoder(
    tol=0.03,
    n_categories=2,
    variables=['cabin', 'pclass', 'embarked'],
    replace_with='Rare',
)

# fit the encoder
encoder.fit(X_train)
```

With `fit()`, the *RareLabelEncoder()* finds the categories present in more than 3% of the observations, that is, those that will not be grouped. These categories can be found in the `encoder_dict_` attribute.

```
encoder.encoder_dict_
```

In the `encoder_dict_` we find the most frequent categories per variable to encode. Any category that is not in this dictionary, will be grouped.

```
{'cabin': ['M', 'C', 'B', 'E', 'D'],
'pclass': [3, 1, 2],
'embarked': ['S', 'C', 'Q']}
```

Now we can go ahead and transform the variables:

```
# transform the data
train_t = encoder.transform(X_train)
test_t = encoder.transform(X_test)
```

Let's now inspect the number of unique categories in the variable "cabin" after the transformation:

```
train_t["cabin"].unique()
```

In the output below, we see that the infrequent categories have been replaced by "Rare".

```
array(['M', 'E', 'C', 'D', 'B', 'Rare'], dtype=object)
```

We can also specify the maximum number of categories that can be considered frequent using the max_n_categories parameter.

Let's begin by creating a toy dataframe and count the values of observations per category:

```
from feature_engine.encoding import RareLabelEncoder
import pandas as pd
data = {'var_
↪A': ['A'] * 10 + ['B'] * 10 + ['C'] * 2 + ['D'] * 1}
data = pd.DataFrame(data)
data['var_A'].value_counts()
```

```
A    10
B    10
C     2
D     1
Name: var_A, dtype: int64
```

In this block of code, we group the categories only for variables with more than 3 unique categories and then we plot the result:

```
rare_encoder = RareLabelEncoder(tol=0.05, n_categories=3)
rare_encoder.fit_transform(data)['var_A'].value_counts()
```

```
A       10
B       10
C        2
Rare     1
Name: var_A, dtype: int64
```

Now, we retain the 2 most frequent categories of the variable and group the rest into the 'Rare' group:

```
rare_encoder = RareLabelEncoder(tol=0.
↪05, n_categories=3, max_n_categories=2)
Xt = rare_encoder.fit_transform(data)
Xt['var_A'].value_counts()
```

```
A       10
B       10
Rare     3
Name: var_A, dtype: int64
```

Tips

The *RareLabelEncoder()* can be used to group infrequent categories and like this control the expansion of the feature space if using one hot encoding.

Some categorical encodings will also return NAN if a category is present in the test set, but was not seen in the train set. This inconvenient can usually be avoided if we group rare labels before training the encoders.

Some categorical encoders will also return NAN if there is not enough observations for a certain category. For example the *WoEEncoder()* and the `PRatioEncoder()`. This behaviour can be also prevented by grouping infrequent labels before the encoding with the *RareLabelEncoder()*.

Additional resources

In the following notebook, you can find more details into the *RareLabelEncoder()* functionality and example plots with the encoded variables:

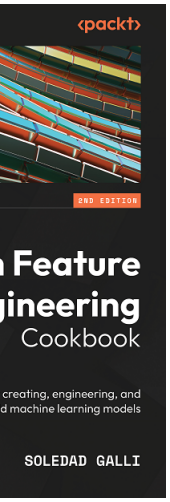
- Jupyter notebook

For more details about this and other feature engineering methods check out these resources:

Or read our book:

Fig. 29: Feature Engineering for Machine Learning

Both our book and course are suitable for beginners and more advanced data scientists alike. By purchasing them you are supporting Sole, the main developer of Feature-engine.



ure Engineering

StringSimilarityEncoder

The *StringSimilarityEncoder()* replaces categorical variables with a set of float variables that capture the similarity between the category names. The new variables have values between 0 and 1, where 0 indicates no similarity and 1 is an exact match between the names of the categories.

To calculate the similarity between the categories, *StringSimilarityEncoder()* uses Gestalt pattern matching. Under the hood, *StringSimilarityEncoder()* uses the `quick_ratio` method from the `SequanceMatcher()` from `difflib`.

The similarity is calculated as:

$$GPM = 2M/T$$

where T is the total number of elements in both sequences and M is the number of matches.

For example, the similarity between the categories "dog" and "dig" is 0.66. T is the total number of elements in both categories, that is 6. There are 2 matches between the words, the letters d and g, so: 2 * M / T = 2 * 2 / 6 = 0.66.

Output of the `StringSimilarityEncoder()`

Let's create a dataframe with the categories "dog", "dig" and "cat":

```
import pandas as pd
from␣
↪feature_engine.encoding import StringSimilarityEncoder

df = pd.DataFrame({"words": ["dog", "dig", "cat"]})
df
```

We see the dataframe in the following output:

```
  words
0   dog
```

(continues on next page)

(continued from previous page)

```
1   dig
2   cat
```

Let's now encode the variable:

```
encoder =  StringSimilarityEncoder()
dft = encoder.fit_transform(df)
dft
```

We see the encoded variables below:

```
   words_dog  words_dig  words_cat
0   1.000000   0.666667        0.0
1   0.666667   1.000000        0.0
2   0.000000   0.000000        1.0
```

Note that *StringSimilarityEncoder()* replaces the original variables by the distance variables.

`StringSimilarityEncoder()` vs One-hot encoding

String similarity encoding is similar to one-hot encoding, in the sense that each category is encoded as a new variable. But the values, instead of 1 or 0, are the similarity between the observation's category and the dummy variable. It is suitable for poorly defined (or 'dirty') categorical variables.

Encoding only popular categories

The *StringSimilarityEncoder()* can also create similarity variables for the n most popular categories, n being determined by the user. For example, if we encode only the 6 more popular categories, by setting the parameter `top_categories=6`, the transformer will add variables only for the 6 most frequent categories. The most frequent categories are those with the largest number of observations. This behaviour is useful when the categorical variables are highly cardinal, to control the expansion of the feature space.

Specifying how `StringSimilarityEncoder()` should deal with missing values

The *StringSimilarityEncoder()* has three options for dealing with missing values, which can be specified with the parameter `missing_values`:

1. Ignore NaNs (option `ignore`) - will leave the NaN in the resulting dataframe after transformation. Could be useful, if the next step in the pipeline is imputation or if the machine learning algorithm can handle missing data out-of-the-box.
2. Impute NaNs (option `impute`) - will impute NaN with an empty string, and then calculate the similarity between the empty string and the variable's categories. Most of the time, the similarity value will be 0 in resulting dataframe. This is the default option.

3. Raise an error (option `raise`) - will raise an error if NaN is present during `fit`, `transform` or `fit_transform`. Could be useful for debugging and monitoring purposes.

Important

StringSimilarityEncoder() will encode unseen categories by out-of-the-box, by measuring the string similarity to the seen categories.

No text preprocessing is applied by *StringSimilarityEncoder()*. Be mindful of preparing string categorical variables if needed.

StringSimilarityEncoder() works with categorical variables by default. And it has the option to encode numerical variables as well. This is useful, when the values of the numerical variables are more useful as strings, than as numbers. For example, for variables like barcode.

Examples

Let's look at an example using the Titanic Dataset. First we load the data and divide it into a train and a test set:

```
import string
from sklearn.model_selection import train_test_split
from feature_engine.datasets import load_titanic
from␣
↪feature_engine.encoding import StringSimilarityEncoder

def clean_titanic():
    translate_
↪table = str.maketrans('' , '', string.punctuation)
    data = load_titanic()
    data['home.dest'] = (
    data['home.dest']
    .str.strip()
    .str.translate(translate_table)
    .str.replace('  ', ' ')
    .str.lower()
    )
    data['name'] = (
    data['name']
    .str.strip()
    .str.translate(translate_table)
    .str.replace('  ', ' ')
    .str.lower()
    )
    data['ticket'] = (
    data['ticket']
    .str.strip()
    .str.translate(translate_table)
    .str.replace('  ', ' ')
    .str.lower()
```

(continues on next page)

(continued from previous page)

```
    )
    return data

data = clean_titanic()
# Separate into train and test sets
X_train, X_test, y_train, y_test = train_test_split(
    data.
↪drop(['survived', 'sex', 'cabin', 'embarked'], axis=1),
    data['survived'],
    test_size=0.3,
    random_state=0
)

X_train.head()
```

Below, we see the first rows of the dataset:

```
    pclass␣
↪                            name  age  sibsp  parch  \
501    ␣
↪   2  mellinger miss madeleine violet   13      0      1
588    ␣
↪   2                  wells miss joan    4      1      1
402    ␣
↪   2     duran y more miss florentina   30      1      0
1193   ␣
↪   3                 scanlan mr james  NaN      0      0
686    ␣
↪   3       bradley miss bridget delia   22      0      0

            ticket     fare boat body  \
501         250644     19.5   14  NaN
588          29103       23   14  NaN
402   scparis 2148  13.8583   12  NaN
1193         36209    7.725  NaN  NaN
686         334914    7.725   13  NaN

                                        home.dest
501                           england bennington vt
588                              cornwall akron oh
402                     barcelona spain havana cuba
1193                                           NaN
686   kingwilliamstown co cork ireland glens falls ny
```

Now, we set up the encoder to encode only the 2 most frequent categories of each of the 3 indicated categorical variables:

```
# set up the encoder
encoder = StringSimilarityEncoder(
    top_categories=2,
    variables=['name', 'home.dest', 'ticket'],
    ignore_format=True
```

(continues on next page)

(continued from previous page)

```
    )

# fit the encoder
encoder.fit(X_train)
```

With `fit()` the encoder will learn the most popular categories of the variables, which are stored in the attribute `encoder_dict_`.

```
encoder.encoder_dict_
```

```
{
  'name': ['mellinger␣
↪miss madeleine violet', 'barbara mrs catherine david'],
  'home.dest': ['', 'new york ny'],
  'ticket': ['ca 2343', 'ca 2144']
}
```

The `encoder_dict_` contains the categories that will derive similarity variables for each categorical variable.

With transform, we go ahead and encode the variables. Note that the *StringSimilarityEncoder()* will drop the original variables.

```
# transform the data
train_t = encoder.transform(X_train)
test_t = encoder.transform(X_test)

test_t.head()
```

Below, we see the resulting dataframe:

```
    pclass  age  sibsp  parch    fare boat body  \
1139       3   38      0      0  7.8958  NaN  NaN
533        2   21      0      1      21   12  NaN
459        2   42      1      0      27  NaN  NaN
1150       3  NaN      0      0    14.5  NaN  NaN
393        2   25      0      0    31.5  NaN  NaN

    name_mellinger miss␣
↪madeleine violet  name_barbara mrs catherine david  \
1139                 ␣
↪                0.454545                          0.550000
533                  ␣
↪                0.615385                          0.524590
459                  ␣
↪                0.596491                          0.603774
1150                 ␣
↪                0.641509                          0.693878
393                  ␣
↪                0.408163                          0.666667

    home.dest_nan␣
↪ home.dest_new york ny  ticket_ca 2343  ticket_ca 2144
```

(continues on next page)

(continued from previous page)

```
1139            1.
↪0                0.000000        0.461538        0.461538
533             0.
↪0                0.370370        0.307692        0.307692
459             0.
↪0                0.352941        0.461538        0.461538
1150            1.
↪0                0.000000        0.307692        0.307692
393             0.
↪0                0.437500        0.666667        0.666667
```

More details

For more details into *StringSimilarityEncoder()*'s functionality visit:

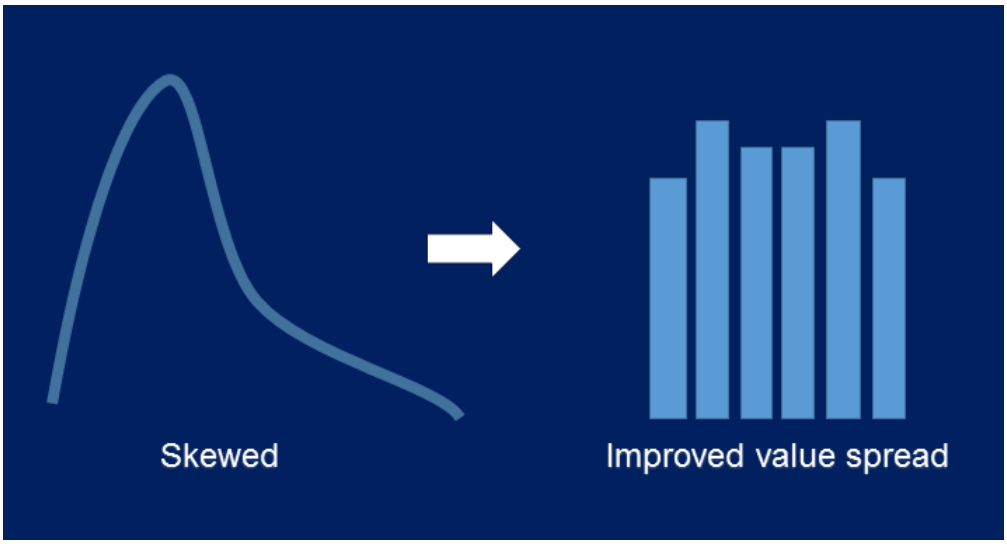
- Jupyter notebook

All notebooks can be found in a dedicated repository.

Discretisation

Feature-engine's variable discretisation transformers transform continuous numerical variables into discrete variables. The discrete variables will contain contiguous intervals in the case of the equal frequency and equal width transformers. The Decision Tree discretiser will return a discrete variable, in the sense that the new feature takes a finite number of values.

The following illustration shows the process of discretisation:



With discretisation, sometimes we can obtain a more homogeneous value spread from an originally skewed variable. But this is not always possible.

Discretisation plus encoding

Very often, after we discretise the numerical continuous variables into discrete intervals we want to proceed their engineering as if they were categorical. This

is common practice. Throughout the user guide, we point to jupyter notebooks that showcase this functionality.

Discretisers

EqualFrequencyDiscretiser

Equal frequency discretization consists of dividing continuous attributes into equal-frequency bins. These bins contain roughly the same number of observations, with boundaries set at specific quantile values determined by the desired number of bins.

Equal frequency discretization ensures a uniform distribution of data points across the range of values, enhancing the handling of skewed data and outliers.

Discretization is a common data preprocessing technique used in data science. It's also known as binning data (or simply "binning").

Advantages and Limitations

Equal frequency discretization has some advantages and shortcomings:

Advantages

Some advantages of equal frequency binning:

- **Algorithm Efficiency:** Enhances the performance of data mining and machine learning algorithms by providing a simplified representation of the dataset.
- **Outlier Management:** Efficiently mitigates the effect of outliers by grouping them into the extreme bins.
- **Data Smoothing:** Helps smooth the data, reduces noise, and improves the model's ability to generalize.
- **Improved value distribution:** Returns an uniform distribution of values across the value range.

  Equal frequency discretization improves the data distribution, **optimizing the spread of values**. This is particularly beneficial for datasets with skewed distributions (see the Python example code).

Limitations

On the other hand, equal frequency binning can lead to a loss of information by aggregating data into broader categories. This is particularly concerning if the data in the same bin has predictive information about the target.

Let's consider a binary classifier task using a decision tree model. A bin with a high proportion of both target categories would potentially impact the model's performance in this scenario.

EqualFrequencyDiscretiser

Feature-engine's *EqualFrequencyDiscretiser* applies equal frequency discretization to numerical variables. It uses the `pandas.qcut()` function under the hood, to determine the interval limits.

You can specify the variables to be discretized by passing their names in a list when you set up the transformer. Alternatively, *EqualFrequencyDiscretiser* will automatically infer the data types to compute the interval limits for all numeric variables.

Optimal number of intervals: With *EqualFrequencyDiscretiser*, the user defines the number of bins. Smaller intervals may be required if the variable is highly skewed or not continuous.

Integration with scikit-learn: *EqualFrequencyDiscretiser* and all other feature-engine transformers seamlessly integrate with scikit-learn pipelines.

Python code example

In this section, we'll show the main functionality of *EqualFrequencyDiscretiser*

Load dataset

In this example, we'll use the Ames House Prices' Dataset. First, let's load the dataset and split it into train and test sets:

```
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split

from feature_
↪engine.discretisation import EqualFrequencyDiscretiser

# Load dataset
X, y = fetch_openml(name='house_
↪prices', version=1, return_X_y=True, as_frame=True)
X.set_index('Id', inplace=True)

# Separate into train and test sets
X_train, X_test, y_train, y_test =␣
↪ train_test_split(X, y, test_size=0.3, random_state=42)
```

Equal-frequency Discretization

In this example, let's discretize two variables, LotArea and GrLivArea, into 10 intervals of approximately equal number of observations.

```
# List the target numeric variables to be transformed
TARGET_NUMERIC_FEATURES= ['LotArea','GrLivArea']

# Set up the discretization transformer
disc = EqualFrequencyDiscretiser(q=10,
↪ variables=TARGET_NUMERIC_FEATURES)

# Fit the transformer
disc.fit(X_train)
```

Note that if we do not specify the variables (default=`None`), *EqualFrequencyDiscretiser* will automatically infer the data types to compute the interval limits for all numeric variables.

With the `fit()` method, the discretizer learns the bin boundaries and saves them into a dictionary so we can use them to transform unseen data:

```
# Learned limits for each variable
disc.binner_dict_
```

```
{'LotArea': [-inf,
  5000.0,
  7105.6,
  8099.200000000003,
  8874.0,
  9600.0,
  10318.400000000001,
  11173.5,
  12208.2,
  14570.699999999999,
  inf],
 'GrLivArea': [-inf,
  918.5,
  1080.4,
  1218.0,
  1348.4,
  1476.5,
  1601.6000000000001,
  1717.6999999999998,
  1893.0000000000005,
  2166.3999999999996,
  inf]}
```

Note that the lower and upper boundaries are set to -inf and inf, respectively. his behavior ensures that the transformer will be able to allocate to the extreme bins values that are smaller or greater than the observed minimum and maximum values in the training set.

EqualFrequencyDiscretiser will not work in the presence of missing values. Therefore, we should either remove or impute missing values before fitting

the transformer.

```
# Transform the data
train_t = disc.transform(X_train)
test_t = disc.transform(X_test)
```

Let's visualize the first rows of the raw data and the transformed data:

```
# Raw data
print(X_train[TARGET_NUMERIC_FEATURES].head())
```

Here we see the original variables:

```
        LotArea  GrLivArea
Id
136     10400       1682
1453     3675       1072
763      8640       1547
933     11670       1905
436     10667       1661
```

```
# Transformed data
print(train_t[TARGET_NUMERIC_FEATURES].head())
```

Here we observe the variables after discretization:

```
          LotArea  GrLivArea
Id
136         6          6
1453        0          1
763         3          5
933         7          8
436         6          6
```

The transformed data now contains discrete values corresponding to the ordered computed buckets (0 being the first and q-1 the last).

Now, let's visualize the plots for equal-width intervals with a histogram and the transformed data with equal-frequency discretiser:

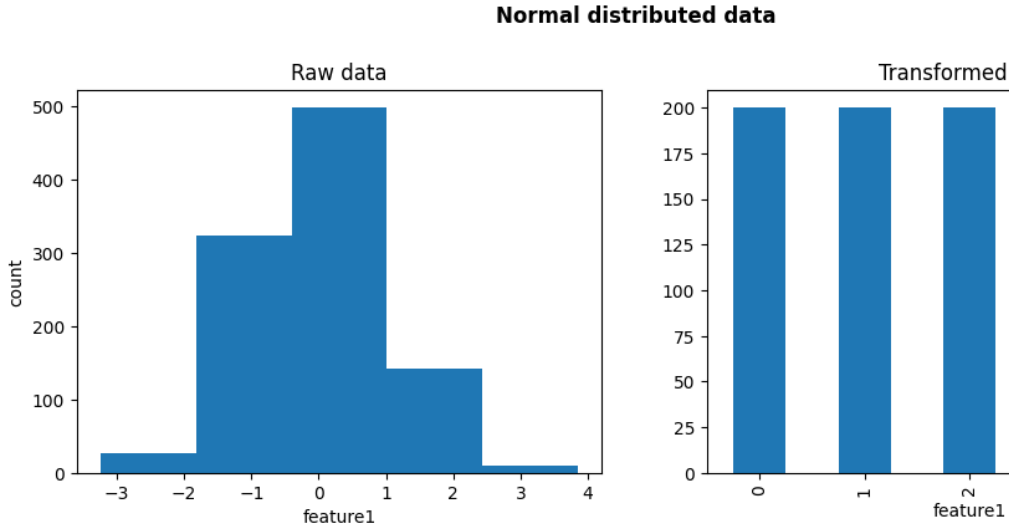
```
# Instantiate a figure with two axes
fig, axes = plt.subplots(ncols=2, figsize=(10,5))

# Plot raw distribution
X_train['GrLivArea'].plot.hist(bins=disc.q, ax=axes[0])
axes[0].set_title('Raw data with equal width binning')
axes[0].set_xlabel('GrLivArea')

# Plot transformed distribution
train_t['GrLivArea
↪'].value_counts().sort_index().plot.bar(ax=axes[1])
axes[1].set_
↪title('Transformed data with equal frequency binning')
```

(continues on next page)

(continued from previous page)

```
plt.tight_layout(w_pad=2)
plt.show()
```

As we see in the following image, the intervals contain approximately the same number of observations:



Finally, as the default value for the `return_object` parameter is `False`, the transformer outputs integer variables:

```
train_t[TARGET_NUMERIC_FEATURES].dtypes
```

```
LotArea      int64
GrLivArea    int64
dtype: object
```

Return variables as object

Categorical encoders in Feature-engine are designed to work by default with variables of type object. Therefore, to further encode the discretised output with Feature-engine, we can set `return_object=True` instead. This will return the transformed variables as object.

Let's say we want to obtain monotonic relationships between the variable and the target. We can do that seamlessly by setting `return_object` to True. A tutorial of how to use this functionality is available here.

Return bin boundaries

If we want to output the intervals limits instead of integers, we can set `return_boundaries` to `True`:

```
# Set up the discretization transformer
disc = EqualFrequencyDiscretiser(
    q=10,
    variables=TARGET_NUMERIC_FEATURES,
    return_boundaries=True)

# Fit the transformer
disc.fit(X_train)

# Transform test set & visualize limit
test_t = disc.transform(X_test)

# Visualize output (boundaries)
print(test_t[TARGET_NUMERIC_FEATURES].head())
```

The transformed variables now show the interval limits in the output. We can immediately see that the bin width for these intervals varies. In other words, they don't have the same width, contrarily to what we see with *equal width discretization*.

Unlike the variables discretized into integers, these variables cannot be used to train machine learning models; however, they are still highly helpful for data analysis in this format, and they may be sent to any Feature-engine encoder for additional processing.

```
          LotArea         GrLivArea
Id
893     (8099.2, 8874.0]   (918.5, 1080.4]
1106  (12208.2, 14570.7]     (2166.4, inf]
414     (8874.0, 9600.0]   (918.5, 1080.4]
523       (-inf, 5000.0]  (1601.6, 1717.7]
1037  (12208.2, 14570.7]  (1601.6, 1717.7]
```

Binning skewed data

Let's now show the benefits of equal frequency discretization for skewed variables. We'll start by importing the libraries and classes:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from feature_
↪engine.discretisation import EqualFrequencyDiscretiser
```

Now, we'll create a toy dataset with a variable that is normally distributed and another one that is skewed:

```
# Set seed for reproducibility
np.random.seed(42)

# Generate a normally distributed data
normal_data = np.random.normal(loc=0, scale=1, size=1000)

# Generate␣
↪a right-skewed data using exponential distribution
skewed_data = np.random.exponential(scale=1, size=1000)

# Create dataframe with simulated data
X = pd.DataFrame(
↪{'feature1': normal_data, 'feature2': skewed_data})
```

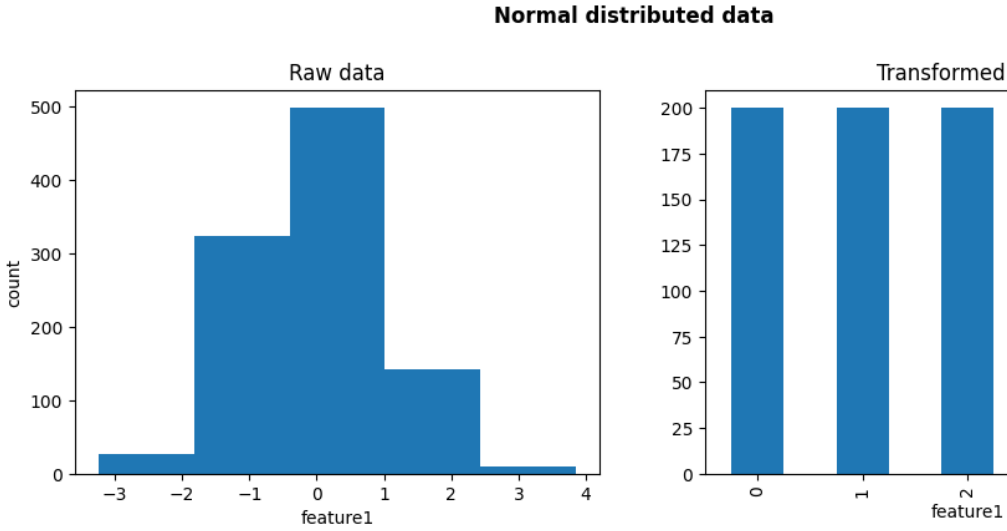
Let's discretize both variables into 5 equal frequency bins:

```
# Instantiate discretizer
disc = EqualFrequencyDiscretiser(q=5)

# Transform simulated data
X_transformed = disc.fit_transform(X)
```

Let's plot the original distribution and the distribution after discretization for the variable that was normally distributed:

```
fig, axes = plt.subplots(1, 2, figsize=(12, 4))

axes[0].hist(X.feature1, bins=disc.q)
axes[0].
↪set(xlabel='feature1', ylabel='count', title='Raw data')

X_transformed.feature1.
↪value_counts().sort_index().plot.bar(ax=axes[1])
axes[1].set_title('Transformed data')

plt.suptitle('Normal␣
↪distributed data', weight='bold', size='large', y=1.05)

plt.show()
```

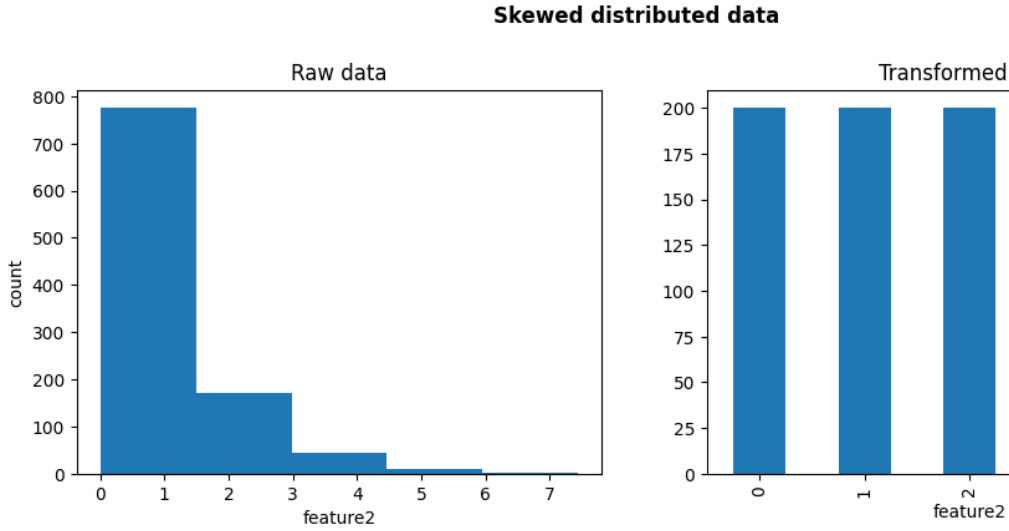
In the following image, we see that after the discretization there is an even distribution of the values across the value range, hence, the variable does no look normally distributed any more.



Let's now plot the original distribution and the distribution after discretization for the variable that was skewed:

```
fig, axes = plt.subplots(1, 2, figsize=(12, 4))

axes[0].hist(X.feature2, bins=disc.q)
axes[0].
↪set(xlabel='feature2', ylabel='count', title='Raw data')

X_transformed.feature2.
↪value_counts().sort_index().plot.bar(ax=axes[1])
axes[1].set_title('Transformed data')

plt.suptitle('Skewed␣
↪distributed data', weight='bold', size='large', y=1.05)

plt.show()
```

In the following image, we see that after the discretization there is an even distribution of the values across the value range.



See Also

For alternative binning techniques, check out the following resources:

- Further feature-engine *discretizers / binning methods*
- Scikit-learn's KBinsDiscretizer.

Check out also:

- Pandas qcut.

Additional resources

Check also for more details on how to use this transformer:

- Jupyter notebook
- Jupyter notebook - Discretizer plus Weight of Evidence encoding

For more details about this and other feature engineering methods check out these resources:

Or read our book:

Fig. 31: Feature Engineering for Machine Learning

Fig. 32: Python Feature Engineering Cookbook

Both our book and course are suitable for beginners and more advanced data scientists alike. By purchasing them you are supporting Sole, the main developer of Feature-engine.

Equal width discretization consist of dividing continuous variables into intervals of equal width, calculated using the following formula:

$$bin_{width} = (max(X) - min(X))/bins$$

Here, `bins` is the number of intervals specified by the user and `max(X)` and `min(X)` are the minimum and maximum values of the variable to discretize.

Discretization is a common data preprocessing technique used in data science. It's also known as data binning (or simply "binning").

Advantages and Limitations

Equal binning discretization has some advantages and also shortcomings.

Advantages

Some advantages of equal width binning:

- **Algorithm Efficiency:** Enhances the performance of data mining and machine learning algorithms by providing a simplified representation of the dataset.
- **Outlier Management:** Efficiently mitigates the effect of outliers by grouping them into the extreme bins, thus preserving the integrity of the main data distribution.
- **Data Smoothing:** Helps smooth the data, reduces noise, and improves the model's ability to generalize.

Limitations

On the other hand, equal width discretzation can lead to a loss of information by aggregating data into broader categories. This is particularly concerning if the data in the same bin has predictive information about the target.

Let's consider a binary classifier task using a decision tree model. A bin with a high proportion of both target categories would potentially impact the model's performance in this scenario.

EqualWidthDiscretiser

Feture-engine's *EqualWidthDiscretiser()* applies equal width discretization to numerical variables. It uses the `pandas.cut()` function under the hood to find the interval limits and then sort the continuous variables into the bins.

You can specify the variables to be discretized by passing their names in a list when you set up the transformer. Alternatively, *EqualWidthDiscretiser()* will automatically infer the data types to compute the interval limits for all numeric variables.

Optimal number of intervals: With *EqualWidthDiscretiser()*, the user defines the number of bins. Smaller intervals may be required if the variable is highly skewed or not continuous.

Integration with scikit-learn: *EqualWidthDiscretiser()* and all other Feature-engine transformers seamlessly integrate with scikit-learn pipelines.

Python code example

In this section, we'll show the main functionality of *EqualWidthDiscretiser()*.

Load dataset

In this example, we'll use the Ames House Prices' Dataset. First, let's load the dataset and split it into train and test sets:

```
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split

from feature_engine.discretisation import EqualFrequencyDiscretiser

# Load dataset
X, y = fetch_openml(name='house_prices', version=1, return_X_y=True, as_frame=True)
X.set_index('Id', inplace=True)

# Separate into train and test sets
X_train, X_test, y_train, y_test =  train_test_split(X, y, test_size=0.3, random_
↪state=42)
```

Equal-width Discretization

In this example, let's discretize two variables, LotArea and GrLivArea, into 10 intervals of equal width:

```
# List the target numeric variables for equal-width discretization
TARGET_NUMERIC_FEATURES= ['LotArea','GrLivArea']

# Set up the discretization transformer
disc = EqualWidthDiscretiser(bins=10, variables=TARGET_NUMERIC_FEATURES)

# Fit the transformer
disc.fit(X_train)
```

Note that if we do not specify the variables (default=`None`), *EqualWidthDiscretiser* will automatically infer the data types to compute the interval limits for all numeric variables.

With the `fit()` method, the discretizer learns the bin boundaries and saves them into a dictionary so we can use them to transform unseen data:

```
# Learned limits for each variable
disc.binner_dict_
```

```
{'LotArea': [-inf,
  22694.5,
  44089.0,
  65483.5,
  86878.0,
  108272.5,
  129667.0,
  151061.5,
  172456.0,
  193850.5,
  inf],
 'GrLivArea': [-inf,
  864.8,
  1395.6,
  1926.3999999999999,
  2457.2,
  2988.0,
  3518.7999999999997,
  4049.5999999999995,
  4580.4,
  5111.2,
  inf]}
```

Note that the lower and upper boundaries are set to -inf and inf, respectively. This behavior ensures that the transformer will be able to allocate to the extreme bins values that are smaller or greater than the observed minimum and maximum values in the training set.

EqualWidthDiscretiser will not work in the presence of missing values. Therefore, we should either remove or impute missing values before fitting the transformer.

```
# Transform the data (data discretization)
train_t = disc.transform(X_train)
test_t = disc.transform(X_test)
```

Let's visualize the first rows of the raw data and the transformed data:

```
# Raw data
print(X_train[TARGET_NUMERIC_FEATURES].head())
```

Here we see the original variables:

```
          LotArea  GrLivArea
Id
136     10400       1682
1453     3675       1072
763      8640       1547
933     11670       1905
436     10667       1661
```

```
# Transformed data
print(train_t[TARGET_NUMERIC_FEATURES].head())
```

Here we observe the variables after discretization:

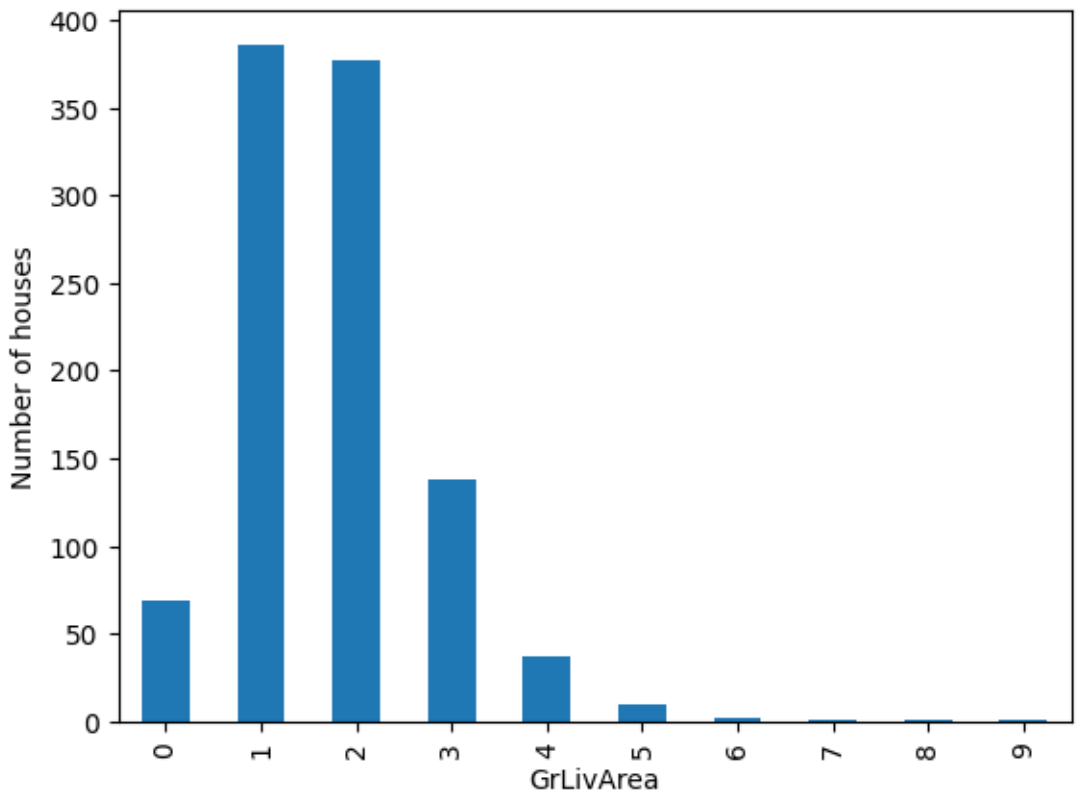
```
          LotArea  GrLivArea
Id
136         0          2
1453        0          1
763         0          2
933         0          2
436         0          2
```

The transformed data now contains discrete values corresponding to the ordered computed buckets (0 being the first and bins-1 the last).

Now, let's check out the number of observations per bin by creating a bar plot:

```
train_t['GrLivArea'].value_counts().sort_index().plot.bar()
plt.ylabel('Number of houses')
plt.show()
```

As we see in the following image, the intervals contain different number of observations. It's a similar output to a histogram:



Equal width discretization does not improve the spread of values over the value range. If the variable is skewed, it will still be skewed after the discretization.

Finally, since the default value for the `return_object` parameter is `False`, the transformer outputs integer variables:

```
train_t[TARGET_NUMERIC_FEATURES].dtypes
```

```
LotArea      int64
GrLivArea    int64
dtype: object
```

Return variables as object

Categorical encoders in Feature-engine are designed to work by default with variables of type object. Therefore, to further encode the discretized output with Feature-engine's encoders, we can set `return_object=True` instead. This will return the transformed variables as object.

Let's say we want to obtain monotonic relationships between the variable and the target. We can do that seamlessly by setting `return_object` to `True`. A tutorial of how to use this functionality is available here.

Return bin boundaries

If we want to output the intervals limits instead of integers, we can set `return_boundaries` to `True`:

```
# Set up the discretization transformer
disc = EqualFrequencyDiscretiser(
    bins=10,
    variables=TARGET_NUMERIC_FEATURES,
    return_boundaries=True)

# Fit the transformer
disc.fit(X_train)

# Transform test set & visualize limit
test_t = disc.transform(X_test)

# Visualize output (boundaries)
print(test_t[TARGET_NUMERIC_FEATURES].head())
```

In the following output we see that the transformed variables now display the interval limits. While we can't use these variables to train machine learning models, as opposed to the variables discretized into integers, they are very useful in this format for data analysis, and they can also be passed on to any Feature-engine encoder for further processing.

```
             LotArea         GrLivArea
Id
893   (-inf, 22694.5]   (864.8, 1395.6]
1106  (-inf, 22694.5]  (2457.2, 2988.0]
414   (-inf, 22694.5]   (864.8, 1395.6]
523   (-inf, 22694.5]  (1395.6, 1926.4]
1037  (-inf, 22694.5]  (1395.6, 1926.4]
```

See Also

For alternative binning techniques, check out the following resources:

- Further feature-engine *discretizers / binning methods*
- Scikit-learn's KBinsDiscretizer.

Check out also:

- Pandas cut.

Additional resources

Check also for more details on how to use this transformer:

- Jupyter notebook
- Jupyter notebook - Discretizer plus Ordinal encoding

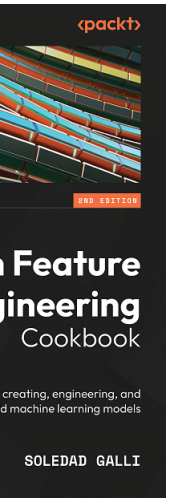
For more details about this and other feature engineering methods check out these resources:

Or read our book:

Fig. 33: Feature Engineering for Machine Learning

Both our book and course are suitable for beginners and more advanced data scientists alike. By purchasing them you are supporting Sole, the main developer of Feature-engine.

ArbitraryDiscretiser



ure Engineering

The *ArbitraryDiscretiser()* sorts the variable values into contiguous intervals which limits are arbitrarily defined by the user. Thus, you must provide a dictionary with the variable names as keys and a list with the limits of the intervals as values, when setting up the discretiser.

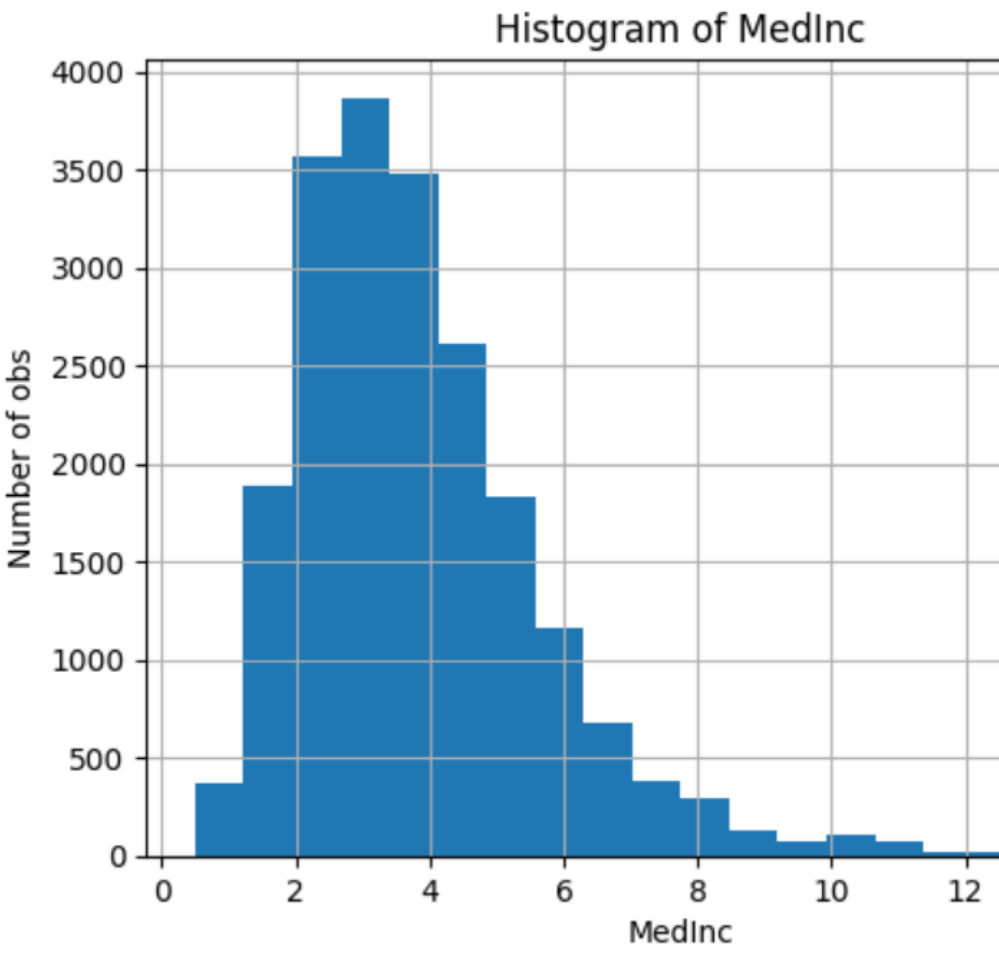
The *ArbitraryDiscretiser()* works only with numerical variables. The discretiser will check that the variables entered by the user are present in the train set and cast as numerical.

Example

Let's take a look at how this transformer works. First, let's load a dataset and plot a histogram of a continuous variable. We use the california housing dataset that comes with Scikit-learn.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_california_housing
from feature_
↪engine.discretisation import ArbitraryDiscretiser

X, y = fetch_
↪california_housing( return_X_y=True, as_frame=True)

X['MedInc'].hist(bins=20)
plt.xlabel('MedInc')
plt.ylabel('Number of obs')
plt.title('Histogram of MedInc')
plt.show()
```

In the following plot we see a histogram of the variable median income:



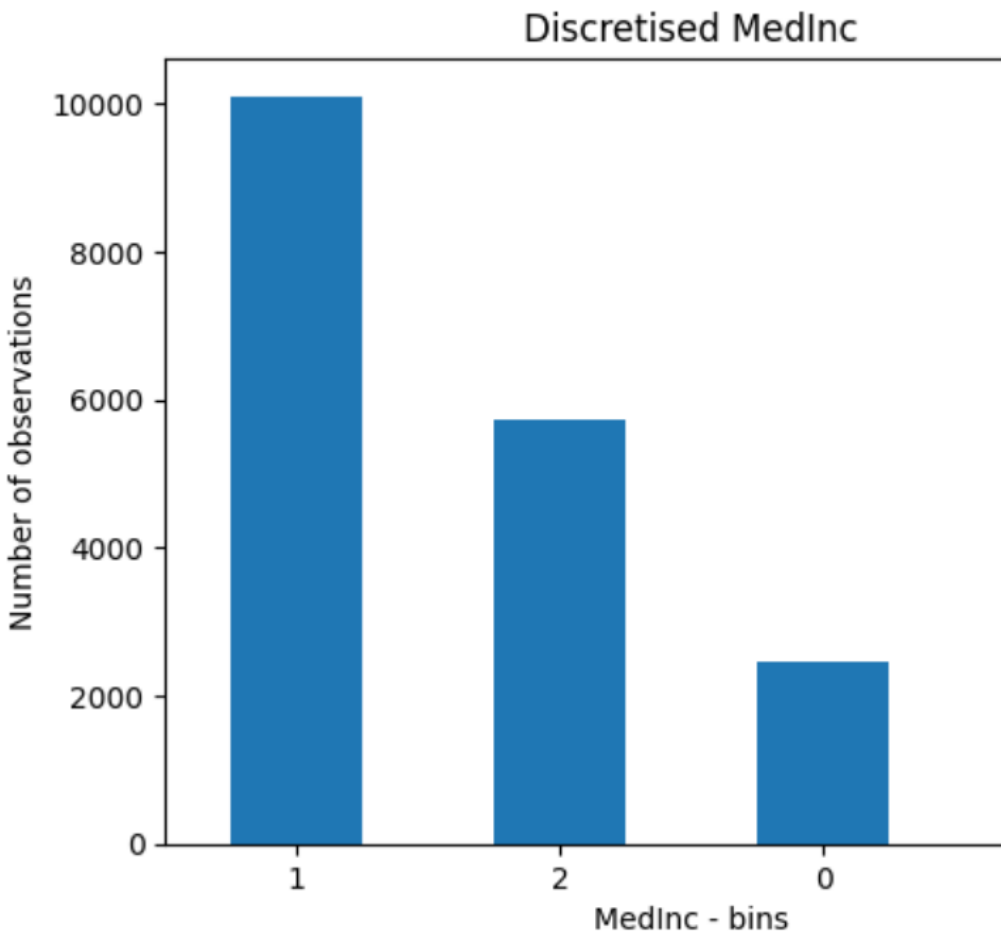
Now, let's discretise the variable into arbitrarily determined intervals. We want the intervals as integers in the resulting transformation, so we set `return_boundaries` to `False`.

```
user_dict = {'MedInc': [0, 2, 4, 6, np.Inf]}

transformer = ArbitraryDiscretiser(
    binning_dict=user_
↪dict, return_object=False, return_boundaries=False)

X = transformer.fit_transform(X)
```

Now, we can go ahead and plot the variable after the transformation:

```
X['MedInc'].value_counts().plot.bar(rot=0)
plt.xlabel('MedInc - bins')
plt.ylabel('Number of observations')
plt.title('Discretised MedInc')
plt.show()
```

In the following plot we see the number of observations per interval:

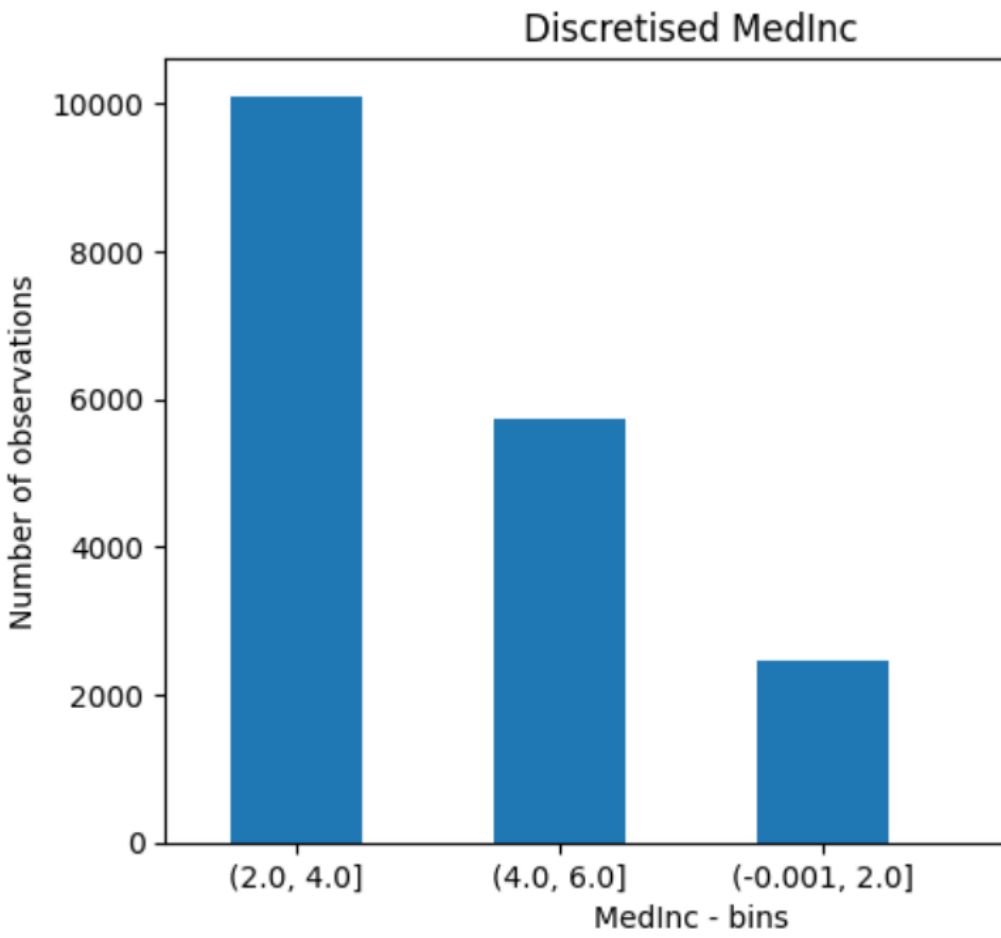


Note that in the above figure the intervals are represented by digits.

Alternatively, we can return the interval limits in the discretised variable by setting `return_boundaries` to `True`.

```
X, y = fetch_
↪california_housing( return_X_y=True, as_frame=True)

user_dict = {'MedInc': [0, 2, 4, 6, np.Inf]}

transformer = ArbitraryDiscretiser(
    binning_dict=user_
↪dict, return_object=False, return_boundaries=True)
X = transformer.fit_transform(X)

X['MedInc'].value_counts().plot.bar(rot=0)
plt.xlabel('MedInc - bins')
plt.ylabel('Number of observations')
plt.title('Discretised MedInc')
plt.show()
```

In the following plot we see the number of observations per interval:



Discretisation plus encoding

If we return the interval values as integers, the discretiser has the option to return the transformed variable as integer or as object. Why would we want the transformed variables as object?

Categorical encoders in Feature-engine are designed to work with variables of type object by default. Thus, if you wish to encode the returned bins further, say to try and obtain monotonic relationships between the variable and the target, you can do so seamlessly by setting `return_object` to True. You can find an example of how to use this functionality here.

Additional resources

Check also:

- Jupyter notebook
- Jupyter notebook - Discretiser plus Mean Encoding

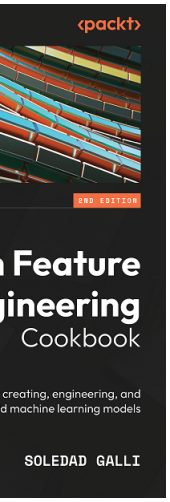
For more details about this and other feature engineering methods check out these resources:

Or read our book:

Fig. 35: Feature Engineering for Machine Learning

Both our book and course are suitable for beginners and more advanced data scientists alike. By purchasing them you are supporting Sole, the main developer of Feature-engine.

DecisionTreeDiscretiser



ure Engineering

The *DecisionTreeDiscretiser()* replaces numerical variables by discrete, i.e., finite variables, which values are the predictions of a decision tree. The method is based on the winning solution of the KDD 2009 competition:

Niculescu-Mizil, et al. "Winning the KDD Cup Orange Challenge with Ensemble Selection". JMLR: Workshop and Conference Proceedings 7: 23-34. KDD 2009.

In the original article, each feature in the challenge dataset was re-coded by training a decision tree of limited depth (2, 3 or 4) using that feature alone, and letting the tree predict the target. The probabilistic predictions of this decision tree were used as an additional feature, that was now linearly (or at least monotonically) correlated with the target.

According to the authors, the addition of these new features had a significant impact on the performance of linear models.

Example

In the following example, we re-code 2 numerical variables using decision trees.

First we load the data and separate it into train and test:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

from feature_
↪engine.discretisation import DecisionTreeDiscretiser

# Load dataset
data = data = pd.read_csv('houseprice.csv')

# Separate into train and test sets
X_train, X_test, y_train, y_test =  train_test_split(
            data.drop(['Id', 'SalePrice'], axis=1),
    ␣
↪       data['SalePrice'], test_size=0.3, random_state=0)
```

Now we set up the discretiser. We will optimise the decision tree's depth using 3 fold cross-validation.

```
# set up the discretisation transformer
disc = DecisionTreeDiscretiser(cv=3,
  ␣
↪                        scoring='neg_mean_squared_error',
    ␣
↪                       variables=['LotArea', 'GrLivArea'],
                          regression=True)

# fit the transformer
disc.fit(X_train, y_train)
```

With `fit()` the transformer fits a decision tree per variable. Then, we can go ahead replace the variable values by the predictions of the trees:

```
# transform the data
train_t= disc.transform(X_train)
test_t= disc.transform(X_test)
```

The `binner_dict_` stores the details of each decision tree.

```
disc.binner_dict_
```

```
{'LotArea
↪': GridSearchCV(cv=3, error_score='raise-deprecating',
             estimator=DecisionTreeRegressor(criterion=
↪'mse', max_depth=None,
       ␣
↪                                      max_features=None,
```

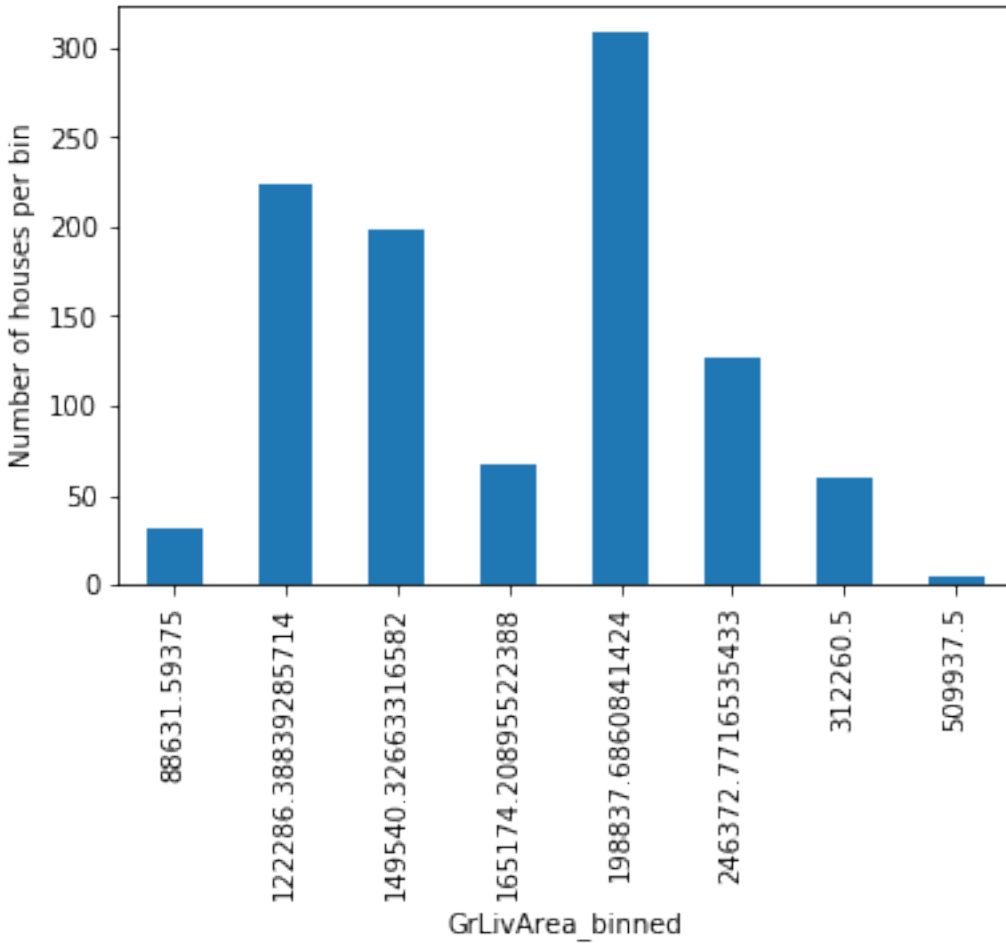
(continues on next page)

(continued from previous page)

```
                                       max_leaf_nodes=None,
                                  min_impurity_decrease=0.0,
                                    min_impurity_split=None,
                                     min_samples_leaf=1,
                                       min_samples_split=2,
                                 min_weight_fraction_leaf=0.0,
                             presort=False, random_state=None,
                                              splitter='best'),
              iid='warn
', n_jobs=None, param_grid={'max_depth': [1, 2, 3, 4]},
              pre_dispatch=
'2*n_jobs', refit=True, return_train_score=False,
              scoring='neg_mean_squared_error', verbose=0),
 'GrLivArea
': GridSearchCV(cv=3, error_score='raise-deprecating',
              estimator=DecisionTreeRegressor(criterion=
'mse', max_depth=None,
                                       max_features=None,
                                       max_leaf_nodes=None,
                                  min_impurity_decrease=0.0,
                                    min_impurity_split=None,
                                     min_samples_leaf=1,
                                       min_samples_split=2,
                                 min_weight_fraction_leaf=0.0,
                             presort=False, random_state=None,
                                              splitter='best'),
              iid='warn
', n_jobs=None, param_grid={'max_depth': [1, 2, 3, 4]},
              pre_dispatch=
'2*n_jobs', refit=True, return_train_score=False,
              scoring='neg_mean_squared_error', verbose=0)}
```

With tree discretisation, each bin, that is, each prediction value, does not necessarily contain the same number of observations.

```
# with tree␣
↪discretisation, each bin does not necessarily contain
# the same number of observations.
train_
↪t.groupby('GrLivArea')['GrLivArea'].count().plot.bar()
plt.ylabel('Number of houses')
```



Note

Our implementation of the *DecisionTreeDiscretiser()* will replace the original values of the variable by the predictions of the trees. This is not strictly identical to what the winners of the KDD competition did. They added the predictions of the features as new variables, while keeping the original ones.

More details

Check also for more details on how to use this transformer:

- Jupyter notebook
- tree_pipe in cell 21 of this Kaggle kernel

For more details about this and other feature engineering methods check out these resources:

- Feature engineering for machine learning, online course.
- Python Feature Engineering Cookbook, book.

GeometricWidthDiscretiser

The *GeometricWidthDiscretiser()* divides continuous numerical variables into intervals of increasing width. The width of each succeeding interval is larger than the previous interval by a constant amount (cw).

The constant amount is calculated as:

$$cw = (Max - Min)^{1/n}$$

were Max and Min are the variable's maximum and minimum value, and n is the number of intervals.

The sizes of the intervals themselves are calculated with a geometric progression:

$$a_{i+1} = a_i cw$$

Thus, the first interval's width equals cw, the second interval's width equals 2 * cw, and so on.

Note that the proportion of observations per interval may vary.

This discretisation technique is great when the distribution of the variable is right skewed.

Note: The width of some bins might be very small. Thus, to allow this transformer to work properly, it might help to increase the precision value, that is, the number of decimal values allowed to define each bin. If the variable has a narrow range or you are sorting into several bins, allow greater precision (i.e., if precision = 3, then 0.001; if precision = 7, then 0.0001).

The *GeometricWidthDiscretiser()* works only with numerical variables. A list of variables to discretise can be indicated, or the discretiser will automatically select all numerical variables in the train set.

Example

Let's look at an example using the house prices dataset (more details about the dataset *here*).

Let's load the house prices dataset and separate it into train and test sets:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

from feature_
↪engine.discretisation import GeometricWidthDiscretiser

# Load dataset
data = pd.read_csv('houseprice.csv')

# Separate into train and test sets
X_train, X_test, y_train, y_test =  train_test_split(
            data.drop(['Id', 'SalePrice'], axis=1),
    ␣
↪       data['SalePrice'], test_size=0.3, random_state=0)
```

Now, we want to discretise the 2 variables indicated below into 10 intervals of increasing width:

```
# set up the discretisation transformer
disc = GeometricWidthDiscretiser(bins=10,
↪ variables=['LotArea', 'GrLivArea'])

# fit the transformer
disc.fit(X_train)
```

With `fit()` the transformer learns the boundaries of each interval. Then, we can go ahead and sort the values into the intervals:

```
# transform the data
train_t= disc.transform(X_train)
test_t= disc.transform(X_test)
```

The `binner_dict_` stores the interval limits identified for each variable.

```
disc.binner_dict_
```

```
'LotArea': [-inf,
1303.412,
1311.643,
1339.727,
1435.557,
1762.542,
2878.27,
6685.32,
19675.608,
64000.633,
inf],
'GrLivArea': [-inf,
336.311,
339.34,
346.34,
```

(continues on next page)

(continued from previous page)

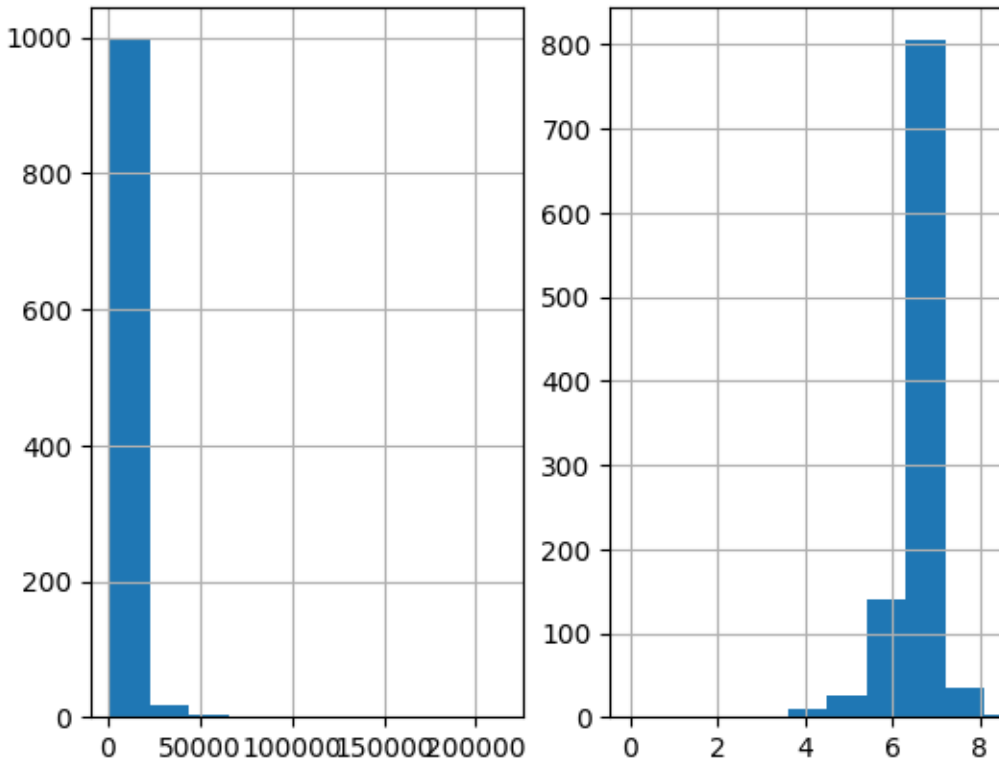
```
362.515,
399.894,
486.27,
685.871,
1147.115,
2212.974,
inf]}
```

With increasing width discretisation, each bin does not necessarily contain the same number of observations. This transformer is suitable for variables with right skewed distributions.

Let's compare the variable distribution before and after the discretization:

```
fig, ax = plt.subplots(1, 2)
X_train['LotArea'].hist(ax=ax[0], bins=10);
train_t['LotArea'].hist(ax=ax[1], bins=10);
```

We can see below that the intervals contain different number of observations. We can also see that the shape from the distribution changed from skewed to a more "bell shaped" distribution.



Discretisation plus encoding

If we return the interval values as integers, the discretiser has the option to return the transformed variable as integer or as object. Why would we want the transformed variables as object?

Categorical encoders in Feature-engine are designed to work with variables of type object by default. Thus, if you wish to encode the returned bins further, say to try and obtain monotonic relationships between the variable and the target, you can do so seamlessly by setting `return_object` to True. You can find an example of how to use this functionality here.

Additional resources

Check also for more details on how to use this transformer:

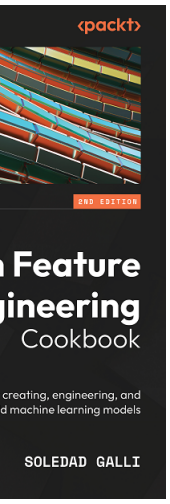
- Jupyter notebook - Geometric Discretiser
- Jupyter notebook - Geometric Discretiser plus Mean encoding

For more details about this and other feature engineering methods check out these resources:

Or read our book:

Fig. 37: Feature Engineering for Machine Learning

Both our book and course are suitable for beginners and more advanced data scientists alike. By purchasing them you are supporting Sole, the main developer of Feature-engine.



ure Engineering

Outlier Handling

Feature-engine's outlier cappers cap maximum or minimum values of a variable at an arbitrary or derived value. The OutlierTrimmer removes outliers from the dataset.

Winsorizer

The *Winsorizer()* caps maximum and/or minimum values of a variable at automatically determined values. The minimum and maximum values can be calculated in 1 of 3 different ways:

Gaussian limits:

- right tail: mean + 3* std
- left tail: mean - 3* std

IQR limits:

- right tail: 75th quantile + 3* IQR
- left tail: 25th quantile - 3* IQR

where IQR is the inter-quartile range: 75th quantile - 25th quantile.

MAD limits:

- right tail: median + 3* MAD
- left tail: median - 3* MAD

where MAD is the median absolute deviation from the median.

percentiles or quantiles:

- right tail: 95th percentile
- left tail: 5th percentile

Example

Let's cap some outliers in the Titanic Dataset. First, let's load the data and separate it into train and test:

```
from sklearn.model_selection import train_test_split
from feature_engine.datasets import load_titanic
from feature_engine.outliers import Winsorizer

X, y = load_titanic(
    return_X_y_frame=True,
    predictors_only=True,
    handle_missing=True,
)
```

(continues on next page)

(continued from previous page)

```
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=0,
)

print(X_train.head())
```

We see the resulting data below:

```
      pclass    ␣
↪ sex         age  sibsp  parch     fare    cabin embarked
501         2  female␣
↪ 13.000000       0      1  19.5000  Missing        S
588         2  female␣
↪  4.000000       1      1  23.0000  Missing        S
402         2  female␣
↪ 30.000000       1      0  13.8583  Missing        C
1193        3    ␣
↪male  29.881135      0      0   7.7250  Missing        Q
686         3  female␣
↪ 22.000000       0      0   7.7250  Missing        Q
```

Now, we will set the *Winsorizer()* to cap outliers at the right side of the distribution only (param `tail`). We want the maximum values to be determined using the mean value of the variable (param `capping_method`) plus 3 times the standard deviation (param `fold`). And we only want to cap outliers in 2 variables, which we indicate in a list.

```
capper = Winsorizer(capping_method='gaussian',
                    tail='right',
                    fold=3,
                    variables=['age', 'fare'])

capper.fit(X_train)
```

With `fit()`, the *Winsorizer()* finds the values at which it should cap the variables. These values are stored in its attribute:

```
capper.right_tail_caps_
```

```
{'age': 67.73951212364803, 'fare': 174.70395336846678}
```

We can now go ahead and censor the outliers:

```
# transform the data
train_t = capper.transform(X_train)
test_t = capper.transform(X_test)
```

If we evaluate now the maximum of the variables in the transformed datasets, they should coincide with the values observed in the attribute `right_tail_caps_`:

```
train_t[['fare', 'age']].max()
```

```
fare    174.703953
age      67.739512
dtype: float64
```

Additional resources

You can find more details about the *Winsorizer()* functionality in the following notebook:

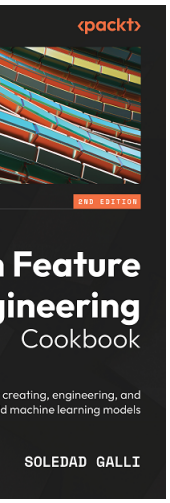
- Jupyter notebook

For more details about this and other feature engineering methods check out these resources:

Or read our book:

Fig. 39: Feature Engineering for Machine Learning

Both our book and course are suitable for beginners and more advanced data scientists alike. By purchasing them you are supporting Sole, the main developer of Feature-engine.



ure Engineering

ArbitraryOutlierCapper

The *ArbitraryOutlierCapper()* caps the maximum or minimum values of a variable at an arbitrary value indicated by the user. The maximum or minimum values should be entered in a dictionary with the form {feature:capping value}.

Let's look at this in an example. First we load the Titanic dataset, and separate it into a train and a test set:

```
from sklearn.model_selection import train_test_split
from feature_engine.datasets import load_titanic
from feature_engine.outliers import ArbitraryOutlierCapper

X, y = load_titanic(
    return_X_y_frame=True,
    predictors_only=True,
    handle_missing=True,
)


X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=0,
)

print(X_train.head())
```

We see the resulting data below:

```
      pclass    ␣
↪ sex         age  sibsp  parch     fare    cabin embarked
501        2  female␣
↪ 13.000000      0      1  19.5000  Missing        S
588        2  female␣
↪  4.000000      1      1  23.0000  Missing        S
402        2  female␣
↪ 30.000000      1      0  13.8583  Missing        C
1193       3    ␣
↪male  29.881135      0      0   7.7250  Missing        Q
686        3  female␣
↪ 22.000000      0      0   7.7250  Missing        Q
```

Now, we set up the *ArbitraryOutlierCapper()* indicating that we want to cap the variable 'age' at 50 and the variable 'Fare' at 200. We do not want to cap these variables on the left side of their distribution.

```
capper = ArbitraryOutlierCapper(
    max_capping_dict={'age': 50, 'fare': 200},
    min_capping_dict=None,
)

capper.fit(X_train)
```

With `fit()` the transformer does not learn any parameter. It just reassigns the entered dictionary to the attribute that will be used in the transformation:

```
capper.right_tail_caps_
```

```
{'age': 50, 'fare': 200}
```

Now, we can go ahead and cap the variables:

```
train_t = capper.transform(X_train)
test_t = capper.transform(X_test)
```

If we now check the maximum values in the transformed data, they should be those entered in the dictionary:

```
train_t[['fare', 'age']].max()
```

```
fare    200.0
age      50.0
dtype: float64
```

Additional resources

You can find more details about the *ArbitraryOutlierCapper()* functionality in the following notebook:

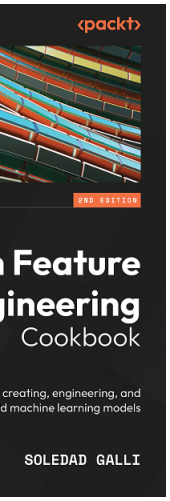
- Jupyter notebook

For more details about this and other feature engineering methods check out these resources:

Or read our book:

Fig. 41: Feature Engineering for Machine Learning

Both our book and course are suitable for beginners and more advanced data scientists alike. By purchasing them you are supporting Sole, the main developer of Feature-engine.



ure Engineering

OutlierTrimmer

Outliers are data points that significantly deviate from the rest of the dataset, potentially indicating errors or rare occurrences. Outliers can distort the learning process of machine learning models by skewing parameter estimates and reducing predictive accuracy. To prevent this, if you suspect that the outliers are errors or rare occurrences, you can remove them from the training data.

In this guide, we show how to remove outliers in Python using the *OutlierTrimmer()*.

The first step to removing outliers consists of identifying those outliers. Outliers can be identified through various statistical methods, such as box plots, z-scores, the interquartile range (IQR), or the median absolute deviation. Additionally, visual inspection of the data using scatter plots or histograms is common practice in data science, and can help detect observations that significantly deviate from the overall pattern of the dataset.

The *OutlierTrimmer()* can identify outliers by using all of these methods and then remove them automatically. Hence, we'll begin this guide with data analysis, showing how we can identify outliers through these statistical methods and boxplots, and then we will remove outliers by using the *OutlierTrimmer()*.

Identifying outliers

Outliers are data points that are usually far greater, or far smaller than some value that determines where most of the values in the distribution lie. These minimum and maximum values, that delimit the data distribution, can be calculated in 4 ways: by using the z-score if the variable is normally distributed, by using the interquartile range proximity rule or the median absolute deviation if the variables are skewed, or by using percentiles.

Gaussian limits or z-score

If the variable shows a normal distribution, most of its values lie between the mean minus 3 times the standard deviation and the mean plus 3 times the standard deviation. Hence, we can determine the limits of the distribution as follows:

- right tail (upper_bound): mean + 3* std
- left tail (lower_bound): mean - 3* std

We can consider outliers those data points that lie beyond these limits.

Interquartile range proximity rule

The interquartile range proximity rule can be used to detect outliers both in variables that show a normal distribution and in variables with a skew. When using the IQR, we detect outliers as those values that lie before the 25th percentile times a factor of the IQR, or after the 75th percentile times a factor of the IQR. This factor is normally 1.5, or 3 if we want to be more stringent. With the IQR method, the limits are calculated as follows:

IQR limits:

- right tail (upper_limit): 75th quantile + 3* IQR
- left tail (lower_limit): 25th quantile - 3* IQR

where IQR is the inter-quartile range:

- IQR = 75th quantile - 25th quantile = third quartile - first quartile.

Observations found beyond those limits can be considered extreme values.

Maximum absolute deviation

Parameters like the mean and the standard deviation are strongly affected by the presence of outliers. Therefore, it might be a better solution to use a metric that is robust against outliers, like the median absolute deviation from the median, commonly shortened to the median absolute deviation (MAD), to delimit the normal data distribution.

When we use MAD, we determine the limits of the distribution as follows:

MAD limits:

- right tail (upper_limit): median + 3* MAD
- left tail (lower_limit): median - 3* MAD

MAD is the median absolute deviation from the median. In other words, MAD is the median value of the absolute difference between each observation and its median.

- MAD = median(abs(X-median(X))

Percentiles

A simpler way to determine the values that delimit the data distribution is by using percentiles. Like this, outlier values would be those that lie before or after a certain percentile or quantiles:

- right tail: 95th percentile
- left tail: 5th percentile

The number of outliers identified by any of these methods will vary. These methods detect outliers, but they can't decide if they are true outliers or faithful data points. That required further examination and domain knowledge.

Let's move on to removing outliers in Python.

Remove outliers in Python

In this demo, we'll identify and remove outliers from the Titanic Dataset. First, let's load the data and separate it into train and test:

```
from sklearn.model_selection import train_test_split
from feature_engine.datasets import load_titanic
from feature_engine.outliers import OutlierTrimmer

X, y = load_titanic(
    return_X_y_frame=True,
    predictors_only=True,
    handle_missing=True,
)


X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=0,
)

print(X_train.head())
```

We see the resulting pandas dataframe below:

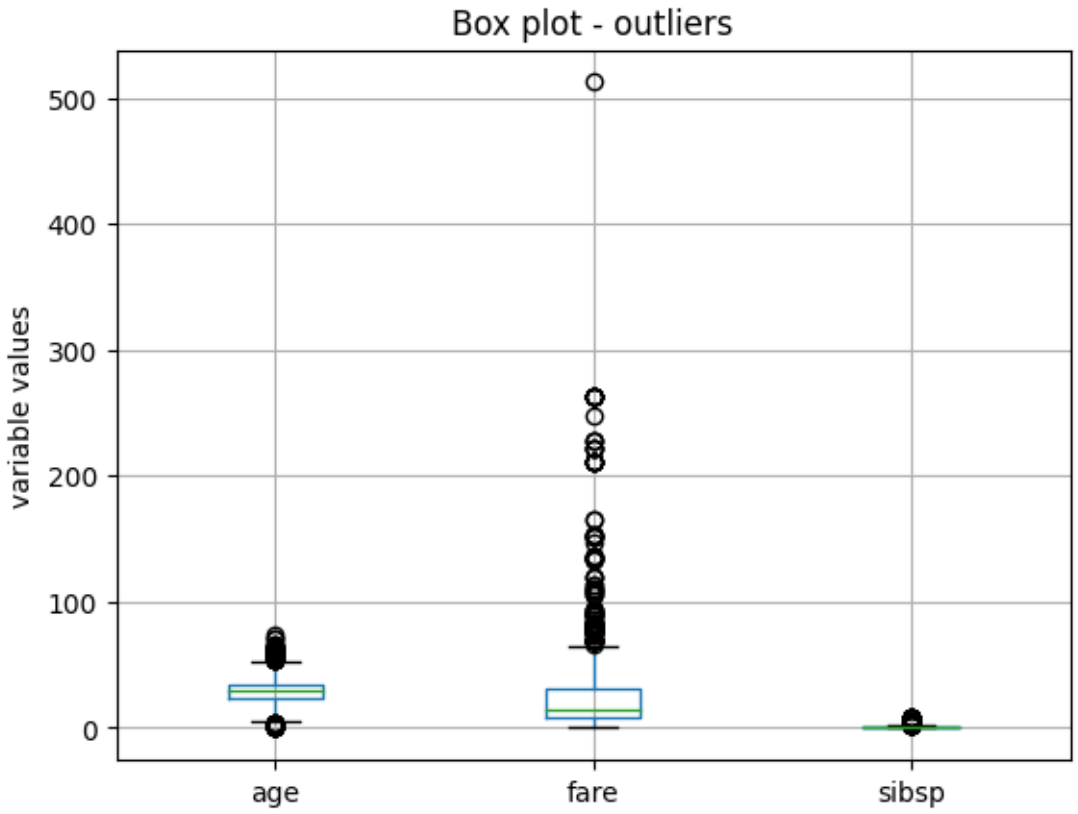
```
      pclass   ␣
↪ sex        age  sibsp  parch     fare    cabin embarked
501        2  female␣
↪ 13.000000      0      1  19.5000  Missing        S
588        2  female␣
↪  4.000000      1      1  23.0000  Missing        S
402        2  female␣
↪ 30.000000      1      0  13.8583  Missing        C
1193       3   ␣
↪male  29.881135      0      0   7.7250  Missing        Q
686        3  female␣
↪ 22.000000      0      0   7.7250  Missing        Q
```

Identifying outliers

Let's now identify potential extreme values in the training set by using boxplots.

```
X_train.boxplot(column=['age', 'fare', 'sibsp'])
plt.title("Box plot - outliers")
plt.ylabel("variable values")
plt.show()
```

In the following boxplots, we see that all three variables have data points that are significantly greater than the majority of the data distribution. The variable age also shows outlier values towards the lower values.

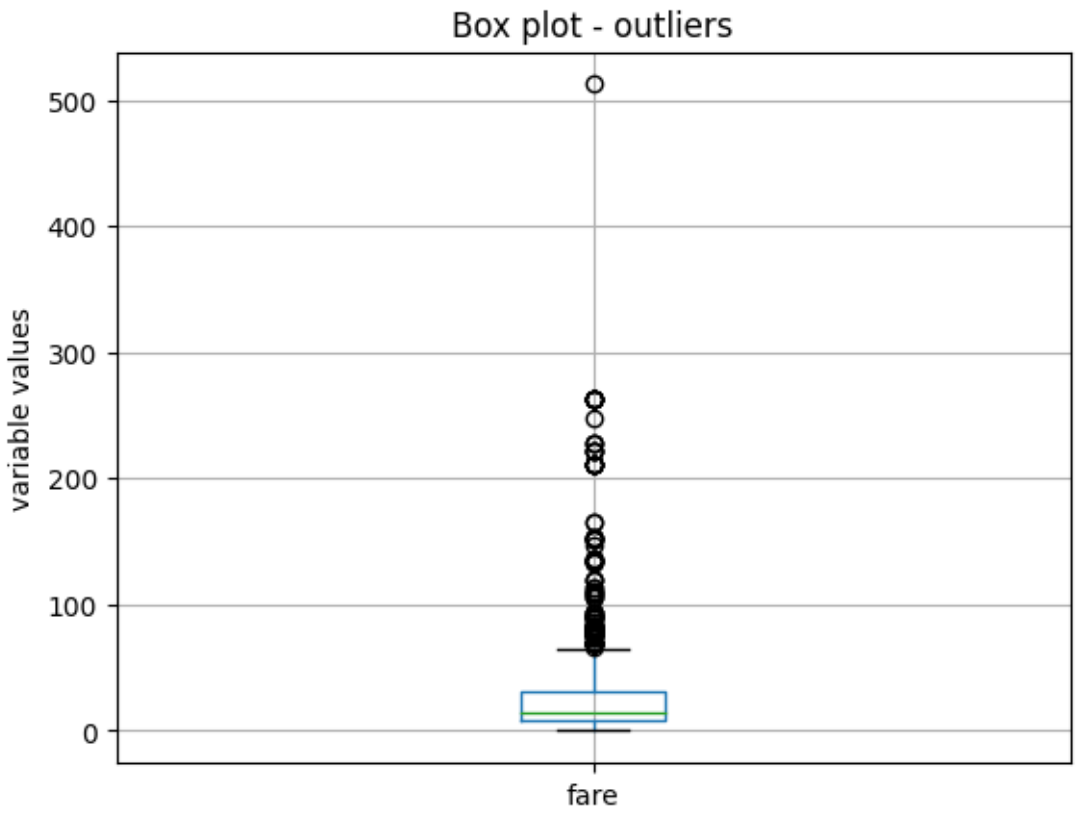


The variables have different scales, so let's plot them individually for better visualization. Let's start by making a boxplot of the variable fare:

```
X_train.boxplot(column=['fare'])
plt.title("Box plot - outliers")
plt.ylabel("variable values")
plt.show()
```

We see the boxplot in the following image:

Next, we plot the variable age:



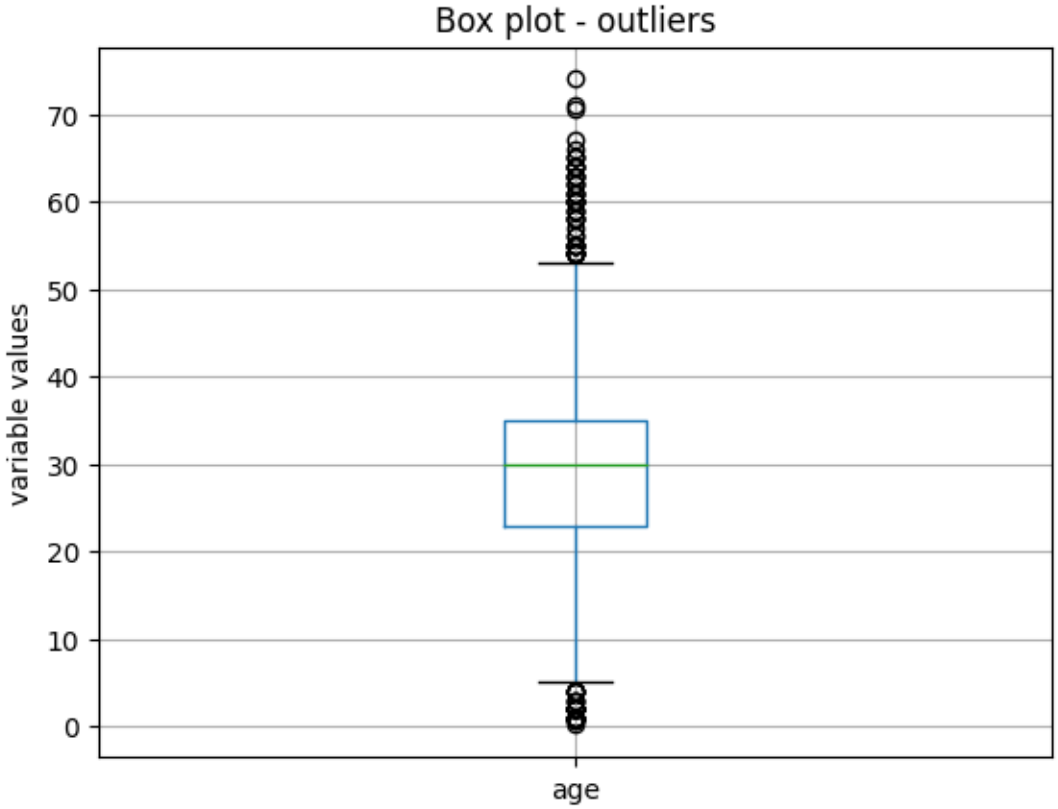

```
X_train.boxplot(column=['age'])
plt.title("Box plot - outliers")
plt.ylabel("variable values")
plt.show()
```

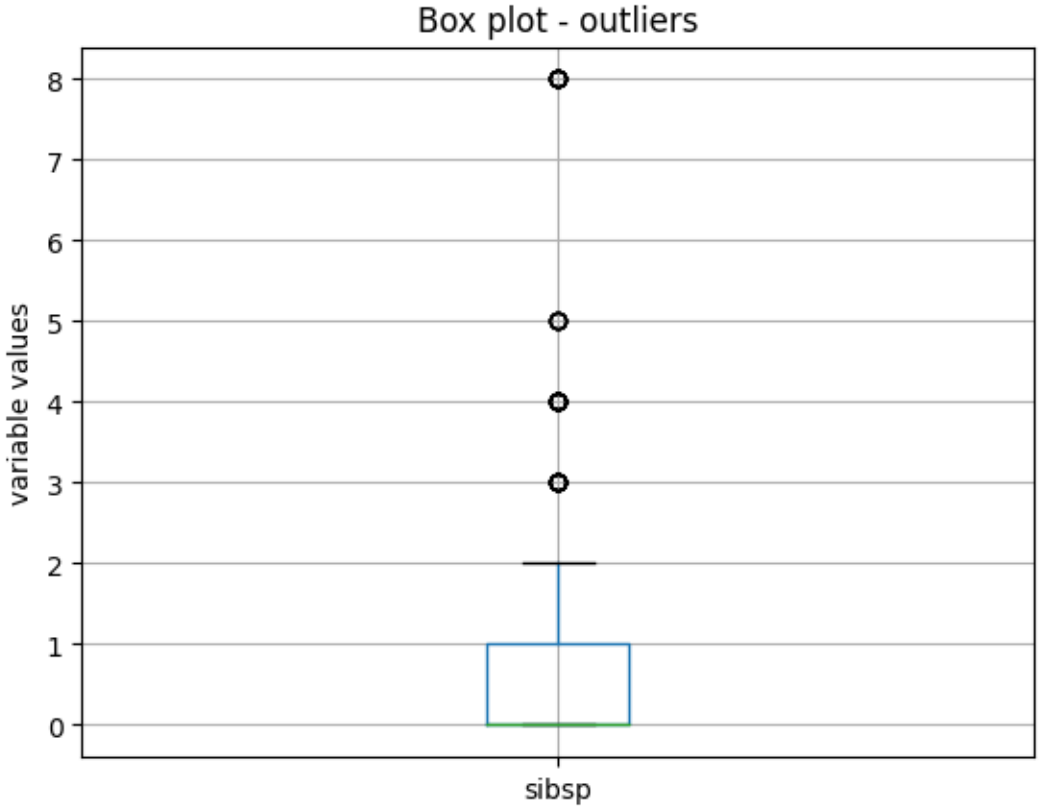
We see the boxplot in the following image:

And finally, we make a boxplot of the variable sibsp:

```
X_train.boxplot(column=['sibsp'])
plt.title("Box plot - outliers")
plt.ylabel("variable values")
plt.show()
```

We see the boxplot and the outlier values in the following image:


Box plot - outliers
variable values
0
10
20
30
40
50
60
70
age


Box plot - outliers
variable values
0
1
2
3
4
5
6
7
8
sibsp

Outlier removal

Now, we will use the *OutlierTrimmer()* to remove outliers. We'll start by using the IQR as outlier detection method.

IQR

We want to remove outliers at the right side of the distribution only (param `tail`). We want the maximum values to be determined using the 75th quantile of the variable (param `capping_method`) plus 1.5 times the IQR (param `fold`). And we only want to cap outliers in 2 variables, which we indicate in a list.

```
ot = OutlierTrimmer(capping_method='iqr',
                    tail='right',
                    fold=1.5,
                    variables=['sibsp', 'fare'],
                    )

ot.fit(X_train)
```

With `fit()`, the *OutlierTrimmer()* finds the values at which it should cap the variables. These values are stored in one of its attributes:

```
ot.right_tail_caps_
```

```
{'sibsp': 2.5, 'fare': 66.34379999999999}
```

We can now go ahead and remove the outliers:

```
train_t = ot.transform(X_train)
test_t = ot.transform(X_test)
```

We can compare the sizes of the original and transformed datasets to check that the outliers were removed:

```
X_train.shape, train_t.shape
```

We see that the transformed dataset contains less rows:

```
((916, 8), (764, 8))
```

If we evaluate now the maximum of the variables in the transformed datasets, they should be <= the values observed in the attribute `right_tail_caps_`:

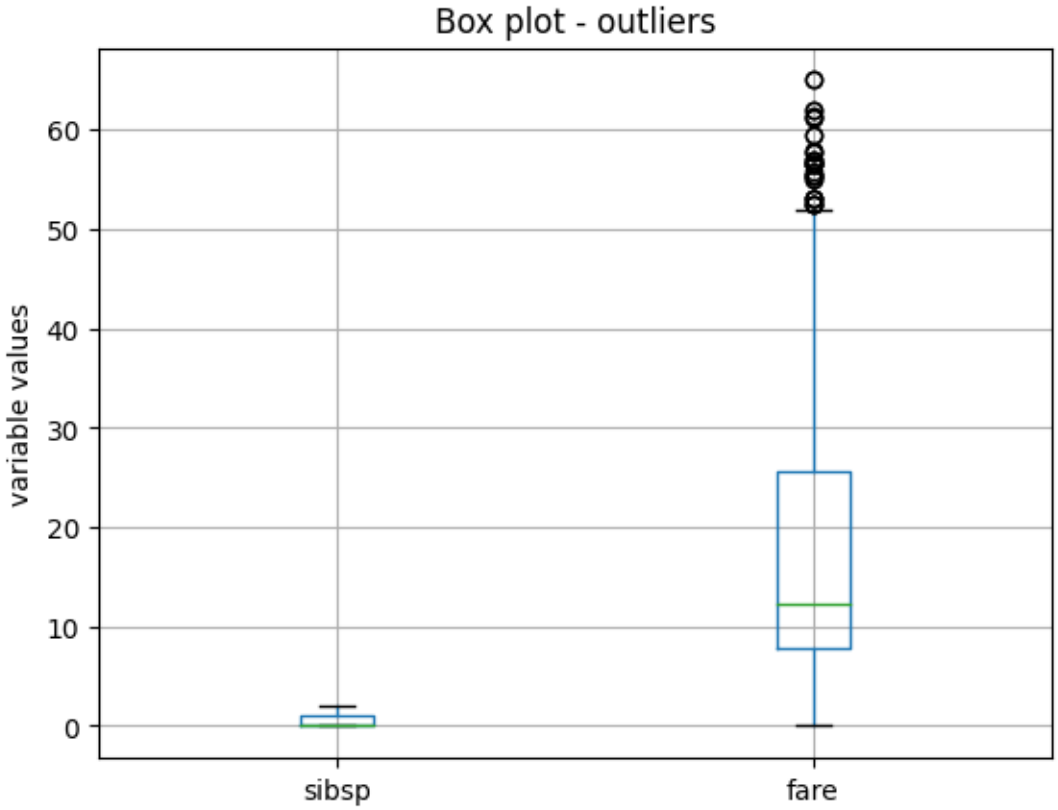
```
train_t[['fare', 'age']].max()
```

```
fare    65.0
age     53.0
dtype: float64
```

Finally, we can check the boxplots of the transformed variables to corroborate the effect on their distribution.

```
train_t.boxplot(column=['sibsp', "fare"])
plt.title("Box plot - outliers")
plt.ylabel("variable values")
plt.show()
```

We see the boxplot and the `sibsp` does no longer have outliers, but as `fare` was very skewed, when removing outliers, the parameters of the IQR change, and we continue to see outliers:



We'll come back to this later, but now let's continue showing the functionality of the *OutlierTrimmer()*.

When we remove outliers from the datasets, we then need to re-align the target variables. We can do this with pandas loc. But the *OutlierTrimmer()* can do that automatically as follows:

```
train_t, y_train_t = ot.transform_x_y(X_train, y_train)
test_t, y_test_t = ot.transform_x_y(X_test, y_test)
```

The method `transform_x_y` will remove outliers from the predictor datasets and then align the target variable. That means, it will remove from the target those rows corresponding to the outlier values.

We can corroborate the size adjustment in the target as follows:

```
y_train.shape, y_train_t.shape,
```

The previous command returns the following output:

```
((916,), (764,))
```

We can obtain the names of the fetaures in the transformed dataset as follows:

```
ot.get_feature_names_out()
```

That returns the following variable namesL

```
['pclass', 'sex
↪', 'age', 'sibsp', 'parch', 'fare', 'cabin', 'embarked']
```

MAD

We saw that the IQR did not work amazingly for the variable fare, because its skew is too big. So let's remove outliers by using the MAD instead:

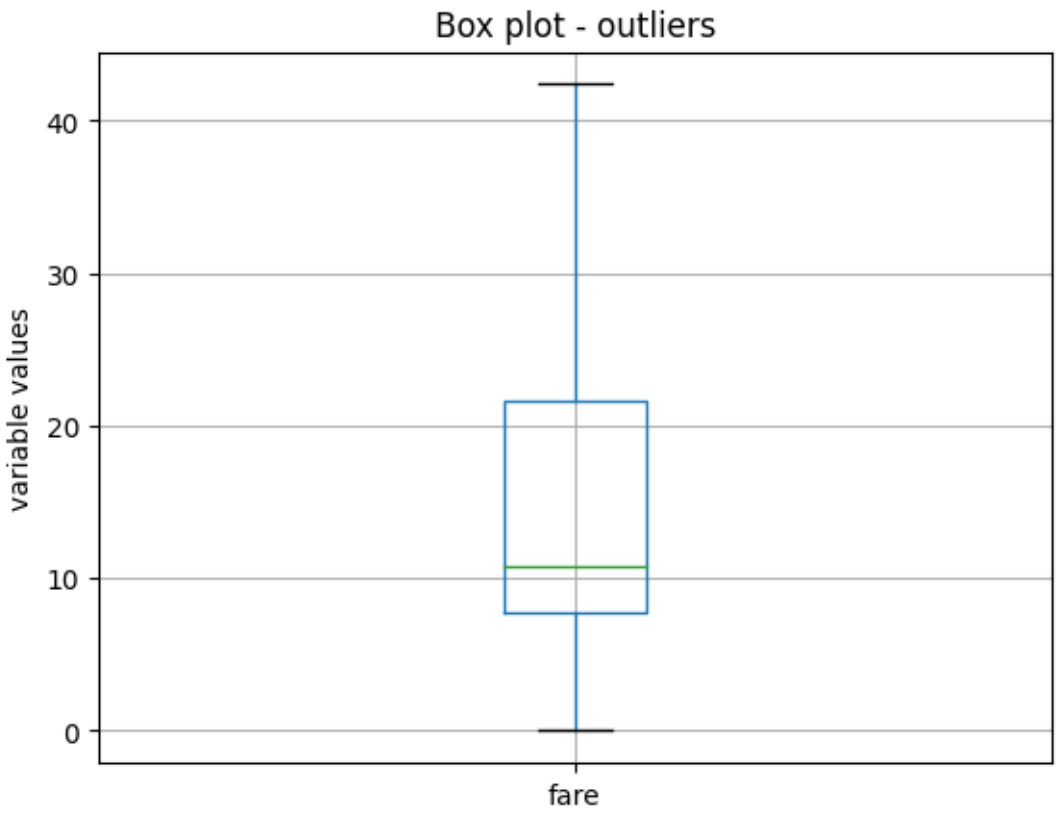
```
ot = OutlierTrimmer(capping_method='mad',
                    tail='right',
                    fold=3,
                    variables=['fare'],
                    )

ot.fit(X_train)

train_t, y_train_t = ot.transform_x_y(X_train, y_train)
test_t, y_test_t = ot.transform_x_y(X_test, y_test)

train_t.boxplot(column=["fare"])
plt.title("Box plot - outliers")
plt.ylabel("variable values")
plt.show()
```

In the following image, we see that after this transformation, the variable fare no longer shows outlier values:

Z-score

The variable age is more homogeneously distributed across its value range, so let's use the z-score or gaussian approximation to detect outliers. We saw in the boxplot that it has outliers at both ends, so we'll cap both ends of the distribution:

```
ot_age = OutlierTrimmer(capping_method='gaussian',
                    tail="both",
                    fold=3,
                    variables=['age'],
                    )


ot_age.fit(X_train)
```



Let's inspect the maximum values beyond which data points will be considered outliers:

```
ot_age.right_tail_caps_
```

```
{'age': 67.73951212364803}
```

And the lower values beyond which data points will be considered outliers:

```
ot_age.left_tail_caps_
```

```
{'age': -7.410476010820627}
```

The minimum value does not make sense, because age can't be negative. So, we'll try capping this variable with percentiles instead.

Percentiles

We'll cap age at the bottom 5 and top 95 percentile:

```
ot = OutlierTrimmer(capping_method='mad',
                    tail='right',
                    fold=3,
                    variables=['fare'],
                    )

ot.fit(X_train)
```

Let's inspect the maximum values beyond which data points will be considered outliers:

```
ot_age.right_tail_caps_
```

```
{'age': 54.0}
```

And the lower values beyond which data points will be considered outliers:

```
ot_age.left_tail_caps_
```

```
{'age': 9.0}
```

Let's tranform the dataset and target:

```
train_
↪t, y_train_t = ot_age.transform_x_y(X_train, y_train)
test_t, y_test_t = ot_age.transform_x_y(X_test, y_test)
```

And plot the resulting variable:

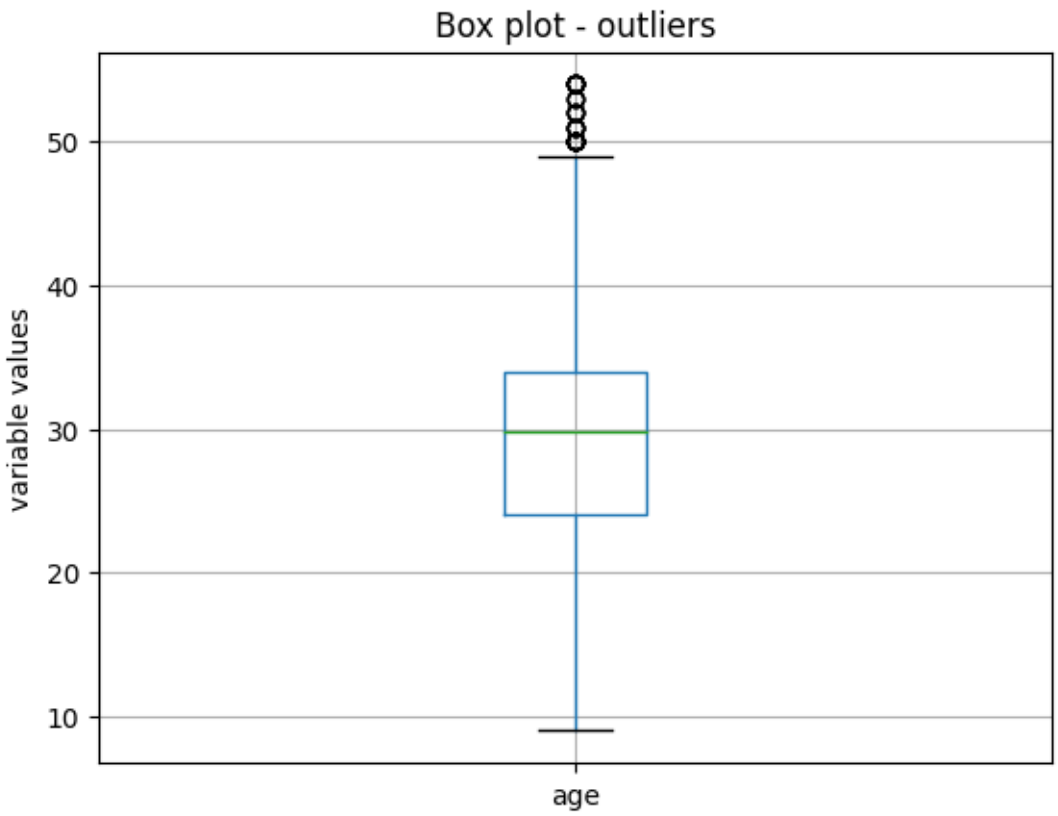
```
train_t.boxplot(column=['age'])
plt.title("Box plot - outliers")
plt.ylabel("variable values")
plt.show()
```

In the following image, we see that after this transformation, the variable age still shows some outlier values towards its higher values, so we should be more stringent with the percentiles or use MAD:

Pipeline

The *OutlierTrimmer()* removes observations from the predictor data sets. If we want to use this transformer within a Pipeline, we can't use Scikit-learn's pipeline because it can't readjust the target. But we can use Feature-engine's pipeline instead.

Let's start by creating a pipeline that removes outliers and then encodes categorical variables:




```
from feature_engine.encoding import OneHotEncoder
from feature_engine.pipeline import Pipeline

pipe = Pipeline(
    [
        ("outliers", ot),
        ("enc", OneHotEncoder()),
    ]
)

pipe.fit(X_train, y_train)
```

The `transform` method will transform only the dataset with the predictors, just like scikit-learn's pipeline:

```
train_t = pipe.transform(X_train)

X_train.shape, train_t.shape
```

We see the adjusted data size compared to the original size here:

```
((916, 8), (736, 76))
```

Feature-engine's pipeline can also adjust the target:

```
train_t, y_train_t = pipe.transform_x_y(X_train, y_train)

y_train.shape, y_train_t.shape
```

We see the adjusted data size compared to the original size here:

```
((916,), (736,))
```

To wrap up, let's add a machine learning algorithm to the pipeline. We'll use logistic regression to predict survival:

```
from sklearn.linear_model import LogisticRegression

pipe = Pipeline(
    [
        ("outliers", ot),
        ("enc", OneHotEncoder()),
        ("logit", LogisticRegression(random_state=10)),
    ]
)

pipe.fit(X_train, y_train)
```

Now, we can predict survival:

```
preds = pipe.predict(X_train)

preds[0:10]
```

We see the following output:

```
array([1, 1, 1, 0, 1, 0, 1, 1, 0, 1], dtype=int64)
```

We can obtain the probability of survival:

```
preds = pipe.predict_proba(X_train)

preds[0:10]
```

We see the following output:

```
array([[0.13027536, 0.86972464],
       [0.14982143, 0.85017857],
       [0.2783799 , 0.7216201 ],
       [0.86907159, 0.13092841],
       [0.31794531, 0.68205469],
       [0.86905145, 0.13094855],
       [0.1396715 , 0.8603285 ],
       [0.48403632, 0.51596368],
       [0.6299007 , 0.3700993 ],
       [0.49712853, 0.50287147]])
```

We can obtain the accuracy of the predictions over the test set:

```
pipe.score(X_test, y_test)
```

That returns the following accuracy:

```
0.7823343848580442
```

We can obtain the names of the features after the trasnformation:

```
pipe[:-1].get_feature_names_out()
```

That returns the following names:

```
['pclass',
 'age',
 'sibsp',
 'parch',
 'fare',
 'sex_female',
 'sex_male',
 'cabin_Missing',
...
```

And finally, we can obtain the transformed dataset and target as follows:

```
X_test_
↪t, y_test_t = pipe[:-1].transform_x_y(X_test, y_test)

X_test.shape, X_test_t.shape
```

We see the resulting sizes here:

```
((393, 8), (317, 76))
```

Tutorials, books and courses

In the following Jupyter notebook, in our accompanying Github repository, you will find more examples using *OutlierTrimmer()*.

- Jupyter notebook

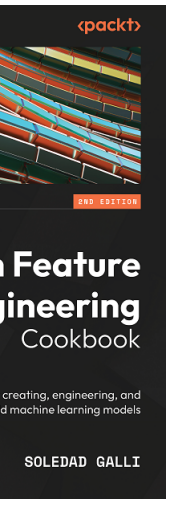
For tutorials about this and other feature engineering methods check out our online course:

Or read our book:

Fig. 43: Feature Engineering for Machine Learning

Both our book and course are suitable for beginners and more advanced data scientists alike. By purchasing them you are supporting Sole, the main developer of Feature-engine.

Variance Stabilizing Transformations



ure Engineering

Feature-engine's variable transformers transform numerical variables with various mathematical transformations.

Variable transformations are commonly used to spread the values of the original variables over a wider value range. See the following illustration:

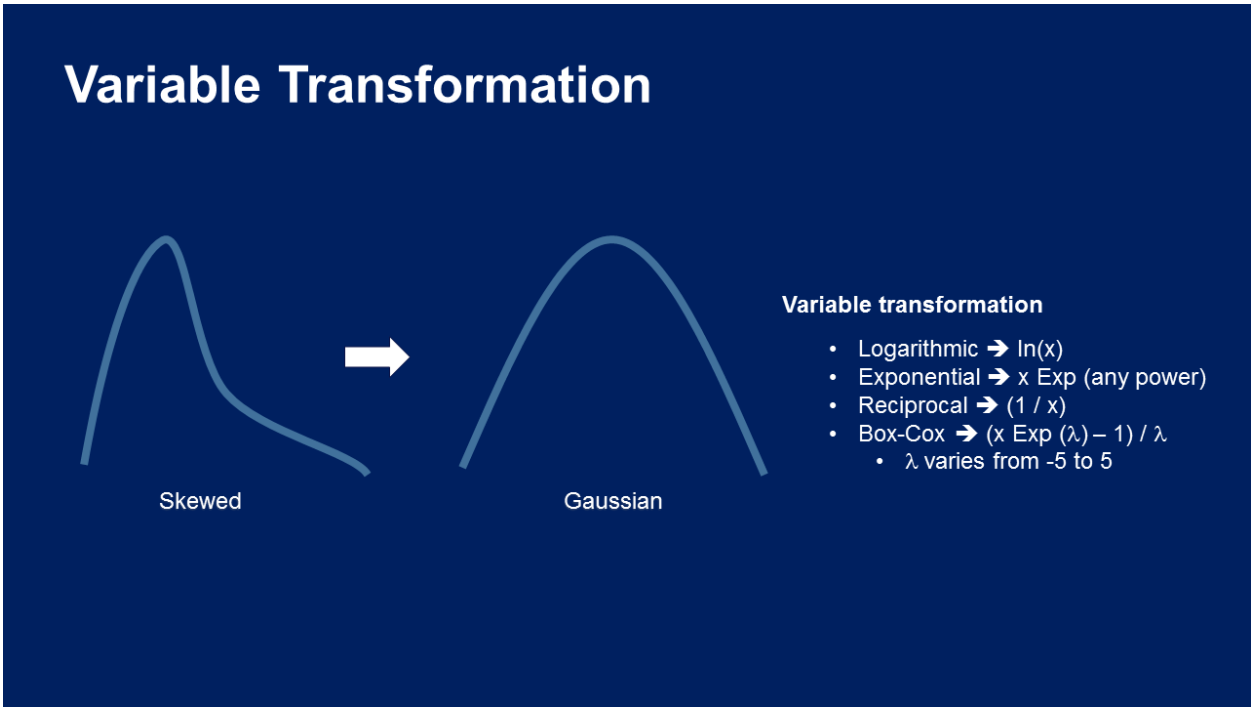
Article

We added a lot of information about **variance stabilizing transformations** in this article.

Note

Note however, that improving the value spread is not always possible and it depends on the nature of the variable.

Transformers



LogTransformer

The log transformation is used to transform skewed data so that the values are more evenly distributed across the value range.

Some regression models, like linear regression, t-test and ANOVA, make assumptions about the data. When the assumptions are not met, we can't trust the results. Applying data transformations is common practice during regression analysis because it can help make the data meet those assumptions and hence obtain more reliable results.

The logarithm function is helpful for dealing with positive data with a right-skewed distribution. That is, those variables whose observations accumulate towards lower values. A common example is the variable income, with a heavy accumulation of values toward lower salaries.

More generally, when data follows a log-normal distribution, then its log-transformed version approximates a normal distribution.

Other useful transformations are the square root transformation, power transformations and the box cox transformation.

In statistical analysis, we can apply the logarithmic transformation to both the dependent variable (that is, the target) and the independent variables (that is, the predictors). These can help meet the linear regression model assumptions and unmask a linear relationship between predictors and response variable.

With Feature-engine, we can only log transform input features. You can easily transform the target variable by applying `np.log(y)`.

The LogTransformer

The *LogTransformer()* applies the natural logarithm or the logarithm in base 10 to numerical variables. Note that the logarithm can only be applied to positive values. Thus, if the variable contains 0 or negative variables, this transformer will return and error.

To transform non-positive variables you can add a constant to shift the data points towards positive values. You can do this from within the transformer by using *LogCpTransformer()*.

Python implementation

In this section, we will apply the logarithmic transformation to some independent variables from the Ames house prices dataset.

Let's start by importing the required libraries and transformers for data analysis and then load the dataset and separate it into train and test sets.

```
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split

from feature_engine.transformation import LogTransformer

data = fetch_openml(name='house_prices', as_frame=True)
data = data.frame

X = data.drop(['SalePrice', 'Id'], axis=1)
y = data['SalePrice']

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42)

print(X_train.head())
```

In the following output we see the predictor variables of the house prices dataset:

```
      MSSubClass␣
↪MSZoning  LotFrontage  LotArea Street Alley LotShape  \
254           ␣
↪ 20       RL         70.0     8400   Pave   NaN      Reg
1066          ␣
↪ 60       RL         59.0     7837   Pave   NaN      IR1
638           ␣
↪ 30       RL         67.0     8777   Pave   NaN      Reg
799           ␣
↪ 50       RL         60.0     7200   Pave   NaN      Reg
380           ␣
↪ 50       RL         50.0     5000   Pave  Pave      Reg

     LandContour Utilities␣
```

(continues on next page)

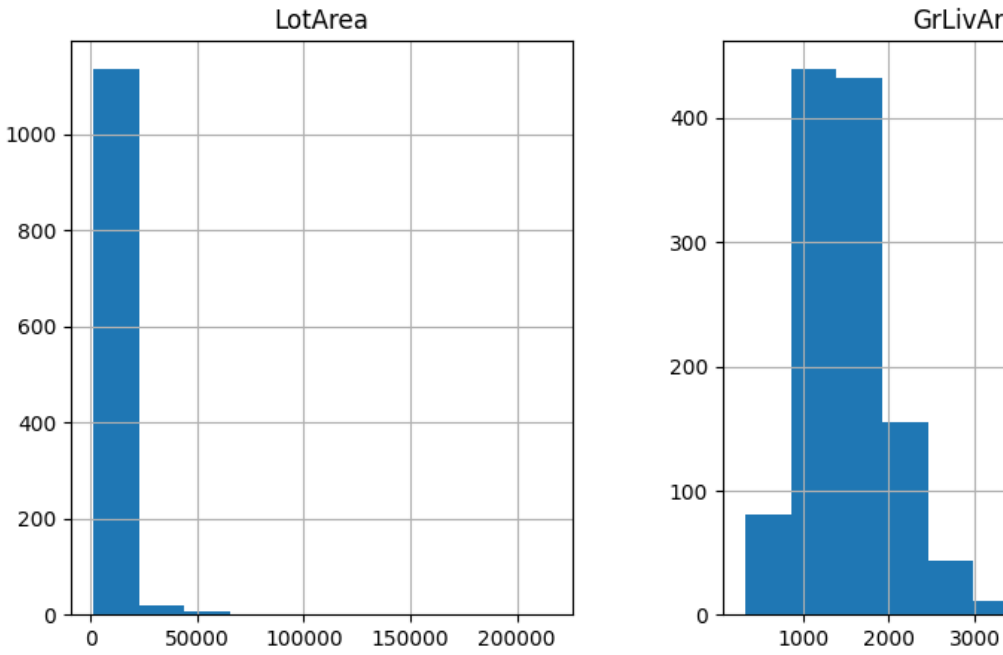
(continued from previous page)

```
↪LotConfig  ... ScreenPorch PoolArea PoolQC  Fence  \
254          Lvl   ␣
↪AllPub    Inside  ...           0        0    NaN    NaN
1066         Lvl   ␣
↪AllPub    Inside  ...           0        0    NaN    NaN
638          Lvl   ␣
↪AllPub    Inside  ...           0        0    NaN  MnPrv
799          Lvl   ␣
↪AllPub    Corner  ...           0        0    NaN  MnPrv
380          Lvl   ␣
↪AllPub    Inside  ...           0        0    NaN    NaN

     MiscFeature␣
↪MiscVal  MoSold  YrSold  SaleType  SaleCondition
254      ␣
↪     NaN       0       6    2010        WD         Normal
1066     ␣
↪     NaN       0       5    2009        WD         Normal
638      ␣
↪     NaN       0       5    2008        WD         Normal
799      ␣
↪     NaN       0       6    2007        WD         Normal
380      ␣
↪     NaN       0       5    2010        WD         Normal

[5 rows x 79 columns]
```

Let's inspect the distribution of 2 variables from the original data with histograms.

```
X_train[['LotArea', 'GrLivArea']].hist(figsize=(10,5))
plt.show()
```

In the following plots we see that the variables show a right-skewed distribution, so they are good candidates for the log transformation:



We want to apply the natural logarithm to these 2 variables in the dataset using the *LogTransformer()*. We set up the transformer as follows:

```
logt␣
↪= LogTransformer(variables = ['LotArea', 'GrLivArea'])

logt.fit(X_train)
```

With `fit()`, this transformer does not learn any parameters, but it checks that the variables you entered are numerical, or if no variable was entered, it will automatically find all numerical variables.

To apply the logarithm in base 10, pass `'10'` to the `base` parameter when setting up the transformer.

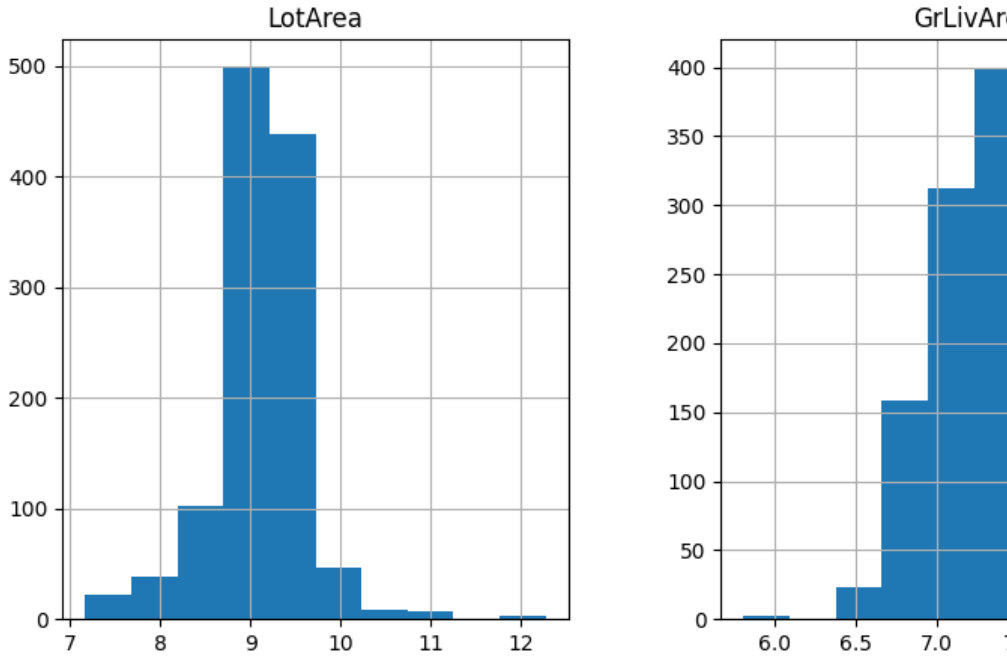
Now, we can go ahead and transform the data:

```
train_t = logt.transform(X_train)
test_t = logt.transform(X_test)
```

Let's now examine the variable distribution in the log-transformed data with histograms:

```
train_t[['LotArea', 'GrLivArea']].hist(figsize=(10,5))
plt.show()
```

In the following histograms we see that the natural log transformation helped make the variables better approximate a normal distribution.

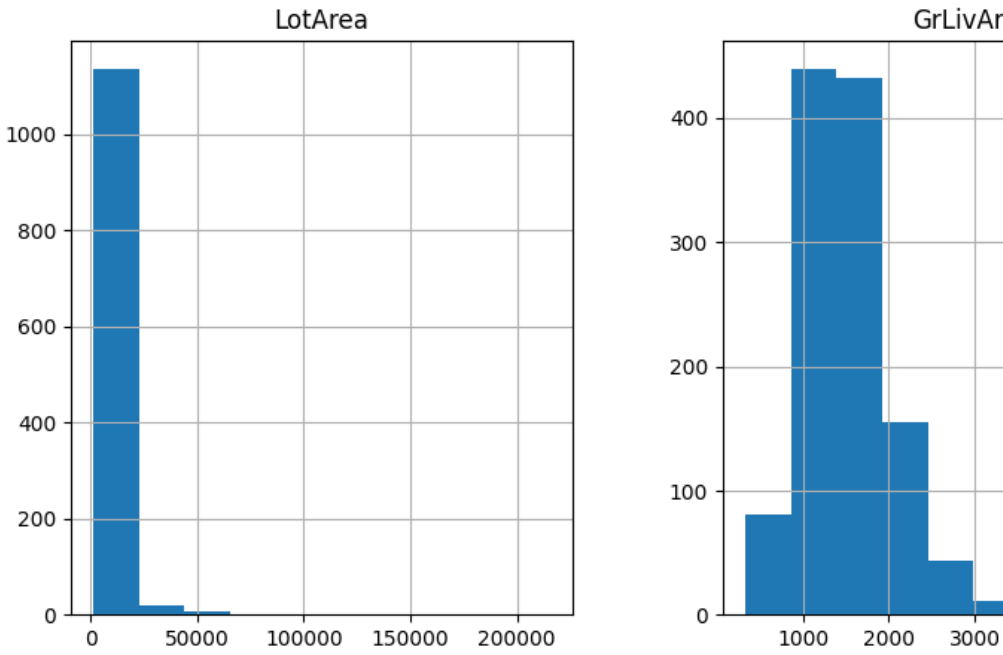


Note that the transformed variable has a more Gaussian looking distribution.

If we want to recover the original data representation, with the method `inverse_transform`, the *LogTransformer()* will apply the exponential function to obtain the variable in its original scale:

```
train_unt = logt.inverse_transform(train_t)
test_unt = logt.inverse_transform(test_t)

train_unt[['LotArea', 'GrLivArea']].hist(figsize=(10,5))
plt.show()
```

In the following plots we see histograms showing the variables in their original scale:



Following the transformations with scatter plots and residual analysis of the regression models helps understand if the transformations are useful in our regression analysis.

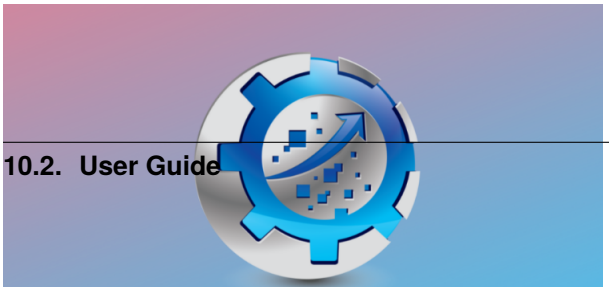
Tutorials, books and courses

You can find more details about the *LogTransformer()* here:

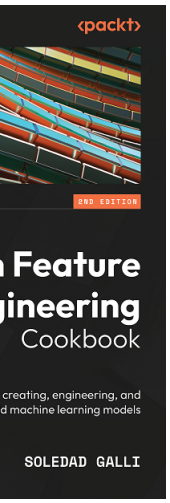
- Jupyter notebook

For tutorials about this and other data transformation methods, like the square root transformation, power transformations, the box cox transformation, check out our online course:

Or read our book:



Both our book and course are suitable for beginners and more advanced data scientists alike. By purchasing them you are supporting Sole, the main developer of Feature-engine.



ure Engineering

LogCpTransformer

The *LogCpTransformer()* applies the transformation log(x + C), where C is a positive constant.

You can enter the positive quantity to add to the variable. Alternatively, the transformer will find the necessary quantity to make all values of the variable positive.

Example

Let's load the California housing dataset that comes with Scikit-learn and separate it into train and test sets.

```
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.datasets import fetch_california_housing

from feature_engine.transformation import LogCpTransformer

# Load dataset
X, y = fetch_
↪california_housing( return_X_y=True, as_frame=True)

# Separate into train and test sets
X_train, X_test, y_train, y_test =  train_test_split(
    X, y, test_size=0.3, random_state=0)
```

Now we want to apply the logarithm to 2 of the variables in the dataset using the *LogCpTransformer()*. We want the transformer to detect automatically the quantity "C" that needs to be added to the variable:

```
# set up the variable transformer
tf = LogCpTransformer(variables
↪= ["MedInc", "HouseAge"], C="auto")

# fit the transformer
tf.fit(X_train)
```

With `fit()` the *LogCpTransformer()* learns the quantity "C" and stores it as an attribute. We can visualise the learned parameters as follows:

```
# learned constant C
tf.C_
```

```
{'MedInc': 1.4999, 'HouseAge': 2.0}
```

Applying the log of a variable plus a constant in this dataset does not make much sense because all variables are positive, that is why the constant values C for the former variables are possible.

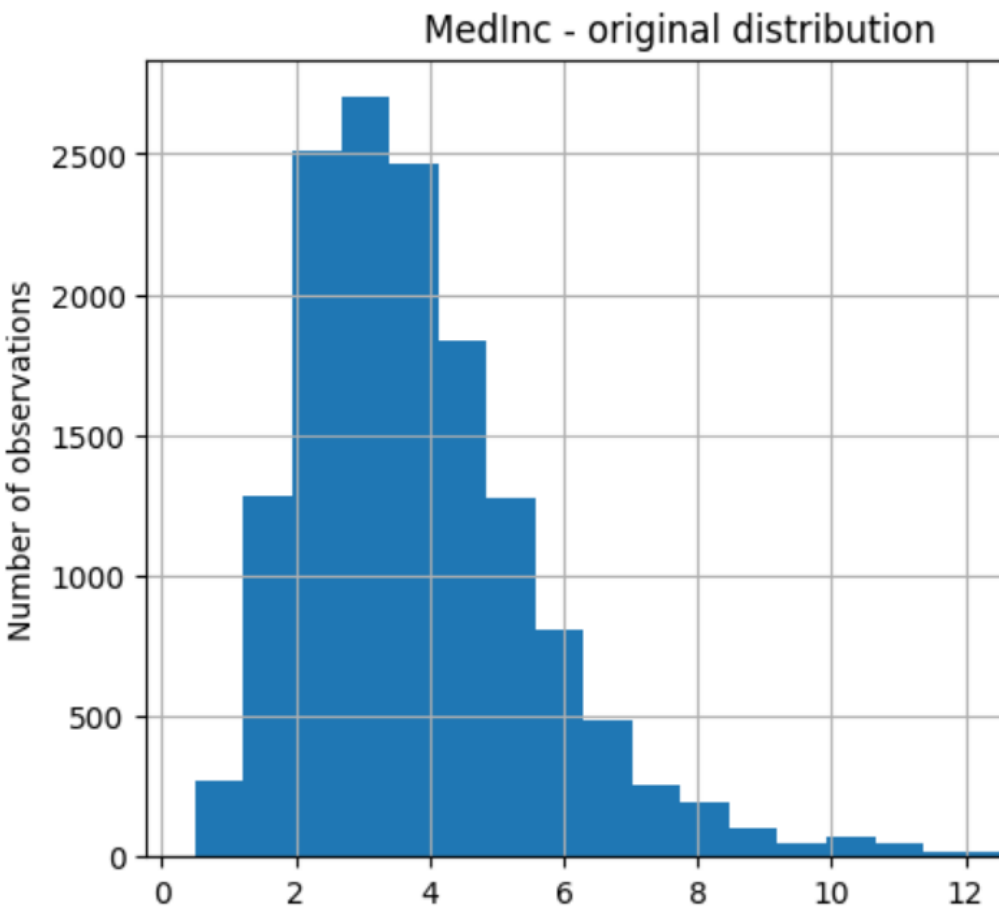
We will carry on with the demo anyways.

We can now go ahead and transform the variables:

```
# transform the data
train_t= tf.transform(X_train)
test_t= tf.transform(X_test)
```

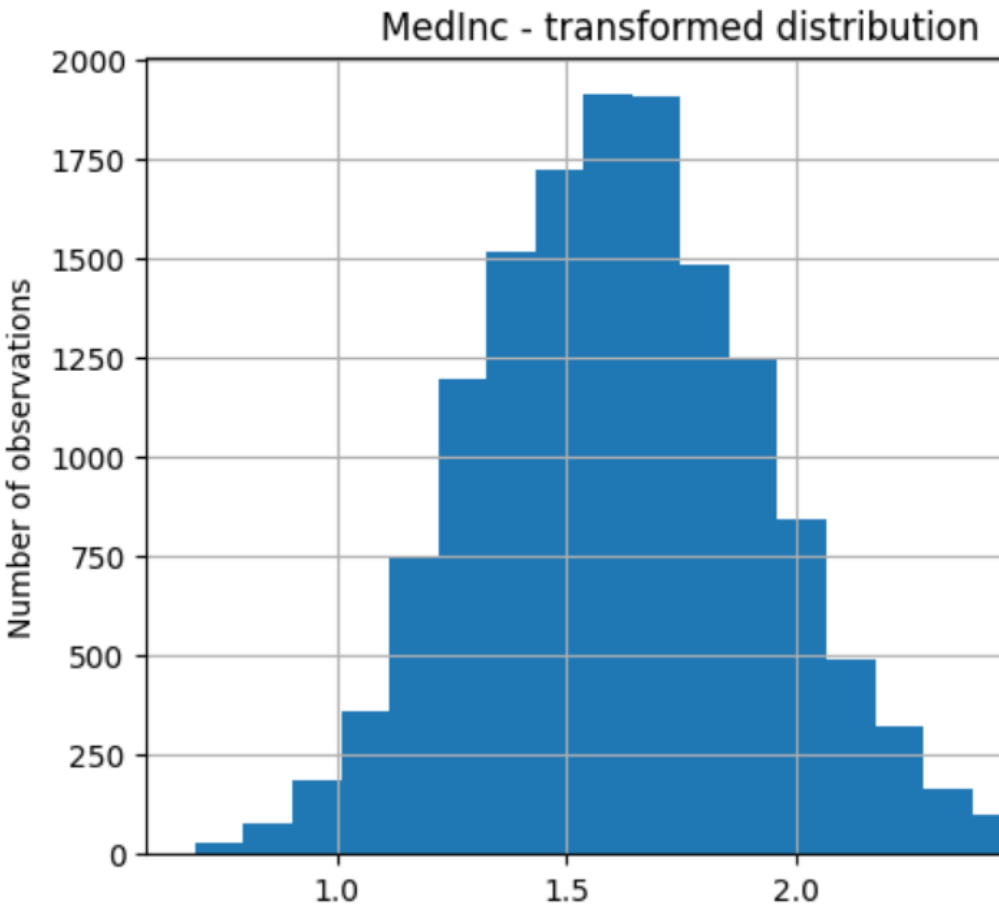
Then we can plot the original variable distribution:

```
# un-transformed variable
X_train["MedInc"].hist(bins=20)
plt.title("MedInc - original distribution")
plt.ylabel("Number of observations")
```



And the distribution of the transformed variable:

```
# transformed variable
train_t["MedInc"].hist(bins=20)
plt.title("MedInc - transformed distribution")
plt.ylabel("Number of observations")
```



Tutorials, books and courses

You can find more details about the *LogCpTransformer()* here:

- Jupyter notebook

For tutorials about this and other data transformation methods, like the square root transformation, power transformations, the box cox transformation, check out our online course:

Or read our book:

Fig. 47: Feature Engineering for Machine Learning

Fig. 48: Python Feature Engineering Cookbook

Both our book and course are suitable for beginners and more advanced data scientists alike.

The *ReciprocalTransformer()* applies the reciprocal transformation 1 / x to numerical variables.

The *ReciprocalTransformer()* only works with numerical variables with non-zero values. If a variable contains the value 0, the transformer will raise an error.

Let's load the house prices dataset and separate it into train and test sets (more details about the dataset *here*).

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

from feature_engine import transformation as vt

# Load dataset
data = data = pd.read_csv('houseprice.csv')

# Separate into train and test sets
X_train, X_test, y_train, y_test =  train_test_split(
            data.drop(['Id', 'SalePrice'], axis=1),
            data['SalePrice'], test_size=0.3, random_
↪state=0)
```

Now we want to apply the reciprocal transformation to 2 variables in the dataframe:

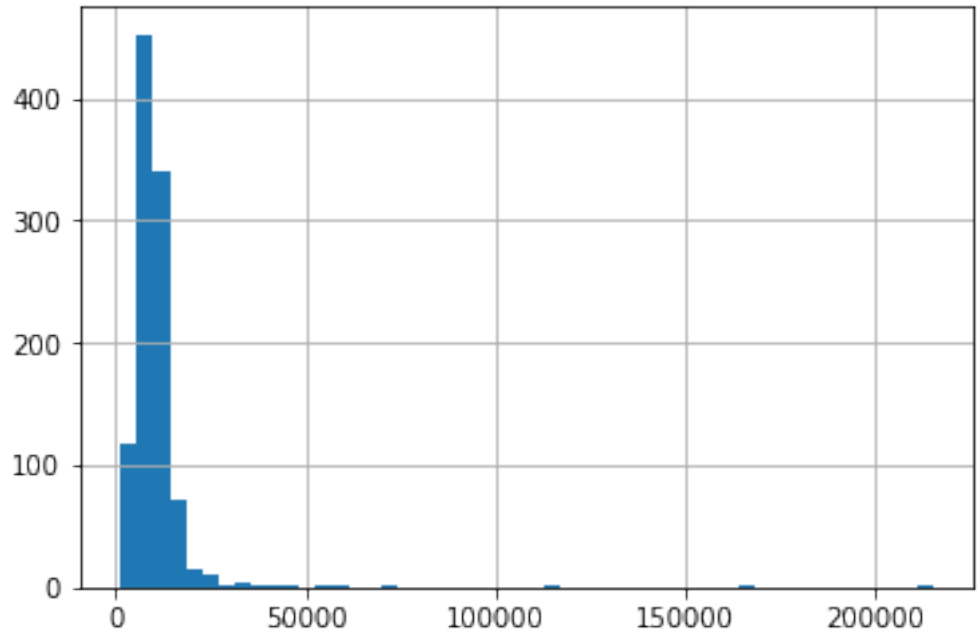
```
# set up the variable transformer
tf = vt.ReciprocalTransformer(variables = ['LotArea', 'GrLivArea'])

# fit the transformer
tf.fit(X_train)
```

The transformer does not learn any parameters. So we can go ahead and transform the variables:

```
# transform the data
train_t= tf.transform(X_train)
test_t= tf.transform(X_test)
```

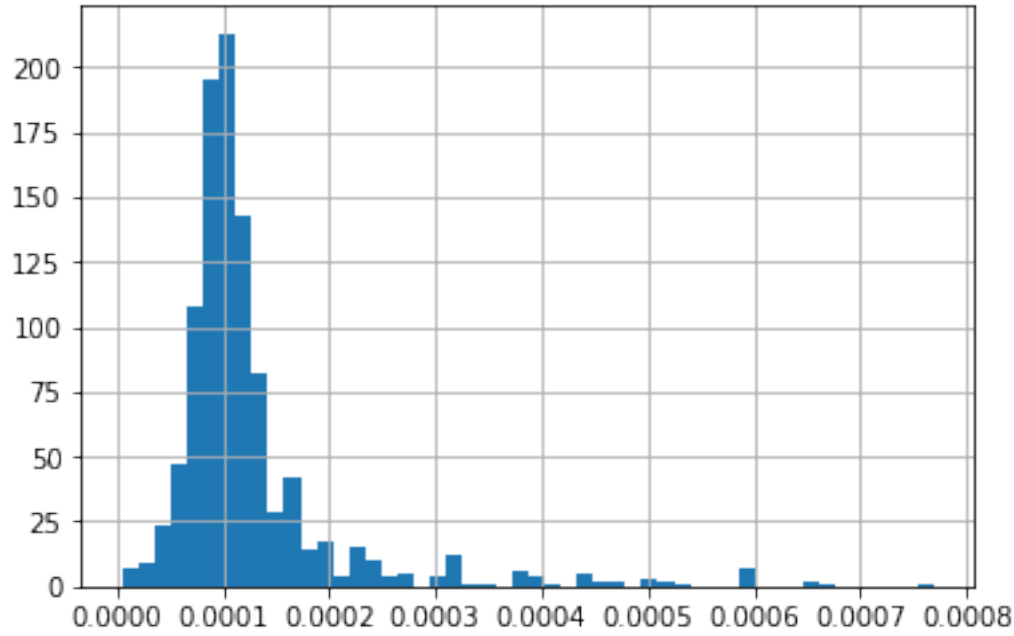
Finally, we can plot the original variable distribution:

```
# un-transformed variable
X_train['LotArea'].hist(bins=50)
```



And now the distribution after the transformation:

```
# transformed variable
train_t['LotArea'].hist(bins=50)
```



Additional resources

You can find more details about the *ReciprocalTransformer()* here:

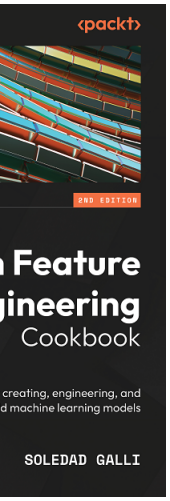
- Jupyter notebook

For more details about this and other feature engineering methods check out these resources:

Or read our book:

Fig. 49: Feature Engineering for Machine Learning

Both our book and course are suitable for beginners and more advanced data scientists alike. By purchasing them you are supporting Sole, the main developer of Feature-engine.



ure Engineering

ArcsinTransformer

The *ArcsinTransformer()* applies the arcsin transformation to numerical variables.

The arcsine transformation, also called arcsin square root transformation, or angular transformation, takes the form of arcsin(sqrt(x)) where x is a real number between 0 and 1.

The arcsin square root transformation helps in dealing with probabilities, percentages, and proportions.

The *ArcsinTransformer()* only works with numerical variables with values between 0 and 1. If the variable contains a value outside of this range, the transformer will raise an error.

Example

Let's load the breast cancer dataset from scikit-learn and separate it into train and test sets.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_breast_cancer

from␣
↪feature_engine.transformation import ArcsinTransformer

#Load dataset
breast_cancer = load_breast_cancer()
X = pd.DataFrame(breast_
↪cancer.data, columns=breast_cancer.feature_names)
y = breast_cancer.target

# Separate data into train and test sets
X_train, X_test,␣
↪y_train, y_test = train_test_split(X, y, random_state=0)
```

Now we want to apply the arcsin transformation to some of the variables in the dataframe. These variables values are in the range 0-1, as we will see in coming histograms.

First, let's make a list with the variable names:

```
vars_ = [
  'mean compactness',
  'mean concavity',
  'mean concave points',
  'mean fractal dimension',
  'smoothness error',
  'compactness error',
  'concavity error',
  'concave points error',
  'symmetry error',
  'fractal dimension error',
  'worst symmetry',
  'worst fractal dimension']
```

Now, let's set up the arscin transformer to modify only the previous variables:

```
# set up the arcsin transformer
tf = ArcsinTransformer(variables = vars_)

# fit the transformer
tf.fit(X_train)
```

The transformer does not learn any parameters when applying the fit method. It does check however that the variables are numericals and with the correct value range.

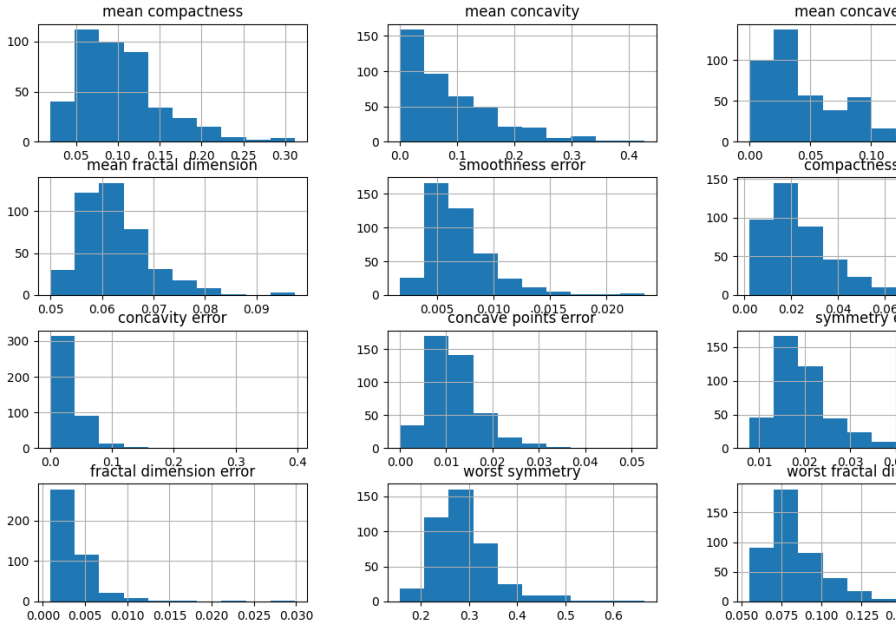
We can now go ahead and transform the variables:

```
# transform the data
train_t = tf.transform(X_train)
test_t = tf.transform(X_test)
```

And that's it, now the variables have been transformed with the arscin formula.

Finally, let's make a histogram for each of the original variables to examine their distribution:

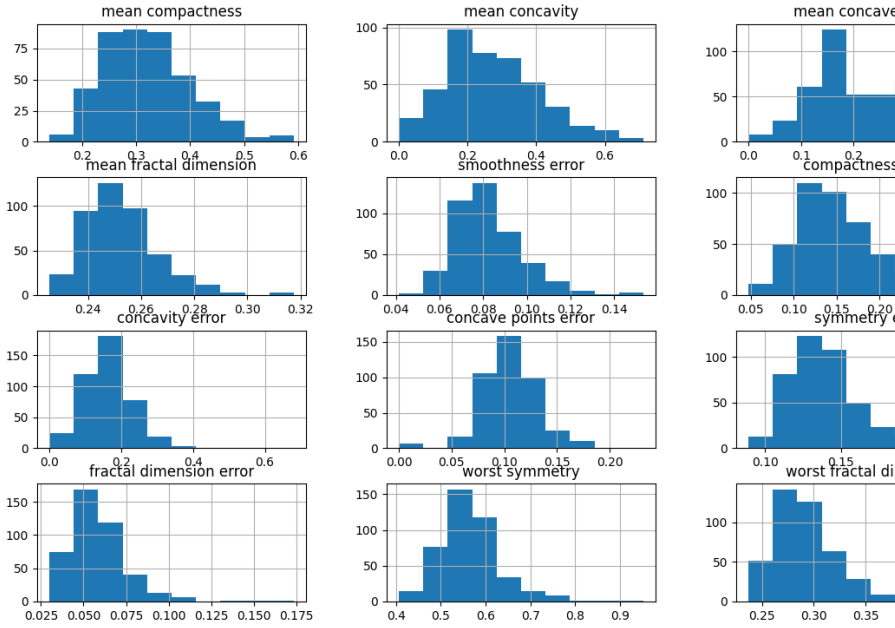
```
# original variables
X_train[vars_].hist(figsize=(20,20))
```



You can see in the previous image that many of the variables are skewed. Note however, that all variables had values between 0 and 1.

Now, let's examine the distribution after the transformation:

```
# transformed variable
train_t[vars_].hist(figsize=(20,20))
```



You can see in the previous image that many variables have after the transformation a more Gaussian looking shape.

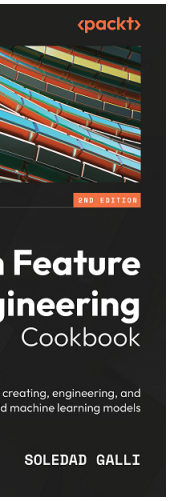
Additional resources

For more details about this and other feature engineering methods check out these resources:

Or read our book:

Both our book and course are suitable for beginners and more advanced data scientists alike. By purchasing them you are supporting Sole, the main developer of Feature-engine.

PowerTransformer



ure Engineering

The *PowerTransformer()* applies power or exponential transformations to numerical variables.

Let's load the house prices dataset and separate it into train and test sets (more details about the dataset *here*).

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

from feature_engine import transformation as vt

# Load dataset
data = data = pd.read_csv('houseprice.csv')

# Separate into train and test sets
X_train, X_test, y_train, y_test =  train_test_split(
            data.drop(['Id', 'SalePrice'], axis=1),
    ␣
↪       data['SalePrice'], test_size=0.3, random_state=0)
```

Now we want to apply the square root to 2 variables in the dataframe:

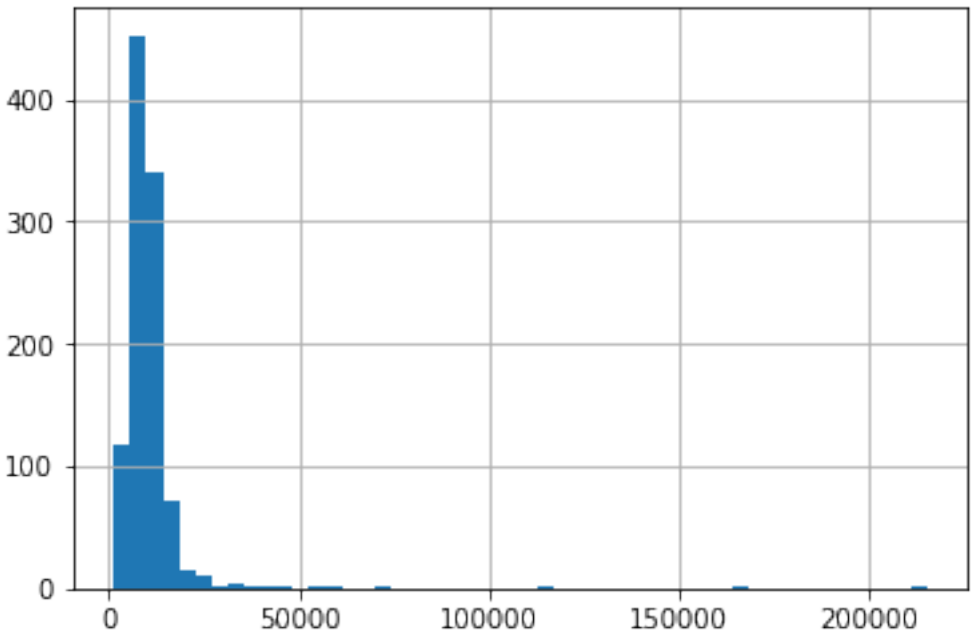
```
# set up the variable transformer
tf = vt.PowerTransformer(variables␣
↪= ['LotArea', 'GrLivArea'], exp=0.5)

# fit the transformer
tf.fit(X_train)
```

The transformer does not learn any parameters. So we can go ahead and transform the variables:

```
# transform the data
train_t= tf.transform(X_train)
test_t= tf.transform(X_test)
```

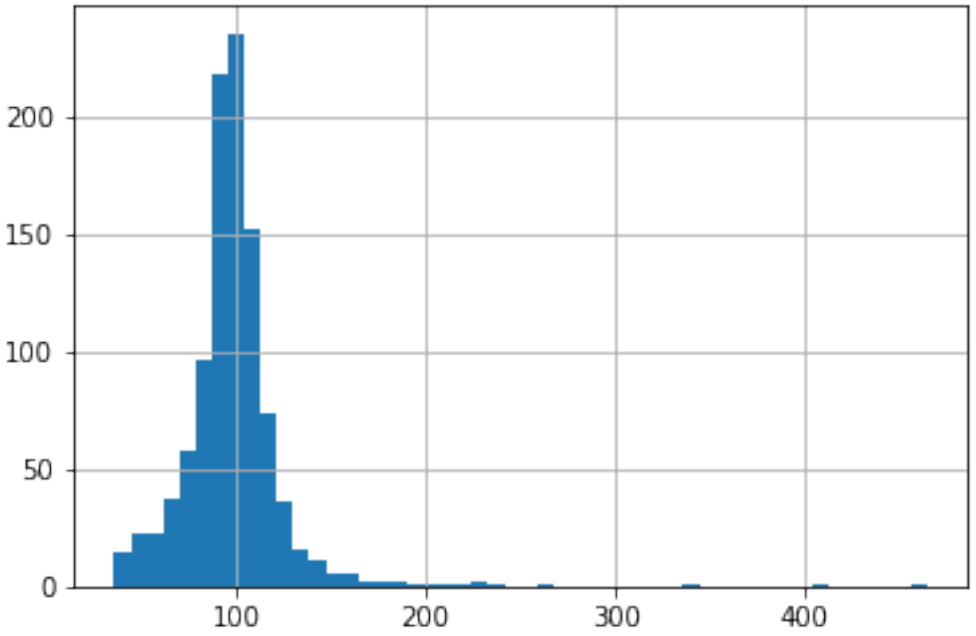
Finally, we can plot the original variable distribution:

```
# un-transformed variable
X_train['LotArea'].hist(bins=50)
```



And now the distribution after the transformation:

```
# transformed variable
train_t['LotArea'].hist(bins=50)
```



Additional resources

You can find more details about the *PowerTransformer()* here:

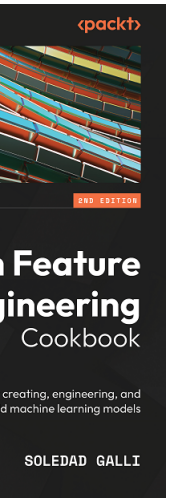
- Jupyter notebook

For more details about this and other feature engineering methods check out these resources:

Or read our book:

Fig. 53: Feature Engineering for Machine Learning

Both our book and course are suitable for beginners and more advanced data scientists alike. By purchasing them you are supporting Sole, the main developer of Feature-engine.



ure Engineering

BoxCoxTransformer

The Box-Cox transformation is a generalization of the power transformations family and is defined as follows:

```
y = (x** - 1) / ,       for  != 0
y = log(x),               for  = 0
```

Here, y is the transformed data, x is the variable to transform and is the transformation parameter.

The Box Cox transformation is used to reduce or eliminate variable skewness and obtain features that better approximate a normal distribution.

The Box Cox transformation evaluates commonly used transformations. When = 1 then we have the original variable, when = 0, we have the logarithm transformation, when = - 1 we have the reciprocal transformation, and when = 0.5 we have the square root.

The Box-Cox transformation evaluates several values of using the maximum likelihood, and selects the optimal value of the parameter, which is the one that returns the best transformation. The best transformation occurs when the transformed data better approximates a normal distribution.

The Box Cox transformation is defined for strictly positive variables. If your variables are not strictly positive, you can add a constant or use the Yeo-Johnson transformation instead.

Uses of the Box Cox Transformation

Many statistical methods that we use for data analysis make assumptions about the data. For example, the linear regression model assumes that the values of the dependent variable are independent, that there is a linear relationship between the response variable and the independent variables, and that the residuals are normally distributed and centered at 0.

When these assumptions are not met, we can't fully trust the results of our regression analyses. To make data meet the assumptions and improve the trust in the models, it is common practice in data science projects to transform the variables before the analysis.

In time series forecasting, we use the Box Cox transformation to make non-stationary time series stationary.

References

George Box and David Cox. "An Analysis of Transformations". Read at a RESEARCH MEETING, 1964. https://rss.onlinelibrary.wiley.com/doi/abs/10.1111/j.2517-6161.1964.tb00553.x

BoxCoxTransformer

The *BoxCoxTransformer()* applies the BoxCox transformation to numerical variables. It uses SciPy.stats under the hood to apply the transformation.

The BoxCox transformation works only for strictly positive variables (>=0). If the variable contains 0 or negative values, the *BoxCoxTransformer()* will return an error. To apply this transformation to non-positive variables, you can add a constant value. Alternatively, you can apply the Yeo-Johnson transformation with the *YeoJohnsonTransformer()*.

Python code examples

In this section, we will apply this data transformation to 2 variables of the Ames house prices dataset.

Let's start by importing the modules, classes and functions and then loading the house prices dataset and separating it into train and test sets.

```
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split

from␣
↪feature_engine.transformation import BoxCoxTransformer

data = fetch_openml(name='house_prices', as_frame=True)
data = data.frame

X = data.drop(['SalePrice', 'Id'], axis=1)
y = data['SalePrice']


X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42)

print(X_train.head())
```

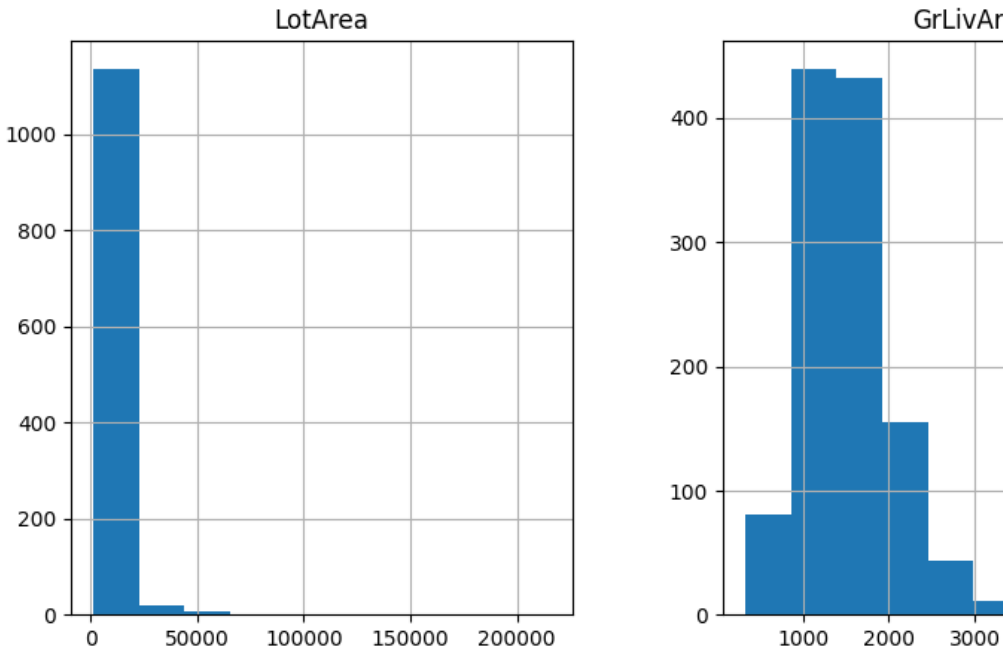
In the following output we see the predictor variables of the house prices dataset:

```
      MSSubClass␣
↪MSZoning  LotFrontage  LotArea Street Alley LotShape  \
254          ␣
↪ 20       RL         70.0     8400   Pave   NaN      Reg
1066         ␣
↪ 60       RL         59.0     7837   Pave   NaN      IR1
638          ␣
↪ 30       RL         67.0     8777   Pave   NaN      Reg
799          ␣
↪ 50       RL         60.0     7200   Pave   NaN      Reg
380          ␣
↪ 50       RL         50.0     5000   Pave  Pave      Reg

     LandContour Utilities␣
↪LotConfig  ... ScreenPorch PoolArea PoolQC  Fence  \
254          Lvl   ␣
↪AllPub    Inside  ...           0        0    NaN    NaN
1066         Lvl   ␣
↪AllPub    Inside  ...           0        0    NaN    NaN
638          Lvl   ␣
↪AllPub    Inside  ...           0        0    NaN  MnPrv
799          Lvl   ␣
↪AllPub    Corner  ...           0        0    NaN  MnPrv
380          Lvl   ␣
↪AllPub    Inside  ...           0        0    NaN    NaN

     MiscFeature␣
↪MiscVal  MoSold  YrSold  SaleType  SaleCondition
254      ␣
↪     NaN       0       6    2010        WD         Normal
1066     ␣
↪     NaN       0       5    2009        WD         Normal
638      ␣
↪     NaN       0       5    2008        WD         Normal
799      ␣
↪     NaN       0       6    2007        WD         Normal
380      ␣
↪     NaN       0       5    2010        WD         Normal

[5 rows x 79 columns]
```

Let's inspect the distribution of 2 variables in the original data with histograms.

```
X_train[['LotArea', 'GrLivArea']].hist(figsize=(10,5))
plt.show()
```

In the following plots we see that the variables are non-normally distributed:



Now we apply the BoxCox transformation to the 2 indicated variables. First, we set up the transformer and fit it to the train set, so that it finds the optimal lambda value.

```
boxcox =␣
↪BoxCoxTransformer(variables = ['LotArea', 'GrLivArea'])
boxcox.fit(X_train)
```

With `fit()`, the *BoxCoxTransformer()* learns the optimal lambda for the transformation. We can inspect these values as follows:

```
boxcox.lambda_dict_
```

We see the optimal lambda values below:

```
{'LotArea': 0.
↪0028222323212918547, 'GrLivArea': -0.006312580181375803}
```

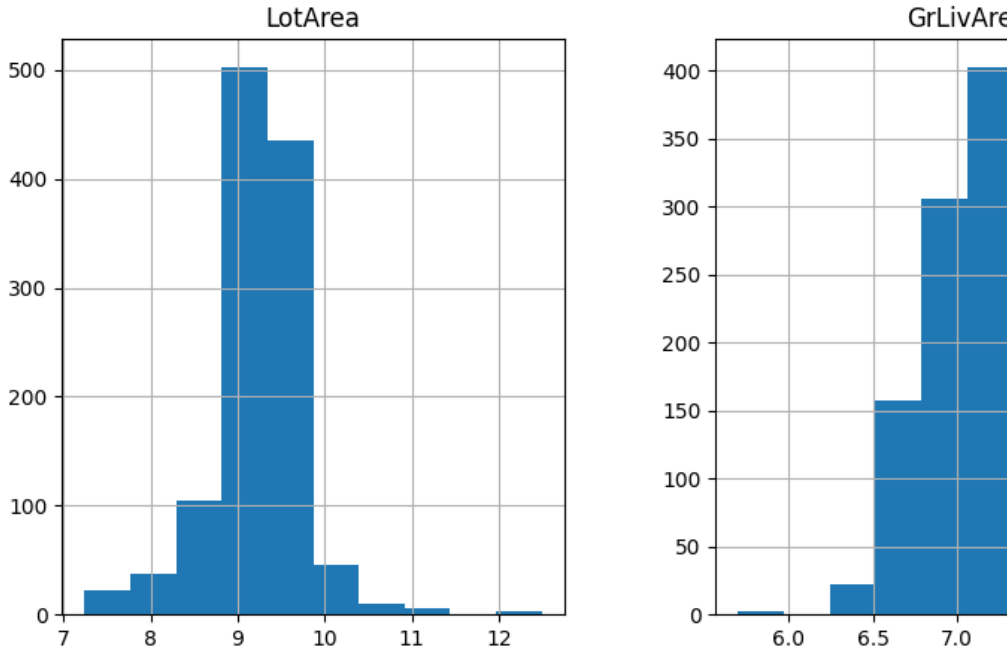
Now, we can go ahead and transform the data:

```
train_t = boxcox.transform(X_train)
test_t = boxcox.transform(X_test)
```

Let's now examine the variable distribution after the transformation with histograms:

```
train_t[['LotArea', 'GrLivArea']].hist(figsize=(10,5))
plt.show()
```

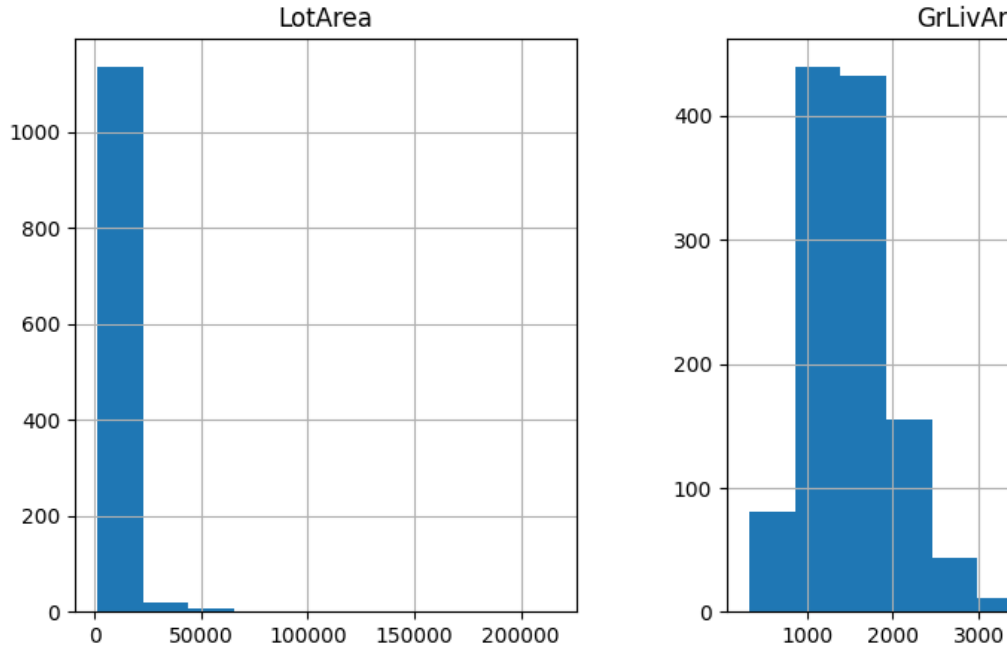
In the following histograms we see that the variables approximate better the normal distribution.



If we want to recover the original data representation, we can also do so as follows:

```
train_unt = boxcox.inverse_transform(train_t)
test_unt = boxcox.inverse_transform(test_t)

train_unt[['LotArea', 'GrLivArea']].hist(figsize=(10,5))
plt.show()
```

In the following plots we see that the variables are non-normally distributed, because they contain the original values, prior to the data transformation:



Tutorials, books and courses

You can find more details about the Box Cox transformation technique with the *BoxCoxTransformer()* here:

- Jupyter notebook

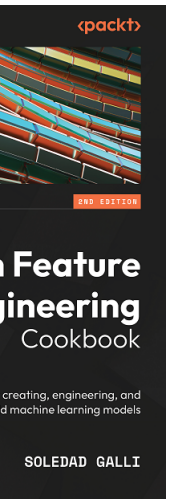
For tutorials about this and other data transformation techniques and feature engineering methods check out our online courses:

Fig. 55: Feature Engineering for Machine Learning

Or read our book:

Fig. 56: Feature Engineering for Time Series Forecasting

YeoJohnsonTransformer



ure Engineering

The *YeoJohnsonTransformer()* applies the Yeo-Johnson transformation to the numerical variables.

The Yeo-Johnson transformation is defined as:

$$\psi(\lambda, y) = \begin{cases} ((y+1)^{\lambda} - 1)/\lambda & \text{if } \lambda \neq 0, y \geq 0 \\ \log(y+1) & \text{if } \lambda = 0, y \geq 0 \\ -[(-y+1)^{2-\lambda} - 1)]/(2-\lambda) & \text{if } \lambda \neq 2, y < 0 \\ -\log(-y+1) & \text{if } \lambda = 2, y < 0 \end{cases}$$

where Y is the response variable and is the transformation parameter.

The Yeo-Johnson transformation implemented by this transformer is that of SciPy.stats.

Example

Let's load the house prices dataset and separate it into train and test sets (more details about the dataset *here*).

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

from feature_engine import transformation as vt

# Load dataset
data = data = pd.read_csv('houseprice.csv')

# Separate into train and test sets
X_train, X_test, y_train, y_test =  train_test_split(
            data.drop(['Id', 'SalePrice'], axis=1),
    ␣
↪       data['SalePrice'], test_size=0.3, random_state=0)
```

Now we apply the Yeo-Johnson transformation to the 2 indicated variables:

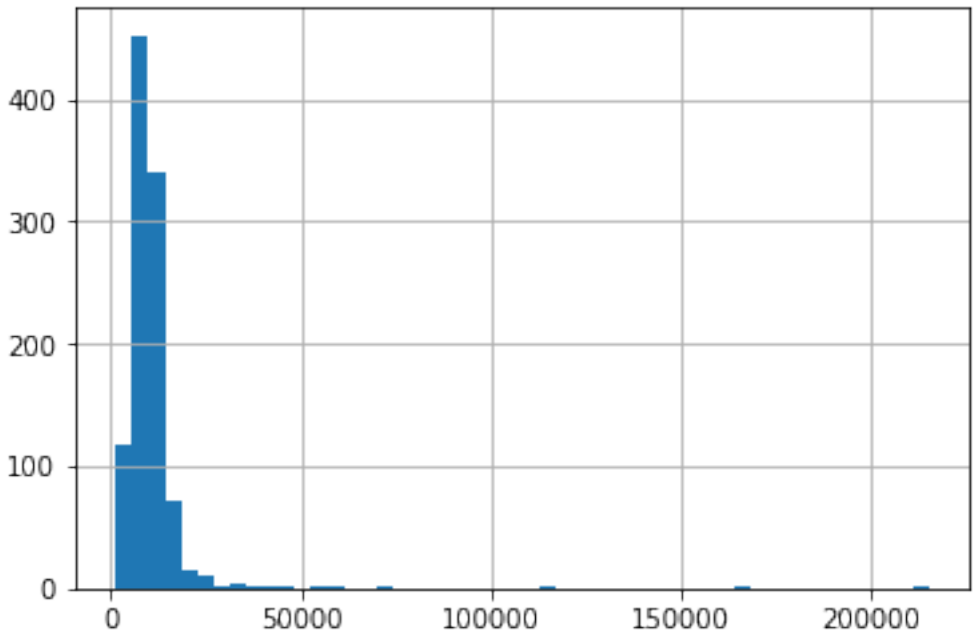
```
# set up the variable transformer
tf = vt.YeoJohnsonTransformer(variables␣
↪= ['LotArea', 'GrLivArea'])

# fit the transformer
tf.fit(X_train)
```

With `fit()`, the *YeoJohnsonTransformer()* learns the optimal lambda for the transformation. Now we can go ahead and trasnform the data:

```
# transform the data
train_t= tf.transform(X_train)
test_t= tf.transform(X_test)
```

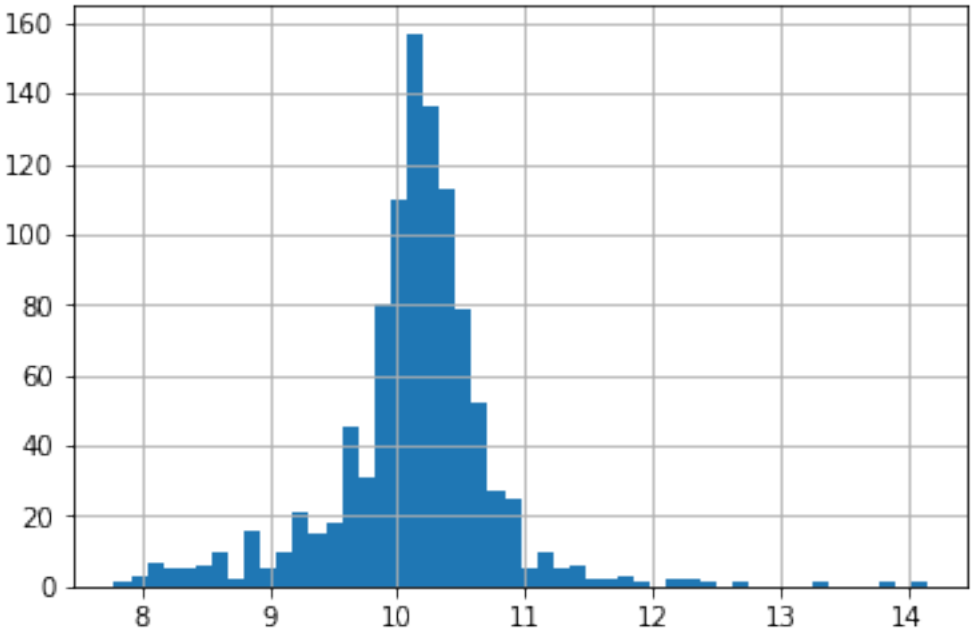
Next, we make a histogram of the original variable distribution:

```
# un-transformed variable
X_train['LotArea'].hist(bins=50)
```



And now, we can explore the distribution of the variable after the transformation:

```
# transformed variable
train_t['LotArea'].hist(bins=50)
```



Additional resources

You can find more details about the `YeoJohnsonTransformer()` here:

- Jupyter notebook

For more details about this and other feature engineering methods check out these resources:

Or read our book:

Fig. 58: Feature Engineering for Machine Learning

Both our book and course are suitable for beginners and more advanced data scientists alike. By purchasing them you are supporting Sole, the main developer of Feature-engine.

10.2.2 Creation

Feature Creation

Feature creation, is a common step during data preprocessing, and consists of constructing new variables from the dataset's original features. By combining two or more variables, we develop new features that can improve the performance of a machine learning model, capture additional information or relationships among variables, or simply make more sense within the domain we are working on.

One of the most common feature creation methods in data science is one-hot encoding, which is a feature engineering technique used to transform a categorical feature into multiple binary variables that represent each category.

Another common feature extraction procedure consist of creating new features from past values of time series data, for example through the use of lags and windows.

In general, creating features requires a dose of domain knowledge and significant time invested in analyzing the raw data, including evaluating the relationship between the independent or predictor variables and the dependent or target variable in the dataset.

Feature creation can be one of the more creative aspects of feature engineering, and the new features can help improve a predictive model's performance.

Lastly, a data scientist should be mindful that creating new features may increase the dimensionality of the dataset quite dramatically. For example, one hot encoding of highly cardinal categorical features results in lots of binary variables, and so does polynomial combinations of high powers. This may have downstream effects depending on the machine learning algorithm being used. For example, decision trees are known for not being able to cope with huge number of features.

Creating New Features with Feature-engine

Feature-engine has several transformers that create and add new features to the dataset. One of the most popular ones is the OneHotEncoder that creates dummy variables from categorical features.

With Feature-engine we can also create new features from time series data through lags and windows by using LagFeatures or WindowFeatures.

Feature-engine's creation module, supports transformers that create and add new features to a pandas dataframe by either combining existing features through different mathematical or statistical operations, or through feature transformations. These transformers operate with numerical variables, that is, those with integer and float data types.

Summary of Feature-engine's feature-creation transformers:

- **CyclicalFeatures** - Creates two new features per variable by applying the trigonometric operations sine and cosine to the original feature.
- **MathFeatures** - Combines a set of features into new variables by applying basic mathematical functions like the sum, mean, maximum or standard deviation.
- **RelativeFeatures** - Utilizes basic mathematical functions between a group of variables and one or more reference features, appending the new features to the pandas dataframe.

Feature creation module

CyclicalFeatures

Some features are inherently cyclical. Clear examples are **time features**, i.e., those features derived from datetime variables like the *hours of the day*, the *days of the week*, or the *months of the year*.

But that's not the end of it. Many variables related to natural processes are also cyclical, like, for example, *tides*, *moon cycles*, or *solar energy generation* (which coincides with light periods, which are cyclical).

In cyclical features, higher values of the variable are closer to lower values. For example, December (12) is closer to January (1) than to June (6).

How can we convey to machine learning models like linear regression the cyclical nature of the features?

In the article "Advanced machine learning techniques for building performance simulation," the authors engineered cyclical variables by representing them as (x,y) coordinates on a circle. The idea was that, after preprocessing the cyclical data, the lowest value of every cyclical feature would appear right next to the largest value.

To represent cyclical features in (x, y) coordinates, the authors created two new features, deriving the sine and cosine components of the cyclical variable. We can call this procedure **"cyclical encoding."**

Cyclical encoding

The trigonometric functions sine and cosine are periodic and repeat their values every 2 pi radians. Thus, to transform cyclical variables into (x, y) coordinates using these functions, first we need to normalize them to 2 pi radians.

We achieve this by dividing the variables' values by their maximum value. Thus, the two new features are derived as follows:

- var_sin = sin(variable * (2. * pi / max_value))
- var_cos = cos(variable * (2. * pi / max_value))

In Python, we can encode cyclical features by using the Numpy functions `sin` and `cos`:

```
import numpy as np

X[f"{variable}_sin"] = np.
↪sin(X["variable"] * (2.0 * np.pi / X["variable"]).max())
X[f"{variable}_cos"] = np.
↪cos(X["variable"] * (2.0 * np.pi / X["variable"]).max())
```

We can also use Feature-Engine to automate this process.

Cyclical encoding with Feature-engine

CyclicalFeatures() creates two new features from numerical variables to better capture the cyclical nature of the original variable. *CyclicalFeatures()* returns two new features per variable, according to:

- var_sin = sin(variable * (2. * pi / max_value))
- var_cos = cos(variable * (2. * pi / max_value))

where max_value is the maximum value in the variable, and pi is 3.14…

Example

In this example, we obtain cyclical features from the variables *days of the week* and *months*. We first create a toy dataframe with the variables "days" and "months":

```
import pandas as pd
from feature_engine.creation import CyclicalFeatures

df = pd.DataFrame({
    'day': [6, 7, 5, 3, 1, 2, 4],
    'months': [3, 7, 9, 12, 4, 6, 12],
    })
```

Now we set up the transformer to find the maximum value of each variable automatically:

```
cyclical␣
↪= CyclicalFeatures(variables=None, drop_original=False)

X = cyclical.fit_transform(df)
```

The maximum values used for the transformation are stored in the attribute `max_values_`:

```
print(cyclical.max_values_)
```

```
{'day': 7, 'months': 12}
```

Let's have a look at the transformed dataframe:

```
print(X.head())
```

We can see that the new variables were added at the right of our dataframe.

```
   day  months␣
↪      day_sin   day_cos    months_sin    months_cos
0    6   ␣
↪    3 -7.818315e-01  0.623490  1.000000e+00  6.123234e-17
1    7   ␣
↪    7 -2.449294e-16  1.000000 -5.000000e-01 -8.660254e-01
2    5   ␣
↪    9 -9.749279e-01 -0.222521 -1.000000e+00 -1.836970e-16
3    3   ␣
↪   12  4.338837e-01 -0.900969 -2.449294e-16  1.000000e+00
4    1   ␣
↪    4  7.818315e-01  0.623490  8.660254e-01 -5.000000e-01
```

We set the parameter `drop_original` to `False`, which means that we keep the original variables. If we want them dropped after the feature creation, we can set the parameter to `True`.

We can now use the new features, which convey the cyclical nature of the data, to train machine learning algorithms, like linear or logistic regression, among others.

Finally, we can obtain the names of the variables of the transformed dataset as follows:

```
cyclical.get_feature_names_out()
```

This returns the name of all the variables in the final output, original and and new:

```
['day', 'months
↪', 'day_sin', 'day_cos', 'months_sin', 'months_cos']
```

Cyclical feature visualization

We now know how to convert cyclical variables into (x, y) coordinates of a circle by using the sine and cosine functions. Let's now carry out some visualizations to better understand the effect of this transformation.

Let's create a toy dataframe:

```
import pandas as pd
import matplotlib.pyplot as plt

df␣
↪= pd.DataFrame([i for i in range(24)], columns=['hour'])
```

Our dataframe looks like this:

```
df.head()

   hour
0     0
1     1
2     2
3     3
4     4
```

Let's now compute the sine and cosine features:

```
cyclical = CyclicalFeatures(variables=None)

df = cyclical.fit_transform(df)

print(df.head())
```

These are the sine and cosine features that represent the hour:

```
   hour  hour_sin  hour_cos
0     0  0.000000  1.000000
1     1  0.269797  0.962917
2     2  0.519584  0.854419
3     3  0.730836  0.682553
4     4  0.887885  0.460065
```

Let's now plot the hour variable against its sine transformation. We add perpendicular lines to flag the hours 0 and 22.

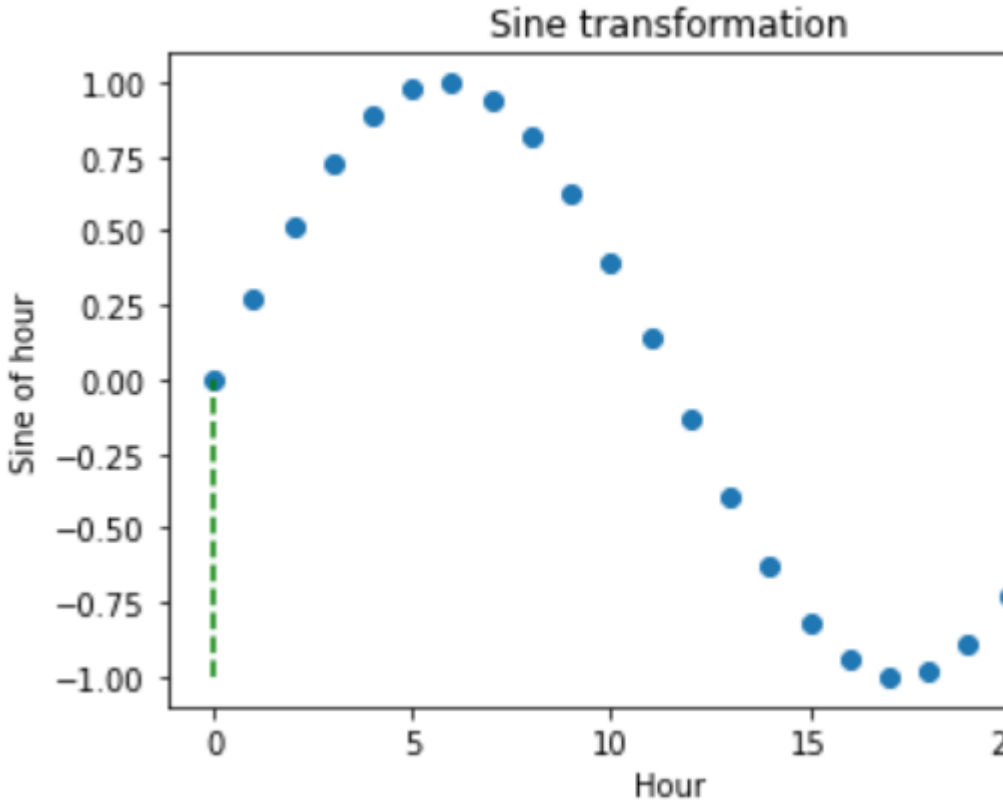
```
plt.scatter(df["hour"], df["hour_sin"])

# Axis labels
plt.ylabel('Sine of hour')
plt.xlabel('Hour')
plt.title('Sine transformation')

plt.vlines(x=0,
↪ ymin=-1, ymax=0, color='g', linestyles='dashed')
```

(continues on next page)

(continued from previous page)

```
plt.vlines(x=22,
↪ ymin=-1, ymax=-0.25, color='g', linestyles='dashed')
```

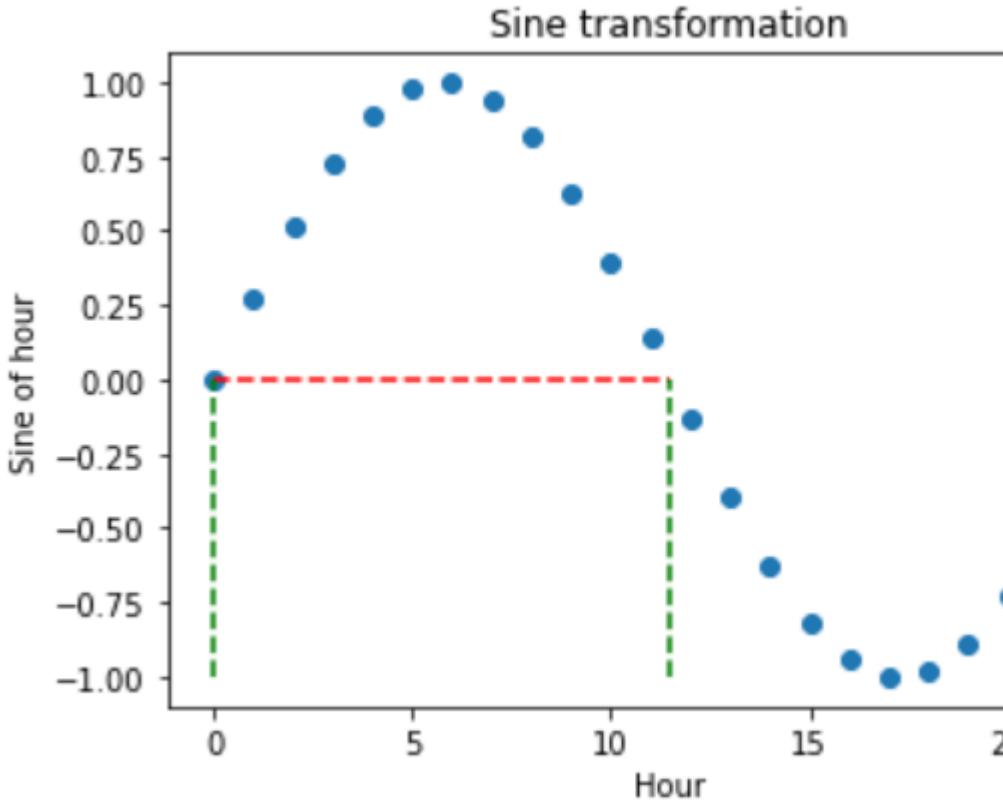
After the transformation using the sine function, we see that the new values for the hours 0 and 22 are closer to each other (follow the dashed lines), which was the expectation:



The problem with trigonometric transformations, is that, because they are periodic, 2 different observations can also return similar values after the transformation. Let's explore that:

```
plt.scatter(df["hour"], df["hour_sin"])

# Axis labels
plt.ylabel('Sine of hour')
plt.xlabel('Hour')
plt.title('Sine transformation')

plt.hlines(y=0,
↪ xmin=0, xmax=11.5, color='r', linestyles='dashed')

plt.vlines(x=0,
↪ ymin=-1, ymax=0, color='g', linestyles='dashed')
plt.vlines(x=11.
↪5, ymin=-1, ymax=0, color='g', linestyles='dashed')
```

In the plot below, we see that the hours 0 and 11.5 obtain very similar values after the sine transformation. So how can we differentiate them?



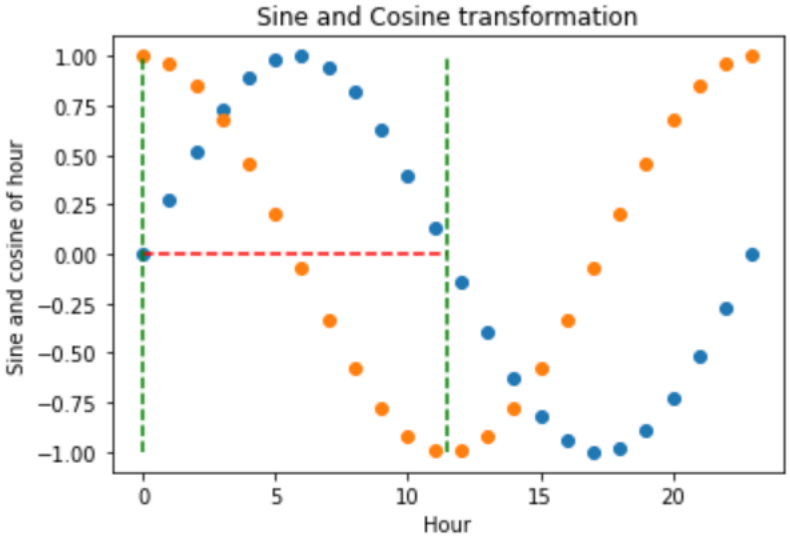
To fully code the information of the hour, we must use the sine and cosine trigonometric transformations together. Adding the cosine function, which is out of phase with the sine function, breaks the symmetry and assigns a unique codification to each hour.

Let's explore that:

```
plt.scatter(df["hour"], df["hour_sin"])
plt.scatter(df["hour"], df["hour_cos"])

# Axis labels
plt.ylabel('Sine and cosine of hour')
plt.xlabel('Hour')
plt.title('Sine and Cosine transformation')


plt.hlines(y=0,
↪ xmin=0, xmax=11.5, color='r', linestyles='dashed')

plt.vlines(x=0,
↪ ymin=-1, ymax=1, color='g', linestyles='dashed')
plt.vlines(x=11.
↪5, ymin=-1, ymax=1, color='g', linestyles='dashed')
```

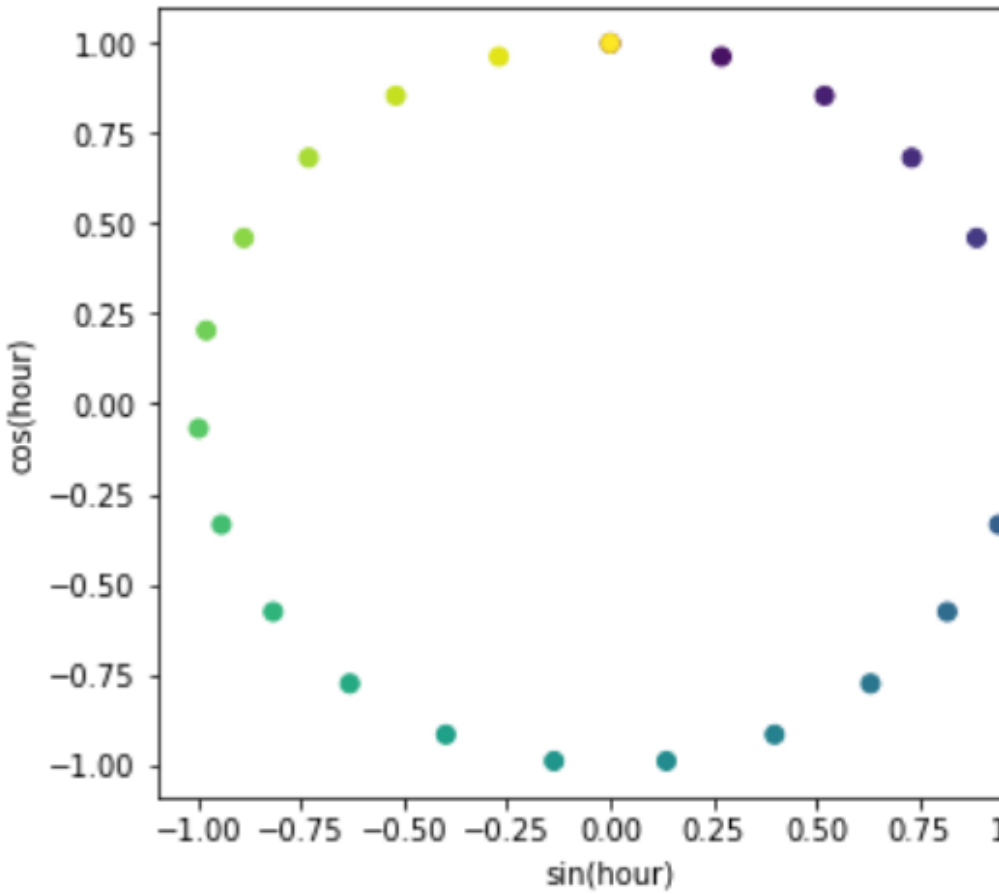
The hour 0, after the transformation, takes the values of sine 0 and cosine 1,

which makes it different from the hour 11.5, which takes the values of sine 0 and cosine -1. In other words, with the two functions together, we are able to distinguish all observations within our original variable.



Finally, let's vizualise the (x, y) circle coordinates generated by the sine and cosine features.

```
fig, ax = plt.subplots(figsize=(7, 5))
sp =␣
↪ax.scatter(df["hour_sin"], df["hour_cos"], c=df["hour"])
ax.set(
    xlabel="sin(hour)",
    ylabel="cos(hour)",
)
_ = fig.colorbar(sp)
```



That's it, you now know how to represent cyclical data through the use of trigonometric functions and cyclical encoding.

Additional resources

For tutorials on how to create cyclical features, check out the following courses:

Fig. 60: Feature Engineering for Machine Learning

For a comparison between one-hot encoding, ordinal encoding, cyclical encoding and spline encoding of cyclical features check out the following sklearn demo.

Check also these Kaggle demo on the use of cyclical encoding with neural networks:

- Encoding Cyclical Features for Deep Learning.

Fig. 61: Feature Engineering for Time Series Forecasting

MathFeatures

MathFeatures() applies basic functions to groups of features, returning one or more additional variables as a result. It uses `pandas.agg()` to create the features, so in essence, you can pass any function that is accepted by this method. One exception is that *MathFeatures()* does not accept dictionaries for the parameter `func`.

The functions can be passed as strings, numpy methods, i.e., np.mean, or any function that you create, as long as, it returns a scalar from a vector.

For supported aggregation functions, see pandas documentation.

As an example, if we have the variables:

- **number_payments_first_quarter**
- **number_payments_second_quarter**
- **number_payments_third_quarter**
- **number_payments_fourth_quarter**

we can use *MathFeatures()* to calculate the total number of payments and mean number of payments as follows:

```
transformer = MathFeatures(
    variables=[
        'number_payments_first_quarter',
        'number_payments_second_quarter',
        'number_payments_third_quarter',
        'number_payments_fourth_quarter'
    ],
    func=['sum','mean'],
    new_variables_name=[
        'total_number_payments',
```

(continues on next page)

(continued from previous page)

```
        'mean_number_payments'
    ]
)

Xt = transformer.fit_transform(X)
```

The transformed dataset, Xt, will contain the additional features **total_number_payments** and **mean_number_payments**, plus the original set of variables.

The variable **total_number_payments** is obtained by adding up the features indicated in `variables`, whereas the variable **mean_number_payments** is the mean of those 4 features.

Examples

Let's dive into how we can use *MathFeatures()* in more details. Let's first create a toy dataset:

```
import numpy as np
import pandas as pd
from feature_engine.creation import MathFeatures

df = pd.DataFrame.from_dict(
    {
        "Name": ["tom", "nick", "krish", "jack"],
        "City": ["London", "Manchester", "Liverpool", "Bristol"],
        "Age": [20, 21, 19, 18],
        "Marks": [0.9, 0.8, 0.7, 0.6],
        "dob": pd.date_range("2020-02-24", periods=4, freq="T"),
    })

print(df)
```

The dataset looks like this:

```
    Name        City  Age  Marks                 dob
0    tom      London   20    0.9 2020-02-24 00:00:00
1   nick  Manchester   21    0.8 2020-02-24 00:01:00
2  krish   Liverpool   19    0.7 2020-02-24 00:02:00
3   jack     Bristol   18    0.6 2020-02-24 00:03:00
```

We can now apply several functions over the numerical variables Age and Marks using strings to indicate the functions:

```
transformer = MathFeatures(
    variables=["Age", "Marks"],
    func = ["sum", "prod", "min", "max", "std"],
)

df_t = transformer.fit_transform(df)

print(df_t)
```

And we obtain the following dataset, where the new variables are named after the function used to obtain them, plus the group of variables that were used in the computation:

```
    Name        City  Age  Marks                 dob  sum_Age_Marks  \
0    tom      London   20    0.9 2020-02-24 00:00:00           20.9
1   nick  Manchester   21    0.8 2020-02-24 00:01:00           21.8
2  krish   Liverpool   19    0.7 2020-02-24 00:02:00           19.7
3   jack     Bristol   18    0.6 2020-02-24 00:03:00           18.6

   prod_Age_Marks  min_Age_Marks  max_Age_Marks  std_Age_Marks
0            18.0            0.9           20.0      13.505740
1            16.8            0.8           21.0      14.283557
2            13.3            0.7           19.0      12.940054
3            10.8            0.6           18.0      12.303658
```

For more flexibility, we can pass existing functions to the `func` argument as follows:

```
transformer = MathFeatures(
    variables=["Age", "Marks"],
    func = [np.sum, np.prod, np.min, np.max, np.std],
)

df_t = transformer.fit_transform(df)

print(df_t)
```

And we obtain the following dataframe:

```
    Name        City  Age  Marks                 dob  sum_Age_Marks  \
0    tom      London   20    0.9 2020-02-24 00:00:00           20.9
1   nick  Manchester   21    0.8 2020-02-24 00:01:00           21.8
2  krish   Liverpool   19    0.7 2020-02-24 00:02:00           19.7
3   jack     Bristol   18    0.6 2020-02-24 00:03:00           18.6

   prod_Age_Marks  amin_Age_Marks  amax_Age_Marks  std_Age_Marks
0            18.0             0.9            20.0      13.505740
1            16.8             0.8            21.0      14.283557
2            13.3             0.7            19.0      12.940054
3            10.8             0.6            18.0      12.303658
```

We have the option to set the parameter `drop_original` to True to drop the variables after performing the calculations.

We can obtain the names of all the features in the transformed data as follows:

```
transformer.get_feature_names_out(input_features=None)
```

Which will return the names of all the variables in the transformed data:

```
['Name',
 'City',
 'Age',
 'Marks',
 'dob',
 'sum_Age_Marks',
 'prod_Age_Marks',
 'amin_Age_Marks',
```

(continues on next page)

(continued from previous page)

```
 'amax_Age_Marks',
 'std_Age_Marks']
```

New variables names

Even though the transfomer allows to combine variables automatically, its use is intended to combine variables with domain knowledge. In this case, we normally want to give meaningful names to the variables. We can do so through the parameter `new_variables_names`.

`new_variables_names` takes a list of strings, with the new variable names. In this parameter, you need to enter a list of names for the newly created features. You must enter one name for each function indicated in the `func` parameter. That is, if you want to perform mean and sum of features, you should enter 2 new variable names. If you compute only the mean of features, enter 1 variable name.

The name of the variables should coincide with the order of the functions in `func`. That is, if you set `func = ['mean', 'prod']`, the first new variable name will be assigned to the mean of the variables and the second variable name to the product of the variables.

Let's look at an example. In the following code snippet, we add up, and find the maximum and minimum value of 2 variables, which results in 3 new features. We add the names for the new features in a list:

```
transformer = MathFeatures(
    variables=["Age", "Marks"],
    func = ["sum", "min", "max"],
    new_variables_names = ["sum_vars", "min_vars", "max_vars"]
)

df_t = transformer.fit_transform(df)

print(df_t)
```

The resulting dataframe contains the new features under the variable names that we provided:

```
    Name        City  Age  Marks                 dob  sum_vars  min_vars  \
0    tom      London   20    0.9 2020-02-24 00:00:00      20.9       0.9
1   nick  Manchester   21    0.8 2020-02-24 00:01:00      21.8       0.8
2  krish   Liverpool   19    0.7 2020-02-24 00:02:00      19.7       0.7
3   jack     Bristol   18    0.6 2020-02-24 00:03:00      18.6       0.6

   max_vars
0      20.0
1      21.0
2      19.0
3      18.0
```

Additional resources

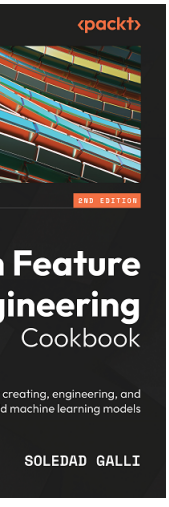
For more details about this and other feature engineering methods check out these resources:

Or read our book:

Fig. 62: Feature Engineering for Machine Learning

Both our book and course are suitable for beginners and more advanced data scientists alike. By purchasing them you are supporting Sole, the main developer of Feature-engine.

RelativeFeatures



RelativeFeatures() applies basic mathematical operations between a group of variables and one or more reference features, adding the resulting features to the dataframe.

RelativeFeatures() uses the pandas methods `pd.DataFrame.add()`, `pd.DataFrame.sub()`, `pd.DataFrame.mul()`, `pd.DataFrame.div()`, `pd.DataFrame.truediv()`, `pd.DataFrame.floordiv()`, `pd.DataFrame.mod()` and `pd.DataFrame.pow()` to transform a group of variables by a group of reference variables.

For example, if we have the variables:

- **number_payments_first_quarter**
- **number_payments_second_quarter**
- **number_payments_third_quarter**
- **number_payments_fourth_quarter**
- **total_payments**,

we can use *RelativeFeatures()* to determine the percentage of payments per quarter as follows:

```
transformer = RelativeFeatures(
    variables=[
        'number_payments_first_quarter',
        'number_payments_second_quarter',
        'number_payments_third_quarter',
        'number_payments_fourth_quarter',
    ],
    reference=['total_payments'],
    func=['div'],
)

Xt = transformer.fit_transform(X)
```

The precedent code block will return a new dataframe, Xt, with 4 new variables that are calculated as the division of each one of the variables in `variables` and 'total_payments'.

Examples

Let's dive into how we can use *RelativeFeatures()* in more details. Let's first create a toy dataset:

```
import pandas as pd
from feature_engine.creation import RelativeFeatures

df = pd.DataFrame.from_dict(
    {
        "Name": ["tom", "nick", "krish", "jack"],
        "City
↪": ["London", "Manchester", "Liverpool", "Bristol"],
        "Age": [20, 21, 19, 18],
        "Marks": [0.9, 0.8, 0.7, 0.6],

↪"dob": pd.date_range("2020-02-24", periods=4, freq="T"),
    })

print(df)
```

The dataset looks like this:

```
    Name        City  Age  Marks                 dob
0    tom      London   20    0.9 2020-02-24 00:00:00
1   nick  Manchester   21    0.8 2020-02-24 00:01:00
2  krish   Liverpool   19    0.7 2020-02-24 00:02:00
3   jack     Bristol   18    0.6 2020-02-24 00:03:00
```

We can now apply several functions between the numerical variables Age and Marks and Age as follows:

```
transformer = RelativeFeatures(
    variables=["Age", "Marks"],
    reference=["Age"],
    func = ["sub", "div", "mod", "pow"],
)

df_t = transformer.fit_transform(df)

print(df_t)
```

And we obtain the following dataset, where the new variables are named after the variables that were used for the calculation and the function in the middle of their names. Thus, `Mark_sub_Age` means `Mark - Age`, and `Marks_mod_Age` means `Mark % Age`.

```
    Name
↪    City  Age  Marks                 dob  Age_sub_Age  \
0    tom
↪     London   20    0.9 2020-02-24 00:00:00            0
1   nick
↪ Manchester   21    0.8 2020-02-24 00:01:00            0
2  krish
↪  Liverpool   19    0.7 2020-02-24 00:02:00            0
3   jack
↪    Bristol   18    0.6 2020-02-24 00:03:00            0

   Marks_sub_Age  Age_
↪div_Age  Marks_div_Age  Age_mod_Age  Marks_mod_Age  \
0          -19.1
↪          1.0       0.045000            0            0.9
1          -20.2
↪          1.0       0.038095            0            0.8
2          -18.3
↪          1.0       0.036842            0            0.7
3          -17.4
↪          1.0       0.033333            0            0.6

           Age_pow_Age  Marks_pow_Age
0 -2101438300051996672       0.121577
1 -1595931050845505211       0.009223
2  6353754964178307979       0.001140
3  -497033925936021504       0.000102
```

We can obtain the names of all the features in the transformed data as follows:

```
transformer.get_feature_names_out(input_features=None)
```

Which will return the names of all the variables in the transformed data:

```
['Name',
 'City',
 'Age',
 'Marks',
 'dob',
 'Age_sub_Age',
 'Marks_sub_Age',
 'Age_div_Age',
 'Marks_div_Age',
 'Age_mod_Age',
 'Marks_mod_Age',
 'Age_pow_Age',
 'Marks_pow_Age']
```

Additional resources

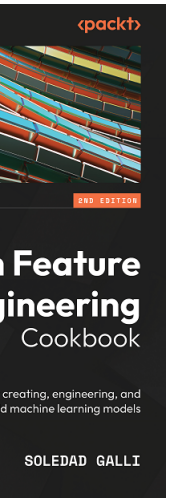
For more details about this and other feature engineering methods check out these resources:

Or read our book:

Fig. 64: Feature Engineering for Machine Learning

Both our book and course are suitable for beginners and more advanced data scientists alike. By purchasing them you are supporting Sole, the main developer of Feature-engine.

Feature-engine in Practice



ure Engineering

Here, you'll get a taste of the transformers from the feature creation module from Feature-engine. We'll use the wine quality dataset. The dataset is comprised of 11 features, including `alcohol`, `ash`, and `flavonoids`, and has `quality` as its target variable.

Through exploratory data analysis and our domain knowledge which includes real-world experimentation, i.e., drinking various brands/types of wine, we believe that we can create better features to train our algorithm by combining original features with various mathematical operations.

Let's load the dataset from Scikit-learn.

```
import pandas as pd
from sklearn.datasets import load_wine
from feature_
↪engine.creation import RelativeFeatures, MathFeatures

X, y = load_wine(return_X_y=True, as_frame=True)
print(X.head())
```

Below we see the wine quality dataset:

```
   alcohol  malic_acid␣
↪  ash  alcalinity_of_ash  magnesium  total_phenols  \
0    14.23      ␣
↪ 1.71  2.43               15.6      127.0          2.80
1    13.20      ␣
↪ 1.78  2.14               11.2      100.0          2.65
2    13.16      ␣
↪ 2.36  2.67               18.6      101.0          2.80
3    14.37      ␣
↪ 1.95  2.50               16.8      113.0          3.85
4    13.24      ␣
↪ 2.59  2.87               21.0      118.0          2.80

   flavanoids  nonflavanoid_
↪phenols  proanthocyanins  color_intensity   hue  \
0        3.06     ␣
↪            0.28             2.29             5.64  1.04
1        2.76     ␣
↪            0.26             1.28             4.38  1.05
2        3.24     ␣
```

(continues on next page)

(continued from previous page)

```
↪            0.30             2.81             5.68  1.03
3        3.49     ␣
↪            0.24             2.18             7.80  0.86
4        2.69     ␣
↪            0.39             1.82             4.32  1.04

   od280/od315_of_diluted_wines  proline
0                          3.92   1065.0
1                          3.40   1050.0
2                          3.17   1185.0
3                          3.45   1480.0
4                          2.93    735.0
```

Now, we create a new feature by removing non-flavonoid phenols from the total phenols to obtain the phenols that are not flavonoid.

```
rf = RelativeFeatures(
    variables=["total_phenols"],
    reference=["nonflavanoid_phenols"],
    func=["sub"],
)

rf.fit(X)
X_tr = rf.transform(X)

print(X_tr.head())
```

We see the new feature and its data points at the right of the pandas dataframe:

```
   alcohol  malic_acid␣
↪  ash  alcalinity_of_ash  magnesium  total_phenols  \
0    14.23      ␣
↪ 1.71  2.43               15.6      127.0          2.80
1    13.20      ␣
↪ 1.78  2.14               11.2      100.0          2.65
2    13.16      ␣
↪ 2.36  2.67               18.6      101.0          2.80
3    14.37      ␣
↪ 1.95  2.50               16.8      113.0          3.85
4    13.24      ␣
↪ 2.59  2.87               21.0      118.0          2.80

   flavanoids  nonflavanoid_
↪phenols  proanthocyanins  color_intensity   hue  \
0        3.06     ␣
↪            0.28             2.29             5.64  1.04
1        2.76     ␣
↪            0.26             1.28             4.38  1.05
2        3.24     ␣
↪            0.30             2.81             5.68  1.03
3        3.49     ␣
↪            0.24             2.18             7.80  0.86
```

(continues on next page)

(continued from previous page)

```
4        2.69      ␣
↪            0.39             1.82             4.32  1.04

   od280/od315_of_diluted_wines  proline  \
0                          3.92   1065.0
1                          3.40   1050.0
2                          3.17   1185.0
3                          3.45   1480.0
4                          2.93    735.0

   total_phenols_sub_nonflavanoid_phenols
0                                    2.52
1                                    2.39
2                                    2.50
3                                    3.61
4                                    2.41
```

Let's now create new features by combining a subset of 3 existing variables:

```
mf = MathFeatures(
  ␣
↪ variables=["flavanoids", "proanthocyanins", "proline"],
    func=["sum", "mean"],
)

mf.fit(X_tr)
X_tr = mf.transform(X_tr)

print(X_tr.head())
```

We see the new features at the right of the resulting pandas dataframe:

```
   alcohol  malic_acid␣
↪  ash  alcalinity_of_ash  magnesium  total_phenols  \
0    14.23      ␣
↪ 1.71  2.43               15.6      127.0           2.80
1    13.20      ␣
↪ 1.78  2.14               11.2      100.0           2.65
2    13.16      ␣
↪ 2.36  2.67               18.6      101.0           2.80
3    14.37      ␣
↪ 1.95  2.50               16.8      113.0           3.85
4    13.24      ␣
↪ 2.59  2.87               21.0      118.0           2.80

   flavanoids  nonflavanoid_
↪phenols  proanthocyanins  color_intensity   hue  \
0        3.06      ␣
↪            0.28             2.29             5.64  1.04
1        2.76      ␣
↪            0.26             1.28             4.38  1.05
2        3.24      ␣
```

(continues on next page)

(continued from previous page)

```
↪            0.30             2.81             5.68  1.03
3        3.49     ␣
↪            0.24             2.18             7.80  0.86
4        2.69     ␣
↪            0.39             1.82             4.32  1.04

   od280/od315_of_diluted_wines  proline  \
0                          3.92   1065.0
1                          3.40   1050.0
2                          3.17   1185.0
3                          3.45   1480.0
4                          2.93    735.0

   total_phenols_sub_nonflavanoid_phenols  \
0                                    2.52
1                                    2.39
2                                    2.50
3                                    3.61
4                                    2.41

   sum_flavanoids_proanthocyanins_proline  \
0                                 1070.35
1                                 1054.04
2                                 1191.05
3                                 1485.67
4                                  739.51

   mean_flavanoids_proanthocyanins_proline
0                                356.783333
1                                351.346667
2                                397.016667
3                                495.223333
4                                246.503333
```

In the above examples, we used `RelativeFeature()` and `MathFeatures` to perform automated feature engineering on the input data by applying the transformations defined in the `func` parameter on the features identified in `variables` and `reference` parameters.

The original and new features can now be used to train a regression model, or a multiclass classification algorithm, to predict the `quality` of the wine.

Summary

Through feature engineering and feature creation, we can optimize the machine learning algorithm's learning process and improve its performance metrics.

We'd strongly recommend the creation of features based on domain knowledge, exploratory data analysis and thorough data mining. We also understand that this is not always possible, particularly with big datasets and limited time allocated to each project. In this situation, we can combine the creation of features with feature selection procedures to let machine learning algorithms select what works best for them.

Good luck with your models!

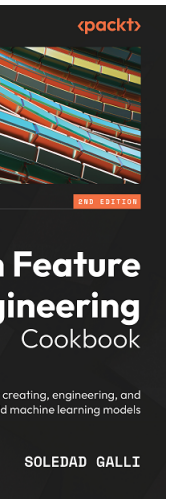
Tutorials, books and courses

For tutorials about this and other feature engineering for machine learning methods check out our online course:

Or read our book:

Fig. 66: Feature Engineering for Machine Learning

Both our book and course are suitable for beginners and more advanced data scientists alike. By purchasing them you are supporting Sole, the main developer of Feature-engine.



ure Engineering

Transformers in other Libraries

Check also the following transformer from Scikit-learn:

- PolynomialFeatures
- SplineTransformer

Datetime Features

Feature-engine's datetime transformers are able to extract a wide variety of datetime features from existing datetime or object-like data.

DatetimeFeatures

In datasets commonly used in data science and machine learning projects, the variables very often contain information about date and time. **Date of birth** and **time of purchase** are two examples of these variables. They are commonly referred to as "datetime features", that is, data whose data type is date and time.

We don't normally use datetime variables in their raw format to train machine learning models, like those for regression, classification, or clustering. Instead, we can extract a lot of information from these variables by extracting the different date and time components of the datetime variable.

Examples of date and time components are the year, the month, the week_of_year, the day of the week, the hour, the minutes, and the seconds.

Datetime features with pandas

In Python, we can extract date and time components through the `dt` module of the open-source library pandas. For example, by executing the following:

```
data = pd.DataFrame({"date
↪": pd.date_range("2019-03-05", periods=20, freq="D")})

data["year"] = data["date"].dt.year
data["quarter"] = data["date"].dt.quarter
data["month"] = data["date"].dt.month
```

In the former code block we created 3 features from the timestamp variable: the *year*, the *quarter* and the *month*.

Datetime features with Feature-engine

DatetimeFeatures() automatically extracts several date and time features from datetime variables. It works with variables whose dtype is datetime, as well as with object-like and categorical variables, provided that they can be parsed into datetime format. It *cannot* extract features from numerical variables.

DatetimeFeatures() uses the pandas `dt` module under the hood, therefore automating datetime feature engineering. In two lines of code and by specifying which features we want to create with *DatetimeFeatures()*, we can create multiple date and time variables from various variables simultaneously.

DatetimeFeatures() can automatically create all features supported by pandas `dt` and a few more, like, for example, a binary feature indicating if the event occurred on a weekend and also the semester.

With *DatetimeFeatures()* we can choose which date and time features to extract from the datetime variables. We can also extract date and time features from one or more datetime variables at the same time.

Through the following examples we highlight the functionality and versatility of *DatetimeFeatures()* for tabular data.

Extract date features

In this example, we are going to extract three **date features** from a specific variable in the dataframe. In particular, we are interested in the *month*, the *day of the year*, and whether that day was the *last day the month*.

First, we will create a toy dataframe with 2 date variables:

```
import pandas as pd
from feature_engine.datetime import DatetimeFeatures

toy_df = pd.DataFrame({
    "var_date1
↪": ['May-1989', 'Dec-2020', 'Jan-1999', 'Feb-2002'],
    "var_date2": [
↪'06/21/2012', '02/10/1998', '08/03/2010', '10/31/2020'],
})
```

Now, we will extract the variables month, month-end and the day of the year from the second datetime variable in our dataset.

```
dtfs = DatetimeFeatures(
    variables="var_date2",
    features_
↪to_extract=["month", "month_end", "day_of_year"]
)

df_transf = dtfs.fit_transform(toy_df)

df_transf
```

With `transform()`, the features extracted from the datetime variable are added to the dataframe.

We see the new features in the following output:

```
  var_date1  var_
↪date2_month  var_date2_month_end  var_date2_day_of_year
0  May-1989    ␣
↪           6                    0                     173
1  Dec-2020    ␣
↪           2                    0                      41
2  Jan-1999    ␣
↪           8                    0                     215
3  Feb-2002    ␣
↪          10                    1                     305
```

By default, *DatetimeFeatures()* drops the variable from which the date and time features were extracted, in this case, *var_date2*. To keep the variable, we just need to indicate `drop_original=False` when initializing the transformer.

Finally, we can obtain the name of the variables in the returned data as follows:

```
dtfs.get_feature_names_out()
```

```
['var_date1',
 'var_date2_month',
 'var_date2_month_end',
 'var_date2_day_of_year']
```

Extract time features

In this example, we are going to extract the feature *minute* from the two time variables in our dataset.

First, let's create a toy dataset with 2 time variables and an object variable.

```
import pandas as pd
from feature_engine.datetime import DatetimeFeatures

toy_df = pd.DataFrame({
    "not_a_dt": ['not', 'a', 'date', 'time'],
    "var_time1
↪": ['12:34:45', '23:01:02', '11:59:21', '08:44:23'],
    "var_time2
↪": ['02:27:26', '10:10:55', '17:30:00', '18:11:18'],
})
```

DatetimeFeatures() automatically finds all variables that can be parsed to datetime. So if we want to extract time features from all our datetime variables, we don't need to specify them.

Note that from version 2.0.0 pandas deprecated the parameter `infer_datetime_format`. Hence, if you want pandas to infer the

datetime format and you have different formats, you need to explicitly say so by passing "mixed" to the format parameter as shown below.

```
dfts = DatetimeFeatures(features_
↪to_extract=["minute"], format="mixed")

df_transf = dfts.fit_transform(toy_df)

df_transf
```

We see the new features in the following output:

```
  not_a_dt  var_time1_minute  var_time2_minute
0      not                34                27
1        a                 1                10
2     date                59                30
3     time                44                11
```

The transformer found two variables in the dataframe that can be cast to datetime and proceeded to extract the requested feature from them.

The variables detected as datetime are stored in the transformer's variables_ attribute:

```
dfts.variables_
```

```
['var_time1', 'var_time2']
```

The original datetime variables are dropped from the data by default. This leaves the dataset ready to train machine learning algorithms like linear regression or random forests.

If we want to keep the datetime variables, we just need to indicate drop_original=False when initializing the transformer.

Finally, if we want to obtain the names of the variables in the output data, we can use:

```
dfts.get_feature_names_out()
```

```
['not_a_dt', 'var_time1_minute', 'var_time2_minute']
```

Extract date and time features

In this example, we will combine what we have seen in the previous two examples and extract a date feature - *year* - and time feature - *hour* - from two variables that contain both date and time information.

Let's go ahead and create a toy dataset with 3 datetime variables.

```
import pandas as pd
from feature_engine.datetime import DatetimeFeatures

toy_df = pd.DataFrame({
```

(continues on next page)

(continued from previous page)

```
    "var_
↪dt1": pd.date_range("2018-01-01", periods=3, freq="H"),
    "var_dt2": ['08/31/
↪00 12:34:45', '12/01/90 23:01:02', '04/25/01 11:59:21'],
    "var_dt3": ['03/02/
↪15 02:27:26', '02/28/97 10:10:55', '11/11/03 17:30:00'],
})
```

Now, we set up the *DatetimeFeatures()* to extract features from 2 of the datetime variables. In this case, we do not want to drop the datetime variable after extracting the features.

```
dfts = DatetimeFeatures(
    variables=["var_dt1", "var_dt3"],
    features_to_extract=["year", "hour"],
    drop_original=False,
    format="mixed",
)
df_transf = dfts.fit_transform(toy_df)

df_transf
```

We can see the resulting dataframe in the following output:

```
              var_dt1␣
↪            var_dt2             var_dt3  var_dt1_year  \
0 2018-01-01 00:00:00␣
↪ 08/31/00 12:34:45  03/02/15 02:27:26          2018
1 2018-01-01 01:00:00␣
↪ 12/01/90 23:01:02  02/28/97 10:10:55          2018
2 2018-01-01 02:00:00␣
↪ 04/25/01 11:59:21  11/11/03 17:30:00          2018

   var_dt1_hour  var_dt3_year  var_dt3_hour
0             0          2015             2
1             1          1997            10
2             2          2003            17
```

And that is it. The new features are now added to the dataframe.

Time series

Time series data consists of datapoints indexed in time order. The time is usually in the index of the dataframe. We can extract features from the timestamp index and use them for time series regression or classification, as well as for time series forecasting.

With *DatetimeFeatures()* we can also create date and time features from the dataframe index.

Let's create a toy dataframe with datetime in the index.

```
import pandas as pd

X = {"ambient_temp":␣
↪[31.31, 31.51, 32.15, 32.39, 32.62, 32.5, 32.52, 32.68],
     "module_temp": [49.
↪18, 49.84, 52.35, 50.63, 49.61, 47.01, 46.67, 47.52],
     "irradiation
↪": [0.51, 0.79, 0.65, 0.76, 0.42, 0.49, 0.57, 0.56],
     "color": ["green"] * 4 + ["blue"] * 4,
     }

X = pd.DataFrame(X)
X.index = pd.date_
↪range("2020-05-15 12:00:00", periods=8, freq="15min")

X.head()
```

Below we see the output of our toy dataframe:

```
                ␣
↪           ambient_temp  module_temp  irradiation  color
2020-05-15␣
↪12:00:00         31.31        49.18         0.51  green
2020-05-15␣
↪12:15:00         31.51        49.84         0.79  green
2020-05-15␣
↪12:30:00         32.15        52.35         0.65  green
2020-05-15␣
↪12:45:00         32.39        50.63         0.76  green
2020-05-15␣
↪13:00:00         32.62        49.61         0.42   blue
```

We can extract features from the index as follows:

```
from feature_engine.datetime import DatetimeFeatures

dtf = DatetimeFeatures(variables="index")

Xtr = dtf.fit_transform(X)

Xtr
```

We can see that the transformer created the default time features and added them at the end of the dataframe.

```
                    ␣
↪ ambient_temp  module_temp  irradiation  color  month  \
2020-05-15 12:00:00␣
↪        31.31        49.18         0.51  green      5
2020-05-15 12:15:00␣
↪        31.51        49.84         0.79  green      5
2020-05-15 12:30:00␣
↪        32.15        52.35         0.65  green      5
```

(continues on next page)

(continued from previous page)

```
2020-05-15 12:45:00␣
↪        32.39        50.63         0.76  green      5
2020-05-15 13:00:00␣
↪        32.62        49.61         0.42   blue      5
2020-05-15 13:15:00␣
↪        32.50        47.01         0.49   blue      5
2020-05-15 13:30:00␣
↪        32.52        46.67         0.57   blue      5
2020-05-15 13:45:00␣
↪        32.68        47.52         0.56   blue      5


                ␣
↪    year  day_of_week  day_of_month  hour  minute  second
2020-05-15 12:00:00␣
↪ 2020             4            15    12       0       0
2020-05-15 12:15:00␣
↪ 2020             4            15    12      15       0
2020-05-15 12:30:00␣
↪ 2020             4            15    12      30       0
2020-05-15 12:45:00␣
↪ 2020             4            15    12      45       0
2020-05-15 13:00:00␣
↪ 2020             4            15    13       0       0
2020-05-15 13:15:00␣
↪ 2020             4            15    13      15       0
2020-05-15 13:30:00␣
↪ 2020             4            15    13      30       0
2020-05-15 13:45:00␣
↪ 2020             4            15    13      45       0
```

We can obtain the name of all the variables in the output dataframe as follows:

```
dtf.get_feature_names_out()
```

```
['ambient_temp',
 'module_temp',
 'irradiation',
 'color',
 'month',
 'year',
 'day_of_week',
 'day_of_month',
 'hour',
 'minute',
 'second']
```

Important

We highly recommend specifying the date and time features that you would like to extract from your datetime variables.

If you have too many time variables, this might not be possible. In this case, keep in mind that if you extract date features from variables that have only time, or time features from variables that have only dates, your features will be meaningless.

Let's explore the outcome with an example. We create a dataset with only time variables.

```
import pandas as pd
from feature_engine.datetime import DatetimeFeatures

toy_df = pd.DataFrame({
    "not_a_dt": ['not', 'a', 'date', 'time'],
    "var_time1
↪": ['12:34:45', '23:01:02', '11:59:21', '08:44:23'],
    "var_time2
↪": ['02:27:26', '10:10:55', '17:30:00', '18:11:18'],
})
```

And now we mistakenly extract only date features:

```
dfts = DatetimeFeatures(
    features_to_extract=["year", "month", "day_of_week"],
    format="mixed",
)
df_transf = dfts.fit_transform(toy_df)

df_transf
```

```
  not_a_dt  var_time1_year ␣
↪var_time1_month  var_time1_day_of_week  var_time2_year \
0      not             2021␣
↪               12                       2             2021
1        a             2021␣
↪               12                       2             2021
2     date             2021␣
↪               12                       2             2021
3     time             2021␣
↪               12                       2             2021

   var_time2_month  var_time2_day_of_week
0               12                      2
1               12                      2
2               12                      2
3               12                      2
```

The transformer will still create features derived from today's date (the date of creating the docs).

If instead we have a dataframe with only date variables:

```
import pandas as pd
from feature_engine.datetime import DatetimeFeatures

toy_df = pd.DataFrame({
    "var_date1
↪": ['May-1989', 'Dec-2020', 'Jan-1999', 'Feb-2002'],
    "var_date2
↪": ['06/21/12', '02/10/98', '08/03/10', '10/31/20'],
})
```

And we mistakenly extract the hour and the minute:

```
dfts = DatetimeFeatures(
    features_to_extract=["hour", "minute"],
    format="mixed",
)
df_transf = dfts.fit_transform(toy_df)

print(df_transf)
```

```
   var_date1_
↪hour  var_date1_minute  var_date2_hour  var_date2_minute
0           ␣
↪    0                0               0                 0
1           ␣
↪    0                0               0                 0
2           ␣
↪    0                0               0                 0
3           ␣
↪    0                0               0                 0
```

The new features will contain the value 0.

Automating feature extraction

We can indicate which features we want to extract from the datetime variables as we did in the previous examples, by passing the feature names in lists.

Alternatively, *DatetimeFeatures()* has default options to extract a group of commonly used features, or all supported features.

Let's first create a toy dataframe:

```
import pandas as pd
from feature_engine.datetime import DatetimeFeatures

toy_df = pd.DataFrame({
    "var_
↪dt1": pd.date_range("2018-01-01", periods=3, freq="H"),
    "var_dt2": ['08/31/
↪00 12:34:45', '12/01/90 23:01:02', '04/25/01 11:59:21'],
    "var_dt3": ['03/02/
```

(continues on next page)

(continued from previous page)

```
↪15 02:27:26', '02/28/97 10:10:55', '11/11/03 17:30:00'],
})
```

Most common features

Now, we will extract the **most common** date and time features from one of the variables. To do this, we leave the parameter `features_to_extract` to `None`.

```
dfts = DatetimeFeatures(
    variables=["var_dt1"],
    features_to_extract=None,
    drop_original=False,
)

df_transf = dfts.fit_transform(toy_df)

df_transf
```

```
             var_dt1␣
↪            var_dt2            var_dt3  var_dt1_month  \
0 2018-01-01 00:00:00␣
↪ 08/31/00 12:34:45  03/02/15 02:27:26              1
1 2018-01-01 01:00:00␣
↪ 12/01/90 23:01:02  02/28/97 10:10:55              1
2 2018-01-01 02:00:00␣
↪ 04/25/01 11:59:21  11/11/03 17:30:00              1

   var_dt1_year  var_
↪dt1_day_of_week  var_dt1_day_of_month  var_dt1_hour  \
0          2018␣
↪                    0                     1             0
1          2018␣
↪                    0                                   1
2          2018␣
↪                    0                   1               2

    var_dt1_minute    var_dt1_second
0                0                 0
1                0                 0
2                0                 0
```

Our new dataset contains the original features plus the new variables extracted from them.

We can find the group of features extracted by the transformer in its attribute:

```
dfts.features_to_extract_
```

```
['month',
 'year',
```

(continues on next page)

(continued from previous page)

```
 'day_of_week',
 'day_of_month',
 'hour',
 'minute',
 'second']
```

All supported features

We can also extract all supported features automatically, by setting the parameter `features_to_extract` to `"all"`:

```
dfts = DatetimeFeatures(
    variables=["var_dt1"],
    features_to_extract='all',
    drop_original=False,
)

df_transf = dfts.fit_transform(toy_df)

print(df_transf)
```

```
             var_dt1␣
↪           var_dt2            var_dt3  var_dt1_month  \
0 2018-01-01 00:00:00␣
↪ 08/31/00 12:34:45  03/02/15 02:27:26              1
1 2018-01-01 01:00:00␣
↪ 12/01/90 23:01:02  02/28/97 10:10:55              1
2 2018-01-01 02:00:00␣
↪ 04/25/01 11:59:21  11/11/03 17:30:00              1

   var_dt1_quarter  var_dt1_semester  var_dt1_year  \
0                1                 1          2018
1                1                 1          2018
2                1                 1          2018

   var_dt1_week  var_dt1_day_of_
↪week  ...  var_dt1_month_end  var_dt1_quarter_start  \
0             1           ␣
↪        0  ...                 0                      1
1             1           ␣
↪        0  ...                 0                      1
2             1           ␣
↪        0  ...                 0                      1

   var_
↪dt1_quarter_end  var_dt1_year_start  var_dt1_year_end  \
0  ␣
↪                 0                  1                0
1  ␣
↪                 0                  1                0
```

(continues on next page)

(continued from previous page)

```
2  ␣
↪                 0                   1                 0

   var_dt1_leap_year␣
↪ var_dt1_days_in_month  var_dt1_hour  var_dt1_minute  \
0                ␣
↪  0                    31             0               0
1                ␣
↪  0                    31             1               0
2                ␣
↪  0                    31             2               0

   var_dt1_second
0               0
1               0
2               0
```

We can find the group of features extracted by the transformer in its attribute:

```
dfts.features_to_extract_
```

```
['month',
 'quarter',
 'semester',
 'year',
 'week',
 'day_of_week',
 'day_of_month',
 'day_of_year',
 'weekend',
 'month_start',
 'month_end',
 'quarter_start',
 'quarter_end',
 'year_start',
 'year_end',
 'leap_year',
 'days_in_month',
 'hour',
 'minute',
 'second']
```

Extract and select features automatically

If we have a dataframe with date variables, time variables and date and time variables, we can extract all features, or the most common features from all the variables, and then go ahead and remove the irrelevant features with the `DropConstantFeatures()` class.

Let's create a dataframe with a mix of datetime variables.

```
import pandas as pd
from sklearn.pipeline import Pipeline
from feature_engine.datetime import DatetimeFeatures
from feature_engine.selection import DropConstantFeatures

toy_df = pd.DataFrame({
    "var_
↪date": ['06/21/12', '02/10/98', '08/03/10', '10/31/20'],
    "var_time1
↪": ['12:34:45', '23:01:02', '11:59:21', '08:44:23'],
    "var_dt": ['08/31/00 12:34:45', '12/01/
↪90 23:01:02', '04/25/01 11:59:21', '04/25/01 11:59:21'],
})
```

Now, we line up in a Scikit-learn pipeline the *DatetimeFeatures* and the `DropConstantFeatures()`. The *DatetimeFeatures* will create date features derived from today for the time variable, and time features with the value 0 for the date only variable. `DropConstantFeatures()` will identify and remove these features from the dataset.

```
pipe = Pipeline([
    ('datetime', DatetimeFeatures(format="mixed")),
    ('drop_constant', DropConstantFeatures()),
])

pipe.fit(toy_df)
```

```
Pipeline(steps=[('datetime', DatetimeFeatures()),
  ␣
↪             ('drop_constant', DropConstantFeatures())])
```

```
df_transf = pipe.transform(toy_df)

print(df_transf)
```

```
   var_date_month  var_date_
↪year  var_date_day_of_week  var_date_day_of_month  \
0               6   ␣
↪       2012                     3                    21
1               2   ␣
↪       1998                     1                    10
2               8   ␣
↪       2010                     1                     3
3              10   ␣
```

(continues on next page)

(continued from previous page)

```
↪       2020                    5                    31

   var_time1_hour␣
↪ var_time1_minute  var_time1_second  var_dt_month  \
0          ␣
↪    12                34                45             8
1          ␣
↪    23                 1                 2            12
2          ␣
↪    11                59                21             4
3          ␣
↪     8                44                23             4

   var_dt_year␣
↪ var_dt_day_of_week  var_dt_day_of_month  var_dt_hour  \
0          2000␣
↪                   3                   31           12
1          1990␣
↪                   5                    1           23
2          2001␣
↪                   2                   25           11
3          2001␣
↪                   2                   25           11

   var_dt_minute   var_dt_second
0             34              45
1              1               2
2             59              21
3             59              21
```

As you can see, we do not have the constant features in the transformed dataset.

Working with different timezones

Time-aware datetime variables can be particularly cumbersome to work with as far as the format goes. We will briefly show how *DatetimeFeatures()* deals with such variables in three different scenarios.

Case 1: our dataset contains a time-aware variable in object format, with potentially different timezones across different observations. We pass `utc=True` when initializing the transformer to make sure it converts all data to UTC timezone.

```
import pandas as pd
from feature_engine.datetime import DatetimeFeatures

toy_df = pd.DataFrame({
    "var_tz":␣
↪['12:34:45+3', '23:01:02-6', '11:59:21-8', '08:44:23Z']
})

dfts = DatetimeFeatures(
```

(continues on next page)

(continued from previous page)

```
    features_to_extract=["hour", "minute"],
    drop_original=False,
    utc=True,
    format="mixed",
)

df_transf = dfts.fit_transform(toy_df)

df_transf
```

```
       var_tz  var_tz_hour  var_tz_minute
0  12:34:45+3            9             34
1  23:01:02-6            5              1
2  11:59:21-8           19             59
3   08:44:23Z            8             44
```

Case 2: our dataset contains a variable that is cast as a localized datetime in a particular timezone. However, we decide that we want to get all the datetime information extracted as if it were in UTC timezone.

```
import pandas as pd
from feature_engine.datetime import DatetimeFeatures

var_tz = pd.Series(['08/31/
↪00 12:34:45', '12/01/90 23:01:02', '04/25/01 11:59:21'])
var_tz = pd.to_datetime(var_tz, format="mixed")
var_tz = var_tz.dt.tz_localize("US/eastern")
var_tz
```

```
0   2000-08-31 12:34:45-04:00
1   1990-12-01 23:01:02-05:00
2   2001-04-25 11:59:21-04:00
dtype: datetime64[ns, US/Eastern]
```

We need to pass `utc=True` when initializing the transformer to revert back to the UTC timezone.

```
toy_df = pd.DataFrame({"var_tz": var_tz})

dfts = DatetimeFeatures(
    features_to_extract=["day_of_month", "hour"],
    drop_original=False,
    utc=True,
)

df_transf = dfts.fit_transform(toy_df)

df_transf
```

```
    ␣
↪                 var_tz  var_tz_day_of_month  var_tz_hour
0 2000-
```

(continues on next page)

(continued from previous page)

```
↪08-31 12:34:45-04:00                    31           16
1 1990-
↪12-01 23:01:02-05:00                     2            4
2 2001-
↪04-25 11:59:21-04:00                    25           15
```

Case 3: given a variable like *var_tz* in the example above, we now want to extract the features keeping the original timezone localization, therefore we pass `utc=False` or `None`. In this case, we leave it to `None` which is the default option.

```
dfts = DatetimeFeatures(
    features_to_extract=["day_of_month", "hour"],
    drop_original=False,
    utc=None,
)

df_transf = dfts.fit_transform(toy_df)

print(df_transf)
```

```
    ␣
↪                var_tz  var_tz_day_of_month  var_tz_hour
0 2000-
↪08-31 12:34:45-04:00                    31           12
1 1990-
↪12-01 23:01:02-05:00                     1           23
2 2001-
↪04-25 11:59:21-04:00                    25           11
```

Note that the hour extracted from the variable differ in this dataframe respect to the one obtained in **Case 2**.

Missing timestamps

DatetimeFeatures has the option to ignore missing timestamps, or raise an error when a missing value is encountered in a datetime variable.

Additional resources

You can find an example of how to use *DatetimeFeatures()* with a real dataset in the following Jupyter notebook

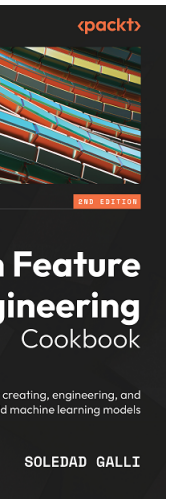
For tutorials on how to create and use features from datetime columns, check the following courses:

Fig. 68: Feature Engineering for Machine Learning

Or read our book:

Fig. 69: Feature Engineering for Time Series Forecasting

Both our book and course are suitable for beginners and
more advanced data scientists alike. By purchasing them you are supporting Sole, the main developer of Feature-engine.



ure Engineering

DatetimeSubtraction

Very often, we have datetime variables in our datasets, and we want to determine the time difference between them. For example, if we work with financial data, we may have the variable **date of loan application**, with the date and time when the customer applied for a loan, and also the variable **date of birth**, with the customer's date of birth. With those two variables, we want to infer the **age of the customer** at the time of application. In order to do this, we can compute the difference in years between `date_of_loan_application` and `date_of_birth` and capture it in a new variable.

In a different example, if we are trying to predict the price of the house and we have information about the year in which the house was built, we can infer the age of the house at the point of sale. Generally, older houses cost less. To calculate the age of the house, we'd simply compute the difference in years between the sale date and the date at which it was built.

The Python program offers many options for making operations between datetime objects, like, for example, the datetime module. Since most likely you will be working with Pandas dataframes, we will focus this guide on pandas and then how we can automate the procedure with Feature-engine.

Subtracting datetime features with pandas

In Python, we can subtract datetime objects with pandas. To work with datetime variables in pandas, we need to make sure that the timestamp, which can be represented in various formats, like strings (str), objects (`"O"`), or datetime, is cast as a datetime. If not, we can convert strings to datetime objects by executing `pd.to_datetime(df[variable_of_interest])`.

Let's create a toy dataframe with 2 datetime variables for a short demo:

```
import numpy as np
import pandas as pd

data = pd.DataFrame({
    "date1
↪": pd.date_range("2019-03-05", periods=5, freq="D"),
    "date2
↪": pd.date_range("2018-03-05", periods=5, freq="W")})

print(data)
```

This is the data that we created, containing two datetime variables:

```
       date1      date2
0 2019-03-05 2018-03-11
1 2019-03-06 2018-03-18
2 2019-03-07 2018-03-25
3 2019-03-08 2018-04-01
4 2019-03-09 2018-04-08
```

Now, we can subtract `date2` from `date1` and capture the difference in a new variable by utilizing the pandas subtraction operator:

```
data["diff"] = data["date1"].sub(data["date2"])

print(data)
```

The new variable, which expresses the difference in number of days, is at the right of the dataframe:

```
       date1      date2     diff
0 2019-03-05 2018-03-11 359 days
1 2019-03-06 2018-03-18 353 days
2 2019-03-07 2018-03-25 347 days
3 2019-03-08 2018-04-01 341 days
4 2019-03-09 2018-04-08 335 days
```

If we want the units in something other than days, we can use numpy's timedelta. The following example shows how to use this syntax:

```
data[
↪"diff"] = data["date1"].sub(data["date2"], axis=0).div(
    np.timedelta64(1, "Y").astype("timedelta64[ns]"))

print(data)
```

We see the new variable now expressing the difference in years, at the right of the dataframe:

```
       date1      date2      diff
0 2019-03-05 2018-03-11  0.982909
1 2019-03-06 2018-03-18  0.966481
2 2019-03-07 2018-03-25  0.950054
3 2019-03-08 2018-04-01  0.933626
4 2019-03-09 2018-04-08  0.917199
```

If you wanted to subtract various datetime variables, you would have to write lines of code for every subtraction. Fortunately, we can automate this procedure with `DatetimeSubtraction()`.

Datetime subtraction with Feature-engine

`DatetimeSubtraction()` automatically subtracts several date and time features from each other. You just need to indicate the features at the right of the subtraction operation in the `variables` parameters and those on the left in the `reference parameter`. You can also change the output unit through the `output_unit` parameter.

`DatetimeSubtraction()` works with variables whose `dtype` is datetime, as well as with object-like and categorical variables, provided that they can be parsed into datetime format. This will be done under the hood by the transformer.

Following up with the former example, here is how we obtain the difference in number of days using `DatetimeSubtraction()`:

```
import pandas as pd
from feature_engine.datetime import DatetimeSubtraction

data = pd.DataFrame({
    "date1
↪": pd.date_range("2019-03-05", periods=5, freq="D"),
    "date2
↪": pd.date_range("2018-03-05", periods=5, freq="W")})

dtf = DatetimeSubtraction(
    variables="date1",
    reference="date2",
    output_unit="Y")

data = dtf.fit_transform(data)

print(data)
```

With `transform()`, `DatetimeSubstraction()` returns a new dataframe containing the original variables and also the new variables with the time difference:

```
       date1      date2  date1_sub_date2
0 2019-03-05 2018-03-11         0.982909
1 2019-03-06 2018-03-18         0.966481
2 2019-03-07 2018-03-25         0.950054
3 2019-03-08 2018-04-01         0.933626
4 2019-03-09 2018-04-08         0.917199
```

Drop original variables after computation

We have the option to drop the original datetime variables after the computation:

```
import pandas as pd
from feature_engine.datetime import DatetimeSubtraction

data = pd.DataFrame({
    "date1
↪": pd.date_range("2019-03-05", periods=5, freq="D"),
    "date2
↪": pd.date_range("2018-03-05", periods=5, freq="W")})

dtf = DatetimeSubtraction(
    variables="date1",
    reference="date2",
    output_unit="M",
    drop_original=True
)

data = dtf.fit_transform(data)
```

(continues on next page)

(continued from previous page)

```
print(data)
```

In this case, the resulting dataframe contains only the time difference between the two original variables:

```
   date1_sub_date2
0        11.794903
1        11.597774
2        11.400645
3        11.203515
4        11.006386
```

Subtract multiple variables simultaneously

We can perform multiple subtractions at the same time. In this example, we will add new datetime variables to the toy dataframe as strings. The idea is to show that `DatetimeSubstraction()` will convert those strings to datetime under the hood to carry out the subtraction operation.

```
import pandas as pd
from feature_engine.datetime import DatetimeSubtraction

data = pd.DataFrame({
    "date1" : ["2022-09-01", "2022-10-01", "2022-12-01"],
    "date2" : ["2022-09-15", "2022-10-15", "2022-12-15"],
    "date3" : ["2022-08-01", "2022-09-01", "2022-11-01"],
    "date4" : ["2022-08-15", "2022-09-15", "2022-11-15"],
})

dtf = DatetimeSubtraction(variables=[
↪"date1", "date2"], reference=["date3", "date4"])

data = dtf.fit_transform(data)

print(data)
```

The resulting dataframe contains the original variables plus the new variables expressing the time difference between the date objects.

```
        date1␣
↪       date2       date3       date4  date1_sub_date3  \
0  2022-09-
↪01  2022-09-15  2022-08-01  2022-08-15             31.0
1  2022-10-
↪01  2022-10-15  2022-09-01  2022-09-15             30.0
2  2022-12-
↪01  2022-12-15  2022-11-01  2022-11-15             30.0

   date2_sub_date3  date1_sub_date4  date2_sub_date4
0             45.0             17.0             31.0
```

(continues on next page)

(continued from previous page)

```
1             44.0             16.0             30.0
2             44.0             16.0             30.0
```

Working with missing values

By default, `DatetimeSubstraction()` will raise an error if the dataframe passed to the `fit()` or `transform()` methods contains NA in the variables to subtract. We can override this behaviour and allow computations between variables with nan by setting the parameter `missing_values` to `"ignore"`. Here is a code example:

```
import numpy as np
import pandas as pd
from feature_engine.datetime import DatetimeSubtraction

data = pd.DataFrame({
    "date1" : ["2022-09-01", "2022-10-01", "2022-12-01"],
    "date2" : ["2022-09-15", np.nan, "2022-12-15"],
    "date3" : ["2022-08-01", "2022-09-01", "2022-11-01"],
    "date4" : ["2022-08-15", "2022-09-15", np.nan],
})

dtf = DatetimeSubtraction(
    variables=["date1", "date2"],
    reference=["date3", "date4"],
    missing_values="ignore")

data = dtf.fit_transform(data)

print(data)
```

When any of the variables contains NAN, the new features with the time difference will also display NANs:

```
        date1␣
↪       date2       date3       date4  date1_sub_date3  \
0  2022-09-
↪01  2022-09-15  2022-08-01  2022-08-15             31.0
1  2022-10-
↪01         NaN  2022-09-01  2022-09-15             30.0
2  2022-12-
↪01  2022-12-15  2022-11-01         NaN             30.0

   date2_sub_date3  date1_sub_date4  date2_sub_date4
0             45.0             17.0             31.0
1              NaN             16.0              NaN
2             44.0              NaN              NaN
```

Working with different timezones

If we have timestamps in different timezones or variables in different timezones, we can still perform subtraction operations with `DatetimeSubstraction()` by first setting all timestamps to the universal central time zone. Here is a code example, were we return the time difference in microseconds:

```
import pandas as pd
from feature_engine.datetime import DatetimeSubtraction

data = pd.DataFrame({
    "date1":␣
↪['12:34:45+3', '23:01:02-6', '11:59:21-8', '08:44:23Z'],
    "date2": ['09:34:45+1
↪', '23:01:02-6+1', '11:59:21-8-2', '08:44:23+3']
})

dfts = DatetimeSubtraction(
    variables="date1",
    reference="date2",
    utc=True,
    output_unit="ms",
    format="mixed"
)

new = dfts.fit_transform(data)

print(new)
```

We see the resulting dataframe with the time difference in microseconds:

```
        date1         date2  date1_sub_date2
0  12:34:45+3    09:34:45+1        3600000.0
1  23:01:02-6  23:01:02-6+1       25200000.0
2  11:59:21-8  11:59:21-8-2       21600000.0
3   08:44:23Z    08:44:23+3       10800000.0
```

Adding arbitrary names to the new variables

Often, we want to compute just a few time differences. In this case, we may want as well to assign the new variables specific names. In this code example, we do so:

```
import pandas as pd
from feature_engine.datetime import DatetimeSubtraction

data = pd.DataFrame({
    "date1
↪": pd.date_range("2019-03-05", periods=5, freq="D"),
    "date2
↪": pd.date_range("2018-03-05", periods=5, freq="W")})
```

(continues on next page)

(continued from previous page)

```
dtf = DatetimeSubtraction(
    variables="date1",
    reference="date2",
    new_variables_names=["my_new_var"]
    )

data = dtf.fit_transform(data)

print(data)
```

In the resulting dataframe, we see that the time difference was captured in a variable called `my_new_var`:

```
       date1      date2  my_new_var
0 2019-03-05 2018-03-11       359.0
1 2019-03-06 2018-03-18       353.0
2 2019-03-07 2018-03-25       347.0
3 2019-03-08 2018-04-01       341.0
4 2019-03-09 2018-04-08       335.0
```

We should be mindful to pass a list of variales containing as many names as new variables. The number of variables that will be created is obtained by multiplying the number of variables in the parameter `variables` by the number of variables in the parameter `reference`.

get_feature_names_out()

Finally, we can extract the names of the transformed dataframe for compatibility with the Scikit-learn pipeline:

```
import pandas as pd
from feature_engine.datetime import DatetimeSubtraction

data = pd.DataFrame({
    "date1" : ["2022-09-01", "2022-10-01", "2022-12-01"],
    "date2" : ["2022-09-15", "2022-10-15", "2022-12-15"],
    "date3" : ["2022-08-01", "2022-09-01", "2022-11-01"],
    "date4" : ["2022-08-15", "2022-09-15", "2022-11-15"],
})

dtf = DatetimeSubtraction(variables=[
↪"date1", "date2"], reference=["date3", "date4"])
dtf.fit(data)

dtf.get_feature_names_out()
```

Below the name of the variables that will appear in any dataframe resulting from applying the `transform()` method:

```
['date1',
 'date2',
```

(continues on next page)

(continued from previous page)

```
 'date3',
 'date4',
 'date1_sub_date3',
 'date2_sub_date3',
 'date1_sub_date4',
 'date2_sub_date4']
```

Combining extraction and subtraction of datetime features

We can also combine the creation of numerical variables from datetime features with the creation of new features by subtraction of datetime variables:

```
import pandas as pd
from sklearn.pipeline import Pipeline
from feature_engine.
↪datetime import DatetimeFeatures, DatetimeSubtraction

data = pd.DataFrame({
    "date1" : ["2022-09-01", "2022-10-01", "2022-12-01"],
    "date2" : ["2022-09-15", "2022-10-15", "2022-12-15"],
    "date3" : ["2022-08-01", "2022-09-01", "2022-11-01"],
    "date4" : ["2022-08-15", "2022-09-15", "2022-11-15"],
})

dtf = DatetimeFeatures(variables=[
↪"date1", "date2"], drop_original=False)
dts = DatetimeSubtraction(
    variables=["date1", "date2"],
    reference=["date3", "date4"],
    drop_original=True,
)

pipe = Pipeline([
    ("features", dtf),("subtraction", dts)
])

data = pipe.fit_transform(data)

print(data)
```

In the following output we see the new dataframe contaning the features that were extracted from the different datetime variables followed by those created by capturing the time difference:

```
   date1_month  date1_year␣
↪ date1_day_of_week  date1_day_of_month  date1_hour  \
0            9      ␣
↪ 2022                  3                  1           0
1           10      ␣
↪ 2022                  5                  1           0
2           12      ␣
```

(continues on next page)

(continued from previous page)

```
↪ 2022                   3                    1           0

   date1_minute  date1_
↪second  date2_month  date2_year  date2_day_of_week  \
0             0 ␣
↪           0            9        2022                  3
1             0 ␣
↪           0           10        2022                  5
2             0 ␣
↪           0           12        2022                  3

   date2_
↪day_of_month  date2_hour  date2_minute  date2_second  \
0    ␣
↪              15           0             0             0
1    ␣
↪              15           0             0             0
2    ␣
↪              15           0             0             0

   date1_sub_
↪date3  date2_sub_date3  date1_sub_date4  date2_sub_date4
0           ␣
↪ 31.0             45.0             17.0             31.0
1           ␣
↪ 30.0             44.0             16.0             30.0
2           ␣
↪ 30.0             44.0             16.0             30.0
```

Additional resources

For tutorials on how to create and use features from datetime columns, check the following courses:

Fig. 71: Feature Engineering for Machine Learning

Or read our book:

Fig. 72: Feature Engineering for Time Series Forecasting

Both our book and course are suitable for beginners and
more advanced data scientists alike. By purchasing them you are supporting Sole, the main developer of Feature-engine.

10.2.3 Selection

Feature Selection

Feature-engine's feature selection transformers identify features with low predictive performance and drop them from the dataset. Most of the feature selection algorithms supported by Feature-engine are not yet available in other libraries. These algorithms have been gathered from data science competitions or used in the industry.

Selection Mechanism Overview

Feature-engine's transformers select features based on different strategies.

The first strategy evaluates the features intrinsic characteristics, like their distributions. For example, we can remove constant or quasi-constant features. Or we can remove features whose distribution in unstable in time by using the Population Stability Index.

A second strategy consists in determining the relationships between features. Among these, we can remove features that are duplicated or correlated.

We can also select features based on their relationship with the target. To assess this, we can replace the feature values by the target mean, or calculate the information value.

Some feature selection procedures involve training machine learning models. We can assess features individually, or collectively, through various algorithms, as shown in the following diagram:

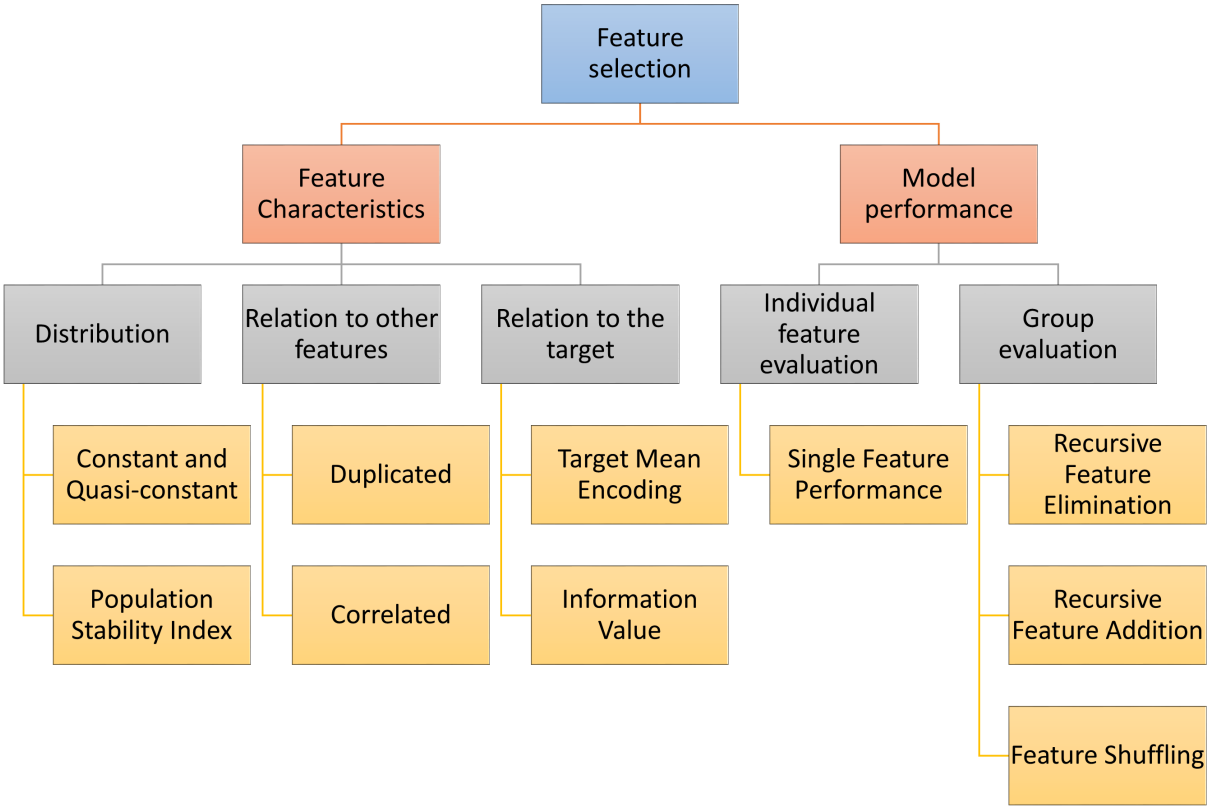


Fig. 74: Selection mechanisms - Overview

Algorithms that select features based on their performance within a group of variables, will normally train a model with all the features, and then remove or add or shuffle a feature and re-evaluate the model performance.

These methods are normally geared towards improving the overall performance of the final machine learning model as well as reducing the feature space.

Selectors Characteristics Overview

Some Feature-engine's selectors work with categorical variables off-the-shelf and/or allow missing data in the variables. These gives you the opportunity to quickly screen features before jumping into any feature engineering.

In the following tables, we highlight the main Feature-engine selectors characteristics:

Selection based on feature characteristics

Transformer	Categorical variables	Allows NA	Description
DropFeatures()			Drops arbitrary features determined by user
DropConstantFeatures()			Drops constant and quasi-constant features
DropDuplicateFeatures()			Drops features that are duplicated
DropCorrelatedFeatures()	×		Drops features that are correlated
SmartCorrelatedSelection()	×		From a correlated feature group drops the less useful features

Methods that determine duplication or the number of unique values, can work with both numerical and categorical variables and support missing data as well.

Selection procedures based on correlation work only with numerical variables but allow missing data.

Selection based on a machine learning model

Transformer	Categorical variables	Allows NA	Description
SelectBySingleFeaturePerformance()	×	×	Selects features based on single feature model performance
RecursiveFeatureElimination()	×	×	Removes features recursively by evaluating model performance
RecursiveFeatureAddition()	×	×	Adds features recursively by evaluating model performance

Selection procedures that require training a machine learning model from Scikit-learn require numerical variables without missing data.

Selection methods commonly used in finance

Transformer	Categorical variables	Allows NA	Description
DropHighPSIFeatures()	×		Drops features with high Population Stability Index
SelectByInformationValue()		x	Drops features with low information value

DropHighPSIFeatures() allows to remove features with changes in their distribution. This is done by splitting the input dataframe in two parts and comparing the distribution of each feature in the two parts. The metric used to assess distribution shift is the Population Stability Index (PSI). Removing unstable features may lead to more robust models. In fields like Credit Risk Modelling, the Regulator often requires the PSI of the final feature set to be below are given threshold.

Alternative feature selection methods

Transformer	Categorical variables	Allows NA	Description
SelectByShuffling()	×	×	Selects features if shuffling their values causes a drop in model performance
SelectByTargetMeanPerformance()		×	Using the target mean as performance proxy, selects high performing features
ProbeFeatureSelection()	×	×	Selects features whose importance is greater than those of random variables

The *SelectByTargetMeanPerformance()* uses the target mean value as proxy for prediction, replacing categories or variable intervals by these values and then determining a performance metric. Thus, it is suitable for both categorical and numerical variables. In its current implementation, it does not support missing data.

The *ProbeFeatureSelection()* introduces random variables to the dataset, then creates a model and derives the feature importance. It selects all variables whose importance is grater than the mean importance of the random features.

Throughout the rest of user guide, you will find more details about each of the feature selection procedures.

Feature Selection Algorithms

Click below to find more details on how to use each one of the transformers.

DropFeatures

The *DropFeatures()* drops a list of variables indicated by the user from the original dataframe. The user can pass a single variable as a string or list of variables to be dropped.

DropFeatures() offers similar functionality to pandas.dataframe.drop, but the difference is that *DropFeatures()* can be integrated into a Scikit-learn pipeline.

When is this transformer useful?

Sometimes, we create new variables combining other variables in the dataset, for example, we obtain the variable `age` by subtracting `date_of_application` from `date_of_birth`. After we obtained our

new variable, we do not need the date variables in the dataset any more. Thus, we can add *DropFeatures()* in the Pipeline to have these removed.

Example

Let's see how to use *DropFeatures()* in an example with the Titanic dataset. We first load the data and separate it into train and test:

```
from sklearn.model_selection import train_test_split
from feature_engine.datasets import load_titanic
from feature_engine.selection import DropFeatures

X, y = load_titanic(
    return_X_y_frame=True,
    handle_missing=True,
)


X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=0,
)

print(X_train.head())
```

Now, we go ahead and print the dataset column names:

```
X_train.columns
```

```
Index(['pclass', 'name
↪', 'sex', 'age', 'sibsp', 'parch', 'ticket', 'fare',
       'cabin', 'embarked', 'boat', 'body', 'home.dest'],
      dtype='object')
```

Now, with *DropFeatures()* we can very easily drop a group of variables. Below we set up the transformer to drop a list of 6 variables:

```
# set up the transformer
transformer = DropFeatures(
    features_to_drop=[
↪'sibsp', 'parch', 'ticket', 'fare', 'body', 'home.dest']
)

# fit the transformer
transformer.fit(X_train)
```

With `fit()` this transformer does not learn any parameter. We can go ahead and remove the variables as follows:

```
train_t = transformer.transform(X_train)
test_t = transformer.transform(X_test)
```

And now, if we print the variable names of the transformed dataset, we see that it has been reduced:

```
train_t.columns
```

```
Index(['pclass', 'name', 'sex
↪', 'age', 'cabin', 'embarked', 'boat'], dtype='object')
```

Additional resources

In this Kaggle kernel we feature 3 different end-to-end machine learning pipelines using *DropFeatures()*:

- Kaggle Kernel

All notebooks can be found in a dedicated repository.

For more details about this and other feature selection methods check out these resources:

Or read our book:

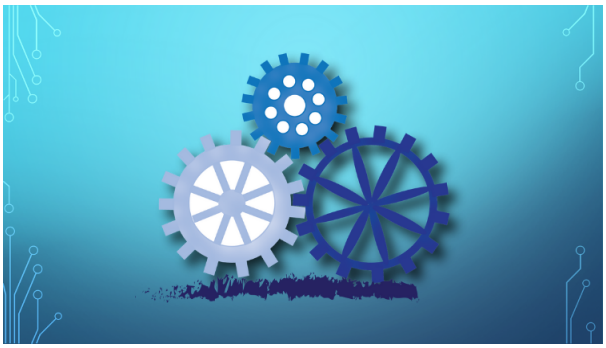
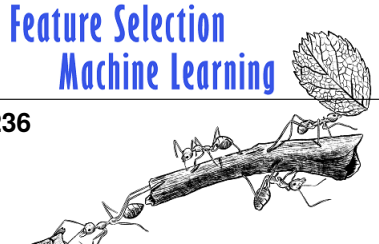


Fig. 75: Feature Selection for Machine Learning

Both our book and course are suitable for beginners and more advanced data scientists alike. By purchasing them you are supporting Sole, the main developer of Feature-engine.



DropConstantFeatures

Constant features are variables that show zero variability, or, in other words, have the same value in all rows. A key step towards training a machine learning model is to identify and remove constant features.

Features with no or low variability rarely constitute useful predictors. Hence, removing them right at the beginning of the data science project is a good way of simplifying your dataset and subsequent data preprocessing pipelines.

Filter methods are selection algorithms that select or remove features based solely on their characteristics. In this light, removing constant features could be considered part of the filter group of selection algorithms.

In Python, we can find constant features by using pandas `std` or `unique` methods, and then remove them with `drop`.

With Scikit-learn, we can find and remove constant variables with `VarianceThreshold` to quickly reduce the number of features. `VarianceThreshold` is part of `sklearn.feature_selection`'s API.

`VarianceThreshold`, however, would only work with numerical variables. Hence, we could only evaluate categorical variables after encoding them, which requires a prior step of data preprocessing just to remove redundant variables.

Feature-engine introduces *DropConstantFeatures()* to find and remove constant and quasi-constant features from a dataframe. *DropConstantFeatures()* works with numerical, categorical, or datetime variables. It is therefore more versatile than Scikit-learn's transformer because it allows us to drop all duplicate variables without the need for prior data transformations.

By default, *DropConstantFeatures()* drops constant variables. We also have the option to drop quasi-constant features, which are those that show mostly constant values and some other values in a very small percentage of rows.

Because *DropConstantFeatures()* works with numerical and categorical variables alike, it offers a straightforward way of reducing the feature subset.

Be mindful, though, that depending on the context, quasi-constant variables could be useful.

Example

Let's see how to use *DropConstantFeatures()* by using the Titanic dataset. This dataset does not contain constant or quasi-constant variables, so for the sake of the demonstration, we will consider quasi-constant those features that show the same value in more than 70% of the rows.

We first load the data and separate it into a training set and a test set:

```
from sklearn.model_selection import train_test_split
from feature_engine.datasets import load_titanic
from feature_engine.selection import DropConstantFeatures

X, y = load_titanic(
    return_X_y_frame=True,
    handle_missing=True,
)


X_train, X_test, y_train, y_test = train_test_split(
```

(continues on next page)

(continued from previous page)

```
    X, y, test_size=0.3, random_state=0,
)
```

Now, we set up the *DropConstantFeatures()* to remove features that show the same value in more than 70% of the observations. We do this through the parameter `tol`. The default value for this parameter is zero, in which case it will remove constant features.

```
# set up the transformer
transformer = DropConstantFeatures(tol=0.7)
```

With `fit()` the transformer finds the variables to drop:

```
# fit the transformer
transformer.fit(X_train)
```

The variables to drop are stored in the attribute `features_to_drop_`:

```
transformer.features_to_drop_
```

```
['parch', 'cabin', 'embarked', 'body']
```

We can check that the variables `parch` and `embarked` show the same value in more than 70% of the observations as follows:

```
X_train['embarked'].value_counts(normalize = True)
```

```
S          0.711790
C          0.195415
Q          0.090611
Missing    0.002183
Name: embarked, dtype: float64
```

Based on the previous results, 71% of the passengers embarked in S.

Let's now evaluate `parch`:

```
X_train['parch'].value_counts(normalize = True)
```

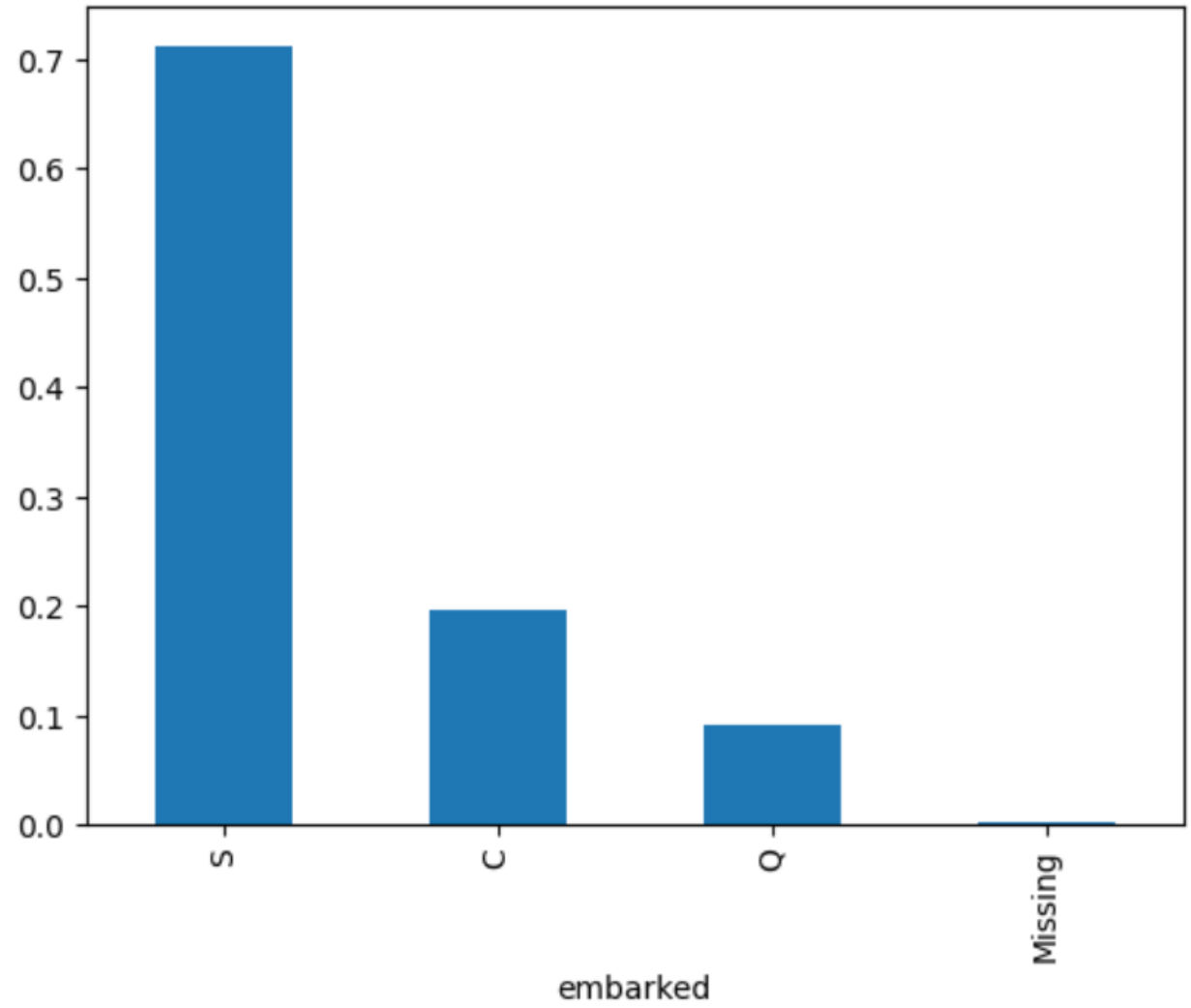
```
0    0.771834
1    0.125546
2    0.086245
3    0.005459
4    0.004367
5    0.003275
6    0.002183
9    0.001092
Name: parch, dtype: float64
```

Based on the previous results, 77% of the passengers had 0 parent or child. Because of this, these features were deemed quasi-constant and will be removed in the next step.

We can also identify quasi-constant variables as follows:

```
import pandas

X_train["embarked"].value_counts(normalize=True).plot.bar()
```

After executing the previous code, we observe the following plot, with more than 70% of passengers embarking in S:



With `transform()`, we drop the quasi-constant variables from the dataset:

```
train_t = transformer.transform(X_train)
test_t = transformer.transform(X_test)

print(train_t.head())
```

We see the resulting dataframe below:

```
      pclass                                name     sex        age  sibsp  \
501        2   Mellinger, Miss. Madeleine Violet  female  13.000000      0
588        2                  Wells, Miss. Joan  female   4.000000      1
402        2     Duran y More, Miss. Florentina  female  30.000000      1
```

(continues on next page)

(continued from previous page)

```
1193       3                 Scanlan, Mr. James    male  29.881135      0
686        3       Bradley, Miss. Bridget Delia  female  22.000000      0

             ticket     fare     boat  \
501          250644  19.5000       14
588           29103  23.0000       14
402   SC/PARIS 2148  13.8583       12
1193          36209   7.7250  Missing
686          334914   7.7250       13

                                          home.dest
501                          England / Bennington, VT
588                              Cornwall / Akron, OH
402                    Barcelona, Spain / Havana, Cuba
1193                                          Missing
686   Kingwilliamstown, Co Cork, Ireland Glens Falls...
```

Like sklearn, Feature-engine transformers have the `fit_transform` method that allows us to find and remove constant or quasi-constant variables in a single line of code for convenience.

Like sklearn as well, `DropConstantFeatures()` has the `get_support()` method, which returns a vector with values `True` for features that will be retained and `False` for those that will be dropped.

```
transformer.get_support()
```

```
[True, True, True, True, True, False, True, True, False, False,
 True, False, True]
```

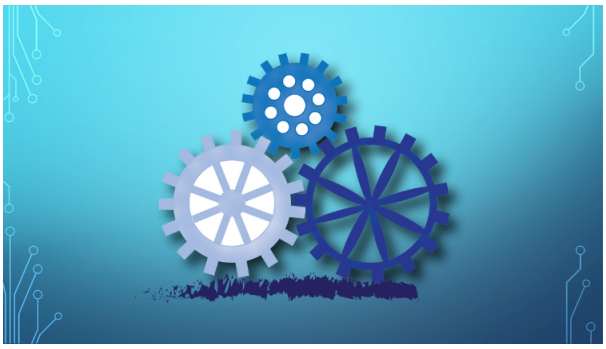
This and other feature selection methods may not necessarily avoid overfitting, but they contribute to simplifying our machine learning pipelines and creating more interpretable machine learning models.

Additional resources

In this Kaggle kernel we use *DropConstantFeatures()* together with other feature selection algorithms and then train a Logistic regression estimator:

- Kaggle kernel

For more details about this and other feature selection methods check out these resources:



Or read our book:

Fig. 77: Feature Selection for Machine Learning

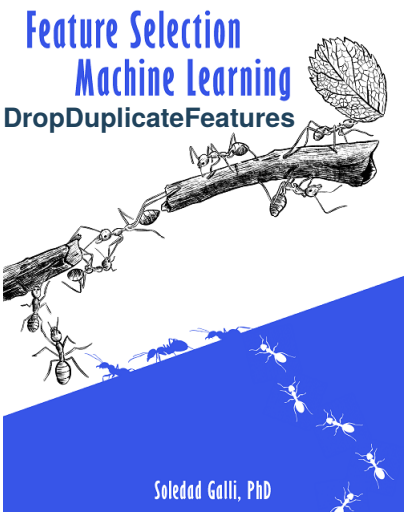


Fig. 78: Feature Selection in Machine Learning

Both our book and course are suitable for beginners and more advanced data scientists alike. By purchasing them you are supporting Sole, the main developer of Feature-engine.

Duplicate features are columns in a dataset that are identical, or, in other words, that contain exactly the same values. Duplicate features can be introduced accidentally, either through poor data management processes or during data manipulation.

For example, duplicated new records can be created by one-hot encoding a categorical variable or by adding missing data indicators. We can also accidentally generate duplicate records when we merge different data sources that show some variable overlap.

Checking for and removing duplicate features is a standard procedure in any data analysis workflow that helps us reduce the dimension of the dataset quickly and ensure data quality. In Python, we can find duplicate values in an attribute table very easily with Pandas. Dropping those duplicate features, however, requires a few more lines of code.

Feature-engine aims to accelerate the process of data validation by finding and removing duplicate features with the `DropDuplicateFeatures()` class, which is part of the selection API.

`DropDuplicateFeatures()` does exactly that; it finds and removes duplicated variables from a dataframe. DropDuplicateFeatures() will automatically evaluate all variables, or alternatively, you can pass a list with the variables you wish to have examined. And it works with numerical and categorical features alike.

So let's see how to set up `DropDuplicateFeatures()`.

Example

In this demo, we will use the Titanic dataset and introduce a few duplicated features manually:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from feature_engine.datasets import load_titanic
from feature_engine.selection import DropDuplicateFeatures

data = load_titanic(
    handle_missing=True,
    predictors_only=True,
)

# Lets duplicate some columns
data = pd.concat([data, data[['sex', 'age', 'sibsp']]], axis=1)
data.columns = ['pclass', 'survived', 'sex', 'age',
                'sibsp', 'parch', 'fare','cabin', 'embarked',
                'sex_dup', 'age_dup', 'sibsp_dup']
```

We then split the data into a training and a testing set:

```
# Separate into train and test sets
X_train, X_test, y_train, y_test = train_test_split(
    data.drop(['survived'], axis=1),
    data['survived'],
    test_size=0.3,
    random_state=0,
)

print(X_train.head())
```

Below we see the resulting data:

```
      pclass     sex        age  sibsp  parch     fare    cabin embarked  \
501        2  female  13.000000      0      1  19.5000  Missing        S
588        2  female   4.000000      1      1  23.0000  Missing        S
402        2  female  30.000000      1      0  13.8583  Missing        C
1193       3    male  29.881135      0      0   7.7250  Missing        Q
686        3  female  22.000000      0      0   7.7250  Missing        Q

     sex_dup    age_dup  sibsp_dup
501   female  13.000000          0
588   female   4.000000          1
402   female  30.000000          1
1193    male  29.881135          0
686   female  22.000000          0
```

As expected, the variables `sex` and `sex_dup` have duplicate field values throughout all the rows. The same is true for the variables `age` and `age_dup`.

Now, we set up *DropDuplicateFeatures()* to find the duplicate features:

```
transformer = DropDuplicateFeatures()
```

With `fit()` the transformer finds the duplicated features:

```
transformer.fit(X_train)
```

The features that are duplicated and will be removed are stored in the `features_to_drop_` attribute:

```
transformer.features_to_drop_
```

```
{'age_dup', 'sex_dup', 'sibsp_dup'}
```

With `transform()` we remove the duplicated variables:

```
train_t = transformer.transform(X_train)
test_t = transformer.transform(X_test)
```

We can go ahead and check the variables in the transformed dataset, and we will see that the duplicated features are not there any more:

```
train_t.columns
```

```
Index(['pclass', 'sex', 'age', 'sibsp', 'parch', 'fare', 'cabin', 'embarked'], dtype=
↪'object')
```

The transformer also stores the groups of duplicated features, which is useful for data analysis and validation.

```
transformer.duplicated_feature_sets_
```

```
[{'sex', 'sex_dup'}, {'age', 'age_dup'}, {'sibsp', 'sibsp_dup'}]
```

Additional resources

In this Kaggle kernel we use *DropDuplicateFeatures()* in a pipeline with other feature selection algorithms:

- Kaggle kernel

For more details about this and other feature selection methods check out these resources:

Or read our book:

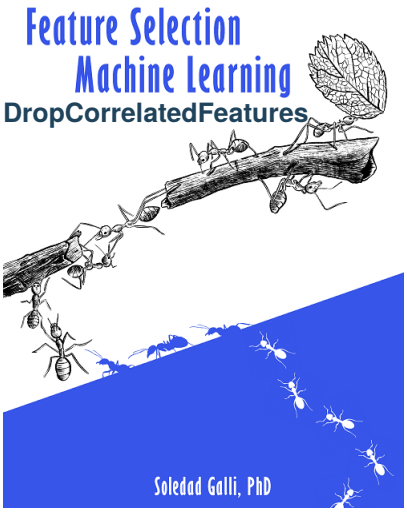


Fig. 80: Feature Selection in Machine Learning

Both our book and course are suitable for beginners and more advanced data scientists alike. By purchasing them you are supporting Sole, the main developer of Feature-engine.

The *DropCorrelatedFeatures()* finds and removes correlated variables from a dataframe. Correlation is calculated with `pandas.corr()`. All correlation methods supported by `pandas.corr()` can be used in the selection, including Spearman, Kendall, or Spearman. You can also pass a bespoke correlation function, provided it returns a value between -1 and 1.

Features are removed on first found first removed basis, without any further insight. That is, the first feature will be retained an all subsequent features that are correlated with this, will be removed.

The transformer will examine all numerical variables automatically. Note that you could pass a dataframe with categorical and datetime variables, and these will be ignored automatically. Alternatively, you can pass a list with the variables you wish to evaluate.

Example

Let's create a toy dataframe where 4 of the features are correlated:

```
import pandas as pd
from sklearn.datasets import make_classification
from␣
↪feature_engine.selection import DropCorrelatedFeatures

# make dataframe with some correlated variables
def make_data():
    X, y = make_classification(n_samples=1000,
                               n_features=12,
                               n_redundant=4,
                               n_clusters_per_class=1,
                               weights=[0.50],
                               class_sep=2,
```

(continues on next page)

(continued from previous page)

```
                                random_state=1)

    # trasform arrays into pandas df and series
    colnames = ['var_'+str(i) for i in range(12)]
    X = pd.DataFrame(X, columns =colnames)
    return X

X = make_data()
```

Now, we set up *DropCorrelatedFeatures()* to find and remove variables which (absolute) correlation coefficient is bigger than 0.8:

```
tr = DropCorrelatedFeatures(variables=None, method='pearson', threshold=0.8)
```

With `fit()` the transformer finds the correlated variables and with `transform()` it drops them from the dataset:

```
Xt = tr.fit_transform(X)
```

The correlated feature groups are stored in the transformer's attributes:

```
tr.correlated_feature_sets_
```

```
[{'var_0', 'var_8'}, {'var_4', 'var_6', 'var_7', 'var_9'}]
```

We can identify from each group which feature will be retained and which ones removed by inspecting the dictionary:

```
tr.correlated_feature_dict_
```

In the dictionary below we see that from the first correlated group, `var_0` is a key, hence it will be retained, whereas `var_8` is a value, which means that it is correlated to `var_0` and will therefore be removed.

```
{'var_0': {'var_8'}, 'var_4': {'var_6', 'var_7', 'var_9'}}
```

Similarly, `var_4` is a key and will be retained, whereas the variables 6, 7 and 8 were found correlated to `var_4` and will therefore be removed.

The features that will be removed from the dataset are stored in a different attribute as well:

```
tr.features_to_drop_
```

```
['var_8', 'var_6', 'var_7', 'var_9']
```

If we now go ahead and print the transformed data, we see that the correlated features have been removed.

```
print(print(Xt.head()))
```

```
      var_0     var_1     var_2     var_3     var_4     var_5    var_10  \
0  1.471061 -2.376400 -0.247208  1.210290 -3.247521  0.091527  2.070526
1  1.819196  1.969326 -0.126894  0.034598 -2.910112 -0.186802  1.184820
2  1.625024  1.499174  0.334123 -2.233844 -3.399345 -0.313881 -0.066448
3  1.939212  0.075341  1.627132  0.943132 -4.783124 -0.468041  0.713558
4  1.579307  0.372213  0.338141  0.951526 -3.199285  0.729005  0.398790
```

(continues on next page)

(continued from previous page)

```
     var_11
0 -1.989335
1 -1.309524
2 -0.852703
3  0.484649
4 -0.186530
```

Additional resources

In this notebook, we show how to use *DropCorrelatedFeatures()* with a different relation metric:

- Jupyter notebook

All notebooks can be found in a dedicated repository.

For more details about this and other feature selection methods check out these resources:

Or read our book:

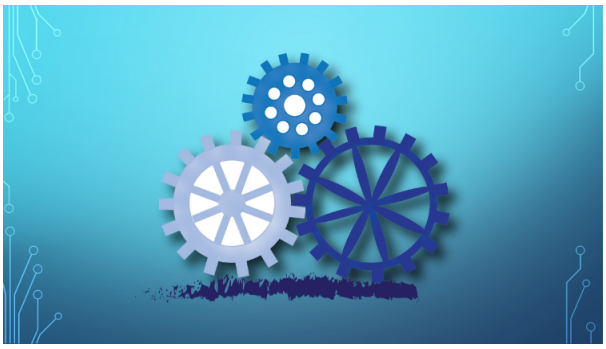


Fig. 81: Feature Selection for Machine Learning

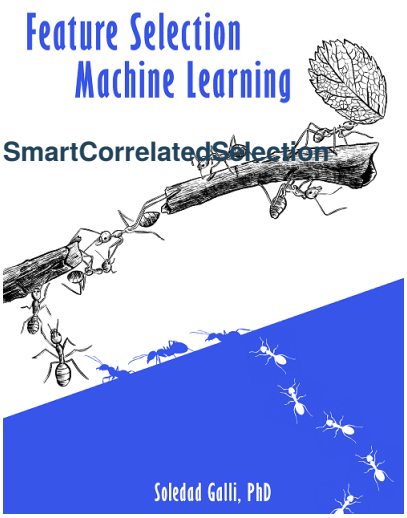


Fig. 82: Feature Selection in Machine Learning

Both our book and course are suitable for beginners and more advanced data scientists alike. By purchasing them you are supporting Sole, the main developer of Feature-engine.

When dealing with datasets containing numerous features, it's common for more than two features to exhibit correlations with each other. This correlation might manifest among three, four, or even more features within the dataset. Consequently, determining which features to retain and which ones to eliminate becomes a crucial consideration.

Deciding which features to retain from a correlated group involves several strategies, such us:

1. **Model Performance**: Some features returns model with higher performance than others.
2. **Variability and Cardinality**: Features with higher variability or cardinality often provide more information about the target variable.
3. **Missing Data**: Features with less missing data are generally more reliable and informative.

We can apply this selection strategies out of the box with the *SmartCorrelatedSelection*.

From a group of correlated variables, the *SmartCorrelatedSelection* will retain the variable with:

- the highest variance
- the highest cardinality
- the least missing data
- the best performing model (based on a single feature)

The remaining features within each correlated group will be dropped.

Features with higher diversity of values (higher variance or cardinality), tend to be more predictive, whereas features with least missing data, tend to be more useful.

Alternatively, directly training a model using each feature within the group and retaining the one that trains the best performing model, directly evaluates the influence of the feature on the target.

Procedure

SmartCorrelatedSelection first finds correlated feature groups using any correlation method supported by `pandas.corr()`, or a user defined function that returns a value between -1 and 1.

Then, from each group of correlated features, it will try and identify the best candidate based on the above criteria.

If the criteria is based on model performance, *SmartCorrelatedSelection* will train a single feature machine learning model, using each one of the features in a correlated group, calculate the model's performance, and select the feature that returned the highest performing model. In simpler words, it trains single feature models, and retains the feature of the highest performing model.

If the criteria is based on variance or cardinality, *SmartCorrelatedSelection* will determine these attributes for each feature in the group and retain that one with the highest. Note however, that variability is dominated by the variable's scale. Hence, **variables with larger scales will dominate the selection procedure**, unless you have a scaled dataset.

If the criteria is based on missing data, *SmartCorrelatedSelection* will determine the number of NA in each feature from the correlated group and keep the one with less NA.

Variance

Let's see how to use *SmartCorrelatedSelection* in a toy example. Let's create a toy dataframe with 4 correlated features:

```
import pandas as pd
from sklearn.datasets import make_classification
from feature_engine.selection import SmartCorrelatedSelection

# make dataframe with some correlated variables
def make_data():
    X, y = make_classification(n_samples=1000,
                               n_features=12,
                               n_redundant=4,
                               n_clusters_per_class=1,
                               weights=[0.50],
                               class_sep=2,
                               random_state=1)

    # transform arrays into pandas df and series
    colnames = ['var_'+str(i) for i in range(12)]
    X = pd.DataFrame(X, columns=colnames)
    return X

X = make_data()
```

Now, we set up *SmartCorrelatedSelection* to find features groups which (absolute) correlation coefficient is >0.8. From these groups, we want to retain the feature with highest variance:

```
# set up the selector
tr = SmartCorrelatedSelection(
    variables=None,
    method="pearson",
    threshold=0.8,
    missing_values="raise",
    selection_method="variance",
    estimator=None,
)
```

With `fit()`, the transformer finds the correlated variables and selects the ones to keep. With `transform()`, it drops the remaining features in the correlated group from the dataset:

```
Xt = tr.fit_transform(X)
```

The correlated feature groups are stored in the one of the transformer's attributes:

```
tr.correlated_feature_sets_
```

In the first group, 4 features are correlated to at least one of them. In the second group, 2 features are correlated.

```
[{'var_4', 'var_6', 'var_7', 'var_9'}, {'var_0', 'var_8'}]
```

SmartCorrelatedSelection picks a feature, and then determines the correlation of other features in the dataframe to it. Hence, all features in a group will be correlated to this one feature, but they may or may not be correlated to the other features within the group, because correlation is not transitive.

This feature that was used in the assessment, was either the one with the higher variance, higher cardinality or smaller number of missing data. Or, if model performance was selected, it was the one that came first in alphabetic order.

We can identify from each group which feature will be retained and which ones removed by inspecting the following attribute:

```
tr.correlated_feature_dict_
```

In the dictionary below we see that from the first correlated group, `var_7` is a key, hence it will be retained, whereas variables 4, 6 and 9 are values, which means that they are correlated to `var_7` and will therefore be removed.

Because we are selecting features based on variability, `var_7` has the higher variability from the group.

```
{'var_7': {'var_4', 'var_6', 'var_9'}, 'var_8': {'var_0'}}
```

Similarly, `var_8` is a key and will be retained, whereas the `var_0` is a value, which means that it was found correlated to `var_8` and will therefore be removed.

We can corroborate that, for example, `var_7` had the highest variability as follows:

```
X[list(tr.correlated_feature_sets_[0])].std()
```

That command returns the following output, where we see that the variability of `var_7` is the highest:

```
var_4    1.810273
var_7    2.159634
var_9    1.764249
var_6    2.032947
dtype: float64
```

The features that will be removed from the dataset are stored in the following attribute:

```
tr.features_to_drop_
```

```
['var_6', 'var_4', 'var_9', 'var_0']
```

If we now go ahead and print the transformed data, we see that the correlated features have been removed.

```
print(Xt.head())
```

```
      var_1     var_2     var_3     var_5     var_7     var_8    var_10  \
0 -2.376400 -0.247208  1.210290  0.091527 -2.230170  2.070483  2.070526
1  1.969326 -0.126894  0.034598 -0.186802 -1.447490  2.421477  1.184820
2  1.499174  0.334123 -2.233844 -0.313881 -2.240741  2.263546 -0.066448
3  0.075341  1.627132  0.943132 -0.468041 -3.534861  2.792500  0.713558
```

(continues on next page)

(continued from previous page)

```
4  0.372213  0.338141  0.951526  0.729005 -2.053965  2.186741  0.398790

     var_11
0 -1.989335
1 -1.309524
2 -0.852703
3  0.484649
4 -0.186530
```

Performance

Let's now select the feature that returns a machine learning model with the highest performance, from each group. We'll use a decision tree.

We start by creating a toy dataframe:

```
import pandas as pd
from sklearn.datasets import make_classification
from sklearn.tree import DecisionTreeClassifier
from feature_engine.selection import SmartCorrelatedSelection

# make dataframe with some correlated variables
def make_data():
    X, y = make_classification(n_samples=1000,
                               n_features=12,
                               n_redundant=4,
                               n_clusters_per_class=1,
                               weights=[0.50],
                               class_sep=2,
                               random_state=1)

    # transform arrays into pandas df and series
    colnames = ['var_'+str(i) for i in range(12)]
    X = pd.DataFrame(X, columns=colnames)
    return X, y

X, y = make_data()
```

Let's now set up the selector:

```
tr = SmartCorrelatedSelection(
    variables=None,
    method="pearson",
    threshold=0.8,
    missing_values="raise",
    selection_method="model_performance",
    estimator=DecisionTreeClassifier(random_state=1),
    scoring='roc_auc',
    cv=3,
)
```

Next, we fit the selector to the data. Here, as we are training a model, we also need to pass the target variable:

```
Xt = tr.fit_transform(X, y)
```

Let's explore the correlated feature groups:

```
tr.correlated_feature_sets_
```

We see that the groups of correlated features are slightly different, because in this cases, the features were assessed in alphabetical order, whereas when we used the variance the features we sorted based on their standard deviation for the assessment.

```
[{'var_0', 'var_8'}, {'var_4', 'var_6', 'var_7', 'var_9'}]
```

We can find the feature that will be retained as the key in the following attribute:

```
tr.correlated_feature_dict_
```

The variables `var_0` and `var_7` will be retained, and the remaining ones will be dropped.

```
{'var_0': {'var_8'}, 'var_7': {'var_4', 'var_6', 'var_9'}}
```

We find the variables that will be dropped in the following attribute:

```
tr.features_to_drop_
```

```
['var_8', 'var_4', 'var_6', 'var_9']
```

And now we can print the resulting dataframe after the transformation:

```
print(Xt.head())
```

```
      var_0     var_1     var_2     var_3     var_5     var_7    var_10  \
0  1.471061 -2.376400 -0.247208  1.210290  0.091527 -2.230170  2.070526
1  1.819196  1.969326 -0.126894  0.034598 -0.186802 -1.447490  1.184820
2  1.625024  1.499174  0.334123 -2.233844 -0.313881 -2.240741 -0.066448
3  1.939212  0.075341  1.627132  0.943132 -0.468041 -3.534861  0.713558
4  1.579307  0.372213  0.338141  0.951526  0.729005 -2.053965  0.398790

     var_11
0 -1.989335
1 -1.309524
2 -0.852703
3  0.484649
4 -0.186530
```

Let's examine other attributes that may be useful. Like with any Scikit-learn transformer we can obtain the names of the features in the resulting dataframe as follows:

```
tr.get_feature_names_out()
```

```
['var_0', 'var_1', 'var_2', 'var_3', 'var_5', 'var_7', 'var_10', 'var_11']
```

We also find the `get_support` method that flags the features that will be retained from the dataframe:

```
tr.get_support()
```

```
[True, True, True, True, False, True, False, True, False, False, True, True]
```

And that's it!

Additional resources

In this notebook, we show how to use *SmartCorrelatedSelection* with a different relation metric:

- Jupyter notebook

All notebooks can be found in a dedicated repository.

For more details about this and other feature selection methods check out these resources:

Or read our book:

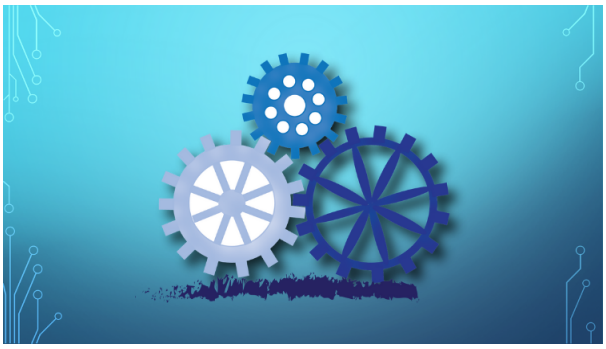
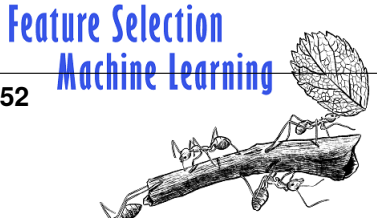


Fig. 83: Feature Selection for Machine Learning

Both our book and course are suitable for beginners and more advanced data



scientists alike. By purchasing them you are supporting Sole, the main developer of Feature-engine.

SelectBySingleFeaturePerformance

The *SelectBySingleFeaturePerformance()* selects features based on the performance of machine learning models trained using individual features. That is, it selects features based on their individual performance. In short, the selection algorithms works as follows:

1. Train a machine learning model per feature (using only 1 feature)
2. Determine the performance metric of choice
3. Retain features which performance is above a threshold

If the parameter `threshold` is left to None, it will select features which performance is above the mean performance of all features.

Example

Let's see how to use this transformer with the diabetes dataset that comes in Scikit-learn. First, we load the data:

```
import pandas as pd
from sklearn.datasets import load_diabetes
from sklearn.linear_model import LinearRegression
from feature_
↪engine.selection import SelectBySingleFeaturePerformance

# load dataset
diabetes_X, diabetes_y = load_diabetes(return_X_y=True)
X = pd.DataFrame(diabetes_X)
y = pd.Series(diabetes_y)
```

Now, we start *SelectBySingleFeaturePerformance()* to select features based on the r2 returned by a Linear regression, using 3 fold cross-validation. We want to select features which r2 > 0.01.

```
# initialize feature selector
sel = SelectBySingleFeaturePerformance(
        estimator=LinearRegression(), scoring="r2", cv=3, threshold=0.01)
```

With `fit()` the transformer fits 1 model per feature, determines the performance and selects the important features:

```
# fit transformer
sel.fit(X, y)
```

The features that will be dropped are stored in an attribute:

```
sel.features_to_drop_
```

```
[1]
```

SelectBySingleFeaturePerformance() also stores the performace of each one of the models, in case we want to study those further:

```
sel.feature_performance_
```

```
{0: 0.029231969375784466,
 1: -0.003738551760264386,
 2: 0.336620809987693,
 3: 0.19219056680145055,
 4: 0.037115559827549806,
 5: 0.017854228256932614,
 6: 0.15153886177526896,
 7: 0.17721609966501747,
 8: 0.3149462084418813,
 9: 0.13876602125792703}
```

With `transform()` we go ahead and remove the features from the dataset:

```
# drop variables
Xt = sel.transform(X)
```

If we now print the transformed data, we see that the features above were removed.

```
print(Xt.head())
```

```
          0         2         3         4         5         6         7  \
0  0.038076  0.061696  0.021872 -0.044223 -0.034821 -0.043401 -0.002592
1 -0.001882 -0.051474 -0.026328 -0.008449 -0.019163  0.074412 -0.039493
2  0.085299  0.044451 -0.005670 -0.045599 -0.034194 -0.032356 -0.002592
3 -0.089063 -0.011595 -0.036656  0.012191  0.024991 -0.036038  0.034309
4  0.005383 -0.036385  0.021872  0.003935  0.015596  0.008142 -0.002592

          8         9
0  0.019907 -0.017646
1 -0.068332 -0.092204
2  0.002861 -0.025930
3  0.022688 -0.009362
4 -0.031988 -0.046641
```

Additional resources

Check also:

- Jupyter notebook

All notebooks can be found in a dedicated repository.

For more details about this and other feature selection methods check out these resources:

For more details about this and other feature selection methods check out these resources:

Or read our book:

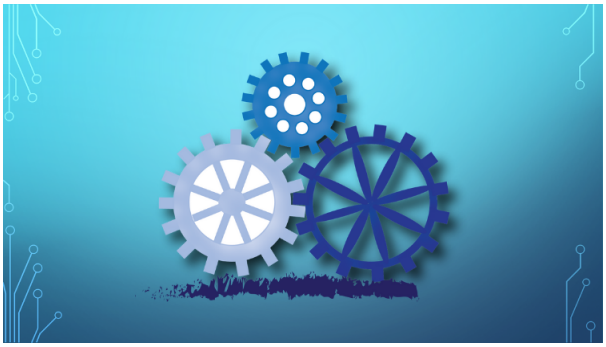


Fig. 85: Feature Selection for Machine Learning

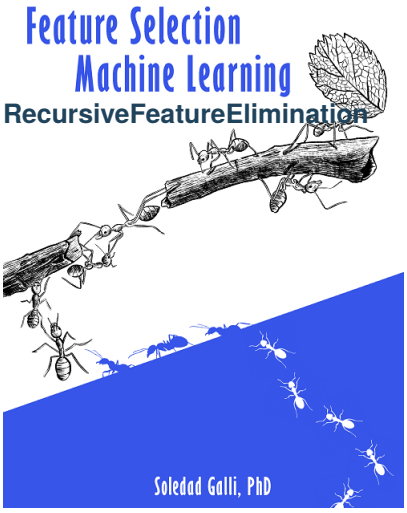


Fig. 86: Feature Selection in Machine Learning

Both our book and course are suitable for beginners and more advanced data scientists alike. By purchasing them you are supporting Sole, the main developer of Feature-engine.

RecursiveFeatureElimination implements recursive feature elimination. Recursive feature elimination (RFE) is a backward feature selection process. In Feature-engine's implementation of RFE, a feature will be kept or removed based on the performance of a machine learning model without that feature. This differs from Scikit-learn's implementation of RFE where a feature will be kept or removed based on the feature importance.

This technique begins by building a model on the entire set of variables, then calculates and stores a model performance metric, and finally computes an importance score for each variable. Features are ranked by the model's `coef_` or `feature_importances_` attributes.

In the next step, the least important feature is removed, the model is re-built, and a new performance metric is determined. If this performance metric is worse than the original one, then, the feature is kept, (because eliminating the feature clearly caused a drop in model performance) otherwise, it removed.

The procedure removes now the second to least important feature, trains a new model, determines a new performance metric, and so on, until it evaluates all

the features, from the least to the most important.

Note that, in Feature-engine's implementation of RFE, the feature importance is used just to rank features and thus determine the order in which the features will be eliminated. But whether to retain a feature is determined based on the decrease in the performance of the model after the feature elimination.

By recursively eliminating features, RFE attempts to eliminate dependencies and collinearity that may exist in the model.

Parameters

Feature-engine's RFE has 2 parameters that need to be determined somewhat arbitrarily by the user: the first one is the machine learning model which performance will be evaluated. The second is the threshold in the performance drop that needs to occur, to remove a feature.

RFE is not machine learning model agnostic, this means that the feature selection depends on the model, and different models may have different subsets of optimal features. Thus, it is recommended that you use the machine learning model that you finally intend to build.

Regarding the threshold, this parameter needs a bit of hand tuning. Higher thresholds will of course return fewer features.

Example

Let's see how to use this transformer with the diabetes dataset that comes in Scikit-learn. First, we load the data:

```
import pandas as pd
from sklearn.datasets import load_diabetes
from sklearn.linear_model import LinearRegression
from feature_engine.selection import RecursiveFeatureElimination

# load dataset
diabetes_X, diabetes_y = load_diabetes(return_X_y=True)
X = pd.DataFrame(diabetes_X)
y = pd.Series(diabetes_y)
```

Now, we set up *RecursiveFeatureElimination* to select features based on the r2 returned by a Linear Regression model, using 3 fold cross-validation. In this case, we leave the parameter `threshold` to the default value which is 0.01.

```
# initialize linear regresion estimator
linear_model = LinearRegression()

# initialize feature selector
tr = RecursiveFeatureElimination(estimator=linear_model, scoring="r2", cv=3)
```

With `fit()` the model finds the most useful features, that is, features that when removed cause a drop in model performance bigger than 0.01. With `transform()`, the transformer removes the features from the dataset.

```
# fit transformer
Xt = tr.fit_transform(X, y)
```

RecursiveFeatureElimination stores the performance of the model trained using all the features in its attribute:

```
# get the initial linear model performance, using all features
tr.initial_model_performance_
```

```
0.488702767247119
```

RecursiveFeatureElimination also stores the change in the performance caused by removing every feature.

```
# Get the performance drift of each feature
tr.performance_drifts_
```

```
{0: -0.0032796652347705235,
 9: -0.00028200591588534163,
 6: -0.0006752869546966522,
 7: 0.00013883578730117252,
 1: 0.011956170569096924,
 3: 0.028634492035512438,
 5: 0.012639090879036363,
 2: 0.06630127204137715,
 8: 0.1093736570697495,
 4: 0.024318093565432353}
```

RecursiveFeatureElimination also stores the features that will be dropped based n the given threshold.

```
# the features to remove
tr.features_to_drop_
```

```
[0, 6, 7, 9]
```

If we now print the transformed data, we see that the features above were removed.

```
print(Xt.head())
```

```
          1         2         3         4         5         8
0  0.050680  0.061696  0.021872 -0.044223 -0.034821  0.019907
1 -0.044642 -0.051474 -0.026328 -0.008449 -0.019163 -0.068332
2  0.050680  0.044451 -0.005670 -0.045599 -0.034194  0.002861
3 -0.044642 -0.011595 -0.036656  0.012191  0.024991  0.022688
4 -0.044642 -0.036385  0.021872  0.003935  0.015596 -0.031988
```

Additional resources

More details on recursive feature elimination in this article:

- Recursive feature elimination with Python

For more details about this and other feature selection methods check out these resources:

For more details about this and other feature selection methods check out these resources:

Or read our book:

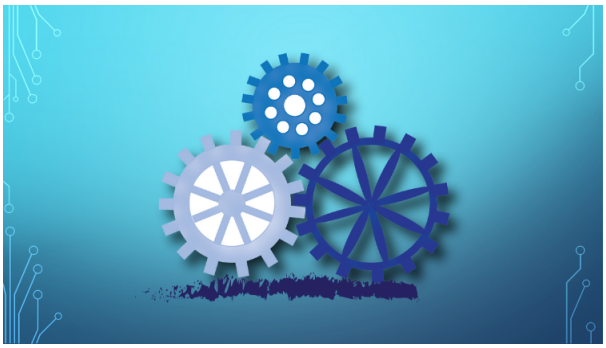


Fig. 87: Feature Selection for Machine Learning

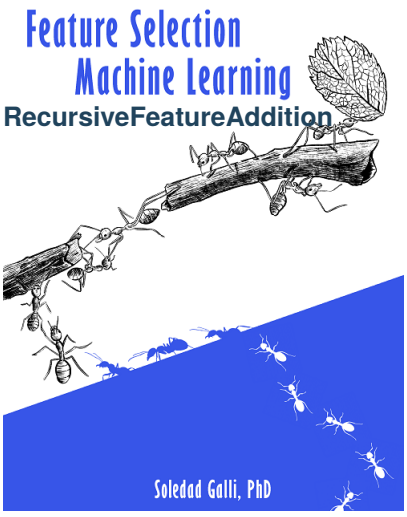


Fig. 88: Feature Selection in Machine Learning

Both our book and course are suitable for beginners and more advanced data scientists alike. By purchasing them you are supporting Sole, the main developer of Feature-engine.

RecursiveFeatureAddition implements recursive feature addition. Recursive feature addition (RFA) is a forward feature selection process.

This technique begins by building a model on the entire set of variables and computing an importance score for each variable. Features are ranked by the model's `coef_` or `feature_importances_` attributes.

In the next step, it trains a model only using the feature with the highest importance and stores the model performance.

Then, it adds the second most important, trains a new model and determines a new performance metric. If the performance increases beyond the threshold, compared to the previous model, then that feature is important and will be kept. Otherwise, that feature is removed.

It proceeds to evaluate the next most important feature, and so on, until all features are evaluated.

Note that feature importance is used just to rank features and thus determine the order in which the features will be added. But whether to retain a feature

is determined based on the increase in the performance of the model after the feature addition.

Parameters

Feature-engine's RFA has 2 parameters that need to be determined somewhat arbitrarily by the user: the first one is the machine learning model which performance will be evaluated. The second is the threshold in the performance increase that needs to occur, to keep a feature.

RFA is not machine learning model agnostic, this means that the feature selection depends on the model, and different models may have different subsets of optimal features. Thus, it is recommended that you use the machine learning model that you finally intend to build.

Regarding the threshold, this parameter needs a bit of hand tuning. Higher thresholds will of course return fewer features.

Example

Let's see how to use this transformer with the diabetes dataset that comes in Scikit-learn. First, we load the data:

```
import pandas as pd
from sklearn.datasets import load_diabetes
from sklearn.linear_model import LinearRegression
from feature_engine.selection import RecursiveFeatureAddition

# load dataset
diabetes_X, diabetes_y = load_diabetes(return_X_y=True)
X = pd.DataFrame(diabetes_X)
y = pd.Series(diabetes_y)
```

Now, we set up *RecursiveFeatureAddition* to select features based on the r2 returned by a Linear Regression model, using 3 fold cross-validation. In this case, we leave the parameter `threshold` to the default value which is 0.01.

```
# initialize linear regresion estimator
linear_model = LinearRegression()

# initialize feature selector
tr = RecursiveFeatureAddition(estimator=linear_model, scoring="r2", cv=3)
```

With `fit()` the model finds the most useful features, that is, features that when added cause an increase in model performance bigger than 0.01. With `transform()`, the transformer removes the features from the dataset.

```
# fit transformer
Xt = tr.fit_transform(X, y)
```

RecursiveFeatureAddition stores the performance of the model trained using all the features in its attribute:

```
# get the initial linear model performance, using all features
tr.initial_model_performance_
```

```
0.488702767247119
```

RecursiveFeatureAddition also stores the change in the performance caused by adding each feature.

```
# Get the performance drift of each feature
tr.performance_drifts_
```

```
{4: 0,
 8: 0.28371458794131676,
 2: 0.1377714799388745,
 5: 0.0023327265047610735,
 3: 0.018759914615172735,
 1: 0.0027996354657459643,
 7: 0.00269514944002163 8,
 6: 0.0026839341346303 06,
 9: 0.000304067408860742,
 0: -0.007387230783454768}
```

RecursiveFeatureAddition also stores the features that will be dropped based n the given threshold.

```
# the features to drop
tr.features_to_drop_
```

```
[0, 1, 5, 6, 7, 9]
```

If we now print the transformed data, we see that the features above were removed.

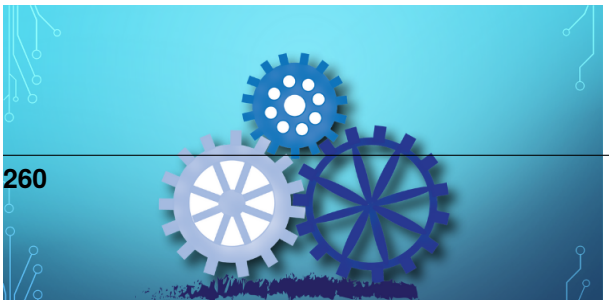
```
print(Xt.head())
```

```
          2         3         4         8
0  0.061696  0.021872 -0.044223  0.019907
1 -0.051474 -0.026328 -0.008449 -0.068332
2  0.044451 -0.005670 -0.045599  0.002861
3 -0.011595 -0.036656  0.012191  0.022688
4 -0.036385  0.021872  0.003935 -0.031988
```

Additional resources

For more details about this and other feature selection methods check out these resources:

Or read our book:



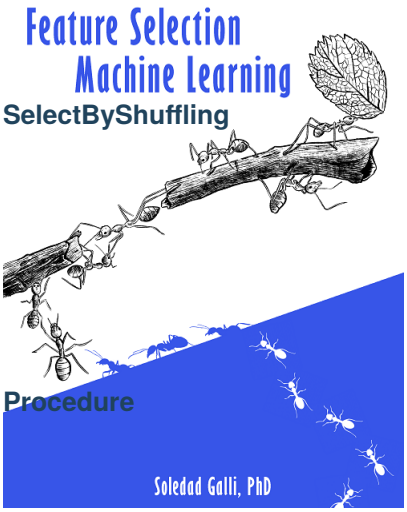


Fig. 90: Feature Selection in Machine Learning

Both our book and course are suitable for beginners and more advanced data scientists alike. By purchasing them you are supporting Sole, the main developer of Feature-engine.

The *SelectByShuffling()* selects important features if a random permutation of their values decreases the model performance. If the feature is predictive, a random shuffle of the values across the rows, should return predictions that are off the truth. If the feature is not predictive, their values should have a minimal impact on the prediction.

The algorithm works as follows:

1. Train a machine learning model using all features
2. Determine a model performance metric of choice
3. Shuffle the order of 1 feature values
4. Use the model trained in 1 to obtain new predictions
5. Determine the performance with the predictions in 4
6. If there is a drop in performance beyond a threshold, keep the feature.
7. Repeat 3-6 until all features are examined.

Example

Let's see how to use this transformer with the diabetes dataset that comes in Scikit-learn. First, we load the data:

```
import pandas as pd
from sklearn.datasets import load_diabetes
from sklearn.linear_model import LinearRegression
from feature_engine.selection import SelectByShuffling
```

(continues on next page)

(continued from previous page)

```
# load dataset
diabetes_X, diabetes_y = load_diabetes(return_X_y=True)
X = pd.DataFrame(diabetes_X)
y = pd.Series(diabetes_y)
```

Now, we set up the model for which we want to have the performance drop evaluated:

```
# initialize linear regresion estimator
linear_model = LinearRegression()
```

Now, we instantiate *SelectByShuffling()* to select features by shuffling, based on the r2 of the model from the previous cell, using 3 fold cross-validation. The parameter `threshold` was left to None, which means that features will be selected if the performance drop is bigger than the mean drop caused by all features.

```
# initialize feature selector
tr = SelectByShuffling(estimator=linear_model, scoring="r2", cv=3)
```

With `fit()` the transformer finds the important variables, that is, those which values permutations caused a drop in the model performance. With `transform()` it drops them from the dataset:

```
# fit transformer
Xt = tr.fit_transform(X, y)
```

SelectByShuffling() stores the performance of the model trained using all the features in its attribute:

```
tr.initial_model_performance_
```

```
0.488702767247119
```

SelectByShuffling() also stores the performance change caused by every single feature after shuffling. In case you are not satisfied with the threshold used, you can get an idea of where the threshold could be by looking at these values:

```
tr.performance_drifts_
```

```
{0: -0.0035681361984126747,
 1: 0.041170843574652394,
 2: 0.1920054944393057,
 3: 0.07007527443645178,
 4: 0.49871458125373913,
 5: 0.1802858704499694,
 6: 0.025536233845966705,
 7: 0.024058931694668884,
 8: 0.40901959802129045,
 9: 0.004487448637912506}
```

SelectByShuffling() also stores the features that will be dropped based on the threshold indicated.

```
tr.features_to_drop_
```

```
[0, 1, 3, 6, 7, 9]
```

If we now print the transformed data, we see that the features above were removed.

```
print(Xt.head())
```

```
          2         4         5         8
0  0.061696 -0.044223 -0.034821  0.019907
1 -0.051474 -0.008449 -0.019163 -0.068332
2  0.044451 -0.045599 -0.034194  0.002861
3 -0.011595  0.012191  0.024991  0.022688
4 -0.036385  0.003935  0.015596 -0.031988
```

Additional resources

For more details about this and other feature selection methods check out these resources:

Or read our book:

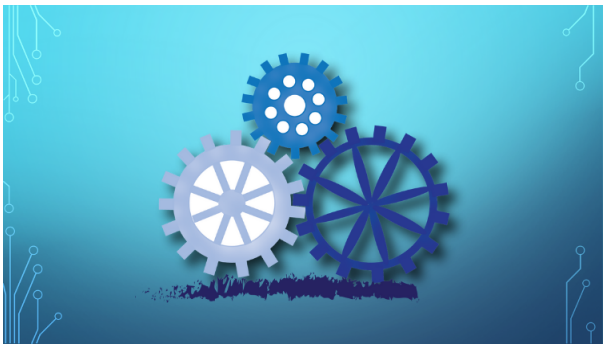
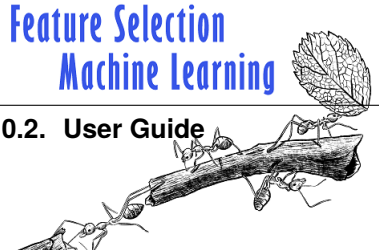


Fig. 91: Feature Selection for Machine Learning



Both our book and course are suitable for beginners and more advanced data scientists alike. By purchasing them you are supporting Sole, the main developer of Feature-engine.

SelectByTargetMeanPerformance

SelectByTargetMeanPerformance() selects features based on performance metrics like the ROC-AUC or accuracy for classification, or mean squared error and R-squared for regression.

To obtain performance metrics, we compare an estimate of the target, returned by a machine learning model, with the real target. The closer the values of the estimate to the real target, the better the performance of the model.

SelectByTargetMeanPerformance(), like *SelectBySingleFeaturePerformance()* train models based on single features. Or in other words, they train and test one model per feature. With *SelectBySingleFeaturePerformance()*, we can use any machine learning classifier or regressor available in Scikit-learn to evaluate each feature's performance. The downside is that Scikit-learn models only work with numerical variables, thus, if our data has categorical variables, we need to encode them into numbers first.

SelectByTargetMeanPerformance(), on the other hand, can select both numerical and categorical variables. *SelectByTargetMeanPerformance()* uses a very simple "machine learning model" to estimate the target. It estimates the target by returning the mean target value per category or per interval. And with this prediction, it determines a performance metric for each feature.

These feature selection idea is very simple; it involves taking the mean of the responses (target) for each level (category or interval), and so amounts to a least squares fit on a single categorical variable against a response variable, with the categories in the continuous variables defined by intervals.

SelectByTargetMeanPerformance() works with cross-validation. It uses the k-1 folds to define the numerical intervals and learn the mean target value per category or interval. Then, it uses the remaining fold to evaluate the performance of the feature: that is, in the last fold it sorts numerical variables into the bins, replaces bins and categories by the learned target estimates, and calculates the performance of each feature.

Despite its simplicity, the method has a number of advantages:

- Speed: Computing means and intervals is fast, straightforward and efficient.
- Stability with respect to feature magnitude: Extreme values for continuous variables do not skew predictions as they would in many models.
- Comparability between continuous and categorical variables.
- Accommodation of non-linearities.
- Does not require encoding categorical variables into numbers.

The method has also some limitations. First, the selection of the number of intervals as well as the threshold is arbitrary. And also, rare categories and very skewed variables will raise errors when NAN are accidentally introduced during the evaluation.

Important

SelectByTargetMeanPerformance() automatically identifies numerical and categorical variables. It will select as categorical variables, those cast as object or categorical, and as numerical variables those of type numeric. Therefore, make sure that your variables are of the correct data type.

Troubleshooting

The main problem that you may encounter using this selector is having missing data introduced in the variables when replacing the categories or the intervals by the target mean estimates.

Categorical variables

NAN are introduced in categorical variables when a category present in the kth fold was not present in the k-1 fold used to calculate the mean target value per category. This is probably due to the categorical variable having high cardinality (a lot of categories) or rare categories, that is, categories present in a small fraction of the observations.

If this happens, try reducing the cardinality of the variable, for example by grouping rare labels into a single group. Check the *RareLabelEncoder* for more details.

Numerical variables

NAN are introduced in numerical variables when an interval present in the kth cross-validation fold was not present in the k-1 fold used to calculate the mean target value per interval. This is probably due to the numerical variable being highly skewed, or having few unique values, for example, if the variable is discrete instead of continuous.

If this happens, check the distribution of the problematic variable and try to identify the problem. Try using equal-frequency intervals instead of equal-width and also reducing the number of bins.

If the variable is discrete and has few unique values, another thing you could do is casting the variable as object, so that the selector evaluates the mean target value per unique value.

Finally, if a numerical variable is truly continuous and not skewed, check that it is not accidentally cast as object.

Example

Let's see how to use this method to select variables in the Titanic dataset. This data has a mix of numerical and categorical variables, then it is a good option to showcase this selector.

Let's import the required libraries and classes, and prepare the titanic dataset:

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split

from feature_engine.datasets import load_titanic
from feature_engine.encoding import RareLabelEncoder
from feature_engine.selection import SelectByTargetMeanPerformance

data = load_titanic(
    handle_missing=True,
    predictors_only=True,
```

(continues on next page)

(continued from previous page)

```
    cabin="letter_only",
)

# replace infrequent cabins by N
data['cabin'] = np.where(data['cabin'].isin(['T', 'G']), 'N', data['cabin'])

# cap maximum values
data['parch'] = np.where(data['parch']>3,3,data['parch'])
data['sibsp'] = np.where(data['sibsp']>3,3,data['sibsp'])

# cast variables as object to treat as categorical
data[['pclass','sibsp','parch']] = data[['pclass','sibsp','parch']].astype('O')

print(data.head())
```

We can see the first 5 rows of data below:

```
  pclass  survived     sex      age sibsp parch      fare cabin embarked
0      1         1  female  29.0000     0     0  211.3375     B        S
1      1         1    male   0.9167     1     2  151.5500     C        S
2      1         0  female   2.0000     1     2  151.5500     C        S
3      1         0    male  30.0000     1     2  151.5500     C        S
4      1         0  female  25.0000     1     2  151.5500     C        S
```

Let's now go ahead and split the data into train and test sets:

```
# separate train and test sets
X_train, X_test, y_train, y_test = train_test_split(
    data.drop(['survived'], axis=1),
    data['survived'],
    test_size=0.1,
    random_state=0)

X_train.shape, X_test.shape
```

We see the sizes of the datasets below:

```
((1178, 8), (131, 8))
```

Now, we set up *SelectByTargetMeanPerformance()*. We will examine the roc-auc using 3 fold cross-validation. We will separate numerical variables into equal-frequency intervals. And we will retain those variables where the roc-auc is bigger than the mean ROC-AUC of all features (default functionality).

```
sel = SelectByTargetMeanPerformance(
    variables=None,
    scoring="roc_auc",
    threshold=None,
    bins=3,
    strategy="equal_frequency",
    cv=3,
    regression=False,
)
```

(continues on next page)

(continued from previous page)

```
sel.fit(X_train, y_train)
```

With `fit()` the transformer:

- replaces categories by the target mean
- sorts numerical variables into equal-frequency bins
- replaces bins by the target mean
- calculates the the roc-auc for each transformed variable
- selects features which roc-auc bigger than the average

In the attribute `variables_` we find the variables that were evaluated:

```
sel.variables_
```

```
['pclass', 'sex', 'age', 'sibsp', 'parch', 'fare', 'cabin', 'embarked']
```

In the attribute `features_to_drop_` we find the variables that were not selected:

```
sel.features_to_drop_
```

```
['age', 'sibsp', 'parch', 'embarked']
```

In the attribute `feature_performance_` we find the ROC-AUC for each feature. Remember that this is the average ROC-AUC in each cross-validation fold:

```
sel.feature_performance_
```

```
{'pclass': 0.668151138112005,
'sex': 0.764831274819234,
'age': 0.535490029737471,
'sibsp': 0.5815934176199077,
'parch': 0.5721327969642238,
'fare': 0.6545985745474006,
'cabin': 0.630092526712033,
'embarked': 0.5765961846034091}
```

The mean ROC-AUC of all features is 0.62, we can calculate it as follows:

```
pd.Series(sel.feature_performance_).mean()

0.6229357428894605
```

So we can see that the transformer correclty selected the features with ROC-AUC above that value.

With `transform()` we can go ahead and drop the features:

```
Xtr = sel.transform(X_test)

Xtr.head()
```

```
     pclass     sex     fare cabin
1139      3    male   7.8958     M
533       2  female  21.0000     M
459       2    male  27.0000     M
1150      3    male  14.5000     M
393       2    male  31.5000     M
```

And finally, we can also obtain the names of the features in the final transformed data:

```
sel.get_feature_names_out()
```

```
['pclass', 'sex', 'fare', 'cabin']
```

Additional resources

Check also:

- Jupyter notebook

All notebooks can be found in a dedicated repository.

For more details about this and other feature selection methods check out these resources: For more details about this and other feature selection methods check out these resources:

Or read our book:

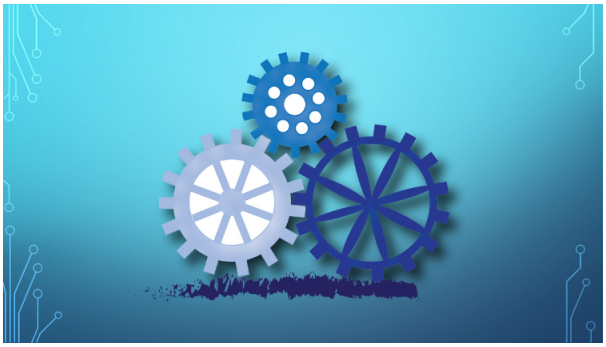


Fig. 93: Feature Selection for Machine Learning

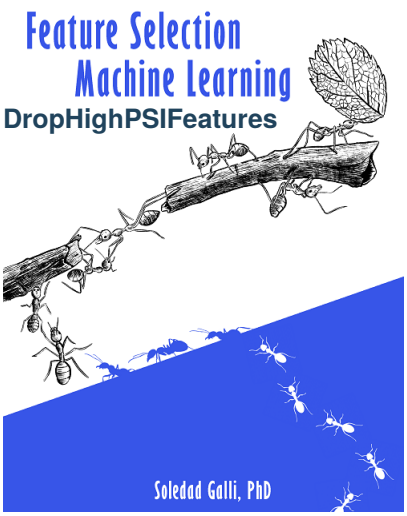


Fig. 94: Feature Selection in Machine Learning

Both our book and course are suitable for beginners and more advanced data scientists alike. By purchasing them you are supporting Sole, the main developer of Feature-engine.

The *DropHighPSIFeatures()* finds and removes features with changes in their distribution, i.e. "unstable values", from a pandas dataframe. The stability of the distribution is computed using the **Population Stability Index (PSI)** and all features having a PSI value above a given threshold are removed.

Unstable features may introduce an additional bias in a model if the training population significantly differs from the population in production. Removing features for which a shift in the distribution is suspected leads to more robust models and therefore to better performance. In the field of Credit Risk modelling, eliminating features with high PSI is common practice and usually required by the Regulator.

Population Stability Index - PSI

The PSI is a measure of how much a population has changed in time or how different the distributions are between two different population samples.

To determine the PSI, continuous features are sorted into discrete intervals, the fraction of observations per interval is then determined, and finally those values are compared between the 2 groups, or as we call them in Feature-engine, between the basis and test sets, to obtain the PSI.

In other words, the PSI is computed as follows:

- Define the intervals into which the observations will be sorted.
- Sort the feature values into those intervals.
- Determine the fraction of observations within each interval.
- Compute the PSI.

The PSI is determined as:

$$PSI = \sum_{i=1}^{n} (test_i - basis_i).ln(\frac{test_i}{basis_i})$$

where `basis` and `test` are the "reference" and "evaluation" datasets, respectively, and `i` refers to the interval.

In other words, the PSI determines the difference in the proportion of observations in each interval, between the reference (aka, original) and test datasets.

In the PSI equation, `n` is the total number of intervals.

Important

When working with the PSI it is worth highlighting the following:

- The PSI is not symmetric; switching the order of the basis and test dataframes in the PSI calculation will lead to different values.
- The number of bins used to define the distributions has an impact on the PSI values.
- The PSI is a suitable metric for numerical features (i.e., either continuous or with high cardinality).
- For categorical or discrete features, the change in distributions is better assessed with Chi-squared.

Threshold

Different thresholds can be used to assess the magnitude of the distribution shift according to the PSI value. The most commonly used thresholds are:

- Below 10%, the variable has not experienced a significant shift.
- Above 25%, the variable has experienced a major shift.
- Between those two values, the shift is intermediate.
- 'auto': the threshold will be calculated based on the size of the base and target datasets and the number of bins.

When 'auto', the threshold is calculated using the chi2 approximation, proposed by B. Yurdakul:

$$threshold = \chi^2_{(q,B-1)} \cdot (\frac{1}{N} + \frac{1}{M})$$

where q is the percentile, B is the number of bins, N is the size of basis dataset, N is the size of test dataset.

In our implementation, we are using the 99.9th percentile.

As mentioned above, the number of bins has an impact on PSI value, because with a higher number of bins it is easier to find divergence in data and vice versa. The same could be said about dataset size - the more data we have, the harder it is to find the difference (if the shift is not drastic). This formula tries to catch these relationships and adjust threshold to correctly detect feature drift.

Procedure

To compute the PSI, the *DropHighPSIFeatures()* splits the input dataset in two: a basis data set (aka the reference data) and a test set. The basis data set is assumed to contain the expected or original feature distributions. The test set will be assessed against the basis data set.

In the next step, the interval boundaries are determined based on the features in the basis or reference data. These intervals can be determined to be of equal with, or equal number of observations.

Next, *DropHighPSIFeatures()* sorts each of the variable values into those intervals, both in the basis and test datasets, and then determines the proportion (percentage) of observations within each interval.

Finally, the PSI is determined as indicated in the previous paragraph for each feature. With the PSI value per feature, *DropHighPSIFeatures()* can now select the features that are unstable and drop them, based on a threshold.

Splitting the data

DropHighPSIFeatures() allows us to determine how much a feature distribution has changed in time, or how much it differs between 2 groups.

If we want to evaluate the distribution change in time, we can use a datetime variable as splitting reference and provide a datetime cut-off as split point.

If we want to compare the distribution change between 2 groups, *DropHighPSIFeatures()* offers 3 different approaches to split the input dataframe:

- Based on proportion of observations.
- Based on proportions of unique observations.
- Using a cut-off value.

Proportion of observations

Splitting by proportion of observations will result in a certain proportion of observations allocated to either the reference and test datasets. For example, if we set `split_frac=0.75`, then 75% and 25% of the observations will be put into the reference and test data, respectively.

If we select this method, we can pass a variable in the parameter `split_col` or leave it to None.

Note that the data split is not done at random, but instead guided by the values in the reference variable indicated in `split_col`. Under the hood, the reference variable indicated in `split_col` is ordered, and the percentage of observations is determined with NumPy quantile. This means that the observations with smaller values in `split_col` will land in the reference dataset, and those with bigger values will go to the test set.

If the rows in your dataset are sorted in time, this could be a good default option to split the dataframe in 2 and compute the PSI. This will for example be the case if your data set contains daily (or any other frequency) sales information on a company's products.

Proportions of unique observations

If we split based on proportion of unique observations, it is important that we indicate which column we want to use as reference in the `split_col` parameter, to make a meaningful split. If we leave this to None, *DropHighPSIFeatures()* will use the dataframe index as reference. This makes sense only if the index in the dataframe has meaningful values.

DropHighPSIFeatures() will first identify the unique values of the variable in `split_col`. Then it will put a certain proportion of those values into the reference dataset and the remaining to the test dataset. The proportion is indicated in the parameter `split_frac`.

Under the hood, *DropHighPSIFeatures()* will sort the unique values of the reference variable, and then use NumPy quantiles to determine the fraction that should be allocated to the reference and test sets. Thus, it is important to consider that the order of the unique values matters in the split.

This split makes sense when we have for example unique customer identifiers and multiple rows per customer in the dataset. We want to make sure that all rows belonging to the same customer are allocated either in the reference or test data, but the same customer cannot be in both data sets. This way of splitting the data will also ensure that we have a certain percentage, indicated in `split_frac` of customers in either data set after the split.

Thus, if `split_frac=0.6` and `split_distinct=True`, *DropHighPSIFeatures()* will send the first 60% of customers to the reference data set, and the remaining 40% to the test set. And it will ensure that rows belonging to the same customer are just in one of the 2 data sets.

Using a cut-off value

We have the option to pass a reference variable to use to split the dataframe using `split_col` and also a cut-off value in the `cut_off` parameter. The cut-off value can be a number, integer or float, a date or a list of values.

If we pass a datetime column in `split_col` and a datetime value in the `cut_off`, we can split the data in a temporal manner. Observations collected before the time indicated will be sent to the reference dataframe, and the remaining to the test set.

If we pass a list of values in the `cut_off` all observations which values are included in the list will go into the reference data set, and the remaining to the test set. This split is useful if we have a categorical variable indicating a portfolio from which the observations have been collected. For example, if we set `split_col='portfolio'` and `cut_off=['port_1', 'port_2']`, all observations that belong to the first and second portfolio will be sent to the reference data set, and the observations from other portfolios to the test set.

Finally, if we pass a number to `cut_off`, all observations which value in the variable indicated in `split_col` is <= cut-off, will be sent to the reference data set, alternatively to the test set. This can be useful for example when dates are defined as integer (for example 20200411) or when using an ordinal customer segmentation to split the dataframe (1: retail customers, 2: private banking customers, 3: SME and 4: Wholesale).

split_col

To split the data set, we recommend that you indicate which column you want to use as reference in the `split_col` parameter. If you don't, the split will be done based on the values of the dataframe index. This might be a good option if the index contains meaningful values or if splitting just based on `split_frac`.

Examples

The versatility of the class lies in the different options to split the input dataframe in a reference or basis data set with the "expected" distributions, and a test set which will be evaluated against the reference.

After splitting the data, *DropHighPSIFeatures()* goes ahead and compares the feature distributions in both data sets by computing the PSI.

To illustrate how to best use *DropHighPSIFeatures()* depending on your data, we provide various examples illustrating the different possibilities.

Case 1: split data based on proportions (split_frac)

In this case, *DropHighPSIFeatures()* will split the dataset in 2, based on the indicated proportion. The proportion is indicated in the `split_frac` parameter. You have the option to select a variable in `split_col` or leave it to None. In the latter, the dataframe index will be used to split.

Let's first create a toy dataframe containing 5 random variables and 1 variable with a shift in its distribution (*var_3* in this case).

```
import pandas as pd
import seaborn as sns

from sklearn.datasets import make_classification
from feature_engine.selection import DropHighPSIFeatures

# Create a dataframe with 500 observations and 6 random variables
```

(continues on next page)

(continued from previous page)

```
X, y = make_classification(
    n_samples=500,
    n_features=6,
    random_state=0
)

colnames = ["var_" + str(i) for i in range(6)]
X = pd.DataFrame(X, columns=colnames)

# Add a column with a shift.
X['var_3'][250:] = X['var_3'][250:] + 1
```

The default approach in *DropHighPSIFeatures()* is to split the input dataframe `X` in two equally sized data sets. You can adjust the proportions by changing the value in the `split_frac` parameter.

For example, let's split the input dataframe into a reference data set containing 60% of the observations and a test set containing 40% of the observations.

```
# Remove the features with high PSI values using a 60-40 split.

transformer = DropHighPSIFeatures(split_frac=0.6)
transformer.fit(X)
```

The value of `split_frac` tells *DropHighPSIFeatures()* to split X according to a 60% - 40% ratio. The `fit()` method performs the split of the dataframe and the calculation of the PSI.

Because we created random variables, these features will have low PSI values (i.e., no distribution change). However, we manually added a distribution shift in the variable *var_3* and therefore expect the PSI for this particular feature to be above the 0.25 PSI threshold.

The PSI values are accessible through the `psi_values_` attribute:

```
transformer.psi_values_
```

The analysis of the PSI values below shows that only feature 3 (called `var_3`) has a PSI above the 0.25 threshold (default value) and will be removed by the `transform` method.

```
{'var_0': 0.07405459925568803,
'var_1': 0.09124093185820083,
'var_2': 0.16985790067687764,
'var_3': 1.342485289730313,
'var_4': 0.0743442762545251,
'var_5': 0.06809060587241555}
```

From the output, we see that the PSI value for *var_0* is around 7%. This means that, when comparing the first 300 and the last 200 observations of the dataframe, there is only a small difference in the distribution of the *var_0* feature. A similar conclusion applies to *var_1, var_2, var_4* and *var_5*. Looking at the PSI value for *var_3*, we see that it exceeds by far the 0.25 threshold. We can then conclude the population of this feature has shifted and it is wise not to include it in the feature set for modelling.

The cut-off value used to split the dataframe is stored in the `cut_off_` attribute:

```
transformer.cut_off_
```

This yields the following answer

```
299.4
```

The value of 299.4 means that observations with index from 0 to 299 are used to define the basis data set. This corresponds to 60% (300 / 500) of the original dataframe (X). The value of 299.4 may seem strange because it is not one of the value present in (the (index of) the dataframe. Intuitively, we would expect the cut_off to be an integer in the present case. However, the cut_off is computed using quantiles and the quantiles are computed using extrapolation.

Splitting with proportions will order the index or the reference column first, and then determine the data that will go into each dataframe. In other words, the order of the index or the variable indicated in `split_col` matters. Observations with the lowest values will be sent to the basis dataframe and the ones with the highest values to the test set.

The `features_to_drop_` attribute provides the list with the features to be dropped when executing the `transform` method.

The command

```
transformer.features_to_drop_
```

Yields the following result:

```
['var_3']
```

That the *var_3* feature is dropped during the procedure is illustrated when looking at the columns from the `X_transformed` dataframe.

```
X_transformed = transformer.transform(X)

X_transformed.columns

Index(['var_0', 'var_1', 'var_2', 'var_4', 'var_5'], dtype='object')
```

DropHighPSIFeatures() also contains a `fit_transform` method that combines the `fit` and the `transform` methods.

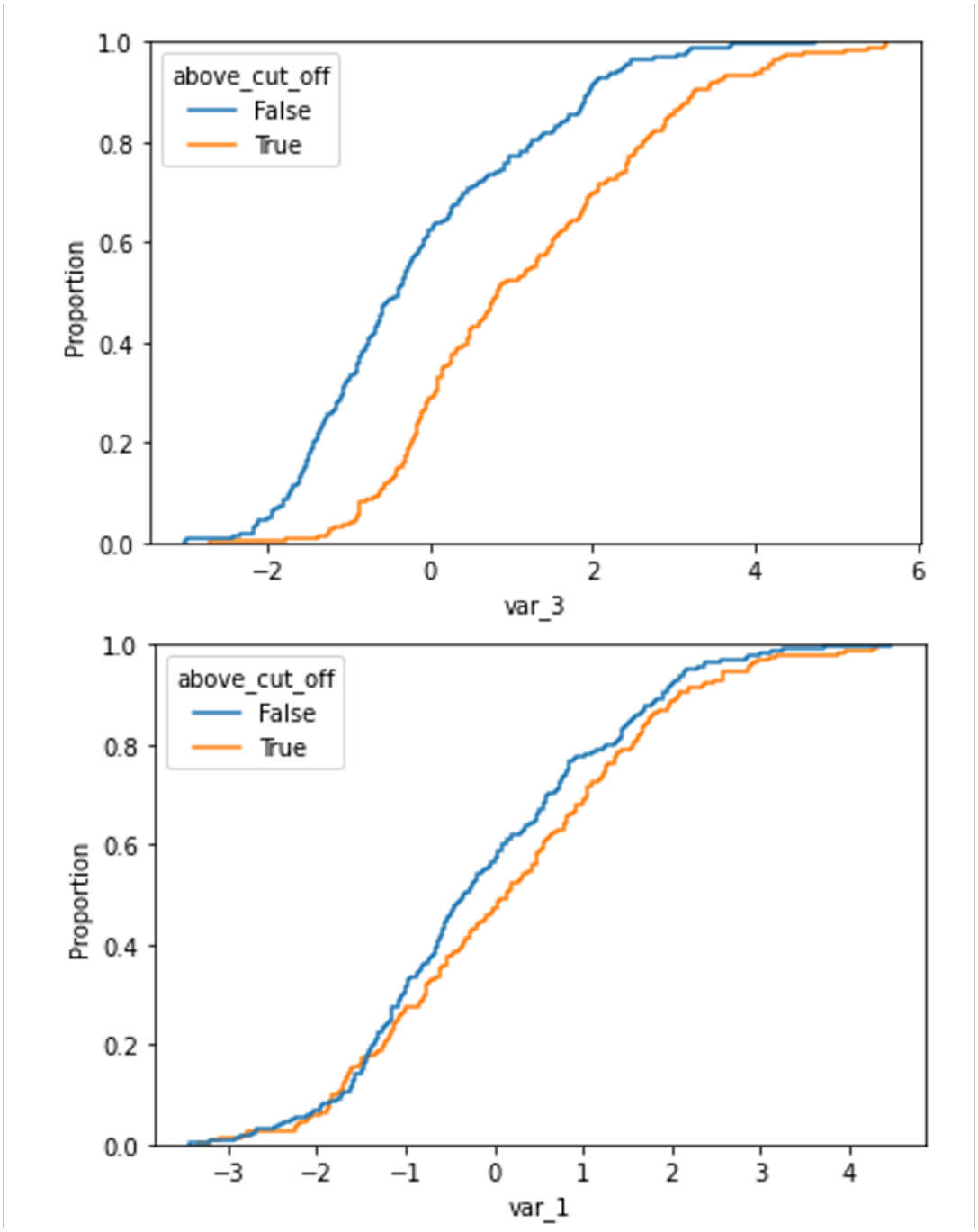
The difference in distribution between a non-shifted and a shifted distribution is clearly visible when plotting the cumulative density function.

For the shifted variable:

```
X['above_cut_off'] = X.index > transformer.cut_off_
sns.ecdfplot(data=X, x='var_3', hue='above_cut_off')
```

and a non-shifted variable (for example *var_1*)

```
sns.ecdfplot(data=X, x='var_1', hue='above_cut_off')
```


above_cut_off
False
True
Proportion
1.0
0.8
0.6
0.4
0.2
0.0
−2
0
2
4
6
var_3
above_cut_off
False
True
Proportion
1.0
0.8
0.6
0.4
0.2
0.0
−3
−2
−1
0
1
2
3
4
var_1

Case 2: split data based on variable (numerical cut_off)

In the previous example, we wanted to split the input dataframe in 2 datasets, with the reference dataset containing 60% of the observations. We let *DropHighPSIFeatures()* find the cut-off to achieve this.

We can instead, provide ourselves the numerical cut-off that determines which observations will go to the reference or basis data set, and which to the test set. Using the `cut_off` parameter, we can define the specific threshold for the split.

A real life example for this case is the use of the customer ID or contract ID to split the dataframe. These IDs are often increasing in value over time which justifies their use to assess distribution shifts in the features.

Let's create a toy dataframe representing the customers' characteristics of a company. This dataset contains six random variables (in real life this are variables like age or postal code), the seniority of the customer (i.e. the number of months since the start of the relationship between the customer and the company) and the customer ID (i.e. the number (integer) used to identify the customer). Generally the customer ID grows over time which means that early customers have a lower customer ID than late customers.

From the definition of the variables, we expect the *seniority* to increase with the customer ID and therefore to have a high PSI value when comparing early and late customer,

```
import pandas as pd
from sklearn.datasets import make_classification
from feature_engine.selection import DropHighPSIFeatures

X, y = make_classification(
        n_samples=500,
        n_features=6,
        random_state=0
    )

colnames = ["var_" + str(i) for i in range(6)]
X = pd.DataFrame(X, columns=colnames)

# Let's add a variable for the customer ID
X['customer_id'] = [customer_id for customer_id in range(1, 501)]

# Add a column with the seniority... that is related to the customer ID
X['seniority'] = 100 - X['customer_id'] // 10

transformer = DropHighPSIFeatures(split_col='customer_id', cut_off=250)
transformer.fit(X)
```

In this case, *DropHighPSIFeatures()* will allocate in the basis or reference data set, all observations which values in `customer_id` are <= 250. The test dataframe contains the remaining observations.

The method `fit()` will determine the PSI values, which are stored in the class:

```
transformer.psi_values_
```

We see that *DropHighPSIFeatures()* does not provide any PSI value for the `customer_id` feature, because this variable was used as a reference to split the data.

```
{'var_0': 0.07385590683974477,
'var_1': 0.06115637727757485,
'var_2': 0.1736694458621651,
'var_3': 0.044965387331530465,
```

(continues on next page)

(continued from previous page)

```
'var_4': 0.0904519893659045,
'var_5': 0.027545195437270797,
'seniority': 7.8688986006052035}
```

```
transformer.features_to_drop_
```

Gives

```
['seniority']
```

Executing the dataframe transformation leads to the exclusion of the *seniority* feature but not to the exclusion of the *customer_id*.

```
X_transformed = transformer.transform(X)

X_transformed.columns

Index(['var_0', 'var_1', 'var_2', 'var_3', 'var_4', 'var_5', 'customer_id'], dtype=
↪'object')
```

Case 3: split data based on time (date as cut_off)

DropHighPSIFeatures() can handle different types of `split_col` variables. The following case illustrates how it works with a date variable. In fact, we often want to determine if the distribution of a feature changes in time, for example after a certain event like the start of the Covid-19 pandemic.

This is how to do it. Let's create a toy dataframe with 6 random numerical variables and two date variables. One will be use to specific the split of the dataframe while the second one is expected to have a high PSI value.

```
import pandas as pd
from datetime import date
from sklearn.datasets import make_classification
from feature_engine.selection import DropHighPSIFeatures

X, y = make_classification(
        n_samples=1000,
        n_features=6,
        random_state=0
    )

colnames = ["var_" + str(i) for i in range(6)]
X = pd.DataFrame(X, columns=colnames)

# Add two time variables to the dataframe
X['time'] = [date(year, 1, 1) for year in range(1000, 2000)]
X['century'] = X['time'].apply(lambda x: ((x.year - 1) // 100) + 1)

# Let's shuffle the dataframe and reset the index to remove the correlation
# between the index and the time variables.

X = X.sample(frac=1).reset_index(drop=True)
```

Dropping features with high PSI values comparing two periods of time is done simply by providing the name of the column with the date and a cut-off date. In the example below the PSI calculations will be done comparing the periods up to the French revolution and after.

```
transformer = DropHighPSIFeatures(split_col='time', cut_off=date(1789, 7, 14))
transformer.fit(X)
```

Important: if the date variable is in pandas or NumPy datetime format, you may need to pass the cut_off value as `pd.to_datetime(1789-07-14)`.

The PSI values shows the *century* variables in unstable as its value is above the 0.25 threshold.

```
transformer.psi_values_

{'var_0': 0.0181623637463045,
'var_1': 0.10595496570984747,
'var_2': 0.05425659114295842,
'var_3': 0.09720689210928271,
'var_4': 0.07917647542638032,
'var_5': 0.10122468631060424,
'century': 8.272395772368412}
```

The class has correctly identified the feature to be dropped.

```
transformer.features_to_drop_

['century']
```

And the transform method correctly removes the feature.

```
X_transformed = transformer.transform(X)

X_transformed.columns

Index(['var_0', 'var_1', 'var_2', 'var_3', 'var_4', 'var_5', 'time'], dtype='object')
```

The difference in distribution between a non-shifted and a shifted distribution is clearly visible when plotting the cumulative density function for each of the group.

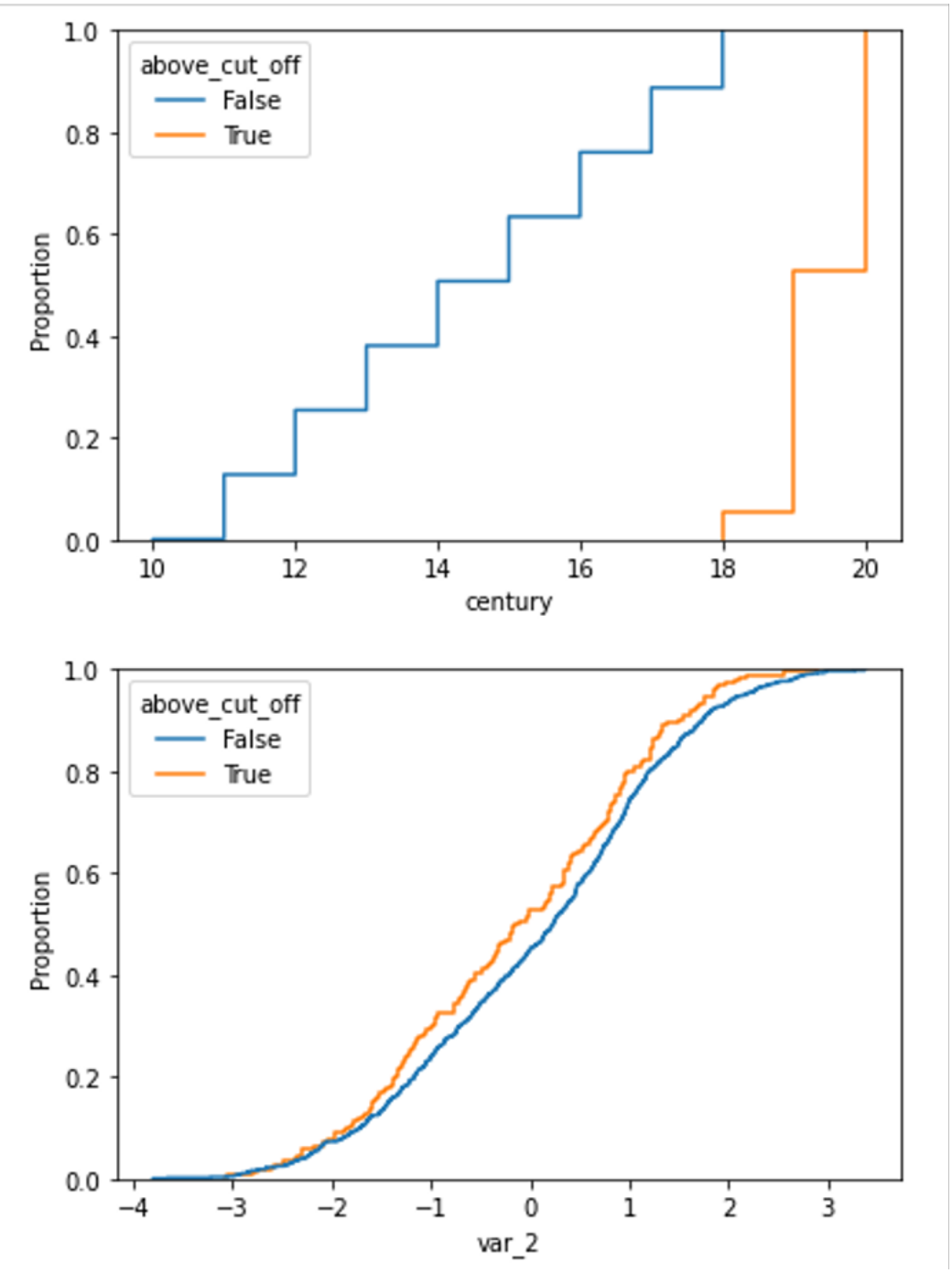
We can plot the cumulative distribution of the shifted variable like this:

```
X['above_cut_off'] = X.time > pd.to_datetime(transformer.cut_off_)
sns.ecdfplot(data=X, x='century', hue='above_cut_off')
```

and the distribution of a non-shifted variable, for example *var_2*, like this:

```
sns.ecdfplot(data=X, x='var_2', hue='above_cut_off')
```

And below we can compare both plots:


above_cut_off
False
True
Proportion
1.0
0.8
0.6
0.4
0.2
0.0
10
12
14
16
18
20
century
above_cut_off
False
True
Proportion
1.0
0.8
0.6
0.4
0.2
0.0
−4
−3
−2
−1
0
1
2
3
var_2

Case 4: split data based on a categorical variable (category or list as cut_off)

DropHighPSIFeatures() can also split the original dataframe based on a categorical variable. The cut-off can then be defined in two ways:

- Using a single string.
- Using a list of values.

In the first case, the column with the categorical variable is sorted alphabetically and the split is determined by the cut-off. We recommend being very careful when using a single category as cut-off, because alphabetical sorting in combination with a cut-off does not always provide obvious results. In other words, for this way of splitting the data to be meaningful, the alphabetical order of the categories in the reference variable should have an intrinsic meaning.

A better purpose for splitting the data based on a categorical variable would be to pass a list with the values of the variable that want in the reference dataframe. A real life example for this case is the computation of the PSI between different customer segments like 'Retail', 'SME' or 'Wholesale'. In this case, if we indicate ['Retail'] as cut-off, observations for Retail will be sent to the basis data set, and those for 'SME' and 'Wholesale' will be added to the test set.

Split passing a category value

Let's show how to set up the transformer in this case. The example data set contains 6 randoms variables, a categorical variable with the labels of the different categories and 2 category related features.

```
import pandas as pd
import seaborn as sns

from sklearn.datasets import make_classification
from feature_engine.selection import DropHighPSIFeatures

X, y = make_classification(
    n_samples=1000,
    n_features=6,
    random_state=0
)

colnames = ["var_" + str(i) for i in range(6)]
X = pd.DataFrame(X, columns=colnames)

# Add a categorical column
X['group'] = ["A", "B", "C", "D", "E"] * 200

# And two category related features
X['group_means'] = X.group.map({"A": 1, "B": 2, "C": 0, "D": 1.5, "E": 2.5})
X['shifted_feature'] = X['group_means'] + X['var_2']
```

We can define a simple cut-off value (for example the letter C). In this case, observations with values that come before C, alphabetically, will be allocated to the reference data set.

```
transformer = DropHighPSIFeatures(split_col='group', cut_off='C')
X_transformed = transformer.fit_transform(X)
```

The PSI values are provided in the `psi_values_` attribute.

```
transformer.psi_values_

{'var_0': 0.06485778974895254,
'var_1': 0.03605540598761757,
'var_2': 0.040632784917352296,
'var_3': 0.023845405645510645,
'var_4': 0.028007185972248064,
'var_5': 0.07009152672971862,
'group_means': 6.601444547497699,
'shifted_feature': 0.48428009522119164}
```

From these values we see that the last 2 features should be removed. We can corroborate that in the `features_to_drop_` attribute:

```
transformer.features_to_drop_

['group_means', 'shifted_feature']
```

And these columns are removed from the original dataframe by the transform method that, in the present case, has been applied through the fit_transform method a couple of block cells above.

```
X_transformed.columns

Index(['var_0', 'var_1', 'var_2', 'var_3', 'var_4', 'var_5', 'group'], dtype='object')
```

Split passing a list of categories

Instead of passing a category value, we can instead pass a list of values to the `cut_off`. Using the same data set let's set up the *DropHighPSIFeatures()* to split the dataframe according to the list ['A', 'C', 'E'] for the categorical variable *group*.

In this case, the PSI's will be computed by comparing two dataframes: the first one containing only the values A, C and E for the *group* variable and the second one containing only the values B and D.

```
trans = DropHighPSIFeatures(split_col='group', cut_off=['A', 'C', 'E'])
X_no_drift = trans.fit_transform(X)
```

```
trans.psi_values_

'var_0': 0.04322345673014104,
'var_1': 0.03534439253617049,
'var_2': 0.05220272785661243,
'var_3': 0.04550964862452317,
'var_4': 0.04492720670343145,
'var_5': 0.044886435640028144,
'group_means': 6.601444547497699,
'shifted_features': 0.3683642099948127}
```

Here again, the object will remove the *group_means* and the *shifted_features* columns from the dataframe.

```
trans.features_to_drop_

['group_means', 'shifted_features']
```

And these columns are removed from the original dataframe by the transform method that has been applied through the `fit_transform` method.

```
X_transformed.columns

Index(['var_0', 'var_1', 'var_2', 'var_3', 'var_4', 'var_5', 'group'], dtype='object')
```

In the following plots, we can compare the distribution of a feature with high PSI and one with low PSI, in the different categories of the categorical variable.

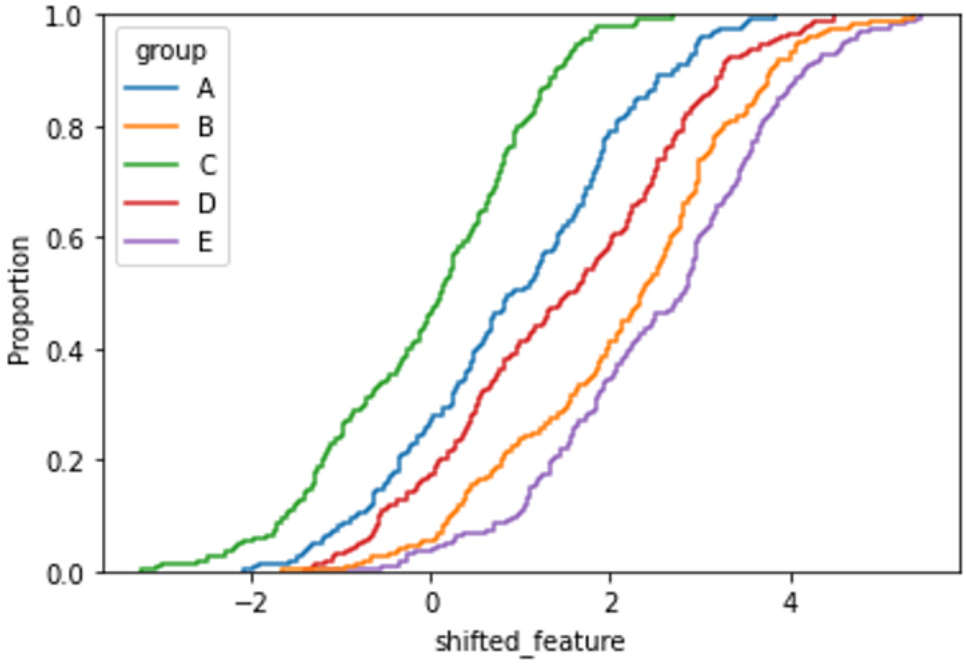
With this code we plot the cumulative distribution of a feature which distribution is different among the different categories of the variable:

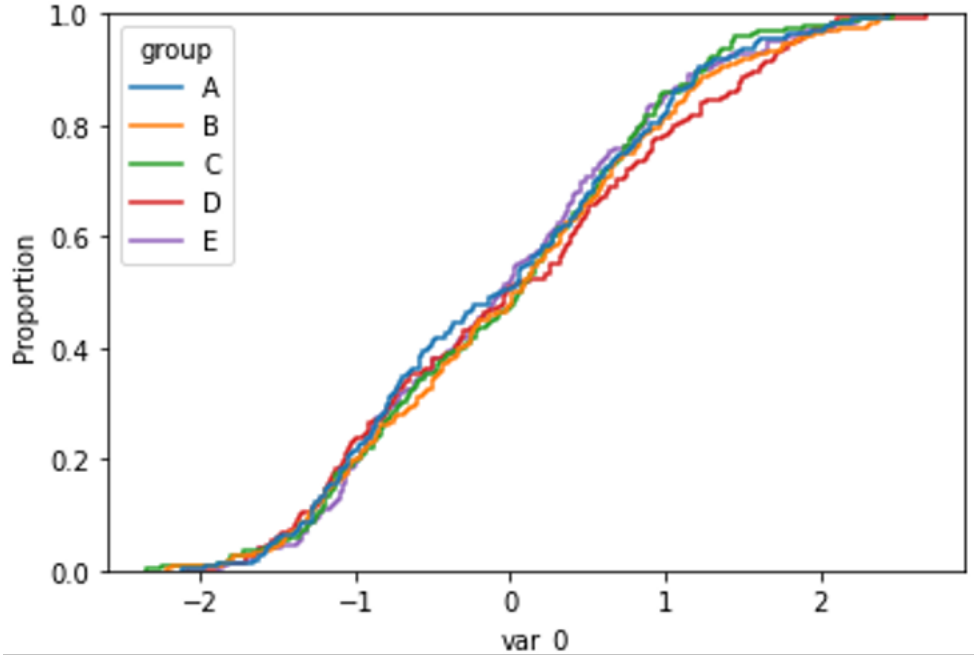
```
sns.ecdfplot(data=X, x='shifted_feature', hue='group')
```

With this code we plot the cumulative distribution of a feature which distribution is the same across the different categories of the categorical variable:

```
sns.ecdfplot(data=X, x='var_0', hue='group')
```

And below we can compare the plots of both features:


group
A
B
C
D
E
Proportion
shifted_feature
1.0
0.8
0.6
0.4
0.2
0.0
−2
0
2
4


group
A
B
C
D
E
Proportion
var_0
1.0
0.8
0.6
0.4
0.2
0.0
−2
−1
0
1
2

Case 5: split data based on unique values (split_distinct)

A variant to the previous example is the use of the `split_distinct` functionality. In this case, the split is not done based on the number observations from `split_col` but from the number of distinct values in the reference variable indicated in `split_col`.

A real life example for this case is when dealing with groups of different sizes like customers income classes ('1000', '2000', '3000', '4000', ...). Split_distinct allows to control the numbers of classes in the basis and test dataframes regardless of the number of observations in each class.

This case is illustrated in the toy data for this case. The data set contains 6 random variable and 1 income variable that is larger for one of the 6 group defined (the F group).

```
import numpy as np
import pandas as pd
import seaborn as sns

from sklearn.datasets import make_classification
from feature_engine.selection import DropHighPSIFeatures

X, y = make_classification(
    n_samples=1000,
    n_features=6,
    random_state=0
)

colnames = ["var_" + str(i) for i in range(6)]
X = pd.DataFrame(X, columns=colnames)

# Add a categorical column
X['group'] = ["A", "B", "C", "D", "E"] * 100 + ["F"] * 500

# And an income variable that is category dependent.
np.random.seed(0)
X['income'] = np.random.uniform(1000, 2000, 500).tolist() +
              np.random.uniform(1250, 2250, 500).tolist()

# Shuffle the dataframe to make the dataset more real life case.
X = X.sample(frac=1).reset_index(drop=True)
```

The `group` column contains 500 observations in the (A, B, C, D, E) group and 500 in the (F) group.

When we pass `split_distinct=True` when initializing the `DropHighPSIFeatures` object, the two dataframes used to compute the PSI will contain the same number of **unique** values in the `group` column (i.e., one dataframe will contain 300 rows associated to groups A, B and C while the other will contain 700 rows associated to groups D, E and F).

```
transformer = DropHighPSIFeatures(split_col='group', split_distinct=True)
transformer.fit(X)

transformer.psi_values_
```

This yields the following PSI values:

```
{'var_0': 0.014825303242393804,
'var_1': 0.03818316821350485,
```

(continues on next page)

(continued from previous page)

```
'var_2': 0.029635981271458896,
'var_3': 0.021700399485890084,
'var_4': 0.061194837255216114,
'var_5': 0.04119583769297253,
'income': 0.46191580731264914}
```

And we can find the feature that will be dropped, income, here:

```
transformer.features_to_drop_

    ['income']
```

The former feature will be removed from the dataset when calling the `transform()` method.

```
X_transformed = transformer.transform(X)

X_transformed.columns

Index(['var_0', 'var_1', 'var_2', 'var_3', 'var_4', 'var_5', 'group'], dtype='object')
```

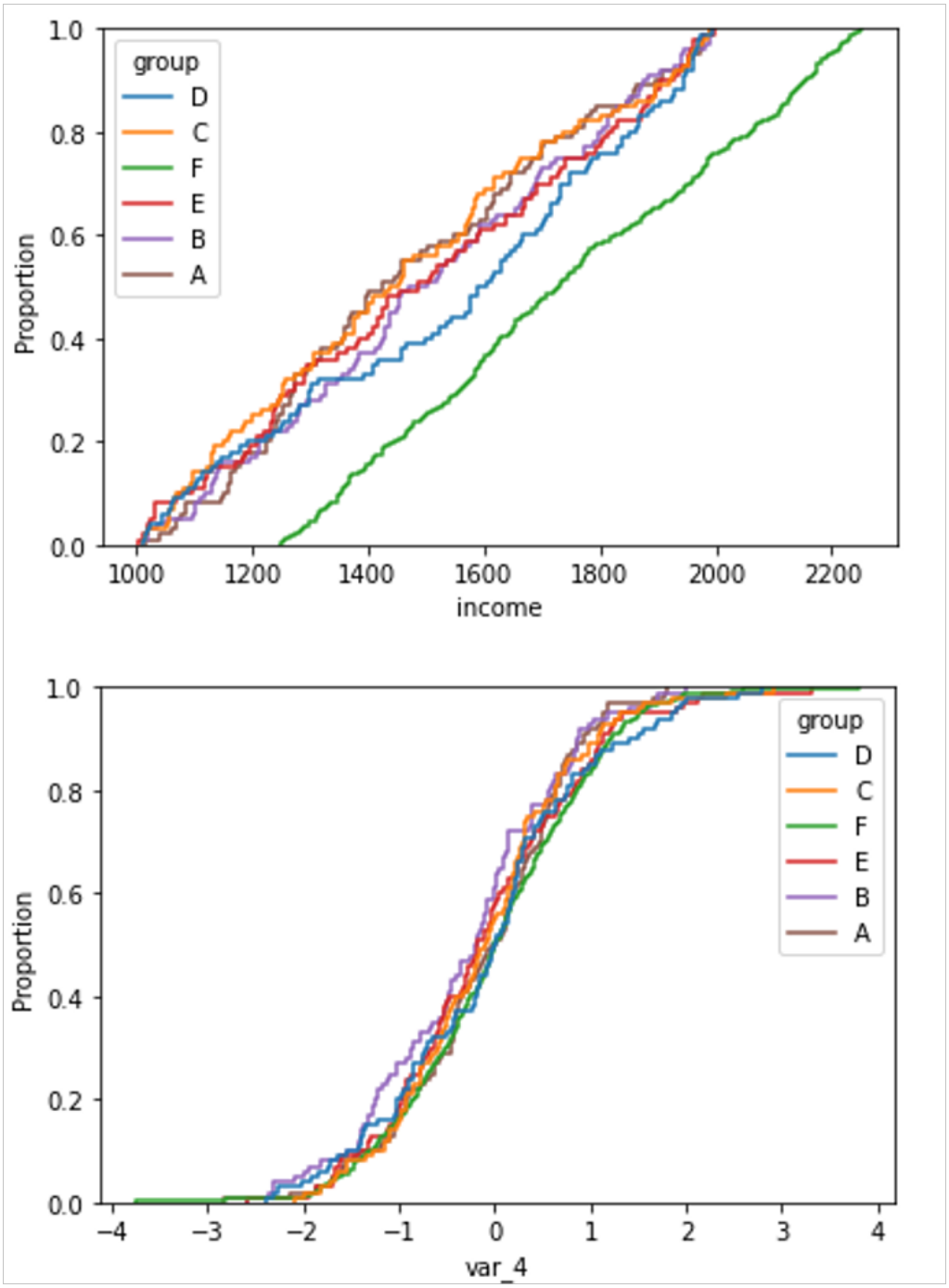
The difference in distribution between a non-shifted and a shifted distribution is clearly visible when plotting the cumulative density function for each of the group.

For the shifted variable (income):

```
sns.ecdfplot(data=X, x='income', hue='group')
```

and a non-shifted variable (for example *var_4*)

```
sns.ecdfplot(data=X, x="var_4", hue="group")
```


group
D
C
F
E
B
A
Proportion
income
var_4

Additional resources

In this notebook, we show how to use `DropHighPSIFeatures` on a real dataset and give more detail about the underlying base and reference sub-dataframes used to determine the PSI.

- Jupyter notebook

All notebooks can be found in a dedicated repository.

For more details about this and other feature selection methods check out these resources:

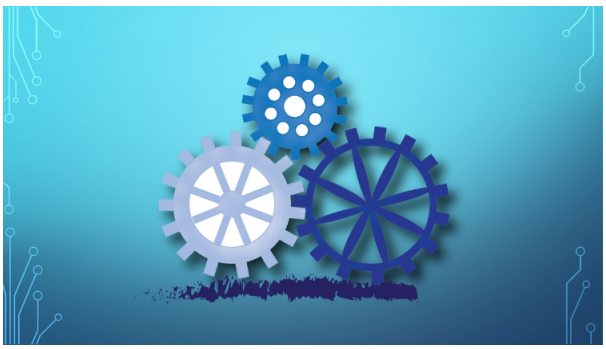
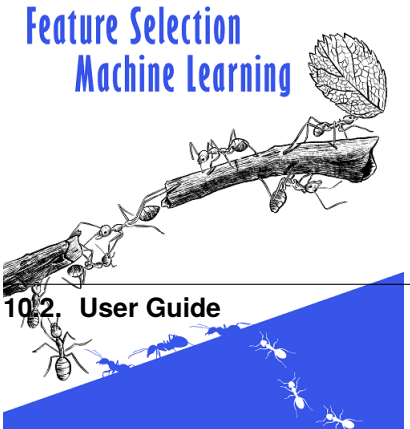


Fig. 95: Feature Selection for Machine Learning

Or read our book:



Both our book and course are suitable for beginners and more advanced data scientists alike. By purchasing them you are supporting Sole, the main developer of Feature-engine.

SelectByInformationValue

`SelectByInformationValue()` selects features based on whether the feature's information value score is greater than the threshold passed by the user.

The IV is calculated as:

$$IV = (fractionofpositivecases - fractionofnegativecases) * WoE$$

where:

- the fraction of positive cases is the proportion of observations of class 1, from the total class 1 observations.
- the fraction of negative cases is the proportion of observations of class 0, from the total class 0 observations.
- WoE is the weight of the evidence.

The WoE is calculated as:

$$WoE = ln(fractionofpositivecases/fractionofnegativecases)$$

Information value (IV) is used to assess a feature's predictive power of a binary-class dependent variable. To derive a feature's IV, the weight of evidence (WoE) must first be calculated for each unique category or bin that comprises the feature. If a category or bin contains a large percentage of true or positive labels compared to the percentage of false or negative labels, then that category or bin will have a high WoE value.

Once the WoE is derived, `SelectByInformationValue()` calculates the IV for each variable. A variable's IV is essentially the weighted sum of the individual WoE values for each category or bin within that variable where the weights incorporate the absolute difference between the numerator and denominator. This value assesses the feature's predictive power in capturing the binary dependent variable.

The table below presents a general framework for using IV to determine a variable's predictive power:

Information Value	Predictive Power
< 0.02	Useless
0.02 to 0.1	Weak
0.1 to 0.3	Medium
0.3 to 0.5	Strong
> 0.5	Suspicious, too good to be true

Table taken from listendata.

Example

Let's see how to use this transformer to select variables from UC Irvine's credit approval data set which can be found here. This dataset concerns credit card applications. All attribute names and values have been changed to meaningless symbols to protect confidentiality.

The data is comprised of both numerical and categorical data.

Let's import the required libraries and classes:

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from feature_engine.selection import SelectByInformationValue
```

Let's now load and prepare the credit approval data:

```
# load data
data = pd.read_csv('crx.data', header=None)

# name variables
var_names = ['A' + str(s) for s in range(1,17)]
data.columns = var_names
data.rename(columns={'A16': 'target'}, inplace=True)

# preprocess data
data = data.replace('?', np.nan)
data['A2'] = data['A2'].astype('float')
data['A14'] = data['A14'].astype('float')
data['target'] = data['target'].map({'+':1, '-':0})

# drop rows with missing data
data.dropna(axis=0, inplace=True)

data.head()
```

Let's now review the first 5 rows of the dataset:

	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	A11	A12	A13	A14	A15	target
0	b	30.83	0.000	u	g	w	v	1.25	t	t	1	f	g	202.0	0	1
1	a	58.67	4.460	u	g	q	h	3.04	t	t	6	f	g	43.0	560	1
2	a	24.50	0.500	u	g	q	h	1.50	t	f	0	f	g	280.0	824	1
3	b	27.83	1.540	u	g	w	v	3.75	t	t	5	t	g	100.0	3	1
4	b	20.17	5.625	u	g	w	v	1.71	t	f	0	f	s	120.0	0	1

Let's now split the data into train and test sets:

```
# separate train and test sets
X_train, X_test, y_train, y_test = train_test_split(
    data.drop(['target'], axis=1),
    data['target'],
    test_size=0.2,
    random_state=0)

X_train.shape, X_test.shape
```

We see the size of the datasets below.

```
((522, 15), (131, 15))
```

Now, we set up *SelectByInformationValue()*. We will pass six categorical variables to the parameter `variables`. We will set the parameter `threshold` to `0.2`. We see from the above mentioned table that an IV score of 0.2 signifies medium predictive power.

```
sel = SelectByInformationValue(
    variables=['A1', 'A6', 'A9', 'A10', 'A12', 'A13'],
    threshold=0.2,
)

sel.fit(X_train, y_train)
```

With `fit()`, the transformer:

- calculates the WoE for each variable
- calculates the the IV for each variable
- identifies the variables that have an IV score below the threshold

In the attribute `variables_`, we find the variables that were evaluated:

```
['A1', 'A6', 'A7', 'A9', 'A10', 'A12', 'A13']
```

In the attribute `features_to_drop_`, we find the variables that were not selected:

```
sel.features_to_drop_

['A1', 'A12', 'A13']
```

The attribute `information_values_` shows the IV scores for each variable.

```
{'A1': 0.0009535686492270659,
 'A6': 0.6006252129425703,
 'A9': 2.9184484098456807,
 'A10': 0.8606638171665587,
 'A12': 0.012251943759377052,
 'A13': 0.04383964979386022}
```

We see that the transformer correctly selected the features that have an IV score greater than the `threshold` which was set to 0.2.

The transformer also has the method `get_support` with similar functionality to Scikit-learn's selectors method. If you execute `sel.get_support()`, you obtain:

```
[False, True, True, True, True, True, True,
 True, True, True, True, False, False, True,
 True]
```

With `transform()`, we can go ahead and drop the features that do not meet the threshold:

```
Xtr = sel.transform(X_test)

Xtr.head()
```

	A2	A3	A4	A5	A6	A7	A8	A9	A10	A11	A14	A15
564	42.17	5.04	u	g	q	h	12.750	t	f	0	92.0	0
519	39.17	1.71	u	g	x	v	0.125	t	t	5	480.0	0
14	45.83	10.50	u	g	q	v	5.000	t	t	7	0.0	0
257	20.00	0.00	u	g	d	v	0.500	f	f	0	144.0	0
88	34.00	4.50	u	g	aa	v	1.000	t	f	0	240.0	0

Note that `Xtr` includes all the numerical features - i.e., A2, A3, A8, A11, and A14 - because we only evaluated a few of the categorical features.

And, finally, we can also obtain the names of the features in the final transformed dataset:

```
sel.get_feature_names_out()

['A2', 'A3', 'A4', 'A5', 'A6', 'A7', 'A8', 'A9', 'A10', 'A11', 'A14', 'A15']
```

If we want to select from categorical and numerical variables, we can do so as well by sorting the numerical variables into bins first. Let's sort them into 5 bins of equal-frequency:

```
sel = SelectByInformationValue(
    bins=5,
    strategy="equal_frequency",
    threshold=0.2,
)

sel.fit(X_train.drop(["A4", "A5", "A7"], axis=1), y_train)
```

If we now inspect the information values:

```
sel.information_values_
```

We see the following:

```
{'A1': 0.0009535686492270659,
 'A2': 0.10319123021570434,
 'A3': 0.2596258749173557,
 'A6': 0.6006252129425703,
 'A8': 0.7291628533346297,
 'A9': 2.9184484098456807,
 'A10': 0.8606638171665587,
 'A11': 1.0634602064399297,
 'A12': 0.012251943759377052,
 'A13': 0.04383964979386022,
 'A14': 0.3316668794040285,
 'A15': 0.6228678069374612}
```

And if we inspect the features to drop:

```
sel.features_to_drop_
```

We see the following:

```
['A1', 'A2', 'A12', 'A13']
```

Note

The WoE is given by a logarithm of a fraction. Thus, if for any category or bin, the fraction of observations of class 0 is 0, the WoE is not defined, and the transformer will raise an error.

If you encounter this problem try grouping variables into fewer bins if they are numerical, or grouping rare categories with the RareLabelEncoder if they are categorical.

Additional resources

For more details about this and other feature selection methods check out these resources:

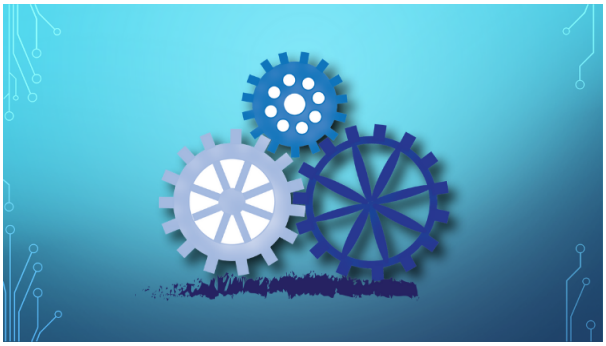
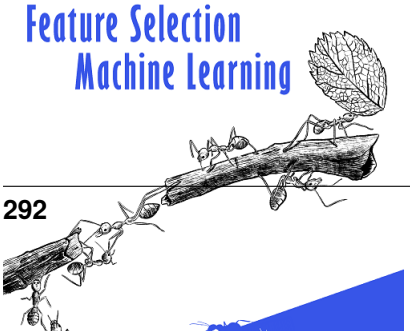


Fig. 97: Feature Selection for Machine Learning

Or read our book:



Both our book and course are suitable for beginners and more advanced data scientists alike. By purchasing them you are supporting Sole, the main developer of Feature-engine.

ProbeFeatureSelection

ProbeFeatureSelection() generates one or more random variables based on the user-selected parameters. Next, the transformer derives the feature importance for each variable and probe feature. Finally, it eliminates the features that have a lower feature importance score than the probe feature(s).

In the case of there being more than one probe feature, the average feature importance score of all the probe features is used.

In summary, this is how *ProbeFeatureSelection()* selects features:

1. Create 1 or more random features
2. Train a machine learning model with all features including the random ones
3. Derive feature importance for all features
4. Take the average importance of the random features (only if more than 1 random feature were used)
5. Select features whose importance is greater than the importance of the random variables (step 4)

One of the primary goals of feature selection is to remove noise from the dataset. A randomly generated variable, i.e., probe feature, inherently possesses a high level of noise. Consequently, any variable that demonstrates less importance than a probe feature is assumed to be noise and can be discarded from the dataset.

When initiating the *ProbeFeatureSelection()* class, the user has the option of selecting which distribution is to be assumed to create the probe feature(s) and the number of probe features to be created. The possible distributions are 'normal', 'binary', 'uniform', or 'all'. 'all' creates 1 or more probe features comprised of each distribution type, i.e., normal, binomial, and uniform.

Example

Let's see how to use this transformer to select variables from UC Irvine's Breast Cancer Wisconsin (Diagnostic) dataset, which can be found here. We will use Scikit-learn to load the dataset. This dataset concerns breast cancer diagnoses. The target variable is binary, i.e., malignant or benign.

The data is solely comprised of numerical data.

Let's import the required libraries and classes:

```
import pandas as pd
from sklearn.datasets import load_breast_cancer
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from feature_engine.selection import ProbeFeatureSelection
```

Let's now load the cancer diagnostic data:

```
cancer_X, cancer_y = load_breast_cancer(return_X_y=True, as_frame=True)
```

Let's check the shape of `cancer_X`:

```
print(cancer_X.shape)
```

We see that the dataset is comprised of 569 observations and 30 features:

```
(569, 30)
```

Let's now split the data into train and test sets:

```
# separate train and test sets
X_train, X_test, y_train, y_test = train_test_split(
    cancer_X,
    cancer_y,
    test_size=0.2,
    random_state=3
)

X_train.shape, X_test.shape
```

We see the size of the datasets below. Note that there are 30 features in both the training and test sets.

```
((455, 30), (114, 30))
```

Now, we set up *ProbeFeatureSelection()*.

We will pass `RandomForestClassifier()` as the `estimator`. We will use `precision` as the `scoring` parameter and 5 as `cv` parameter, both parameters to be used in the cross validation.

In this example, we will introduce just 1 random feature with a normal distribution. Thus, we pass 1 for the `n_probes` parameter and `normal` as the `distribution`.

```
sel = ProbeFeatureSelection(
    estimator=RandomForestClassifier(),
    variables=None,
    scoring="precision",
    n_probes=1,
    distribution="normal",
    cv=5,
    random_state=150,
    confirm_variables=False
)

sel.fit(X_train, y_train)
```

With `fit()`, the transformer:

- creates `n_probes` number of probe features using provided distribution(s)
- uses cross-validation to fit the provided estimator
- calculates the feature importance score for each variable, including probe features
- if there are multiple probe features, the transformer calculates the average importance score
- identifies features to drop because their importance scores are less than that of the probe feature(s)

In the attribute `probe_features`, we find the pseudo-randomly generated variable(s):

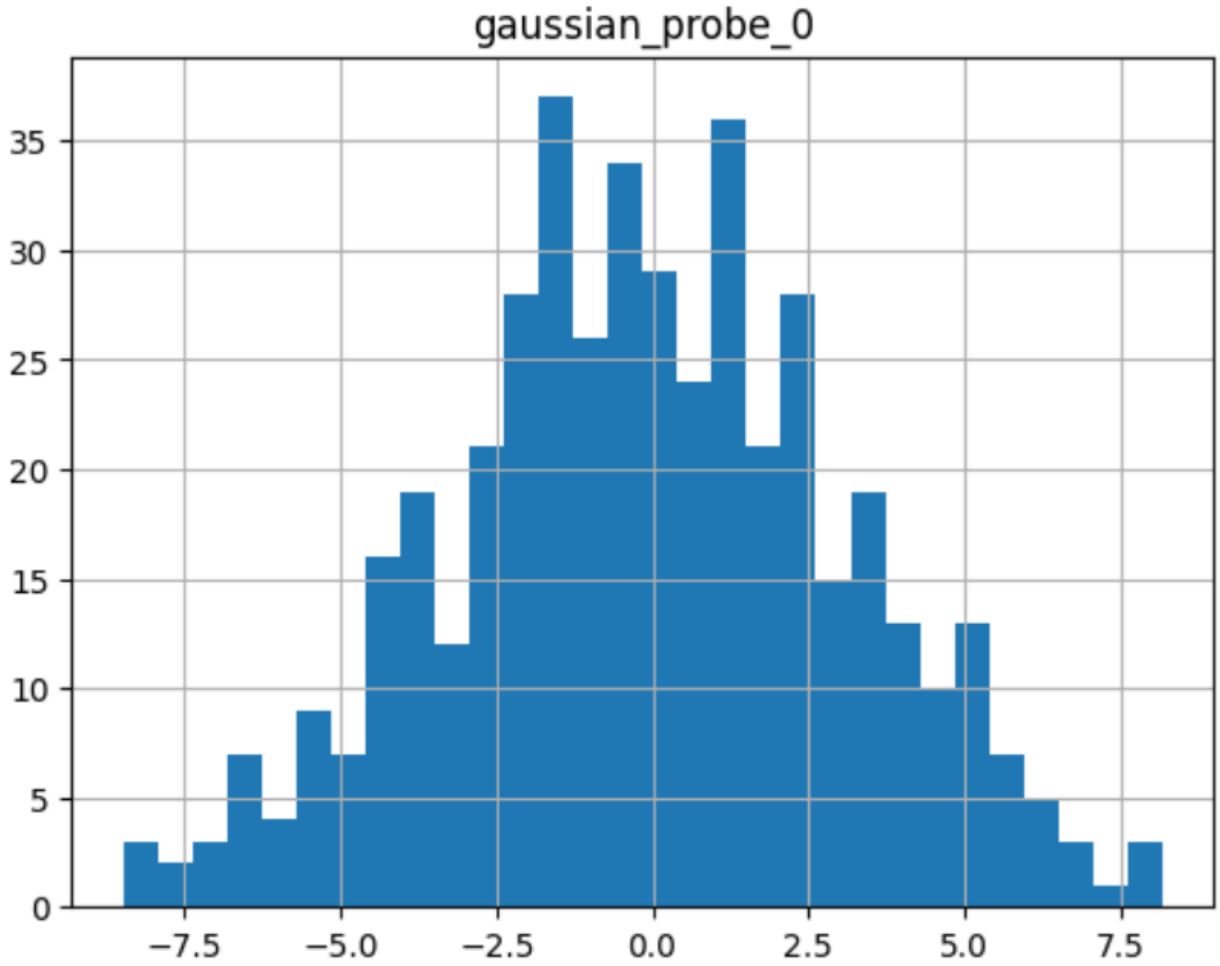
```
sel.probe_features_.head()
```

```
   gaussian_probe_0
0         -0.694150
1          1.171840
2          1.074892
3          1.698733
4          0.498702
```

We can go ahead and display a histogram of the probe feature:

```
sel.probe_features_.hist(bins=30)
```

As we can see, it shows a normal distribution:



The attribute `feature_importances_` shows each variable's feature importance:

```
sel.feature_importances_.head()
```

These are the first 5 features:

```
mean radius        0.058463
mean texture       0.011953
```

(continues on next page)

(continued from previous page)

```
mean perimeter     0.069516
mean area          0.050947
mean smoothness    0.004974
```

At the end of the series, we see the importance of the probe feature:

```
sel.feature_importances_.tail()
```

These are the importance of the last 5 features including the probe:

```
worst concavity            0.037844
worst concave points       0.102769
worst symmetry             0.011587
worst fractal dimension    0.007456
gaussian_probe_0           0.003783
dtype: float64
```

In the attribute `features_to_drop_`, we find the variables that were not selected:

```
sel.features_to_drop_
```

These are the variables that will be removed from the dataframe:

```
['mean symmetry',
 'mean fractal dimension',
 'texture error',
 'smoothness error',
 'concave points error',
 'fractal dimension error']
```

We see that the `features_to_drop_` have feature importance scores that are less than the probe feature's score:

```
sel.feature_importances_.loc[sel.features_to_drop_+["gaussian_probe_0"]]
```

The previous command returns the following output:

```
mean symmetry              0.003698
mean fractal dimension     0.003455
texture error              0.003595
smoothness error           0.003333
concave points error       0.003548
fractal dimension error    0.003576
gaussian_probe_0           0.003783
```

With `transform()`, we can go ahead and drop the six features with feature importance score less than `gaussian_probe_0` variable:

```
Xtr = sel.transform(X_test)

Xtr.shape
```

The final shape of the data after removing the features:

```
(114, 24)
```

And, finally, we can also obtain the names of the features in the final transformed dataset:

```
sel.get_feature_names_out()

['mean radius',
 'mean texture',
 'mean perimeter',
 'mean area',
 'mean smoothness',
 'mean compactness',
 'mean concavity',
 'mean concave points',
 'radius error',
 'perimeter error',
 'area error',
 'compactness error',
 'concavity error',
 'symmetry error',
 'worst radius',
 'worst texture',
 'worst perimeter',
 'worst area',
 'worst smoothness',
 'worst compactness',
 'worst concavity',
 'worst concave points',
 'worst symmetry',
 'worst fractal dimension']
```

For compatibility with Scikit-learn selection transformers, *ProbeFeatureSelection()* also supports the method `get_support()`:

```
sel.get_support()
```

which returns the following output:

```
[True, True, True, True, True, True, True, True, False, False, True, False, True,
 True, False, True, True, False, True, False, True, True, True, True, True, True,
 True, True, True, True]
```

Using several probe features

Let's now repeat the selection process, but using more than 1 probe feature.

```
sel = ProbeFeatureSelection(
    estimator=RandomForestClassifier(),
    variables=None,
    scoring="precision",
    n_probes=3,
    distribution="all",
```

(continues on next page)

(continued from previous page)

```
    cv=5,
    random_state=150,
    confirm_variables=False
)

sel.fit(X_train, y_train)
```

Let's display the random features that the transformer created:

```
sel.probe_features_.head()
```

Here we find some example values of the probe features:

```
   gaussian_probe_0  binary_probe_0  uniform_probe_0
0         -0.694150               1         0.983610
1          1.171840               1         0.765628
2          1.074892               1         0.991439
3          1.698733               0         0.668574
4          0.498702               0         0.192840
```

Let's go ahead and plot histograms:

```
sel.probe_features_.hist(bins=30)
```

In the histograms we recognise the 3 well defined distributions:

Let's display the importance of the random features

```
sel.feature_importances_.tail()
```

```
worst symmetry             0.009176
worst fractal dimension    0.007825
gaussian_probe_0           0.003765
binary_probe_0             0.000354
uniform_probe_0            0.002377
dtype: float64
```

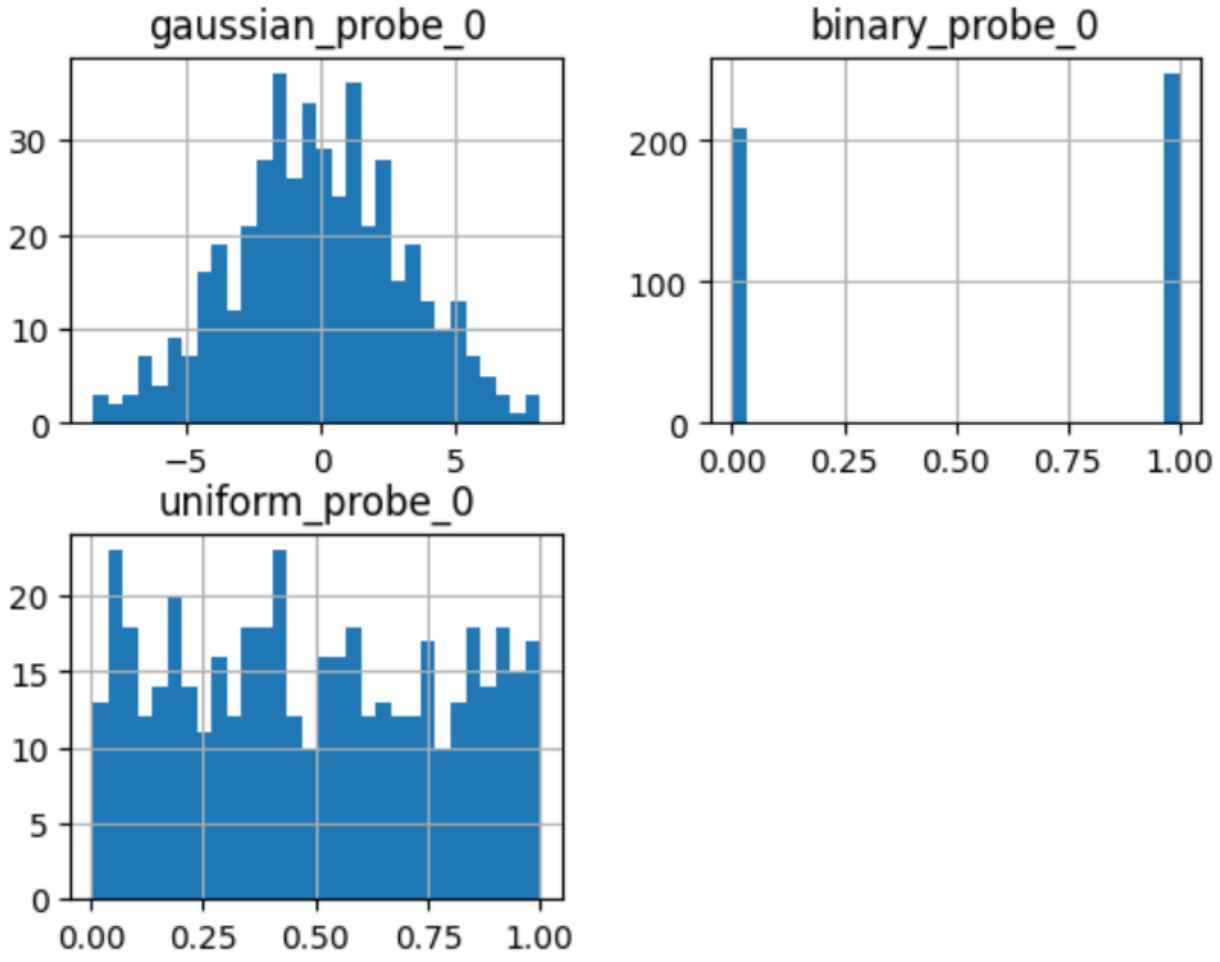
We see that the binary feature has an extremely low importance, hence, when we take the average, the value is so small, that no feature will be dropped:

```
sel.features_to_drop_
```

The previous command returns and empty list:

```
[]
```

It is important to select a suitable probe feature distribution when trying to remove variables. If most variables are continuous, introduce features with normal and uniform distributions. If you have one hot encoded features or sparse matrices, binary features might be a better option.


gaussian_probe_0
30
20
10
0
−5
0
5
binary_probe_0
200
100
0
0.00
0.25
0.50
0.75
1.00
uniform_probe_0
20
15
10
5
0
0.00
0.25
0.50
0.75
1.00

Additional resources

More info about this method can be found in these resources:

- Kaggle Tips for Feature Engineering and Selection, by Gilberto Titericz.
- Feature Selection: Beyond feature importance?, KDDNuggets.

For more details about this and other feature selection methods check out these resources:

Or read our book:

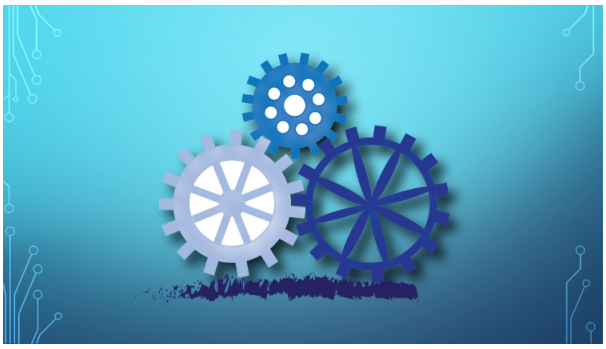
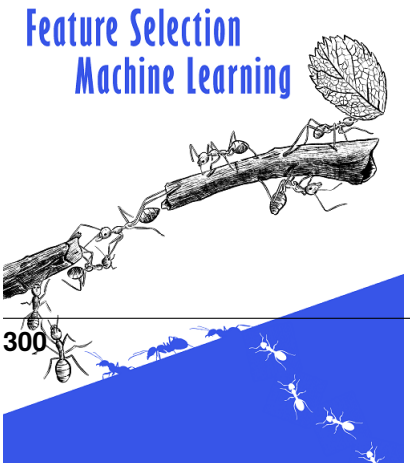


Fig. 99: Feature Selection for Machine Learning



Both our book and course are suitable for beginners and more advanced data scientists alike. By purchasing them you are supporting Sole, the main developer of Feature-engine.

Additional Resources

More details about feature selection can be found in the following resources:

Or read our book:

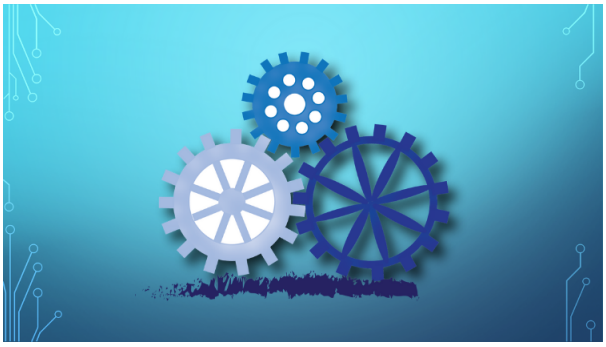


Fig. 101: Feature Selection for Machine Learning

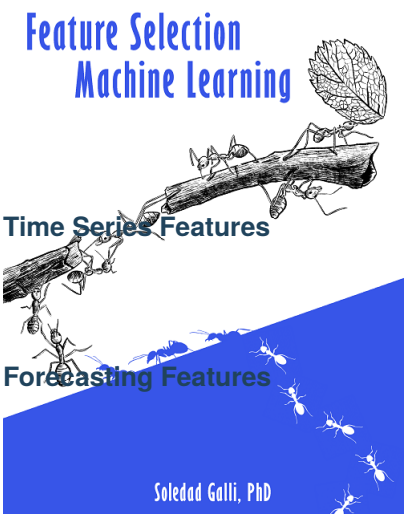


Fig. 102: Feature Selection in Machine Learning

Both our book and course are suitable for beginners and more advanced data scientists alike. By purchasing them you are supporting Sole, the main developer of Feature-engine.

10.2.4 Time series

Feature-engine's time series transformers create features from time series data.

Feature-engine's time series forecasting transformers create and add new features to the dataframe by lagging features or calculating statistics over windows of the features in the past.

LagFeatures

Lag features are commonly used in data science to forecast time series with traditional

machine learning models, like linear regression or random forests. A lag feature is a feature with information about a prior time step of the time series.

When forecasting the future values of a variable, the past values of that same variable are likely to be predictive. Past values of other predictive features can also be useful for our forecast. Thus, in forecasting, it is common practice to create lag features from time series data and use them as input to machine learning algorithms or forecasting models.

What is a lag feature?

A lag feature is the value of the time series **k** period(s) in the past, where **k** is the lag and is to be set by the user. For example, a lag of 1 is a feature that contains the previous time point value of the time series. A lag of 3 contains the value 3 time points before, and so on. By varying k, we can create features with multiple lags.

In Python, we can create lag features by using the pandas method `shift`. For example, by executing `X[my_variable].shift(freq="1H", axis=0)`, we create a new feature consisting of lagged values of `my_variable` by 1 hour.

Feature-engine's *LagFeatures* automates the creation of lag features from multiple variables and by using multiple lags. It uses pandas `shift` under the hood, and automatically concatenates the new features to the input dataframe.

Automating lag feature creation

There are 2 ways in which we can indicate the lag k using *LagFeatures*. Just like with pandas `shift`, we can indicate the lag using the parameter `periods`. This parameter takes integers that indicate the number of rows forward that the features will be lagged.

Alternatively, we can use the parameter `freq`, which takes a string with the period and frequency, and lags features based on the datetime index. For example, if we pass `freq="1D"`, the values of the features will be moved 1 day forward.

The *LagFeatures* transformer works very similarly to `pandas.shift`, but unlike `pandas.shift` we can indicate the lag using either `periods` or `freq` but not both at the same time. Also, unlike `pandas.shift`, we can only lag features forward.

LagFeatures has several advantages over `pandas.shift`:

- First, it can create features with multiple values of k at the same time.
- Second, it adds the features with a name to the original dataframe.
- Third, it has the methods `fit()` and `transform()` that make it compatible with the Scikit-learn's `Pipeline` and cross-validation functions.

Note that, in the current implementation, *LagFeatures* only works with dataframes whose index, containing the time series timestamp, contains unique values and no NaN.

Examples

Let's create a toy dataset to show how to add lag features with *LagFeatures*. The dataframe contains 3 numerical variables, a categorical variable, and a datetime index. We also create an arbitrary target.

```
import pandas as pd

X = {"ambient_temp": [31.31, 31.51, 32.15, 32.39, 32.62, 32.5, 32.52, 32.68],
     "module_temp": [49.18, 49.84, 52.35, 50.63, 49.61, 47.01, 46.67, 47.52],
     "irradiation": [0.51, 0.79, 0.65, 0.76, 0.42, 0.49, 0.57, 0.56],
     "color": ["green"] * 4 + ["blue"] * 4,
     }

X = pd.DataFrame(X)
X.index = pd.date_range("2020-05-15 12:00:00", periods=8, freq="15min")
y = pd.Series([1, 2, 3, 4, 5, 6, 7, 8])
y.index = X.index

X.head()
```

Below we see the output of our toy dataframe:

```
                     ambient_temp  module_temp  irradiation  color
2020-05-15 12:00:00         31.31        49.18         0.51  green
2020-05-15 12:15:00         31.51        49.84         0.79  green
2020-05-15 12:30:00         32.15        52.35         0.65  green
2020-05-15 12:45:00         32.39        50.63         0.76  green
2020-05-15 13:00:00         32.62        49.61         0.42   blue
```

And here we print and show the target variable:

```
y
```

```
2020-05-15 12:00:00    1
2020-05-15 12:15:00    2
2020-05-15 12:30:00    3
2020-05-15 12:45:00    4
2020-05-15 13:00:00    5
2020-05-15 13:15:00    6
2020-05-15 13:30:00    7
2020-05-15 13:45:00    8
Freq: 15min, dtype: int64
```

Shift a row forward

Now we will create lag features by lagging all numerical variables 1 row forward. Note that *LagFeatures* automatically finds all numerical variables.

```
from feature_engine.timeseries.forecasting import LagFeatures

lag_f = LagFeatures(periods=1)

X_tr = lag_f.fit_transform(X)

X_tr.head()
```

We can find the lag features on the right side of the dataframe. Note that the values have been shifted a row forward.

```
                     ambient_temp  module_temp  irradiation  color  \
2020-05-15 12:00:00         31.31        49.18         0.51  green
2020-05-15 12:15:00         31.51        49.84         0.79  green
2020-05-15 12:30:00         32.15        52.35         0.65  green
2020-05-15 12:45:00         32.39        50.63         0.76  green
2020-05-15 13:00:00         32.62        49.61         0.42   blue

                     ambient_temp_lag_1  module_temp_lag_1  irradiation_lag_1
2020-05-15 12:00:00                 NaN                NaN                NaN
2020-05-15 12:15:00               31.31              49.18               0.51
2020-05-15 12:30:00               31.51              49.84               0.79
2020-05-15 12:45:00               32.15              52.35               0.65
2020-05-15 13:00:00               32.39              50.63               0.76
```

The variables to lag are stored in the `variables_` attribute of the *LagFeatures*:

```
lag_f.variables_
```

```
['ambient_temp', 'module_temp', 'irradiation']
```

We can obtain the names of the original variables plus the lag features that are the returned in the transformed dataframe using the `get_feature_names_out()` method:

```
lag_f.get_feature_names_out()
```

```
['ambient_temp',
 'module_temp',
 'irradiation',
 'color',
 'ambient_temp_lag_1',
 'module_temp_lag_1',
 'irradiation_lag_1']
```

When we create lag features, we introduce nan values for the first rows of the training data set, because there are no past values for those data points. We can impute those nan values with an arbitrary value as follows:

```
lag_f = LagFeatures(periods=1, fill_value=0)

X_tr = lag_f.fit_transform(X)

print(X_tr.head())
```

We see that the nan values were replaced by 0:

```
                     ambient_temp  module_temp  irradiation  color  \
2020-05-15 12:00:00         31.31        49.18         0.51  green
2020-05-15 12:15:00         31.51        49.84         0.79  green
2020-05-15 12:30:00         32.15        52.35         0.65  green
2020-05-15 12:45:00         32.39        50.63         0.76  green
2020-05-15 13:00:00         32.62        49.61         0.42   blue

                     ambient_temp_lag_1  module_temp_lag_1  irradiation_lag_1
2020-05-15 12:00:00                0.00               0.00               0.00
2020-05-15 12:15:00               31.31              49.18               0.51
2020-05-15 12:30:00               31.51              49.84               0.79
2020-05-15 12:45:00               32.15              52.35               0.65
2020-05-15 13:00:00               32.39              50.63               0.76
```

Alternatively, we can drop the rows with missing values in the lag features, like this:

```
lag_f = LagFeatures(periods=1, drop_na=True)

X_tr = lag_f.fit_transform(X)

print(X_tr.head())
```

```
                     ambient_temp  module_temp  irradiation  color  \
2020-05-15 12:15:00         31.51        49.84         0.79  green
2020-05-15 12:30:00         32.15        52.35         0.65  green
2020-05-15 12:45:00         32.39        50.63         0.76  green
2020-05-15 13:00:00         32.62        49.61         0.42   blue
2020-05-15 13:15:00         32.50        47.01         0.49   blue

                     ambient_temp_lag_1  module_temp_lag_1  irradiation_lag_1
2020-05-15 12:15:00               31.31              49.18               0.51
2020-05-15 12:30:00               31.51              49.84               0.79
2020-05-15 12:45:00               32.15              52.35               0.65
2020-05-15 13:00:00               32.39              50.63               0.76
```

(continues on next page)

(continued from previous page)

```
2020-05-15 13:15:00              32.62              49.61               0.42
```

We can also drop the rows with nan in the lag features and then adjust the target variable like this:

```
X_tr, y_tr = lag_f.transform_x_y(X, y)

X_tr.shape, y_tr.shape, X.shape, y.shape
```

We created a lag feature of 1, hence there is only 1 row with nan, which was removed from train set and target:

```
((7, 7), (7,), (8, 4), (8,))
```

Create multiple lag features

We can create multiple lag features with one transformer by passing the lag periods in a list.

```
lag_f = LagFeatures(periods=[1, 2])

X_tr = lag_f.fit_transform(X)

X_tr.head()
```

Note how multiple lag features were created for each of the numerical variables and added at the right side of the dataframe.

```
                     ambient_temp  module_temp  irradiation  color  \
2020-05-15 12:00:00         31.31        49.18         0.51  green
2020-05-15 12:15:00         31.51        49.84         0.79  green
2020-05-15 12:30:00         32.15        52.35         0.65  green
2020-05-15 12:45:00         32.39        50.63         0.76  green
2020-05-15 13:00:00         32.62        49.61         0.42   blue

                     ambient_temp_lag_1  module_temp_lag_1  irradiation_lag_1  \
2020-05-15 12:00:00                 NaN                NaN                NaN
2020-05-15 12:15:00               31.31              49.18               0.51
2020-05-15 12:30:00               31.51              49.84               0.79
2020-05-15 12:45:00               32.15              52.35               0.65
2020-05-15 13:00:00               32.39              50.63               0.76

                     ambient_temp_lag_2  module_temp_lag_2  irradiation_lag_2
2020-05-15 12:00:00                 NaN                NaN                NaN
2020-05-15 12:15:00                 NaN                NaN                NaN
2020-05-15 12:30:00               31.31              49.18               0.51
2020-05-15 12:45:00               31.51              49.84               0.79
2020-05-15 13:00:00               32.15              52.35               0.65
```

We can get the names of features in the resulting dataframe as follows:

```
lag_f.get_feature_names_out()
```

```
['ambient_temp',
 'module_temp',
 'irradiation',
 'color',
 'ambient_temp_lag_1',
 'module_temp_lag_1',
 'irradiation_lag_1',
 'ambient_temp_lag_2',
 'module_temp_lag_2',
 'irradiation_lag_2']
```

We can replace the nan introduced in the lag features as well. In this opportunity, we'll use a string. Not that this is a suitable solution to train machine learning algorithms, but the idea here is to showcase *LagFeatures*'s functionality.

```
lag_f = LagFeatures(periods=[1, 2], fill_value='None')

X_tr = lag_f.fit_transform(X)

print(X_tr.head())
```

In this case, we replaced the nan in the lag features with the string None:

```
                     ambient_temp  module_temp  irradiation  color  \
2020-05-15 12:00:00         31.31        49.18         0.51  green
2020-05-15 12:15:00         31.51        49.84         0.79  green
2020-05-15 12:30:00         32.15        52.35         0.65  green
2020-05-15 12:45:00         32.39        50.63         0.76  green
2020-05-15 13:00:00         32.62        49.61         0.42   blue

                    ambient_temp_lag_1 module_temp_lag_1 irradiation_lag_1  \
2020-05-15 12:00:00               None              None              None
2020-05-15 12:15:00              31.31             49.18              0.51
2020-05-15 12:30:00              31.51             49.84              0.79
2020-05-15 12:45:00              32.15             52.35              0.65
2020-05-15 13:00:00              32.39             50.63              0.76

                    ambient_temp_lag_2 module_temp_lag_2 irradiation_lag_2
2020-05-15 12:00:00               None              None              None
2020-05-15 12:15:00               None              None              None
2020-05-15 12:30:00              31.31             49.18              0.51
2020-05-15 12:45:00              31.51             49.84              0.79
2020-05-15 13:00:00              32.15             52.35              0.65
```

Alternatively, we can drop rows containing nan in the lag features and then adjust the target variable:

```
lag_f = LagFeatures(periods=[1, 2], drop_na=True)

lag_f.fit(X)

X_tr, y_tr = lag_f.transform_x_y(X, y)

X_tr.shape, y_tr.shape, X.shape, y.shape
```

We see that 2 rows were dropped from train set and target:

```
((6, 10), (6,), (8, 4), (8,))
```

Lag features based on datetime

We can also lag features utilizing information in the timestamp of the dataframe, which is commonly cast as datetime.

Let's for example create features by lagging 2 of the numerical variables 30 minutes forward.

```
lag_f = LagFeatures(variables = ["module_temp", "irradiation"], freq="30min")

X_tr = lag_f.fit_transform(X)

X_tr.head()
```

Note that the features were moved forward 30 minutes.

```
                     ambient_temp  module_temp  irradiation  color  \
2020-05-15 12:00:00         31.31        49.18         0.51  green
2020-05-15 12:15:00         31.51        49.84         0.79  green
2020-05-15 12:30:00         32.15        52.35         0.65  green
2020-05-15 12:45:00         32.39        50.63         0.76  green
2020-05-15 13:00:00         32.62        49.61         0.42   blue

                     module_temp_lag_30min  irradiation_lag_30min
2020-05-15 12:00:00                    NaN                    NaN
2020-05-15 12:15:00                    NaN                    NaN
2020-05-15 12:30:00                  49.18                   0.51
2020-05-15 12:45:00                  49.84                   0.79
2020-05-15 13:00:00                  52.35                   0.65
```

We can replace the nan in the lag features with a number like this:

```
lag_f = LagFeatures(
    variables=["module_temp", "irradiation"], freq="30min", fill_value=100)

X_tr = lag_f.fit_transform(X)

print(X_tr.head())
```

Here, we replaced nan by 100:

```
                     ambient_temp  module_temp  irradiation  color  \
2020-05-15 12:00:00         31.31        49.18         0.51  green
2020-05-15 12:15:00         31.51        49.84         0.79  green
2020-05-15 12:30:00         32.15        52.35         0.65  green
2020-05-15 12:45:00         32.39        50.63         0.76  green
2020-05-15 13:00:00         32.62        49.61         0.42   blue

                     module_temp_lag_30min  irradiation_lag_30min
2020-05-15 12:00:00                 100.00                 100.00
2020-05-15 12:15:00                 100.00                 100.00
2020-05-15 12:30:00                  49.18                   0.51
```

(continues on next page)

(continued from previous page)

```
2020-05-15 12:45:00                  49.84                  0.79
2020-05-15 13:00:00                  52.35                  0.65
```

Alternatively, we can remove the nan introduced in the lag features and adjust the target:

```
lag_f = LagFeatures(
    variables=["module_temp", "irradiation"], freq="30min", drop_na=True)

lag_f.fit(X)

X_tr, y_tr = lag_f.transform_x_y(X, y)

X_tr.shape, y_tr.shape, X.shape, y.shape
```

Two rows were removed from the training data set and the target:

```
((6, 6), (6,), (8, 4), (8,))
```

Drop variable after lagging features

Similarly, we can lag multiple time intervals forward, but this time, let's drop the original variable after creating the lag features.

```
lag_f = LagFeatures(variables="irradiation",
                    freq=["30min", "45min"],
                    drop_original=True,
                    )

X_tr = lag_f.fit_transform(X)

X_tr.head()
```

We now see the multiple lag features at the back of the dataframe, and also that the original variable is not present in the output dataframe.

```
                     ambient_temp  module_temp  color  irradiation_lag_30min  \
2020-05-15 12:00:00         31.31        49.18  green                    NaN
2020-05-15 12:15:00         31.51        49.84  green                    NaN
2020-05-15 12:30:00         32.15        52.35  green                   0.51
2020-05-15 12:45:00         32.39        50.63  green                   0.79
2020-05-15 13:00:00         32.62        49.61   blue                   0.65

                     irradiation_lag_45min
2020-05-15 12:00:00                    NaN
2020-05-15 12:15:00                    NaN
2020-05-15 12:30:00                    NaN
2020-05-15 12:45:00                   0.51
2020-05-15 13:00:00                   0.79
```

This is super useful in time series forecasting, because the original variable is usually the one that we are trying to forecast, that is, the target variable. The original variables also contain values that are **NOT** available at the time points that we are forecasting.

Working with pandas series

If your time series is a pandas Series instead of a pandas Dataframe, you need to transform it into a dataframe before using *LagFeatures*.

The following is a pandas Series:

```
X['ambient_temp']
```

```
2020-05-15 12:00:00    31.31
2020-05-15 12:15:00    31.51
2020-05-15 12:30:00    32.15
2020-05-15 12:45:00    32.39
2020-05-15 13:00:00    32.62
2020-05-15 13:15:00    32.50
2020-05-15 13:30:00    32.52
2020-05-15 13:45:00    32.68
Freq: 15T, Name: ambient_temp, dtype: float64
```

We can use *LagFeatures* to create, for example, 3 features by lagging the pandas Series if we convert it to a pandas Dataframe using the method `to_frame()`:

```
lag_f = LagFeatures(periods=[1, 2, 3])

X_tr = lag_f.fit_transform(X['ambient_temp'].to_frame())

X_tr.head()
```

```
                     ambient_temp  ambient_temp_lag_1  ambient_temp_lag_2  \
2020-05-15 12:00:00         31.31                 NaN                 NaN
2020-05-15 12:15:00         31.51               31.31                 NaN
2020-05-15 12:30:00         32.15               31.51               31.31
2020-05-15 12:45:00         32.39               32.15               31.51
2020-05-15 13:00:00         32.62               32.39               32.15

                     ambient_temp_lag_3
2020-05-15 12:00:00                 NaN
2020-05-15 12:15:00                 NaN
2020-05-15 12:30:00                 NaN
2020-05-15 12:45:00               31.31
2020-05-15 13:00:00               31.51
```

And if we do not want the original values of time series in the returned dataframe, we just need to remember to drop the original series after the transformation:

```
lag_f = LagFeatures(periods=[1, 2, 3], drop_original=True)

X_tr = lag_f.fit_transform(X['ambient_temp'].to_frame())

X_tr.head()
```

```
                     ambient_temp_lag_1  ambient_temp_lag_2  \
2020-05-15 12:00:00                 NaN                 NaN
```

(continues on next page)

(continued from previous page)

```
2020-05-15 12:15:00                31.31                  NaN
2020-05-15 12:30:00                31.51                31.31
2020-05-15 12:45:00                32.15                31.51
2020-05-15 13:00:00                32.39                32.15

                     ambient_temp_lag_3
2020-05-15 12:00:00                 NaN
2020-05-15 12:15:00                 NaN
2020-05-15 12:30:00                 NaN
2020-05-15 12:45:00               31.31
2020-05-15 13:00:00               31.51
```

Getting the name of the lag features

We can easily obtain the name of the original and new variables with the method `get_feature_names_out`. By using the method with the default parameters, we obtain all the features in the output dataframe.

```
lag_f = LagFeatures(periods=[1, 2])

lag_f.fit(X)

lag_f.get_feature_names_out()
```

```
['ambient_temp',
 'module_temp',
 'irradiation',
 'color',
 'ambient_temp_lag_1',
 'module_temp_lag_1',
 'irradiation_lag_1',
 'ambient_temp_lag_2',
 'module_temp_lag_2',
 'irradiation_lag_2']
```

Determining the right lag

We can create multiple lag features by utilizing various lags. But how do we decide which lag is a good lag?

There are multiple ways to do this.

We can create features by using multiple lags and then determine the best features by using feature selection.

Alternatively, we can determine the best lag through time series analysis by evaluating the autocorrelation or partial autocorrelation of the time series.

For tutorials on how to create lag features for forecasting, check the course Feature Engineering for Time Series Forecasting. In the course, we also show how to detect and remove outliers from time series data, how to use features that capture seasonality and trend, and much more.

Lags from the target vs lags from predictor variables

Very often, we want to forecast the values of just one time series. For example, we want to forecast sales in the next month. The sales variable is our target variable, and we can create features by lagging past sales values.

We could also create lag features from accompanying predictive variables. For example, if we want to predict pollutant concentration in the next few hours, we can create lag features from past pollutant concentrations. In addition, we can create lag features from accompanying time series values, like the concentrations of other gases, or the temperature or humidity.

See also

Check out the additional transformers to create window features through the use of rolling windows (*WindowFeatures*) or expanding windows (*ExpandingWindowFeatures*).

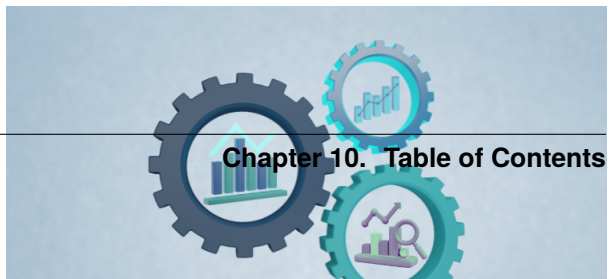
If you want to use *LagFeatures* as part of a feature engineering pipeline, check out Feature-engine's `Pipeline`.

Tutorials and courses

For tutorials about this and other feature engineering methods for time series forecasting check out our online courses:

Fig. 103: Feature Engineering for Time Series Forecasting

Our courses are suitable for beginners and more advanced data scientists looking to forecast time series using traditional machine learning models, like linear regression or gradient boosting machines.



By purchasing them you are supporting Sole, the main developer of Feature-engine.

WindowFeatures

Window features are commonly used in data science to forecast time series with traditional machine learning models, like linear regression or gradient boosting machines. Window features are created by performing mathematical operations over windows of past data.

For example, the mean "sales" value of the previous 3 months of data is a window feature. The maximum "revenue" of the previous three rows of data is another window feature.

In time series forecasting, we want to predict future values of the time series. To do this, we can create window features by performing mathematical operations over windows of past values of the time series data. Then, we would use this features to predict the time series with any regression model.

Rolling window features with pandas

Window features are the result of window operations over the variables. Rolling window operations are operations that perform an aggregation over a **sliding partition** of past values of the time series data.

A window feature is, then, a feature created after computing mathematical functions (e.g., mean, min, max, etc.) within a window over the past data.

In Python, we can create window features by utilizing pandas method `rolling`. For example, by executing:

```
X[["var_1", "var_2"].rolling(window=3).agg(["max", "mean"])
```

With the previous command, we create 2 window features for each variable, `var_1` and `var_2`, by taking the maximum and average value of the current and 2 previous rows of data.

If we want to use those features for forecasting using traditional machine learning algorithms, we also need to shift the window forward with pandas method `shift`:

```
X[["var_1", "var_2"].rolling(window=3).agg(["max", "mean"]).shift(period=1)
```

Shifting is important to ensure that we are using values strictly in the past, respect to the point that we want to forecast.

Sliding window features with Feature-engine

WindowFeatures can automatically create and add window features to the dataframe, by performing multiple mathematical operations over different window sizes over various numerical variables.

Thus, *WindowFeatures* creates and adds new features to the data set automatically through the use of windows over historical data.

Window features: parameters

To create window features we need to determine a number of parameters. First, we need to identify the size of the window or windows in which we will perform the operations. For example, we can take the average of the variable over 3 months, or 6 weeks.

We also need to determine how far back is the window located respect to the data point we want to forecast. For example, I can take the average of the last 3 weeks of data to forecast this week of data, or I can take the average of the last 3 weeks of data to forecast next weeks data, leaving a gap of a window in between the window feature and the forecasting point.

WindowFeatures: under the hood

WindowFeatures works on top of `pandas.rolling`, `pandas.aggregate` and `pandas.shift`. With `pandas.rolling`, *WindowFeatures* determines the size of the windows for the operations. With `pandas.rolling` we can specify the window size with an integer, a string or a function. With *WindowFeatures*, in addition, we can pass a list of integers, strings or functions, to perform computations over multiple window sizes.

WindowFeatures uses `pandas.aggregate` to perform the mathematical operations over the windows. Therefore, you can use any operation supported by pandas. For supported aggregation functions, see Rolling Window Functions.

With `pandas.shift`, *WindowFeatures* places the value derived from the past window, at the place of the value that we want to forecast. So if we want to forecast this week with the average of the past 3 weeks of data, we should shift the value 1 week forward. If we want to forecast next week with the last 3 weeks of data, we should forward the value 2 weeks forward.

WindowFeatures will add the new features with a representative name to the original dataframe. It also has the methods `fit()` and `transform()` that make it compatible with the Scikit-learn's `Pipeline` and cross-validation functions.

Note that, in the current implementation, *WindowFeatures* only works with dataframes whose index, containing the time series timestamp, contains unique values and no NaN.

Examples

Let's create a time series dataset to see how to create window features with *WindowFeatures*. The dataframe contains 3 numerical variables, a categorical variable, and a datetime index. We also create a target variable.

```
import pandas as pd

X = {"ambient_temp": [31.31, 31.51, 32.15, 32.39, 32.62, 32.5, 32.52, 32.68],
     "module_temp": [49.18, 49.84, 52.35, 50.63, 49.61, 47.01, 46.67, 47.52],
     "irradiation": [0.51, 0.79, 0.65, 0.76, 0.42, 0.49, 0.57, 0.56],
     "color": ["green"] * 4 + ["blue"] * 4,
     }

X = pd.DataFrame(X)
X.index = pd.date_range("2020-05-15 12:00:00", periods=8, freq="15min")

y = pd.Series([1,2,3,4,5,6,7,8])
y.index = X.index

X.head()
```

Below we see the dataframe:

```
                     ambient_temp  module_temp  irradiation  color
2020-05-15 12:00:00         31.31        49.18         0.51  green
2020-05-15 12:15:00         31.51        49.84         0.79  green
2020-05-15 12:30:00         32.15        52.35         0.65  green
2020-05-15 12:45:00         32.39        50.63         0.76  green
2020-05-15 13:00:00         32.62        49.61         0.42   blue
```

Let's now print out the target:

```
y
```

Below we see the target variable:

```
2020-05-15 12:00:00    1
2020-05-15 12:15:00    2
2020-05-15 12:30:00    3
2020-05-15 12:45:00    4
2020-05-15 13:00:00    5
2020-05-15 13:15:00    6
2020-05-15 13:30:00    7
2020-05-15 13:45:00    8
Freq: 15min, dtype: int64
```

Now we will create window features from the numerical variables. By setting `window=["30min", "60min"]` we perform calculations over windows of 30 and 60 minutes, respectively.

If you look at our toy dataframe, you'll notice that 30 minutes corresponds to 2 rows of data, and 60 minutes are 4 rows of data. So, we will perform calculations over 2 and then 4 rows of data, respectively.

In `functions`, we indicate all the operations that we want to perform over those windows. In our example below, we want to calculate the mean and the standard deviation of the data within those windows and also find the maximum value within the windows.

With `freq="15min"` we indicate that we need to shift the calculations 15 minutes forward. In other words, we are using the data available in windows defined up to 15 minutes before the forecasting point.

```
from feature_engine.timeseries.forecasting import WindowFeatures

win_f = WindowFeatures(
    window=["30min", "60min"], functions=["mean", "max", "std"], freq="15min",
)

X_tr = win_f.fit_transform(X)

X_tr.head()
```

We find the window features on the right side of the dataframe.

```
                     ambient_temp  module_temp  irradiation  color  \
2020-05-15 12:00:00         31.31        49.18         0.51  green
2020-05-15 12:15:00         31.51        49.84         0.79  green
2020-05-15 12:30:00         32.15        52.35         0.65  green
2020-05-15 12:45:00         32.39        50.63         0.76  green
2020-05-15 13:00:00         32.62        49.61         0.42   blue
```

(continues on next page)

(continued from previous page)

```
                     ambient_temp_window_30min_mean  \
2020-05-15 12:00:00                             NaN
2020-05-15 12:15:00                           31.31
2020-05-15 12:30:00                           31.41
2020-05-15 12:45:00                           31.83
2020-05-15 13:00:00                           32.27

                     ambient_temp_window_30min_max  \
2020-05-15 12:00:00                             NaN
2020-05-15 12:15:00                           31.31
2020-05-15 12:30:00                           31.51
2020-05-15 12:45:00                           32.15
2020-05-15 13:00:00                           32.39

                     ambient_temp_window_30min_std  \
2020-05-15 12:00:00                             NaN
2020-05-15 12:15:00                             NaN
2020-05-15 12:30:00                        0.141421
2020-05-15 12:45:00                        0.452548
2020-05-15 13:00:00                        0.169706

                     module_temp_window_30min_mean  \
2020-05-15 12:00:00                             NaN
2020-05-15 12:15:00                          49.180
2020-05-15 12:30:00                          49.510
2020-05-15 12:45:00                          51.095
2020-05-15 13:00:00                          51.490

                     module_temp_window_30min_max  \
2020-05-15 12:00:00                           NaN
2020-05-15 12:15:00                         49.18
2020-05-15 12:30:00                         49.84
2020-05-15 12:45:00                         52.35
2020-05-15 13:00:00                         52.35

                     module_temp_window_30min_std  ...  \
2020-05-15 12:00:00                           NaN  ...
2020-05-15 12:15:00                           NaN  ...
2020-05-15 12:30:00                      0.466690  ...
2020-05-15 12:45:00                      1.774838  ...
2020-05-15 13:00:00                      1.216224  ...

                     irradiation_window_30min_std  \
2020-05-15 12:00:00                           NaN
2020-05-15 12:15:00                           NaN
2020-05-15 12:30:00                      0.197990
2020-05-15 12:45:00                      0.098995
2020-05-15 13:00:00                      0.077782

                     ambient_temp_window_60min_mean  \
2020-05-15 12:00:00                             NaN
```

(continues on next page)

(continued from previous page)

```
2020-05-15 12:15:00                        31.310000
2020-05-15 12:30:00                        31.410000
2020-05-15 12:45:00                        31.656667
2020-05-15 13:00:00                        31.840000

                     ambient_temp_window_60min_max  \
2020-05-15 12:00:00                            NaN
2020-05-15 12:15:00                          31.31
2020-05-15 12:30:00                          31.51
2020-05-15 12:45:00                          32.15
2020-05-15 13:00:00                          32.39

                     ambient_temp_window_60min_std  \
2020-05-15 12:00:00                            NaN
2020-05-15 12:15:00                            NaN
2020-05-15 12:30:00                       0.141421
2020-05-15 12:45:00                       0.438786
2020-05-15 13:00:00                       0.512640

                     module_temp_window_60min_mean  \
2020-05-15 12:00:00                            NaN
2020-05-15 12:15:00                      49.180000
2020-05-15 12:30:00                      49.510000
2020-05-15 12:45:00                      50.456667
2020-05-15 13:00:00                      50.500000

                     module_temp_window_60min_max  \
2020-05-15 12:00:00                            NaN
2020-05-15 12:15:00                          49.18
2020-05-15 12:30:00                          49.84
2020-05-15 12:45:00                          52.35
2020-05-15 13:00:00                          52.35

                     module_temp_window_60min_std  \
2020-05-15 12:00:00                            NaN
2020-05-15 12:15:00                            NaN
2020-05-15 12:30:00                       0.466690
2020-05-15 12:45:00                       1.672553
2020-05-15 13:00:00                       1.368381

                     irradiation_window_60min_mean  \
2020-05-15 12:00:00                            NaN
2020-05-15 12:15:00                         0.5100
2020-05-15 12:30:00                         0.6500
2020-05-15 12:45:00                         0.6500
2020-05-15 13:00:00                         0.6775

                     irradiation_window_60min_max  \
2020-05-15 12:00:00                            NaN
2020-05-15 12:15:00                           0.51
2020-05-15 12:30:00                           0.79
2020-05-15 12:45:00                           0.79
```

(continues on next page)

(continued from previous page)

```
2020-05-15 13:00:00                          0.79

                     irradiation_window_60min_std
2020-05-15 12:00:00                           NaN
2020-05-15 12:15:00                           NaN
2020-05-15 12:30:00                      0.197990
2020-05-15 12:45:00                      0.140000
2020-05-15 13:00:00                      0.126853

[5 rows x 22 columns]
```

The variables used as input for the window features are stored in the `variables_` attribute of the *WindowFeatures*:

```
win_f.variables_
```

```
['ambient_temp', 'module_temp', 'irradiation']
```

We can obtain the names of the variables in the transformed dataframe using the `get_feature_names_out()` method:

```
win_f.get_feature_names_out()
```

```
['ambient_temp',
 'module_temp',
 'irradiation',
 'color',
 'ambient_temp_window_30min_mean',
 'ambient_temp_window_30min_max',
 'ambient_temp_window_30min_std',
 'module_temp_window_30min_mean',
 'module_temp_window_30min_max',
 'module_temp_window_30min_std',
 'irradiation_window_30min_mean',
 'irradiation_window_30min_max',
 'irradiation_window_30min_std',
 'ambient_temp_window_60min_mean',
 'ambient_temp_window_60min_max',
 'ambient_temp_window_60min_std',
 'module_temp_window_60min_mean',
 'module_temp_window_60min_max',
 'module_temp_window_60min_std',
 'irradiation_window_60min_mean',
 'irradiation_window_60min_max',
 'irradiation_window_60min_std']
```

Dropping rows with nan

When we create window features, we may introduce nan values for those data points where there isn't enough data in the past to create the windows. We can automatically drop the rows with nan values in the window features both in the train set and in the target variable as follows:

```
win_f = WindowFeatures(
    window=["30min", "60min"],
    functions=["mean", ],
    freq="15min",
    drop_na=True,
)

win_f.fit(X)

X_tr, y_tr = win_f.transform_x_y(X, y)

X.shape, y.shape, X_tr.shape, y_tr.shape
```

We see that the resulting dataframe contains less rows than the original dataframe:

```
((8, 4), (8,), (7, 10), (7,))
```

Imputing rows with nan

If instead of removing the row with nan in the window features, we want to impute those values, we can do so with any of Feature-engine's imputers. Here, we will replace nan with the arbitrary value -99, using the `ArbitraryNumberImputer` within a pipeline:

```
from feature_engine.imputation import ArbitraryNumberImputer
from feature_engine.pipeline import Pipeline

win_f = WindowFeatures(
    window=["30min", "60min"],
    functions=["mean", ],
    freq="15min",
)

pipe = Pipeline([
    ("windows", win_f),
    ("imputer", ArbitraryNumberImputer(arbitrary_number=-99))
])

X_tr = pipe.fit_transform(X, y)

print(X_tr.head())
```

We see the resulting dataframe, where the nan values were replaced by -99:

```
                     ambient_temp  module_temp  irradiation  color  \
2020-05-15 12:00:00         31.31        49.18         0.51  green
2020-05-15 12:15:00         31.51        49.84         0.79  green
```

(continues on next page)

(continued from previous page)

```
2020-05-15 12:30:00         32.15         52.35         0.65  green
2020-05-15 12:45:00         32.39         50.63         0.76  green
2020-05-15 13:00:00         32.62         49.61         0.42   blue

                     ambient_temp_window_30min_mean  \
2020-05-15 12:00:00                          -99.00
2020-05-15 12:15:00                           31.31
2020-05-15 12:30:00                           31.41
2020-05-15 12:45:00                           31.83
2020-05-15 13:00:00                           32.27

                     module_temp_window_30min_mean  \
2020-05-15 12:00:00                        -99.000
2020-05-15 12:15:00                         49.180
2020-05-15 12:30:00                         49.510
2020-05-15 12:45:00                         51.095
2020-05-15 13:00:00                         51.490

                     irradiation_window_30min_mean  \
2020-05-15 12:00:00                        -99.000
2020-05-15 12:15:00                          0.510
2020-05-15 12:30:00                          0.650
2020-05-15 12:45:00                          0.720
2020-05-15 13:00:00                          0.705

                     ambient_temp_window_60min_mean  \
2020-05-15 12:00:00                       -99.000000
2020-05-15 12:15:00                        31.310000
2020-05-15 12:30:00                        31.410000
2020-05-15 12:45:00                        31.656667
2020-05-15 13:00:00                        31.840000

                     module_temp_window_60min_mean  \
2020-05-15 12:00:00                      -99.000000
2020-05-15 12:15:00                       49.180000
2020-05-15 12:30:00                       49.510000
2020-05-15 12:45:00                       50.456667
2020-05-15 13:00:00                       50.500000

                     irradiation_window_60min_mean
2020-05-15 12:00:00                        -99.0000
2020-05-15 12:15:00                          0.5100
2020-05-15 12:30:00                          0.6500
2020-05-15 12:45:00                          0.6500
2020-05-15 13:00:00                          0.6775
```

Working with pandas series

If your time series is a pandas Series instead of a pandas Dataframe, you need to transform it into a dataframe before using *WindowFeatures*.

The following is a pandas Series:

```
X['ambient_temp']
```

```
2020-05-15 12:00:00    31.31
2020-05-15 12:15:00    31.51
2020-05-15 12:30:00    32.15
2020-05-15 12:45:00    32.39
2020-05-15 13:00:00    32.62
2020-05-15 13:15:00    32.50
2020-05-15 13:30:00    32.52
2020-05-15 13:45:00    32.68
Freq: 15T, Name: ambient_temp, dtype: float64
```

We can use *WindowFeatures* to create, for example, 2 new window features by finding the mean and maximum value within a 45 minute windows of a pandas Series if we convert it to a pandas Dataframe using the method `to_frame()`:

```
win_f = WindowFeatures(
    window=["45min"],
    functions=["mean", "max"],
    freq="30min",
)

X_tr = win_f.fit_transform(X['ambient_temp'].to_frame())

X_tr.head()
```

```
                     ambient_temp  ambient_temp_window_45min_mean  \
2020-05-15 12:00:00         31.31                             NaN
2020-05-15 12:15:00         31.51                             NaN
2020-05-15 12:30:00         32.15                       31.310000
2020-05-15 12:45:00         32.39                       31.410000
2020-05-15 13:00:00         32.62                       31.656667

                     ambient_temp_window_45min_max
2020-05-15 12:00:00                            NaN
2020-05-15 12:15:00                            NaN
2020-05-15 12:30:00                          31.31
2020-05-15 12:45:00                          31.51
2020-05-15 13:00:00                          32.15
```

And if we do not want the original values of time series in the returned dataframe, we just need to remember to drop the original series after the transformation:

```
win_f = WindowFeatures(
    window=["45min"],
    functions=["mean", "max"],
    freq="30min",
```

(continues on next page)

(continued from previous page)

```
    drop_original=True,
)

X_tr = win_f.fit_transform(X['ambient_temp'].to_frame())

X_tr.head()
```

```
                     ambient_temp_window_45min_mean  \
2020-05-15 12:00:00                             NaN
2020-05-15 12:15:00                             NaN
2020-05-15 12:30:00                       31.310000
2020-05-15 12:45:00                       31.410000
2020-05-15 13:00:00                       31.656667

                     ambient_temp_window_45min_max
2020-05-15 12:00:00                            NaN
2020-05-15 12:15:00                            NaN
2020-05-15 12:30:00                          31.31
2020-05-15 12:45:00                          31.51
2020-05-15 13:00:00                          32.15
```

Getting the name of the new features

We can easily obtain the name of the original and new variables with the method `get_feature_names_out`. By using the method with the default parameters, we obtain all the features in the output dataframe.

```
win_f = WindowFeatures()

win_f.fit(X)

win_f.get_feature_names_out()
```

```
['ambient_temp',
 'module_temp',
 'irradiation',
 'color',
 'ambient_temp_window_3_mean',
 'module_temp_window_3_mean',
 'irradiation_window_3_mean']
```

Windows from the target vs windows from predictor variables

Very often, we work with univariate time series, for example, the total sales revenue of a retail company. We want to forecast future sales values. The sales variable is our target variable, and we can extract features from windows of past sales values.

We could also work with multivariate time series, where we have sales in different countries, or alternatively, multiple time series, like pollutant concentration in the air, accompanied by concentrations of other gases, temperature, and humidity.

When working with multivariate time series, like sales in multiple countries, we would extract features from windows of past data for each country separately.

When working with multiple time series, like pollutant concentration, gas concentration, temperature, and humidity, pollutant concentration is our target variable, and the other time series are accompanying predictive variables. We can create window features from past pollutant concentrations, that is, from past time steps of our target variable. And, in addition, we can create features from windows of past data from accompanying time series, like the concentrations of other gases or the temperature or humidity.

The process of feature extraction from time series data, to create a table of predictors and a target variable to forecast using supervised learning models like linear regression or random forest, is called "tabularizing" the time series.

See also

Check out the additional transformers to create expanding window features (*ExpandingWindowFeatures*) or lag features, by lagging past values of the time series data (*LagFeatures*).

Other open-source packages for window features

- tsfresh
- featuretools

Tutorials and courses

For tutorials about this and other feature engineering methods for time series forecasting check out our online courses:

Fig. 105: Feature Engineering for Time Series Forecasting

Our courses are suitable for beginners and more advanced data scientists looking to forecast time series using traditional machine learning models, like linear regression or gradient boosting machines.

By purchasing them you are supporting Sole, the main developer of Feature-engine.

Fig. 106: Forecasting with Machine Learning

ExpandingWindowFeatures

Window features are variables created by performing mathematical operations over a window of past data in a time series.

Rolling window features are created by performing aggregations over a **sliding partition** (or moving window) of past data points of the time series data. The window size in this case is constant.

Expanding window features are created by performing aggregations over an **expanding partition** of past values of the time series. The window size increases as we approach more recent values.

An example of an expanding window feature is the mean value of all the data points prior to the current row / value. The maximum value of all the rows prior to the current row is another expanding window feature.

For an expanding window feature to be suitable for forecasting, the window can span from the start of the data up to, but not including, the first point of forecast.

Expanding window features can be used for forecasting by using traditional machine learning models, like linear regression.

Expanding window features with pandas

In Python, we can create expanding window features by utilizing pandas method `expanding`. For example, by executing:

```
X[["var_1", "var_2"].expanding(min_periods=3).agg(["max", "mean"])
```

With the previous command, we create 2 window features for each variable, `var_1` and `var_2`, by taking the maximum and average value of all observations up to (and including) a certain row.

If we want to use those features for forecasting using traditional machine learning algorithms, we would also shift the window forward with pandas method `shift`:

```
X[["var_1", "var_2"].expanding(min_periods=3).agg(["max", "mean"]).shift(period=1)
```

Expanding window features with Feature-engine

ExpandingWindowFeatures adds expanding window features to the dataframe.

Window features are the result of applying an aggregation operation (e.g., mean, min, max, etc.) to a variable over a window of past data.

When forecasting the future values of a variable, the past values of that variable are likely to be predictive. To capitalize on the past values of a variable, we can simply lag features with *LagFeatures*. We can also create features that summarise the past values into a single quantity utilising *ExpandingWindowFeatures*.

ExpandingWindowFeatures works on top of `pandas.expanding`, `pandas.aggregate` and `pandas.shift`.

ExpandingWindowFeatures uses `pandas.aggregate` to perform the mathematical operations over the expanding window. Therefore, you can use any operation supported by pandas. For supported aggregation functions, see Expanding Window Functions.

With `pandas.shift`, *ExpandingWindowFeatures* lags the result of the expanding window operation. This is useful to ensure that only the information known at predict time is used to compute the window feature. So if at predict time we only know the value of a feature at the previous time period and before that, then we should lag the the window feature by 1 period. If at predict time we only know the value of a feature from 2 weeks ago and before that, then we should lag the window feature column by 2 weeks. *ExpandingWindowFeatures* uses a default lag of one period.

ExpandingWindowFeatures will add the new variables with a representative name to the original dataframe. It also has the methods `fit()` and `transform()` that make it compatible with the Scikit-learn's `Pipeline` and cross-validation functions.

Note that, in the current implementation, *ExpandingWindowFeatures* only works with dataframes whose index, containing the time series timestamp, contains unique values and no NaN.

Examples

Let's create a toy dataset to demonstrate the functionality of *ExpandingWindowFeatures*. The dataframe contains 3 numerical variables, a categorical variable, and a datetime index.

```
import pandas as pd

X = {"ambient_temp": [31.31, 31.51, 32.15, 32.39, 32.62, 32.5, 32.52, 32.68],
     "module_temp": [49.18, 49.84, 52.35, 50.63, 49.61, 47.01, 46.67, 47.52],
     "irradiation": [0.51, 0.79, 0.65, 0.76, 0.42, 0.49, 0.57, 0.56],
     "color": ["green"] * 4 + ["blue"] * 4,
     }

X = pd.DataFrame(X)
X.index = pd.date_range("2020-05-15 12:00:00", periods=8, freq="15min")

y = pd.Series([1,2,3,4,5,6,7,8])
y.index = X.index

X.head()
```

Below we see the output of our toy dataframe:

```
                     ambient_temp  module_temp  irradiation  color
2020-05-15 12:00:00         31.31        49.18         0.51  green
2020-05-15 12:15:00         31.51        49.84         0.79  green
```

(continues on next page)

(continued from previous page)

```
2020-05-15 12:30:00         32.15         52.35         0.65  green
2020-05-15 12:45:00         32.39         50.63         0.76  green
2020-05-15 13:00:00         32.62         49.61         0.42   blue
```

Let's now print out the target:

```
y
```

Below we see the target variable:

```
2020-05-15 12:00:00    1
2020-05-15 12:15:00    2
2020-05-15 12:30:00    3
2020-05-15 12:45:00    4
2020-05-15 13:00:00    5
2020-05-15 13:15:00    6
2020-05-15 13:30:00    7
2020-05-15 13:45:00    8
Freq: 15min, dtype: int64
```

Now we will create expanding window features from the numerical variables. In `functions`, we indicate all the operations that we want to perform over those windows. In our example below, we want to calculate the mean and the standard deviation of the data within those windows and also find the maximum value within the windows.

```
from feature_engine.timeseries.forecasting import ExpandingWindowFeatures

win_f = ExpandingWindowFeatures(functions=["mean", "max", "std"])

X_tr = win_f.fit_transform(X)

X_tr.head()
```

We can find the window features on the right side of the dataframe.

```
                     ambient_temp  module_temp  irradiation  color  \
2020-05-15 12:00:00         31.31        49.18         0.51  green
2020-05-15 12:15:00         31.51        49.84         0.79  green
2020-05-15 12:30:00         32.15        52.35         0.65  green
2020-05-15 12:45:00         32.39        50.63         0.76  green
2020-05-15 13:00:00         32.62        49.61         0.42   blue

                     ambient_temp_expanding_mean  ambient_temp_expanding_max  \
2020-05-15 12:00:00                          NaN                         NaN
2020-05-15 12:15:00                    31.310000                       31.31
2020-05-15 12:30:00                    31.410000                       31.51
2020-05-15 12:45:00                    31.656667                       32.15
2020-05-15 13:00:00                    31.840000                       32.39

                     ambient_temp_expanding_std  module_temp_expanding_mean  \
2020-05-15 12:00:00                         NaN                         NaN
2020-05-15 12:15:00                         NaN                   49.180000
2020-05-15 12:30:00                    0.141421                   49.510000
2020-05-15 12:45:00                    0.438786                   50.456667
```

(continues on next page)

(continued from previous page)

```
2020-05-15 13:00:00                    0.512640                   50.500000

                     module_temp_expanding_max  module_temp_expanding_std  \
2020-05-15 12:00:00                        NaN                        NaN
2020-05-15 12:15:00                      49.18                        NaN
2020-05-15 12:30:00                      49.84                   0.466690
2020-05-15 12:45:00                      52.35                   1.672553
2020-05-15 13:00:00                      52.35                   1.368381

                     irradiation_expanding_mean  irradiation_expanding_max  \
2020-05-15 12:00:00                         NaN                        NaN
2020-05-15 12:15:00                      0.5100                       0.51
2020-05-15 12:30:00                      0.6500                       0.79
2020-05-15 12:45:00                      0.6500                       0.79
2020-05-15 13:00:00                      0.6775                       0.79

                     irradiation_expanding_std
2020-05-15 12:00:00                        NaN
2020-05-15 12:15:00                        NaN
2020-05-15 12:30:00                   0.197990
2020-05-15 12:45:00                   0.140000
2020-05-15 13:00:00                   0.126853
```

The variables used as input for the window features are stored in the `variables_` attribute of the *ExpandingWindowFeatures*.

```
win_f.variables_
```

```
['ambient_temp', 'module_temp', 'irradiation']
```

We can obtain the names of the variables in the returned dataframe using the `get_feature_names_out()` method:

```
win_f.get_feature_names_out()
```

```
['ambient_temp',
 'module_temp',
 'irradiation',
 'color',
 'ambient_temp_expanding_mean',
 'ambient_temp_expanding_max',
 'ambient_temp_expanding_std',
 'module_temp_expanding_mean',
 'module_temp_expanding_max',
 'module_temp_expanding_std',
 'irradiation_expanding_mean',
 'irradiation_expanding_max',
 'irradiation_expanding_std']
```

Dropping rows with nan

When we create window features using expanding windows, we may introduce nan values for those data points where there isn't enough data in the past to create the windows. We can automatically drop the rows with nan values in the window features both in the train set and in the target variable as follows:

```
win_f = ExpandingWindowFeatures(
    functions=["mean", "max", "std"],
    drop_na=True,
)

win_f.fit(X)

X_tr, y_tr = win_f.transform_x_y(X, y)

X.shape, y.shape, X_tr.shape, y_tr.shape
```

We see that the resulting dataframe contains less rows than the original dataframe:

```
(8, 4), (8,), (6, 13), (6,))
```

Imputing rows with nan

If instead of removing the row with nan in the expanding window features, we want to impute those values, we can do so with any of Feature-engine's imputers. Here, we will replace nan with the median value of the resulting window features, using the `MeanMedianImputer` within a pipeline:

```
from feature_engine.imputation import MeanMedianImputer
from feature_engine.pipeline import Pipeline

win_f = ExpandingWindowFeatures(functions=["mean", "std"])

pipe = Pipeline([
    ("windows", win_f),
    ("imputer", MeanMedianImputer(imputation_method="median"))
])

X_tr = pipe.fit_transform(X, y)

print(X_tr.head())
```

We see the resulting dataframe, where the nan values were replaced with the median:

```
                     ambient_temp  module_temp  irradiation  color  \
2020-05-15 12:00:00         31.31        49.18         0.51  green
2020-05-15 12:15:00         31.51        49.84         0.79  green
2020-05-15 12:30:00         32.15        52.35         0.65  green
2020-05-15 12:45:00         32.39        50.63         0.76  green
2020-05-15 13:00:00         32.62        49.61         0.42   blue

                     ambient_temp_expanding_mean  ambient_temp_expanding_std  \
2020-05-15 12:00:00                    31.840000                    0.518740
```

(continues on next page)

(continued from previous page)

```
2020-05-15 12:15:00                    31.310000                    0.518740
2020-05-15 12:30:00                    31.410000                    0.141421
2020-05-15 12:45:00                    31.656667                    0.438786
2020-05-15 13:00:00                    31.840000                    0.512640

                     module_temp_expanding_mean  module_temp_expanding_std  \
2020-05-15 12:00:00                   49.770000                   1.520467
2020-05-15 12:15:00                   49.180000                   1.520467
2020-05-15 12:30:00                   49.510000                   0.466690
2020-05-15 12:45:00                   50.456667                   1.672553
2020-05-15 13:00:00                   50.500000                   1.368381

                     irradiation_expanding_mean  irradiation_expanding_std
2020-05-15 12:00:00                      0.6260                   0.146424
2020-05-15 12:15:00                      0.5100                   0.146424
2020-05-15 12:30:00                      0.6500                   0.197990
2020-05-15 12:45:00                      0.6500                   0.140000
2020-05-15 13:00:00                      0.6775                   0.126853
```

Working with pandas series

If your time series is a pandas Series instead of a pandas Dataframe, you need to transform it into a dataframe before using *ExpandingWindowFeatures*.

The following is a pandas Series:

```
X['ambient_temp']
```

```
2020-05-15 12:00:00    31.31
2020-05-15 12:15:00    31.51
2020-05-15 12:30:00    32.15
2020-05-15 12:45:00    32.39
2020-05-15 13:00:00    32.62
2020-05-15 13:15:00    32.50
2020-05-15 13:30:00    32.52
2020-05-15 13:45:00    32.68
Freq: 15T, Name: ambient_temp, dtype: float64
```

We can use *ExpandingWindowFeatures* to create, for example, 2 new expanding window features by finding the mean and maximum value of a pandas Series if we convert it to a pandas Dataframe using the method `to_frame()`:

```
win_f = ExpandingWindowFeatures(functions=["mean", "max"])

X_tr = win_f.fit_transform(X['ambient_temp'].to_frame())

X_tr.head()
```

```
                     ambient_temp  ambient_temp_expanding_mean  \
2020-05-15 12:00:00         31.31                          NaN
2020-05-15 12:15:00         31.51                    31.310000
2020-05-15 12:30:00         32.15                    31.410000
```

(continues on next page)

(continued from previous page)

```
2020-05-15 12:45:00         32.39                    31.656667
2020-05-15 13:00:00         32.62                    31.840000

                     ambient_temp_expanding_max
2020-05-15 12:00:00                         NaN
2020-05-15 12:15:00                       31.31
2020-05-15 12:30:00                       31.51
2020-05-15 12:45:00                       32.15
2020-05-15 13:00:00                       32.39
```

And if we do not want the original values of time series in the returned dataframe, we just need to remember to drop the original series after the transformation:

```
win_f = ExpandingWindowFeatures(
    functions=["mean", "max"],
    drop_original=True,
)

X_tr = win_f.fit_transform(X['ambient_temp'].to_frame())

X_tr.head()
```

```
                     ambient_temp_expanding_mean  ambient_temp_expanding_max
2020-05-15 12:00:00                          NaN                         NaN
2020-05-15 12:15:00                    31.310000                       31.31
2020-05-15 12:30:00                    31.410000                       31.51
2020-05-15 12:45:00                    31.656667                       32.15
2020-05-15 13:00:00                    31.840000                       32.39
```

Getting the name of the new features

We can easily obtain the name of the original and new variables with the method `get_feature_names_out`.

```
win_f = ExpandingWindowFeatures()

win_f.fit(X)

win_f.get_feature_names_out()
```

```
['ambient_temp',
 'module_temp',
 'irradiation',
 'color',
 'ambient_temp_expanding_mean',
 'module_temp_expanding_mean',
 'irradiation_expanding_mean']
```

See also

Check out the additional transformers to create rolling window features (*WindowFeatures*) or lag features, by lagging past values of the time series data (*LagFeatures*).

Tutorials and courses

For tutorials about this and other feature engineering methods for time series forecasting check out our online courses:

Fig. 107: Feature Engineering for Time Series Forecasting

Our courses are suitable for beginners and more advanced data scientists looking to forecast time series using traditional machine learning models, like linear regression or gradient boosting machines.

By purchasing them you are supporting Sole, the main developer of Feature-engine.

Fig. 108: Forecasting with Machine Learning

10.2.5 Other

Preprocessing

Feature-engine's preprocessing transformers apply general data pre-processing and transformation procedures.

MatchCategories

MatchCategories() ensures that categorical variables are encoded as pandas 'categorical' dtype instead of generic python 'object' or other dtypes.

Under the hood, 'categorical' dtype is a representation that maps each category to an integer, thus providing a more memory-efficient object structure than, for example, 'str', and allowing faster grouping, mapping, and similar operations on the resulting object.

MatchCategories() remembers the encodings or levels that represent each category, and can thus can be used to ensure that the correct encoding gets applied when passing categorical data to modeling packages that support this dtype, or to prevent unseen categories from reaching a further transformer or estimator in a pipeline, for example.

Let's explore this with an example. First we load the Titanic dataset and split it into a train and a test sets:

```
from feature_engine.preprocessing import MatchCategories
from feature_engine.datasets import load_titanic

# Load dataset
data = load_titanic(
    predictors_only=True,
    handle_missing=True,
    cabin="letter_only",
)

data['pclass'] = data['pclass'].astype('O')

# Split test and train
train = data.iloc[0:1000, :]
test = data.iloc[1000:, :]
```

Now, we set up *MatchCategories()* and fit it to the train set.

```
# set up the transformer
match_categories = MatchCategories(missing_values="ignore")

# learn the mapping of categories to integers in the train set
match_categories.fit(train)
```

MatchCategories() stores the mappings from the train set in its attribute:

```
# the transformer stores the mappings for categorical variables
match_categories.category_dict_
```

```
{'pclass': Int64Index([1, 2, 3], dtype='int64'),
 'sex': Index(['female', 'male'], dtype='object'),
```

(continues on next page)

(continued from previous page)

```
 'cabin': Index(['A', 'B', 'C', 'D', 'E', 'F', 'M', 'T'], dtype='object'),
 'embarked': Index(['C', 'Missing', 'Q', 'S'], dtype='object')}
```

If we transform the test dataframe using the same `match_categories` object, categorical variables will be converted to a 'category' dtype with the same numeration (mapping from categories to integers) that was applied to the train dataset:

```
# encoding that would be gotten from the train set
train.embarked.unique()
```

```
array(['S', 'C', 'Missing', 'Q'], dtype=object)
```

```
# encoding that would be gotten from the test set
test.embarked.unique()
```

```
array(['Q', 'S', 'C'], dtype=object)
```

```
# with 'match_categories', the encoding remains the same
match_categories.transform(train).embarked.cat.categories
```

```
Index(['C', 'Missing', 'Q', 'S'], dtype='object')
```

```
# this will have the same encoding as the train set
match_categories.transform(test).embarked.cat.categories
```

```
Index(['C', 'Missing', 'Q', 'S'], dtype='object')
```

If some category was not present in the training data, it will not mapped to any integer and will thus not get encoded. This behavior can be modified through the parameter `errors`:

```
# categories present in the train data
train.cabin.unique()
```

```
array(['B', 'C', 'E', 'D', 'A', 'M', 'T', 'F'], dtype=object)
```

```
# categories present in the test data - 'G' is new
test.cabin.unique()
```

```
array(['M', 'F', 'E', 'G'], dtype=object)
```

```
match_categories.transform(train).cabin.unique()
```

```
['B', 'C', 'E', 'D', 'A', 'M', 'T', 'F']
Categories (8, object): ['A', 'B', 'C', 'D', 'E', 'F', 'M', 'T']
```

```
# unseen category 'G' will not get mapped to any integer
match_categories.transform(test).cabin.unique()
```

```
['M', 'F', 'E', NaN]
Categories (8, object): ['A', 'B', 'C', 'D', 'E', 'F', 'M', 'T']
```

When to use the transformer

This transformer is useful when creating custom transformers for categorical columns, as well as when passing categorical columns to modeling packages which support them natively but leave the variable casting to the user, such as `lightgbm` or `glum`.

MatchVariables

MatchVariables() ensures that the columns in the test set are identical to those in the train set.

If the test set contains additional columns, they are dropped. Alternatively, if the test set lacks columns that were present in the train set, they will be added with a value determined by the user, for example np.nan. *MatchVariables()* will also return the variables in the order seen in the train set.

Let's explore this with an example. First we load the Titanic dataset and split it into a train and a test set:

```
from feature_engine.preprocessing import MatchVariables
from feature_engine.datasets import load_titanic

# Load dataset
data = load_titanic(
    predictors_only=True,
    cabin="letter_only",
)

data['pclass'] = data['pclass'].astype('O')

# Split test and train
train = data.iloc[0:1000, :]
test = data.iloc[1000:, :]
```

Now, we set up *MatchVariables()* and fit it to the train set.

```
# set up the transformer
match_cols = MatchVariables(missing_values="ignore")

# learn the variables in the train set
match_cols.fit(train)
```

MatchVariables() stores the variables from the train set in its attribute:

```
# the transformer stores the input variables
match_cols.feature_names_in_
```

```
['pclass',
 'survived',
 'sex',
 'age',
```

(continues on next page)

(continued from previous page)

```
 'sibsp',
 'parch',
 'fare',
 'cabin',
 'embarked']
```

Now, we drop some columns in the test set.

```
# Let's drop some columns in the test set for the demo
test_t = test.drop(["sex", "age"], axis=1)

test_t.head()
```

```
     pclass  survived  sibsp  parch     fare cabin embarked
1000      3         1      0      0   7.7500     n        Q
1001      3         1      2      0  23.2500     n        Q
1002      3         1      2      0  23.2500     n        Q
1003      3         1      2      0  23.2500     n        Q
1004      3         1      0      0   7.7875     n        Q
```

If we transform the dataframe with the dropped columns using *MatchVariables()*, we see that the new dataframe contains all the variables, and those that were missing are now back in the data, with np.nan values as default.

```
# the transformer adds the columns back
test_tt = match_cols.transform(test_t)

test_tt.head()
```

```
The following variables are added to the DataFrame: ['age', 'sex']
     pclass  survived  sex  age  sibsp  parch     fare cabin embarked
1000      3         1  NaN  NaN      0      0   7.7500     n        Q
1001      3         1  NaN  NaN      2      0  23.2500     n        Q
1002      3         1  NaN  NaN      2      0  23.2500     n        Q
1003      3         1  NaN  NaN      2      0  23.2500     n        Q
1004      3         1  NaN  NaN      0      0   7.7875     n        Q
```

Note how the missing columns were added back to the transformed test set, with missing values, in the position (i.e., order) in which they were in the train set.

Similarly, if the test set contained additional columns, those would be removed. To test that, let's add some extra columns to the test set:

```
# let's add some columns for the demo
test_t[['var_a', 'var_b']] = 0

test_t.head()
```

```
     pclass  survived  sibsp  parch     fare cabin embarked  var_a  var_b
1000      3         1      0      0   7.7500     n        Q      0      0
1001      3         1      2      0  23.2500     n        Q      0      0
1002      3         1      2      0  23.2500     n        Q      0      0
1003      3         1      2      0  23.2500     n        Q      0      0
1004      3         1      0      0   7.7875     n        Q      0      0
```

And now, we transform the data with *MatchVariables()*:

```
test_tt = match_cols.transform(test_t)

test_tt.head()
```

```
The following variables are added to the DataFrame: ['age', 'sex']
The following variables are dropped from the DataFrame: ['var_b', 'var_a']
     pclass  survived  sex  age  sibsp  parch     fare cabin embarked
1000      3         1  NaN  NaN      0      0   7.7500     n        Q
1001      3         1  NaN  NaN      2      0  23.2500     n        Q
1002      3         1  NaN  NaN      2      0  23.2500     n        Q
1003      3         1  NaN  NaN      2      0  23.2500     n        Q
1004      3         1  NaN  NaN      0      0   7.7875     n        Q
```

Now, the transformer simultaneously added the missing columns with NA as values and removed the additional columns from the resulting dataset.

However, if we look closely, the dtypes for the `sex` variable do not match. This could cause issues if other transformations depend upon having the correct dtypes.

```
train.sex.dtype
```

```
dtype('O')
```

```
test_tt.sex.dtype
```

```
dtype('float64')
```

Set the `match_dtypes` parameter to `True` in order to align the dtypes as well.

```
match_cols_and_dtypes = MatchVariables(missing_values="ignore", match_dtypes=True)
match_cols_and_dtypes.fit(train)

test_ttt = match_cols_and_dtypes.transform(test_t)
```

```
The following variables are added to the DataFrame: ['sex', 'age']
The following variables are dropped from the DataFrame: ['var_b', 'var_a']
The sex dtype is changing from  float64 to object
```

Now the dtype matches.

```
test_ttt.sex.dtype
```

```
dtype('O')
```

By default, *MatchVariables()* will print out messages indicating which variables were added, removed and altered. We can switch off the messages through the parameter `verbose`.

When to use the transformer

These transformer is useful in "predict then optimize type of problems". In such cases, a machine learning model is trained on a certain dataset, with certain input features. Then, test sets are "post-processed" according to scenarios that want to be modelled. For example, "what would have happened if the customer received an email campaign"? where the variable "receive_campaign" would be turned from 0 -> 1.

While creating these modelling datasets, a lot of meta data e.g., "scenario number", "time scenario was generated", etc, could be added to the data. Then we need to pass these data over to the model to obtain the modelled prediction.

MatchVariables() provides an easy an elegant way to remove the additional metadeta, while returning datasets with the input features in the correct order, allowing the different scenarios to be modelled directly inside a machine learning pipeline.

More details

You can also find a similar implementation of the example shown in this page in the following Jupyter notebook:

- Jupyter notebook

All notebooks can be found in a dedicated repository.

Scikit-learn Wrapper

Feature-engine's Scikit-learn wrappers wrap Scikit-learn transformers allowing their implementation only on a selected subset of features.

SklearnTransformerWrapper

The *SklearnTransformerWrapper()* applies Scikit-learn transformers to a selected group of variables. It works with transformers like the SimpleImputer, OrdinalEncoder, OneHotEncoder, KBinsDiscretizer, all scalers and also transformers for feature selection. Other transformers have not been tested, but we think it should work with most of them.

The *SklearnTransformerWrapper()* offers similar functionality to the ColumnTransformer class available in Scikit-learn. They differ in the implementation to select the variables and the output.

The *SklearnTransformerWrapper()* returns a pandas dataframe with the variables in the order of the original data. The ColumnTransformer returns a Numpy array, and the order of the variables may not coincide with that of the original dataset.

In the next code snippet we show how to wrap the SimpleImputer from Scikit-learn to impute only the selected variables.

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.impute import SimpleImputer
from feature_engine.wrappers import SklearnTransformerWrapper

# Load dataset
data = pd.read_csv('houseprice.csv')

# Separate into train and test sets
X_train, X_test, y_train, y_test = train_test_split(
```

(continues on next page)

(continued from previous page)

```
    data.drop(['Id', 'SalePrice'], axis=1),
    data['SalePrice'], test_size=0.3, random_state=0)

# set up the wrapper with the SimpleImputer
imputer = SklearnTransformerWrapper(transformer = SimpleImputer(strategy='mean'),
                                    variables = ['LotFrontage', 'MasVnrArea'])

# fit the wrapper + SimpleImputer
imputer.fit(X_train)

# transform the data
X_train = imputer.transform(X_train)
X_test = imputer.transform(X_test)
```

In the next snippet of code we show how to wrap the StandardScaler from Scikit-learn to standardize only the selected variables.

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from feature_engine.wrappers import SklearnTransformerWrapper

# Load dataset
data = pd.read_csv('houseprice.csv')

# Separate into train and test sets
X_train, X_test, y_train, y_test = train_test_split(
    data.drop(['Id', 'SalePrice'], axis=1),
    data['SalePrice'], test_size=0.3, random_state=0)

# set up the wrapper with the StandardScaler
scaler = SklearnTransformerWrapper(transformer = StandardScaler(),
                                    variables = ['LotFrontage', 'MasVnrArea'])

# fit the wrapper + StandardScaler
scaler.fit(X_train)

# transform the data
X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)
```

In the next snippet of code we show how to wrap the SelectKBest from Scikit-learn to select only a subset of the variables.

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.feature_selection import f_regression, SelectKBest
from feature_engine.wrappers import SklearnTransformerWrapper

# Load dataset
```

(continues on next page)

(continued from previous page)

```
data = pd.read_csv('houseprice.csv')

# Separate into train and test sets
X_train, X_test, y_train, y_test = train_test_split(
    data.drop(['Id', 'SalePrice'], axis=1),
    data['SalePrice'], test_size=0.3, random_state=0)

cols = [var for var in X_train.columns if X_train[var].dtypes !='O']

# let's apply the standard scaler on the above variables

selector = SklearnTransformerWrapper(
    transformer = SelectKBest(f_regression, k=5),
    variables = cols)

selector.fit(X_train.fillna(0), y_train)

# transform the data
X_train_t = selector.transform(X_train.fillna(0))
X_test_t = selector.transform(X_test.fillna(0))
```

Even though Feature-engine has its own implementation of OneHotEncoder, you may want to use Scikit-Learn's transformer in order to access different options, such as drop first Category. In the following example, we show you how to apply Scikit-learn's OneHotEncoder to a subset of categories using the :class:SklearnTransformerWrapper().

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder
from feature_engine.wrappers import SklearnTransformerWrapper

# Load dataset
def load_titanic():
    data = pd.read_csv('https://www.openml.org/data/get_csv/16826755/phpMYEkMl')
    data = data.replace('?', np.nan)
    data['cabin'] = data['cabin'].astype(str).str[0]
    data['pclass'] = data['pclass'].astype('O')
    data['embarked'].fillna('C', inplace=True)
    data.drop(["name", "home.dest", "ticket", "boat", "body"], axis=1, inplace=True)
    return data

df = load_titanic()

X_train, X_test, y_train, y_test= train_test_split(
    df.drop("survived", axis=1),
    df["survived"],
    test_size=0.2,
    random_state=42,
)

ohe = SklearnTransformerWrapper(
        OneHotEncoder(sparse=False, drop='first'),
```

(continues on next page)

(continued from previous page)

```
        variables = ['pclass','sex'])

ohe.fit(X_train)

X_train_transformed = ohe.transform(X_train)
X_test_transformed = ohe.transform(X_test)
```

We can examine the result by executing the following:

```
print(X_train_transformed.head())
```

The resulting dataframe is:

```
     age  sibsp  parch     fare cabin embarked  pclass_2  pclass_3  sex_male
772   17      0      0   7.8958     n        S       0.0       1.0       1.0
543   36      0      0     10.5     n        S       1.0       0.0       1.0
289   18      0      2    79.65     E        S       0.0       0.0       0.0
10    47      1      0  227.525     C        C       0.0       0.0       1.0
147  NaN      0      0     42.4     n        S       0.0       0.0       1.0
```

Let's say you want to use *SklearnTransformerWrapper()* in a more complex context. As you may note there are ? signs to denote unknown values. Due to the complexity of the transformations needed we'll use a Pipeline to impute missing values, encode categorical features and create interactions for specific variables using Scikit-Learn's PolynomialFeatures.

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import Pipeline
from feature_engine.datasets import load_titanic
from feature_engine.imputation import CategoricalImputer, MeanMedianImputer
from feature_engine.encoding import OrdinalEncoder
from feature_engine.wrappers import SklearnTransformerWrapper

X, y = load_titanic(
    return_X_y_frame=True,
    predictors_only=True,
    cabin="letter_only",
)

X_train, X_test, y_train, y_test= train_test_split(X, y, test_size=0.2, random_state=42)

pipeline = Pipeline(steps = [
    ('ci', CategoricalImputer(imputation_method='frequent')),
    ('mmi', MeanMedianImputer(imputation_method='mean')),
    ('od', OrdinalEncoder(encoding_method='arbitrary')),
    ('pl', SklearnTransformerWrapper(
        PolynomialFeatures(interaction_only = True, include_bias=False),
        variables=['pclass','sex']))
])

pipeline.fit(X_train)
```

(continues on next page)

(continued from previous page)

```
X_train_transformed = pipeline.transform(X_train)
X_test_transformed = pipeline.transform(X_test)

print(X_train_transformed.head())
```

```
           age  sibsp  parch      fare  cabin  embarked  pclass  sex  \
772  17.000000      0      0    7.8958      0         0     3.0  0.0
543  36.000000      0      0   10.5000      0         0     2.0  0.0
289  18.000000      0      2   79.6500      1         0     1.0  1.0
10   47.000000      1      0  227.5250      2         1     1.0  0.0
147  29.532738      0      0   42.4000      0         0     1.0  0.0

     pclass sex
772         0.0
543         0.0
289         1.0
10          0.0
147         0.0
```

More details

In the following Jupyter notebooks you can find more details about how to navigate the parameters of the *SklearnTransformerWrapper()* and also access the parameters of the Scikit-learn transformer wrapped, as well as the output of the transformations.

- Wrap sklearn categorical encoder
- Wrap sklearn KBinsDiscretizer
- Wrap sklearn SimpleImputer
- Wrap sklearn feature selectors
- Wrap sklearn scalers

The notebooks can be found in a dedicated repository.

10.2.6 Pipeline

Pipeline

Feature-engine's Pipeline is equivalent to Scikit-learn's pipeline, and in addition, it accepts the method `transform_x_y`, to adjust both X and y, in those cases where rows are removed from X.

Pipeline

Pipeline facilitates the chaining together of multiple estimators into a unified sequence. This proves beneficial as data processing frequently involves a predefined series of actions, such as feature selection, normalization, and training a machine learning model.

Feature-engine's *Pipeline* is different from scikit-learn's Pipeline in that our *Pipeline* supports transformers that remove rows from the dataset, like `DropMissingData`, `OutlierTrimmer`, `LagFeatures` and `WindowFeatures`.

When observations are removed from the training data set, *Pipeline* invokes the method `transform_x_y` available in these transformers, to adjust the target variable to the remaining rows.

The Pipeline serves various functions in this context:

Simplicity and encapsulation:

You need only call the `fit` and `predict` functions once on your data to fit an entire sequence of estimators.

Hyperparameter Optimization:

Grid search and random search can be performed over hyperparameters of all estimators in the pipeline simultaneously.

Safety

Using a pipeline prevent the leakage of statistics from test data into the trained model during cross-validation, by ensuring that the same data is used to fit the transformers and predictors.

Pipeline functions

Calling the `fit` function on the pipeline, is the same as calling `fit` on each individual estimator sequentially, transforming the input data and forwarding it to the subsequent step.

The pipeline will have all the methods present in the final estimator within it. For instance, if the last estimator is a classifier, the Pipeline can function as a classifier. Similarly, if the last estimator is a transformer, the pipeline inherits this functionality as well.

Setting up a Pipeline

The *Pipeline* is constructed utilizing a list of (key, value) pairs, wherein the key represents the desired name for the step, and the value denotes an estimator or a transformer object.

In the following example, we set up a *Pipeline* that drops missing data, then replaces categories with ordinal numbers, and finally fits a Lasso regression model.

```
import numpy as np
import pandas as pd
from feature_engine.imputation import DropMissingData
from feature_engine.encoding import OrdinalEncoder
from feature_engine.pipeline import Pipeline

from sklearn.linear_model import Lasso

X = pd.DataFrame(
    dict(
        x1=[2, 1, 1, 0, np.nan],
        x2=["a", np.nan, "b", np.nan, "a"],
```

(continues on next page)

(continued from previous page)

```
    )
)
y = pd.Series([1, 2, 3, 4, 5])

pipe = Pipeline(
    [
        ("drop", DropMissingData()),
        ("enc", OrdinalEncoder(encoding_method="arbitrary")),
        ("lasso", Lasso(random_state=10)),
    ]
)
# predict
pipe.fit(X, y)
preds_pipe = pipe.predict(X)
preds_pipe
```

In the output we see the predictions made by the pipeline:

```
array([2., 2.])
```

Accessing Pipeline steps

The *Pipeline*'s estimators are stored as a list within the `steps` attribute. We can use slicing notation to obtain a subset or partial pipeline within the Pipeline. This functionality is useful for executing specific transformations or their inverses selectively.

For example, this notation extracts the first step of the pipeline:

```
pipe[:1]
```

```
Pipeline(steps=[('drop', DropMissingData())])
```

This notation extracts the first **two** steps of the pipeline:

```
pipe[:2]
```

```
Pipeline(steps=[('drop', DropMissingData()),
                ('enc', OrdinalEncoder(encoding_method='arbitrary'))])
```

This notation extracts the last step of the pipeline:

```
pipe[-1:]
```

```
Pipeline(steps=[('lasso', Lasso(random_state=10))])
```

We can also select specific steps of the pipeline to check their attributes. For example, we can check the coefficients of the Lasso algorithm as follows:

```
pipe.named_steps["lasso"].coef_
```

And we see the coefficients:

```
array([-0.,  0.])
```

There was no relationship between the target and the variables, so it's fine to obtain these coefficients.

Let's instead check the ordinal encoder mappings for the categorical variables:

```
pipe.named_steps["enc"].encoder_dict_
```

We see the integers used to replace each category:

```
{'x2': {'a': 0, 'b': 1}}
```

Finding feature names in a Pipeline

The *Pipeline* includes a `get_feature_names_out()` method, similar to other transformers. By employing pipeline slicing, you can obtain the feature names entering each step.

Let's set up a Pipeline that adds new features to the dataset to make this more interesting:

```
import numpy as np
import pandas as pd
from feature_engine.imputation import DropMissingData
from feature_engine.encoding import OneHotEncoder
from feature_engine.pipeline import Pipeline

from sklearn.linear_model import Lasso

X = pd.DataFrame(
    dict(
        x1=[2, 1, 1, 0, np.nan],
        x2=["a", np.nan, "b", np.nan, "a"],
    )
)
y = pd.Series([1, 2, 3, 4, 5])

pipe = Pipeline(
    [
        ("drop", DropMissingData()),
        ("enc", OneHotEncoder()),
        ("lasso", Lasso(random_state=10)),
    ]
)
pipe.fit(X, y)
```

In the first step of the pipeline, no features are added, we just drop rows with nan. So if we execute `get_feature_names_out()` we should see just the 2 variables from the input dataframe:

```
pipe[:1].get_feature_names_out()
```

```
['x1', 'x2']
```

In the second step, we add binary variables for each category of x2, so x2 should disappear, and in its place, we should see the binary variables:

```
pipe[:2].get_feature_names_out()
```

```
['x1', 'x2_a', 'x2_b']
```

The last step is an estimator, that is, a machine learning model. Estimators don't support the method `get_feature_names_out()`. So if we apply this method to the entire pipeline, we'll get an error.

Accessing nested parameters

We can re-define, or re-set the parameters of the transformers and estimators within the pipeline. This is done under the hood by the Grid search and random search. But in case you need to change a parameter in a step of the *Pipeline*, this is how you do it:

```
pipe.set_params(lasso__alpha=10)
```

Here, we changed the alpha of the lasso regression algorithm to 10.

Best use: Dropping rows during data preprocessing

Feature-engine's *Pipeline* was designed to support transformers that remove rows from the dataset, like `DropMissingData`, `OutlierTrimmer`, `LagFeatures` and `WindowFeatures`.

We saw earlier in this page how to use *Pipeline* with `DropMissingData`. Let's now take a look at how to combine *Pipeline* with `LagFeatures` and `WindowFeaures` to do multiple step forecasting.

We start by making imports:

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

from sklearn.linear_model import Lasso
from sklearn.metrics import root_mean_squared_error
from sklearn.multioutput import MultiOutputRegressor

from feature_engine.timeseries.forecasting import (
    LagFeatures,
    WindowFeatures,
)
from feature_engine.pipeline import Pipeline
```

We'll use the Australia electricity demand dataset described here:

Godahewa, Rakshitha, Bergmeir, Christoph, Webb, Geoff, Hyndman, Rob, & Montero-Manso, Pablo. (2021). Australian Electricity Demand Dataset (Version 1) [Data set]. Zenodo. https://doi.org/10.5281/zenodo.4659727

```
url = "https://raw.githubusercontent.com/tidyverts/tsibbledata/master/data-raw/vic_elec/
↪VIC2015/demand.csv"
df = pd.read_csv(url)

df.drop(columns=["Industrial"], inplace=True)
```

(continues on next page)

(continued from previous page)

```
# Convert the integer Date to an actual date with datetime type
df["date"] = df["Date"].apply(
    lambda x: pd.Timestamp("1899-12-30") + pd.Timedelta(x, unit="days")
)

# Create a timestamp from the integer Period representing 30 minute intervals
df["date_time"] = df["date"] + \
    pd.to_timedelta((df["Period"] - 1) * 30, unit="m")

df.dropna(inplace=True)

# Rename columns
df = df[["date_time", "OperationalLessIndustrial"]]

df.columns = ["date_time", "demand"]

# Resample to hourly
df = (
    df.set_index("date_time")
    .resample("h")
    .agg({"demand": "sum"})
)

print(df.head())
```

Here, we see the first rows of data:

```
                        demand
date_time
2002-01-01 00:00:00  6919.366092
2002-01-01 01:00:00  7165.974188
2002-01-01 02:00:00  6406.542994
2002-01-01 03:00:00  5815.537828
2002-01-01 04:00:00  5497.732922
```

We'll predict the next 6 hours of energy demand. We'll use direct forecasting. Hence, we need to create 6 target variables, one for each step in the horizon:

```
horizon = 6
y = pd.DataFrame(index=df.index)
for h in range(horizon):
    y[f"h_{h}"] = df.shift(periods=-h, freq="h")
y.dropna(inplace=True)
df = df.loc[y.index]
print(y.head())
```

This is our target variable:

```
                            h_0          h_1          h_2          h_3  \
date_time
2002-01-01 00:00:00  6919.366092  7165.974188  6406.542994  5815.537828
2002-01-01 01:00:00  7165.974188  6406.542994  5815.537828  5497.732922
2002-01-01 02:00:00  6406.542994  5815.537828  5497.732922  5385.851060
```

(continues on next page)

(continued from previous page)

```
2002-01-01 03:00:00  5815.537828  5497.732922  5385.851060  5574.731890
2002-01-01 04:00:00  5497.732922  5385.851060  5574.731890  5457.770634

                              h_4          h_5
date_time
2002-01-01 00:00:00  5497.732922  5385.851060
2002-01-01 01:00:00  5385.851060  5574.731890
2002-01-01 02:00:00  5574.731890  5457.770634
2002-01-01 03:00:00  5457.770634  5698.152000
2002-01-01 04:00:00  5698.152000  5938.337614
```

Next, we split the data into a training set and a test set:

```
end_train = '2014-12-31 23:59:59'
X_train = df.loc[:end_train]
y_train = y.loc[:end_train]

begin_test = '2014-12-31 17:59:59'
X_test  = df.loc[begin_test:]
y_test = y.loc[begin_test:]
```

Next, we set up `LagFeatures` and `WindowFeatures` to create features from lags and windows:

```
lagf = LagFeatures(
    variables=["demand"],
    periods=[1, 2, 3, 4, 5, 6],
    missing_values="ignore",
    drop_na=True,
)


winf = WindowFeatures(
    variables=["demand"],
    window=["3h"],
    freq="1h",
    functions=["mean"],
    missing_values="ignore",
    drop_original=True,
    drop_na=True,
)
```

We wrap the lasso regression within the multioutput regressor to predict multiple targets:

```
lasso = MultiOutputRegressor(Lasso(random_state=0, max_iter=10))
```

Now, we assemble the steps in the *Pipeline* and fit it to the training data:

```
pipe = Pipeline(
    [
        ("lagf", lagf),
        ("winf", winf),
        ("lasso", lasso),
    ]
```

(continues on next page)

(continued from previous page)

```
).set_output(transform="pandas")

pipe.fit(X_train, y_train)
```

We can obtain the datasets with the predictors and the targets like this:

```
Xt, yt = pipe[:-1].transform_x_y(X_test, y_test)

X_test.shape, y_test.shape, Xt.shape, yt.shape
```

We see that the *Pipeline* has dropped some rows during the transformation and re-adjusted the target. The rows that were dropped were those necessary to create the first lags.

```
((1417, 1), (1417, 6), (1410, 7), (1410, 6))
```

We can examine the predictors training set, to make sure we are passing the right variables to the regression model:

```
print(Xt.head())
```

We see the input features:

```
                     demand_lag_1  demand_lag_2  demand_lag_3  demand_lag_4  \
date_time
2015-01-01 01:00:00   7804.086240   8352.992140   7571.301440   7516.472988
2015-01-01 02:00:00   7174.339984   7804.086240   8352.992140   7571.301440
2015-01-01 03:00:00   6654.283364   7174.339984   7804.086240   8352.992140
2015-01-01 04:00:00   6429.598010   6654.283364   7174.339984   7804.086240
2015-01-01 05:00:00   6412.785284   6429.598010   6654.283364   7174.339984

                     demand_lag_5  demand_lag_6  demand_window_3h_mean
date_time
2015-01-01 01:00:00   7801.201802   7818.461408            7804.086240
2015-01-01 02:00:00   7516.472988   7801.201802            7489.213112
2015-01-01 03:00:00   7571.301440   7516.472988            7210.903196
2015-01-01 04:00:00   8352.992140   7571.301440            6752.740453
2015-01-01 05:00:00   7804.086240   8352.992140            6498.888886
```

Now, we can make forecasts for the test set:

```
forecast = pipe.predict(X_test)

forecasts = pd.DataFrame(
    pipe.predict(X_test),
    index=Xt.loc[end_train:].index,
    columns=[f"step_{i+1}" for i in range(6)]

)

print(forecasts.head())
```

We see the 6 hr ahead energy demand prediction for each hour:

```
                        step_1       step_2       step_3       step_4  \
date_time
2015-01-01 01:00:00  7810.769000  7890.897914  8123.247406  8374.365708
2015-01-01 02:00:00  7049.673468  7234.890108  7586.593627  7889.608312
2015-01-01 03:00:00  6723.246357  7046.660134  7429.115933  7740.984091
2015-01-01 04:00:00  6639.543752  6962.661308  7343.941881  7616.240318
2015-01-01 05:00:00  6634.279747  6949.262247  7287.866893  7633.157948

                        step_5       step_6
date_time
2015-01-01 01:00:00  8569.220349  8738.027713
2015-01-01 02:00:00  8116.631154  8270.579148
2015-01-01 03:00:00  7937.918837  8170.531420
2015-01-01 04:00:00  7884.815566  8197.598425
2015-01-01 05:00:00  7979.920512  8321.363714
```

To learn more about direct forecasting and how to create features, check out our courses:

Fig. 109: Feature Engineering for Time Series Forecasting

Hyperparameter optimization

We can optimize the hyperparameters of the transformers and the estimators from a pipeline simultaneously.

We'll start by loading the titanic dataset:

Fig. 110: Forecasting with Machine Learning

```
from feature_
↪engine.datasets import load_titanic
from feature_
↪engine.encoding import OneHotEncoder
from feature_
↪engine.outliers import OutlierTrimmer
from feature_engine.pipeline import Pipeline

from sklearn.
↪linear_model import LogisticRegression
from sklearn.model_selection␣
↪import train_test_split, GridSearchCV
from sklearn.
↪preprocessing import StandardScaler

X, y = load_titanic(
    return_X_y_frame=True,
    predictors_only=True,
    handle_missing=True,
)


X_train, X_
↪test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=0,
)

print(X_train.head())
```

We see the first 5 rows from the training set below:

```
      pclass     sex        age␣
↪ sibsp  parch     fare    cabin embarked
501        2  female  13.000000␣
↪     0      1  19.5000  Missing        S
588        2  female   4.000000␣
↪     1      1  23.0000  Missing        S
402        2  female  30.000000␣
↪     1      0  13.8583  Missing        C
1193       3    male  29.881135␣
↪     0      0   7.7250  Missing        Q
686        3  female  22.000000␣
↪     0      0   7.7250  Missing        Q
```

Now, we set up a Pipeline:

```
pipe = Pipeline(
    [
    ␣
↪    ("outliers", OutlierTrimmer(variables=[
↪"age", "fare"])),
        ("enc", OneHotEncoder()),
        ("scaler", StandardScaler()),
        ("logit
↪", LogisticRegression(random_state=10)),
    ]
)
```

We establish the hyperparameter space to search:

```
param_grid={
    'logit__C': [0.1, 10.],
    'enc__top_categories': [None, 5],
   ␣
↪'outliers__capping_method': ["mad", 'iqr']
}
```

We do the grid search:

```
grid = GridSearchCV(
    pipe,
    param_grid=param_grid,
    cv=2,
    refit=False,
)

grid.fit(X_train, y_train)
```

And we can see the best hyperparameters for each step:

```
grid.best_params_
```

```
{'enc__top_categories': None,
 'logit__C': 0.1,
 'outliers__capping_method': 'iqr'}
```

And the best accuracy obtained with these hyperparameters:

```
grid.best_score_
```

```
0.7843822843822843
```

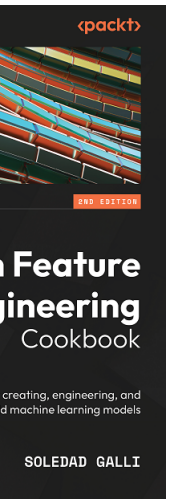
Additional resources

To learn more about feature engineering and data preprocessing, including missing data imputation, outlier removal or capping, variable transformation and encoding, check out our online course and book:

Or read our book:

Fig. 111: Feature Engineering for Machine Learning

Both our book and course are suitable for beginners and more advanced data scientists alike. By purchasing them you are supporting Sole, the main developer of Feature-engine.



ure Engineering

make_pipeline

make_pipeline is a shorthand for *Pipeline*. While to set up *Pipeline* we create tuples with step names and transformers or estimators, with *make_pipeline* we just add a sequence of transformers and estimators, and the names will be added automatically.

Setting up a Pipeline with make_pipeline

In the following example, we set up a `Pipeline` that drops missing data, then replaces categories with ordinal numbers, and finally fits a Lasso regression model.

```
import numpy as np
import pandas as pd
from feature_engine.imputation import DropMissingData
from feature_engine.encoding import OrdinalEncoder
from feature_engine.pipeline import make_pipeline

from sklearn.linear_model import Lasso

X = pd.DataFrame(
    dict(
        x1=[2, 1, 1, 0, np.nan],
        x2=["a", np.nan, "b", np.nan, "a"],
    )
)
y = pd.Series([1, 2, 3, 4, 5])

pipe = make_pipeline(
    DropMissingData(),
    OrdinalEncoder(encoding_method="arbitrary"),
    Lasso(random_state=10),
)
# predict
pipe.fit(X, y)
preds_pipe = pipe.predict(X)
preds_pipe
```

In the output we see the predictions made by the pipeline:

```
array([2., 2.])
```

The names of the pipeline were assigned automatically:

```
print(pipe)
```

```
Pipeline(steps=[('dropmissingdata', DropMissingData()),
                ('ordinalencoder
↪', OrdinalEncoder(encoding_method='arbitrary')),
                ('lasso', Lasso(random_state=10))])
```

The pipeline returned by *make_pipeline* has exactly the same characteristics than *Pipeline*. Hence, for additional guidelines, check out the *Pipeline* documentation.

Forecasting

Let's set up another pipeline to do direct forecasting:

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

from sklearn.linear_model import Lasso
from sklearn.metrics import root_mean_squared_error
from sklearn.multioutput import MultiOutputRegressor

from feature_engine.timeseries.forecasting import (
    LagFeatures,
    WindowFeatures,
)
from feature_engine.pipeline import make_pipeline
```

We'll use the Australia electricity demand dataset described here:

Godahewa, Rakshitha, Bergmeir, Christoph, Webb, Geoff, Hyndman, Rob, & Montero-Manso, Pablo. (2021). Australian Electricity Demand Dataset (Version 1) [Data set]. Zenodo. https://doi.org/10.5281/zenodo.4659727

```
url = "https://raw.githubusercontent.com/tidyverts/
↪tsibbledata/master/data-raw/vic_elec/VIC2015/demand.csv"
df = pd.read_csv(url)

df.drop(columns=["Industrial"], inplace=True)

# Convert␣
↪the integer Date to an actual date with datetime type
df["date"] = df["Date"].apply(
    lambda x: pd.
↪Timestamp("1899-12-30") + pd.Timedelta(x, unit="days")
)

# Create a timestamp␣
↪from the integer Period representing 30 minute intervals
df["date_time"] = df["date"] + \
    pd.to_timedelta((df["Period"] - 1) * 30, unit="m")

df.dropna(inplace=True)

# Rename columns
df = df[["date_time", "OperationalLessIndustrial"]]

df.columns = ["date_time", "demand"]
```

(continues on next page)

(continued from previous page)

```
# Resample to hourly
df = (
    df.set_index("date_time")
    .resample("h")
    .agg({"demand": "sum"})
)

print(df.head())
```

Here, we see the first rows of data:

```
                        demand
date_time
2002-01-01 00:00:00  6919.366092
2002-01-01 01:00:00  7165.974188
2002-01-01 02:00:00  6406.542994
2002-01-01 03:00:00  5815.537828
2002-01-01 04:00:00  5497.732922
```

We'll predict the next 3 hours of energy demand. We'll use direct forecasting. Let's create the target variable:

```
horizon = 3
y = pd.DataFrame(index=df.index)
for h in range(horizon):
    y[f"h_{h}"] = df.shift(periods=-h, freq="h")
y.dropna(inplace=True)
df = df.loc[y.index]
print(y.head())
```

This is our target variable:

```
                            h_0          h_1          h_2
date_time
2002-01-01 00:00:00  6919.366092  7165.974188  6406.542994
2002-01-01 01:00:00  7165.974188  6406.542994  5815.537828
2002-01-01 02:00:00  6406.542994  5815.537828  5497.732922
2002-01-01 03:00:00  5815.537828  5497.732922  5385.851060
2002-01-01 04:00:00  5497.732922  5385.851060  5574.731890
```

Next, we split the data into a training set and a test set:

```
end_train = '2014-12-31 23:59:59'
X_train = df.loc[:end_train]
y_train = y.loc[:end_train]

begin_test = '2014-12-31 17:59:59'
X_test  = df.loc[begin_test:]
y_test = y.loc[begin_test:]
```

Next, we set up `LagFeatures` and `WindowFeatures` to create features from lags and windows:

```
lagf = LagFeatures(
    variables=["demand"],
    periods=[1, 3, 6],
    missing_values="ignore",
    drop_na=True,
)


winf = WindowFeatures(
    variables=["demand"],
    window=["3h"],
    freq="1h",
    functions=["mean"],
    missing_values="ignore",
    drop_original=True,
    drop_na=True,
)
```

We wrap the lasso regression within the multioutput regressor to predict multiple targets:

```
lasso =␣
↪MultiOutputRegressor(Lasso(random_state=0, max_iter=10))
```

Now, we assemble Pipeline:

```
pipe = make_pipeline(lagf, winf, lasso)

print(pipe)
```

The steps' names were assigned automatically:

```
Pipeline(steps=[('lagfeatures',
                 ␣
↪      LagFeatures(drop_na=True, missing_values='ignore',
                       ␣
↪              periods=[1, 3, 6], variables=['demand'])),
                ('windowfeatures',
                 WindowFeatures(drop_
↪na=True, drop_original=True, freq='1h',
                                ␣
↪              functions=['mean'], missing_values='ignore',
                          ␣
↪                  variables=['demand'], window=['3h'])),
                ('multioutputregressor',
                 ␣
↪       MultiOutputRegressor(estimator=Lasso(max_iter=10,
                                       ␣
↪                                    random_state=0)))])
```

Let's fit the Pipeline:

```
pipe.fit(X_train, y_train)
```

Now, we can make forecasts for the test set:

```
forecast = pipe.predict(X_test)

forecasts = pd.DataFrame(
    pipe.predict(X_test),
    columns=[f"step_{i+1}" for i in range(3)]

)

print(forecasts.head())
```

We see the 3 hr ahead energy demand prediction for each hour:

```
        step_1       step_2       step_3
0  8031.043352  8262.804811  8484.551733
1  7017.158081  7160.568853  7496.282999
2  6587.938171  6806.903940  7212.741943
3  6503.807479  6789.946587  7195.796841
4  6646.981390  6970.501840  7308.359237
```

To learn more about direct forecasting and how to create features, check out our courses:

Fig. 113: Feature Engineering for Time Series Forecasting

Fig. 114: Forecasting with Machine Learning

Fig. 115: Feature Engineering for Machine Learning

Both our book and course are suitable for beginners and more advanced data scientists alike. By purchasing them you are supporting Sole, the main developer of Feature-engine.

10.2.7 Tools

Variable handling functions

This set of functions find variables of a specific type in a dataframe, or check that a list of variables is of a specified data type.

The `find` functions take a dataframe as an argument and returns a list with the names of the variables of the desired type.

The `check` functions check that the list of variables are all of the desired data type.

The `retain` functions select the variables in a list if they fulfill a condition.

You can use these functions to identify different sets of variables based on their data type to streamline your feature engineering pipelines or create your own Feature-engine or Scikit-learn compatible transformers.

find_all_variables

With *find_all_variables()* you can automatically capture in a list the names of all the variables in the dataset.

Let's create a toy dataset with numerical, categorical and datetime variables:

```
import pandas as pd
from sklearn.datasets import make_classification

X, y = make_classification(
    n_samples=1000,
    n_features=4,
    n_redundant=1,
    n_clusters_per_class=1,
    weights=[0.50],
    class_sep=2,
    random_state=1,
)

# transform arrays into pandas df and series
colnames = [f"num_var_{i+1}" for i in range(4)]
X = pd.DataFrame(X, columns=colnames)

X["cat_var1"] = ["Hello"] * 1000
X["cat_var2"] = ["Bye"] * 1000

X["date1"] = pd.date_range("2020-02-24", periods=1000,␣
↪freq="T")
X["date2"] = pd.date_range("2021-09-29", periods=1000,␣
↪freq="H")
X["date3"] = ["2020-02-24"] * 1000

print(X.head())
```

We see the resulting dataframe below:

```
   num_var_1  num_var_2  num_var_3  num_var_4 cat_var1 cat_var2  \
0  -1.558594   1.634123   1.556932   2.869318    Hello      Bye
1   1.499925   1.651008   1.159977   2.510196    Hello      Bye
2   0.277127  -0.263527   0.532159   0.274491    Hello      Bye
3  -1.139190  -1.131193   2.296540   1.189781    Hello      Bye
4  -0.530061  -2.280109   2.469580   0.365617    Hello      Bye

                 date1               date2       date3
0 2020-02-24 00:00:00 2021-09-29 00:00:00  2020-02-24
1 2020-02-24 00:01:00 2021-09-29 01:00:00  2020-02-24
2 2020-02-24 00:02:00 2021-09-29 02:00:00  2020-02-24
3 2020-02-24 00:03:00 2021-09-29 03:00:00  2020-02-24
4 2020-02-24 00:04:00 2021-09-29 04:00:00  2020-02-24
```

We can now use *find_all_variables()* to capture all the variable names in a list. So let's do that and then display the items in the list:

```
from feature_engine.variable_handling import find_all_variables

vars_all = find_all_variables(X)

vars_all
```

We see the variable names in the list below:

```
['num_var_1',
 'num_var_2',
 'num_var_3',
 'num_var_4',
 'cat_var1',
 'cat_var2',
 'date1',
 'date2',
 'date3']
```

We have the option to return the name of the variables of type categorical, object and numerical only, or in other words, to exclude datetime variables. We can do so as follows:

```
vars_all = find_all_variables(X, exclude_datetime=True)

vars_all
```

In the list below, we can see that variables of type datetime were ignored:

```
['num_var_1',
 'num_var_2',
 'num_var_3',
 'num_var_4',
 'cat_var1',
 'cat_var2',
 'date3']
```

find_categorical_variables

With *find_categorical_variables()* you can capture in a list the names of all the variables of type object or categorical in the dataset.

Let's create a toy dataset with numerical, categorical and datetime variables:

```
import pandas as pd
from sklearn.datasets import make_classification

X, y = make_classification(
    n_samples=1000,
    n_features=4,
    n_redundant=1,
    n_clusters_per_class=1,
    weights=[0.50],
    class_sep=2,
    random_state=1,
)

# transform arrays into pandas df and series
colnames = [f"num_var_{i+1}" for i in range(4)]
X = pd.DataFrame(X, columns=colnames)

X["cat_var1"] = ["Hello"] * 1000
X["cat_var2"] = ["Bye"] * 1000

X["date1"] = pd.date_range("2020-02-24", periods=1000, freq="T")
X["date2"] = pd.date_range("2021-09-29", periods=1000, freq="H")
X["date3"] = ["2020-02-24"] * 1000

print(X.head())
```

We see the resulting dataframe below:

```
   num_var_1  num_var_2  num_var_3  num_var_4 cat_var1 cat_var2  \
0  -1.558594   1.634123   1.556932   2.869318    Hello      Bye
1   1.499925   1.651008   1.159977   2.510196    Hello      Bye
2   0.277127  -0.263527   0.532159   0.274491    Hello      Bye
3  -1.139190  -1.131193   2.296540   1.189781    Hello      Bye
4  -0.530061  -2.280109   2.469580   0.365617    Hello      Bye

                 date1               date2       date3
0 2020-02-24 00:00:00 2021-09-29 00:00:00  2020-02-24
1 2020-02-24 00:01:00 2021-09-29 01:00:00  2020-02-24
2 2020-02-24 00:02:00 2021-09-29 02:00:00  2020-02-24
3 2020-02-24 00:03:00 2021-09-29 03:00:00  2020-02-24
4 2020-02-24 00:04:00 2021-09-29 04:00:00  2020-02-24
```

We can use *find_categorical_variables()* to capture the names of all variables of type object or categorical in a list.

So let's do that and then display the list:

```
from feature_engine.variable_handling import find_categorical_variables

var_cat = find_categorical_variables(X)

var_cat
```

We see the variable names in the list below:

```
['cat_var1', 'cat_var2']
```

Note that *find_categorical_variables()* will not return variables cast as object or categorical that could be parsed as datetime. That's why, the variable `date3` was excluded from the returned list.

If there are no categorical variables in the dataset, this function will raise an error.

find_datetime_variables

With *find_datetime_variables()* you can automatically capture in a list the names of all datetime variables in a dataset, whether they are parsed as datetime or object.

Let's create a toy dataset with numerical, categorical and datetime variables:

```
import pandas as pd
from sklearn.datasets import make_classification

X, y = make_classification(
    n_samples=1000,
    n_features=4,
    n_redundant=1,
    n_clusters_per_class=1,
    weights=[0.50],
    class_sep=2,
    random_state=1,
)

# transform arrays into pandas df and series
colnames = [f"num_var_{i+1}" for i in range(4)]
X = pd.DataFrame(X, columns=colnames)

X["cat_var1"] = ["Hello"] * 1000
X["cat_var2"] = ["Bye"] * 1000

X["date1"] = pd.date_range("2020-02-24", periods=1000, freq="T")
X["date2"] = pd.date_range("2021-09-29", periods=1000, freq="H")
X["date3"] = ["2020-02-24"] * 1000

print(X.head())
```

We see the resulting dataframe below:

```
   num_var_1  num_var_2  num_var_3  num_var_4 cat_var1 cat_var2  \
0  -1.558594   1.634123   1.556932   2.869318    Hello      Bye
1   1.499925   1.651008   1.159977   2.510196    Hello      Bye
```

(continues on next page)

(continued from previous page)

```
2   0.277127  -0.263527   0.532159   0.274491    Hello      Bye
3  -1.139190  -1.131193   2.296540   1.189781    Hello      Bye
4  -0.530061  -2.280109   2.469580   0.365617    Hello      Bye

                 date1               date2       date3
0 2020-02-24 00:00:00 2021-09-29 00:00:00  2020-02-24
1 2020-02-24 00:01:00 2021-09-29 01:00:00  2020-02-24
2 2020-02-24 00:02:00 2021-09-29 02:00:00  2020-02-24
3 2020-02-24 00:03:00 2021-09-29 03:00:00  2020-02-24
4 2020-02-24 00:04:00 2021-09-29 04:00:00  2020-02-24
```

The dataframe has 3 datetime variables, two of them are of type datetime and one of type object.

We can now use *find_datetime_variables()* to capture all datetime variables regardless of their data type. So let's do that and then display the list:

```
from feature_engine.variable_handling import find_datetime_variables

var_date = find_datetime_variables(X)

var_date
```

Below we see the variable names in the list:

```
['date1', 'date2', 'date3']
```

Note that *find_datetime_variables()* captures all 3 datetime variables. The first 2 are of type datetime, whereas the third variable is of type object. But as it can be parsed as datetime, it will be captured in the list as well.

find_numerical_variables

find_numerical_variables() returns a list with the names of the numerical variables in the dataset.

Let's create a toy dataset with numerical, categorical and datetime variables:

```
import pandas as pd
df = pd.DataFrame({
    "Name": ["tom", "nick", "krish", "jack"],
    "City": ["London", "Manchester", "Liverpool", "Bristol"],
    "Age": [20, 21, 19, 18],
    "Marks": [0.9, 0.8, 0.7, 0.6],
    "dob": pd.date_range("2020-02-24", periods=4, freq="T"),
})

print(df.head())
```

We see the resulting dataframe below:

```
    Name        City  Age  Marks                 dob
0    tom      London   20    0.9 2020-02-24 00:00:00
1   nick  Manchester   21    0.8 2020-02-24 00:01:00
2  krish   Liverpool   19    0.7 2020-02-24 00:02:00
3   jack     Bristol   18    0.6 2020-02-24 00:03:00
```

With *find_numerical_variables()* we capture the names of all numerical variables in a list. So let's do that and then display the list:

```
from feature_engine.variable_handling import find_numerical_variables

var_num = find_numerical_variables(df)

var_num
```

We see the names of the numerical variables in the list below:

```
['Age', 'Marks']
```

If there are no numerical variables in the dataset, *find_numerical_variables()* will raise an error.

find_categorical_and_numerical_variables

With *find_categorical_and_numerical_variables()* you can automatically capture in 2 separate lists the names of all the categorical and numerical variables in the dataset, respectively.

Let's create a toy dataset with numerical, categorical and datetime variables:

```
import pandas as pd
from sklearn.datasets import make_classification

X, y = make_classification(
    n_samples=1000,
    n_features=4,
    n_redundant=1,
    n_clusters_per_class=1,
    weights=[0.50],
    class_sep=2,
    random_state=1,
)

# transform arrays into pandas df and series
colnames = [f"num_var_{i+1}" for i in range(4)]
X = pd.DataFrame(X, columns=colnames)

X["cat_var1"] = ["Hello"] * 1000
X["cat_var2"] = ["Bye"] * 1000

X["date1"] = pd.date_range("2020-02-24", periods=1000, freq="T")
X["date2"] = pd.date_range("2021-09-29", periods=1000, freq="H")
X["date3"] = ["2020-02-24"] * 1000

print(X.head())
```

Below we see the resulting dataframe:

```
   num_var_1  num_var_2  num_var_3  num_var_4 cat_var1 cat_var2  \
0  -1.558594   1.634123   1.556932   2.869318    Hello      Bye
1   1.499925   1.651008   1.159977   2.510196    Hello      Bye
```

(continues on next page)

(continued from previous page)

```
2   0.277127  -0.263527   0.532159   0.274491    Hello      Bye
3  -1.139190  -1.131193   2.296540   1.189781    Hello      Bye
4  -0.530061  -2.280109   2.469580   0.365617    Hello      Bye

                 date1               date2       date3
0 2020-02-24 00:00:00 2021-09-29 00:00:00  2020-02-24
1 2020-02-24 00:01:00 2021-09-29 01:00:00  2020-02-24
2 2020-02-24 00:02:00 2021-09-29 02:00:00  2020-02-24
3 2020-02-24 00:03:00 2021-09-29 03:00:00  2020-02-24
4 2020-02-24 00:04:00 2021-09-29 04:00:00  2020-02-24
```

We can now use *find_categorical_and_numerical_variables()* to capture categorical and numerical variables in separate lists. So let's do that and then display the lists:

```
from feature_engine.variable_handling import find_categorical_and_numerical_variables

var_cat, var_num = find_categorical_and_numerical_variables(X)

var_cat, var_num
```

Below we see the names of the categorical variables, followed by the names of the numerical variables:

```
(['cat_var1', 'cat_var2'],
 ['num_var_1', 'num_var_2', 'num_var_3', 'num_var_4'])
```

We can also use *find_categorical_and_numerical_variables()* with a list of variables, to indentify their types:

```
var_cat, var_num = find_categorical_and_numerical_variables(X, ["num_var_1", "cat_var1"])

var_cat, var_num
```

We see the resulting lists below:

```
(['cat_var1'], ['num_var_1'])
```

If we pass a variable that is not of type numerical or categorical, *find_categorical_and_numerical_variables()* will return an error:

```
find_categorical_and_numerical_variables(X, ["num_var_1", "cat_var1", "date1"])
```

Below the error message:

```
TypeError: Some of the variables are neither numerical nor categorical.
```

check_all_variables

With *check_all_variables()* we can check that the variables in a list are present in the dataframe.

Let's create a toy dataset with numerical, categorical and datetime variables:

```
import pandas as pd
from sklearn.datasets import make_classification

X, y = make_classification(
    n_samples=1000,
    n_features=4,
    n_redundant=1,
    n_clusters_per_class=1,
    weights=[0.50],
    class_sep=2,
    random_state=1,
)

# transform arrays into pandas df and series
colnames = [f"num_var_{i+1}" for i in range(4)]
X = pd.DataFrame(X, columns=colnames)

X["cat_var1"] = ["Hello"] * 1000
X["cat_var2"] = ["Bye"] * 1000

X["date1"] = pd.date_range("2020-02-24", periods=1000, freq="T")
X["date2"] = pd.date_range("2021-09-29", periods=1000, freq="H")
X["date3"] = ["2020-02-24"] * 1000

print(X.head())
```

We see the resulting dataframe below:

```
   num_var_1  num_var_2  num_var_3  num_var_4 cat_var1 cat_var2  \
0  -1.558594   1.634123   1.556932   2.869318    Hello      Bye
1   1.499925   1.651008   1.159977   2.510196    Hello      Bye
2   0.277127  -0.263527   0.532159   0.274491    Hello      Bye
3  -1.139190  -1.131193   2.296540   1.189781    Hello      Bye
4  -0.530061  -2.280109   2.469580   0.365617    Hello      Bye

                 date1               date2       date3
0 2020-02-24 00:00:00 2021-09-29 00:00:00  2020-02-24
1 2020-02-24 00:01:00 2021-09-29 01:00:00  2020-02-24
2 2020-02-24 00:02:00 2021-09-29 02:00:00  2020-02-24
3 2020-02-24 00:03:00 2021-09-29 03:00:00  2020-02-24
4 2020-02-24 00:04:00 2021-09-29 04:00:00  2020-02-24
```

We can use *check_all_variables()* with a list of variable names to verify that the variables in the list are in the dataframe.

```
from feature_engine.variable_handling import check_all_variables

checked_vars = check_all_variables(X, ["num_var_1", "cat_var1", "date1"])
```

(continues on next page)

(continued from previous page)

```
checked_vars
```

The output is the list of variable names passed to the function:

```
['num_var_1', 'cat_var1', 'date1']
```

If we pass the name of a variable that is not in the dataframe, *check_all_variables()* will return an error:

```
check_all_variables(X, ["hola", "cat_var1", "date1"])
```

Below we see the error message:

```
KeyError: 'Some of the variables are not in the dataframe.'
```

check_categorical_variables

check_categorical_variables() checks that the variables in the list are of type object or categorical.

Let's create a toy dataset with numerical, categorical and datetime variables:

```
import pandas as pd
from sklearn.datasets import make_classification

X, y = make_classification(
    n_samples=1000,
    n_features=4,
    n_redundant=1,
    n_clusters_per_class=1,
    weights=[0.50],
    class_sep=2,
    random_state=1,
)

# transform arrays into pandas df and series
colnames = [f"num_var_{i+1}" for i in range(4)]
X = pd.DataFrame(X, columns=colnames)

X["cat_var1"] = ["Hello"] * 1000
X["cat_var2"] = ["Bye"] * 1000

X["date1"] = pd.date_range("2020-02-24", periods=1000, freq="T")
X["date2"] = pd.date_range("2021-09-29", periods=1000, freq="H")
X["date3"] = ["2020-02-24"] * 1000

print(X.head())
```

We see the resulting dataframe below:

```
   num_var_1  num_var_2  num_var_3  num_var_4 cat_var1 cat_var2  \
0  -1.558594   1.634123   1.556932   2.869318    Hello      Bye
1   1.499925   1.651008   1.159977   2.510196    Hello      Bye
```

(continues on next page)

(continued from previous page)

```
2   0.277127  -0.263527   0.532159   0.274491    Hello      Bye
3  -1.139190  -1.131193   2.296540   1.189781    Hello      Bye
4  -0.530061  -2.280109   2.469580   0.365617    Hello      Bye

                 date1               date2       date3
0 2020-02-24 00:00:00 2021-09-29 00:00:00  2020-02-24
1 2020-02-24 00:01:00 2021-09-29 01:00:00  2020-02-24
2 2020-02-24 00:02:00 2021-09-29 02:00:00  2020-02-24
3 2020-02-24 00:03:00 2021-09-29 03:00:00  2020-02-24
4 2020-02-24 00:04:00 2021-09-29 04:00:00  2020-02-24
```

Let's now check that 3 of the variables are of type numerical:

```
from feature_engine.variable_handling import check_categorical_variables

var_cat = check_categorical_variables(X, ["cat_var1", "date3"])

var_cat
```

Both variables are of type object and hence, will be in the resulting list:

```
['cat_var1', 'date3']
```

If we pass a variable that is not of type object or categorical, *check_categorical_variables()* will return an error:

```
check_categorical_variables(X, ["cat_var1", "num_var_1"])
```

Below we see the error message:

```
TypeError: Some of the variables are not categorical. Please cast them as object
or categorical before using this transformer.
```

check_datetime_variables

check_datetime_variables() checks that the variables in the list are, or can be parsed as datetime.

Let's create a toy dataset with numerical, categorical and datetime variables:

```
import pandas as pd
from sklearn.datasets import make_classification

X, y = make_classification(
    n_samples=1000,
    n_features=4,
    n_redundant=1,
    n_clusters_per_class=1,
    weights=[0.50],
    class_sep=2,
    random_state=1,
)

# transform arrays into pandas df and series
```

(continues on next page)

(continued from previous page)

```
colnames = [f"num_var_{i+1}" for i in range(4)]
X = pd.DataFrame(X, columns=colnames)

X["cat_var1"] = ["Hello"] * 1000
X["cat_var2"] = ["Bye"] * 1000

X["date1"] = pd.date_range("2020-02-24", periods=1000, freq="T")
X["date2"] = pd.date_range("2021-09-29", periods=1000, freq="H")
X["date3"] = ["2020-02-24"] * 1000

print(X.head())
```

We see the resulting dataframe below:

```
   num_var_1  num_var_2  num_var_3  num_var_4 cat_var1 cat_var2  \
0  -1.558594   1.634123   1.556932   2.869318    Hello      Bye
1   1.499925   1.651008   1.159977   2.510196    Hello      Bye
2   0.277127  -0.263527   0.532159   0.274491    Hello      Bye
3  -1.139190  -1.131193   2.296540   1.189781    Hello      Bye
4  -0.530061  -2.280109   2.469580   0.365617    Hello      Bye

                 date1               date2       date3
0 2020-02-24 00:00:00 2021-09-29 00:00:00  2020-02-24
1 2020-02-24 00:01:00 2021-09-29 01:00:00  2020-02-24
2 2020-02-24 00:02:00 2021-09-29 02:00:00  2020-02-24
3 2020-02-24 00:03:00 2021-09-29 03:00:00  2020-02-24
4 2020-02-24 00:04:00 2021-09-29 04:00:00  2020-02-24
```

The dataframe has 3 datetime variables, two of them are of type datetime and one of type object.

Let's check that a list of variables can be parsed as datetime:

```
from feature_engine.variable_handling import check_datetime_variables

var_date = check_datetime_variables(X, ["date2", "date3"])

var_date
```

In this case, both variables, if they can be parsed as datetime, will be in the resulting list:

```
['date2', 'date3']
```

If we pass a variable that can't be parsed as datetime, *check_datetime_variables()* will return an error:

```
check_datetime_variables(X, ["date2", "cat_var1"])
```

Below the error message:

```
TypeError: Some of the variables are not or cannot be parsed as datetime.
```

check_numerical_variables

check_numerical_variables() checks that the variables in the list are of type numerical.

Let's create a toy dataset with numerical, categorical and datetime variables:

```
import pandas as pd
df = pd.DataFrame({
    "Name": ["tom", "nick", "krish", "jack"],
    "City": ["London", "Manchester", "Liverpool", "Bristol"],
    "Age": [20, 21, 19, 18],
    "Marks": [0.9, 0.8, 0.7, 0.6],
    "dob": pd.date_range("2020-02-24", periods=4, freq="T"),
})

print(df.head())
```

We see the resulting dataframe below:

```
    Name        City  Age  Marks                 dob
0    tom      London   20    0.9 2020-02-24 00:00:00
1   nick  Manchester   21    0.8 2020-02-24 00:01:00
2  krish   Liverpool   19    0.7 2020-02-24 00:02:00
3   jack     Bristol   18    0.6 2020-02-24 00:03:00
```

Let's now check that 2 of the variables are of type numerical:

```
from feature_engine.variable_handling import check_numerical_variables

var_num = check_numerical_variables(df, ['Age', 'Marks'])

var_num
```

If the variables are numerical, the function returns their names in a list:

```
['Age', 'Marks']
```

If we pass a variable that is not of type numerical, *check_numerical_variables()* will return an error:

```
check_numerical_variables(df, ['Age', 'Name'])
```

Below we see the error message:

```
TypeError: Some of the variables are not numerical. Please cast them as numerical
before using this transformer.
```

retain_variables_if_in_df

retain_variables_if_in_df() returns the subset of variables in a list that is present in the dataset.

Let's create a toy dataset with numerical, categorical and datetime variables:

```
import pandas as pd
df = pd.DataFrame({
    "Name": ["tom", "nick", "krish", "jack"],
    "City": ["London", "Manchester", "Liverpool", "Bristol"],
    "Age": [20, 21, 19, 18],
    "Marks": [0.9, 0.8, 0.7, 0.6],
    "dob": pd.date_range("2020-02-24", periods=4, freq="T"),
})

print(df.head())
```

We see the resulting dataframe below:

```
    Name        City  Age  Marks                 dob
0    tom      London   20    0.9 2020-02-24 00:00:00
1   nick  Manchester   21    0.8 2020-02-24 00:01:00
2  krish   Liverpool   19    0.7 2020-02-24 00:02:00
3   jack     Bristol   18    0.6 2020-02-24 00:03:00
```

With *retain_variables_if_in_df()* we capture in a list, the names of the variables that are present in the dataset. So let's do that and then display the resulting list:

```
from feature_engine.variable_handling import retain_variables_if_in_df

vars_in_df = retain_variables_if_in_df(df, variables = ["Name", "City", "Dogs"])

var_in_df
```

We see the names of the subset of variables that are in the dataframe below:

```
['Name', 'City']
```

If none of variables in the list are in the dataset, *retain_variables_if_in_df()* will raise an error.

Uses

This function was originally developed for internal use.

When we run various feature selection transformers one after the other, for example, `DropConstantFeatures`, then `DropDuplicateFeatures`, and finally `RecursiveFeatureElimination`, we can't anticipate which variables will be dropped by each transformer. Hence, these transformers use *retain_variables_if_in_df()* under the hood, to select those variables that were entered by the user and that still remain in the dataset, before applying the selection algorithm.

We've now decided to expose this function as part of the `variable_handling` module. It might be useful, for example, if you are creating `Feature-engine` compatible selection transformers.

10.3 API

Full API documentation for Feature-engine transformers.

10.3.1 Transformation

Missing Data Imputation

Missing data refers to the absence of observed values in a dataset and is a common occurrence in any real-world data science project. In data science, missing data can lead to biased analysis, inaccurate predictions, and reduced reliability of models. Therefore, handling missing data has become one of the most important steps in a data preprocessing pipeline.

Feature-engine supports several imputation techniques to handle missing data. Here, we provide an overview of each of the supported methods.

MeanMedianImputer

class feature_engine.imputation.**MeanMedianImputer**(*imputation_method='median'*, *variables=None*)

The MeanMedianImputer() replaces missing data by the mean or median value of the variable. It works only with numerical variables.

You can pass a list of variables to impute. Alternatively, the MeanMedianImputer() will automatically select all variables of type numeric in the training set.

More details in the *User Guide*.

Parameters

imputation_method: str, default='median'
Desired method of imputation. Can take 'mean' or 'median'.

variables: list, default=None
The list of numerical variables to transform. If None, the transformer will automatically find and select all numerical variables.

Attributes

imputer_dict_:
Dictionary with the values to replace missing data in each variable.

variables_:
The group of variables that will be transformed.

feature_names_in_:
List with the names of features seen during `fit`.

n_features_in_:
The number of features in the train set used in fit.

Examples

```
>>> import pandas as pd
>>> import numpy as np
>>> from feature_engine.imputation import MeanMedianImputer
>>> X = pd.DataFrame(dict(
>>>         x1 = [np.nan,1,1,0,np.nan],
>>>         x2 = ["a", np.nan, "b", np.nan, "a"],
>>>         ))
>>> mmi = MeanMedianImputer(imputation_method='median')
>>> mmi.fit(X)
>>> mmi.transform(X)
    x1   x2
0  1.0    a
1  1.0  NaN
2  1.0    b
3  0.0  NaN
4  1.0    a
```

Methods

fit:	Learn the mean or median values.
fit_transform:	Fit to data, then transform it.
get_feature_names_out:	Get output feature names for transformation.
get_params:	Get parameters for this estimator.
set_params:	Set the parameters of this estimator.
transform:	Impute missing data.

`fit`(*X*, *y=None*)

Learn the mean or median values.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The training dataset.

y: pandas series or None, default=None
y is not needed in this imputation. You can pass None or y.

`fit_transform`(*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to `X` and `y` with optional parameters `fit_params` and returns a transformed version of `X`.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_feature_names_out`(*input_features=None*)

Get output feature names for transformation. In other words, returns the variable names of transformed dataframe.

Parameters

input_features
[array or list, default=None] This parameter exits only for compatibility with the Scikit-learn pipeline.

- If `None`, then `feature_names_in_` is used as feature names in.
- If an array or list, then `input_features` must match `feature_names_in_`.

Returns

feature_names_out: list
Transformed feature names.

rtype
`List`[`Union`[`str`, `int`]] ..

`get_metadata_routing`()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Replace missing data with the learned parameters.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The data to be transformed.

Returns

X_new: pandas dataframe of shape = [n_samples, n_features]
The dataframe without missing values in the selected variables.

rtype
`DataFrame` ..

ArbitraryNumberImputer

`class feature_engine.imputation.`**`ArbitraryNumberImputer`**(*arbitrary_number=999*, *variables=None*, *imputer_dict=None*)

The ArbitraryNumberImputer() replaces missing data by an arbitrary value determined by the user. It works only with numerical variables.

You can impute all variables with the same number by defining the variables to impute in `variables` and the imputation number in `arbitrary_number`. Alternatively, you can pass a dictionary with the variable names and the numbers to use for their imputation in the `imputer_dict` parameter.

More details in the *User Guide*.

Parameters

arbitrary_number: int or float, default=999
The number to replace the missing data. This parameter is used only if `imputer_dict` is None.

variables: list, default=None
The list of variables to impute. If None, the imputer will select all numerical variables. This parameter is used only if `imputer_dict` is None.

imputer_dict: dict, default=None
The dictionary of variables and the arbitrary numbers for their imputation. If specified, it overrides the above parameters.

Attributes

imputer_dict_:
Dictionary with the values to replace missing data in each variable.

variables_:
The group of variables that will be transformed.

feature_names_in_:
List with the names of features seen during `fit`.

n_features_in_:
The number of features in the train set used in fit.

See also:

feature_engine.imputation.EndTailImputer

Examples

```
>>> import pandas as pd
>>> import numpy as np
>>> from feature_engine.imputation import ArbitraryNumberImputer
>>> X = pd.DataFrame(dict(
>>>         x1 = [np.nan,1,1,0,np.nan],
>>>         x2 = ["a", np.nan, "b", np.nan, "a"],
>>>       ))
>>> ani = ArbitraryNumberImputer(arbitrary_number=-999)
>>> ani.fit(X)
>>> ani.transform(X)
      x1   x2
0 -999.0    a
1    1.0  NaN
2    1.0    b
3    0.0  NaN
4 -999.0    a
```

Methods

fit:	This transformer does not learn parameters.
fit_transform:	Fit to data, then transform it.
get_feature_names_out:	Get output feature names for transformation.
get_params:	Get parameters for this estimator.
set_params:	Set the parameters of this estimator.
transform:	Impute missing data.

`fit`(*X*, *y=None*)

This method does not learn any parameter.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The training dataset.

y: None
y is not needed in this imputation. You can pass None or y.

`fit_transform`(*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to `X` and `y` with optional parameters `fit_params` and returns a transformed version of `X`.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_feature_names_out`(*input_features=None*)

Get output feature names for transformation. In other words, returns the variable names of transformed dataframe.

Parameters

input_features
[array or list, default=None] This parameter exits only for compatibility with the Scikit-learn pipeline.

- If `None`, then `feature_names_in_` is used as feature names in.
- If an array or list, then `input_features` must match `feature_names_in_`.

Returns

feature_names_out: list
Transformed feature names.

rtype
`List`[`Union`[`str`, `int`]] ..

`get_metadata_routing`()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Replace missing data with the learned parameters.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The data to be transformed.

Returns

X_new: pandas dataframe of shape = [n_samples, n_features]
The dataframe without missing values in the selected variables.

rtype
`DataFrame` ..

EndTailImputer

`class` feature_engine.imputation.**EndTailImputer**(*imputation_method='gaussian'*, *tail='right'*, *fold=3*, *variables=None*)

The EndTailImputer() replaces missing data by a value at either tail of the distribution. It works only with numerical variables.

You can indicate the variables to impute in a list. Alternatively, the EndTailImputer() will automatically select all numerical variables.

The imputer first calculates the values at the end of the distribution for each variable (fit). The values at the end of the distribution are determined using the Gaussian limits, the the IQR proximity rule limits, or a factor of the maximum value:

Gaussian limits:

- right tail: mean + 3*std
- left tail: mean - 3*std

IQR limits:

- right tail: 75th quantile + 3*IQR
- left tail: 25th quantile - 3*IQR

where IQR is the inter-quartile range = 75th quantile - 25th quantile

Maximum value:

- right tail: max * 3
- left tail: not applicable

You can change the factor that multiplies the std, IQR or the maximum value using the parameter `fold` (we used `fold=3` in the examples above).

The imputer then replaces the missing data with the estimated values (transform).

More details in the *User Guide*.

Parameters

imputation_method: str, default='gaussian'
Method to be used to find the replacement values. Can take 'gaussian', 'iqr' or 'max'.

'gaussian': the imputer will use the Gaussian limits to find the values to replace missing data.

'iqr': the imputer will use the IQR limits to find the values to replace missing data.

'max': the imputer will use the maximum values to replace missing data. Note that if 'max' is passed, the parameter 'tail' is ignored.

tail: str, default='right'
Indicates if the values to replace missing data should be selected from the right or left tail of the variable distribution. Can take values 'left' or 'right'.

fold: int, default=3
Factor to multiply the std, the IQR or the Max values. Recommended values are 2 or 3 for Gaussian, or 1.5 or 3 for IQR.

variables: list, default=None
The list of numerical variables to transform. If None, the transformer will automatically find and select all numerical variables.

Attributes

imputer_dict_:
Dictionary with the values to replace missing data in each variable.

variables_:
The group of variables that will be transformed.

feature_names_in_:
List with the names of features seen during `fit`.

n_features_in_:
The number of features in the train set used in fit.

Examples

```
>>> import pandas as pd
>>> import numpy as np
>>> from feature_engine.imputation import EndTailImputer
>>> X = pd.DataFrame(dict(x1 = [np.nan,0.5, 0.5, 0,np.nan]))
>>> eti = EndTailImputer(imputation_method='gaussian', tail='right', fold=3)
>>> eti.fit(X)
>>> eti.transform(X)
         x1
0  1.199359
1  0.500000
2  0.500000
```

(continues on next page)

(continued from previous page)

```
3  0.000000
4  1.199359
```

Methods

fit:	Learn values to replace missing data.
fit_transform:	Fit to data, then transform it.
get_feature_names_out:	Get output feature names for transformation.
get_params:	Get parameters for this estimator.
set_params:	Set the parameters of this estimator.
transform:	Impute missing data.

`fit`(*X*, *y=None*)

Learn the values at the end of the variable distribution.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The training dataset.

y: pandas Series, default=None
y is not needed in this imputation. You can pass None or y.

`fit_transform`(*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to `X` and `y` with optional parameters `fit_params` and returns a transformed version of `X`.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_feature_names_out`(*input_features=None*)

Get output feature names for transformation. In other words, returns the variable names of transformed dataframe.

Parameters

input_features
[array or list, default=None] This parameter exits only for compatibility with the Scikit-learn pipeline.

- If `None`, then `feature_names_in_` is used as feature names in.
- If an array or list, then `input_features` must match `feature_names_in_`.

Returns

feature_names_out: list
Transformed feature names.

rtype
List[Union[str, int]] ..

`get_metadata_routing()`

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

`get_params(deep=True)`

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`set_params(**params)`

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform(X)`

Replace missing data with the learned parameters.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The data to be transformed.

Returns

X_new: pandas dataframe of shape = [n_samples, n_features]
The dataframe without missing values in the selected variables.

rtype
DataFrame ..

CategoricalImputer

class feature_engine.imputation.**CategoricalImputer**(*imputation_method='missing'*, *fill_value='Missing'*, *variables=None*, *return_object=False*, *ignore_format=False*)

The CategoricalImputer() replaces missing data in categorical variables by an arbitrary value or by the most frequent category.

The CategoricalImputer() imputes by default only categorical variables (type 'object' or 'categorical'). You can pass a list of variables to impute, or alternatively, the encoder will find and impute all categorical variables.

If you want to impute numerical variables with this transformer, there are 2 ways of doing it:

Option 1: Cast your numerical variables as object in the input dataframe before passing it to the transformer.

Option 2: Set `ignore_format=True`. Note that if you do this and do not pass the list of variables to impute, the imputer will automatically select and impute all variables in the dataframe.

More details in the *User Guide*.

Parameters

imputation_method: str, default='missing'
Desired method of imputation. Can be 'frequent' for frequent category imputation or 'missing' to impute with an arbitrary value.

fill_value: str, int, float, default='Missing'
User-defined value to replace missing data. Only used when `imputation_method='missing'`.

variables: list, default=None
The list of categorical variables that will be imputed. If None, the imputer will find and transform all variables of type object or categorical by default. You can also make the transformer accept numerical variables, see the parameter `ignore_format` below.

return_object: bool, default=False
If working with numerical variables cast as object, decide whether to return the variables as numeric or re-cast them as object. Note that pandas will re-cast them automatically as numeric after the transformation with the mode or with an arbitrary number.

ignore_format: bool, default=False
Whether the format in which the categorical variables are cast should be ignored. If false, the imputer will automatically select variables of type object or categorical, or check that the variables entered by the user are of type object or categorical. If True, the imputer will select all variables or accept all variables entered by the user, including those cast as numeric.

Attributes

imputer_dict_:
Dictionary with the values to replace missing data in each variable.

variables_:
The group of variables that will be transformed.

feature_names_in_:
List with the names of features seen during `fit`.

n_features_in_:
The number of features in the train set used in fit.

Examples

```
>>> import pandas as pd
>>> import numpy as np
>>> from feature_engine.imputation import CategoricalImputer
>>> X = pd.DataFrame(dict(
>>>         x1 = [np.nan,1,1,0,np.nan],
>>>         x2 = ["a", np.nan, "b", np.nan, "a"],
>>>         ))
>>> ci = CategoricalImputer(imputation_method='frequent')
>>> ci.fit(X)
>>> ci.transform(X)
    x1 x2
0  NaN  a
1  1.0  a
2  1.0  b
3  0.0  a
4  NaN  a
```

Methods

fit:	Learn the most frequent category or assign arbitrary value to variable.
fit_transform:	Fit to data, then transform it.
get_feature_names_out:	Get output feature names for transformation.
get_params:	Get parameters for this estimator.
set_params:	Set the parameters of this estimator.
transform:	Impute missing data.

fit(*X*, *y=None*)

Learn the most frequent category if the imputation method is set to frequent.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The training dataset.

y: pandas Series, default=None
y is not needed in this imputation. You can pass None or y.

fit_transform(*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to `X` and `y` with optional parameters `fit_params` and returns a transformed version of `X`.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_feature_names_out`(*input_features=None*)

Get output feature names for transformation. In other words, returns the variable names of transformed dataframe.

Parameters

input_features
[array or list, default=None] This parameter exits only for compatibility with the Scikit-learn pipeline.

- If `None`, then `feature_names_in_` is used as feature names in.
- If an array or list, then `input_features` must match `feature_names_in_`.

Returns

feature_names_out: list
Transformed feature names.

rtype
`List`[`Union`[`str`, `int`]] ..

`get_metadata_routing`()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Replace missing data with the learned parameters.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The data to be transformed.

Returns

X_new: pandas dataframe of shape = [n_samples, n_features]
The dataframe without missing values in the selected variables.

rtype
`DataFrame` ..

RandomSampleImputer

`class` feature_engine.imputation.**RandomSampleImputer**(*variables=None*, *random_state=None*, *seed='general'*, *seeding_method='add'*)

The RandomSampleImputer() replaces missing data with a random sample extracted from the variables in the training set.

The RandomSampleImputer() works with both numerical and categorical variables.

Note

The Random samples used to replace missing values may vary from execution to execution. This may affect the results of your work. Thus, it is advisable to set a seed.

More details in the *User Guide*.

Parameters

variables: list, default=None
The list of variables to be imputed. If None, the imputer will select all variables in the train set.

random_state: int, str or list, default=None
The random_state can take an integer to set the seed when extracting the random samples. Alternatively, it can take a variable name or a list of variables, which values will be used to determine the seed, observation per observation.

seed: str, default='general'
Indicates whether the seed should be set for each observation with missing values, or if one seed should be used to impute all observations in one go.

'general': one seed will be used to impute the entire dataframe. This is equivalent to setting the seed in pandas.sample(random_state).

'observation': the seed will be set for each observation using the values of the variables indicated in the random_state for that particular observation.

seeding_method: str, default='add'
If more than one variable are indicated to seed the random sampling per observation, you can choose to combine those values as an addition or a multiplication. Can take the values 'add' or 'multiply'.

Attributes

X_:
Copy of the training dataframe from which to extract the random samples.

variables_:
The group of variables that will be transformed.

feature_names_in_:
List with the names of features seen during `fit`.

n_features_in_:
The number of features in the train set used in fit.

Examples

```
>>> import pandas as pd
>>> import numpy as np
>>> from feature_engine.imputation import RandomSampleImputer
>>> X = pd.DataFrame(dict(
>>>         x1 = [np.nan,1,1,0,np.nan],
>>>         x2 = ["a", np.nan, "b", np.nan, "a"],
>>>         ))
>>> rsi = RandomSampleImputer()
>>> rsi.fit(X)
>>> rsi.transform(X)
    x1 x2
0  1.0  a
1  1.0  b
2  1.0  b
3  0.0  a
4  1.0  a
```

Methods

fit:	Make a copy of the train set
fit_transform:	Fit to data, then transform it.
get_feature_names_out:	Get output feature names for transformation.
get_params:	Get parameters for this estimator.
set_params:	Set the parameters of this estimator.
transform:	Impute missing data.

`fit`(*X*, *y=None*)

Makes a copy of the train set. Only stores a copy of the variables to impute. This copy is then used to randomly extract the values to fill the missing data during transform.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The training dataset.

y: None
y is not needed in this imputation. You can pass None or y.

`fit_transform`(*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to `X` and `y` with optional parameters `fit_params` and returns a transformed version of `X`.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_feature_names_out`(*input_features=None*)

Get output feature names for transformation. In other words, returns the variable names of transformed dataframe.

Parameters

input_features
[array or list, default=None] This parameter exits only for compatibility with the Scikit-learn pipeline.

- If `None`, then `feature_names_in_` is used as feature names in.
- If an array or list, then `input_features` must match `feature_names_in_`.

Returns

feature_names_out: list
Transformed feature names.

rtype
`List`[`Union`[`str`, `int`]] ..

`get_metadata_routing`()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Replace missing data with random values taken from the train set.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The dataframe to be transformed.

Returns

X_new: pandas dataframe of shape = [n_samples, n_features]
The dataframe without missing values in the transformed variables.

rtype
`DataFrame` ..

AddMissingIndicator

`class` feature_engine.imputation.**AddMissingIndicator**(*missing_only=True*, *variables=None*)

The AddMissingIndicator() adds binary variables that indicate if data is missing (one indicator per variable). The added variables (missing indicators) are named with the original variable name plus '_na'.

The AddMissingIndicator() works for both numerical and categorical variables. You can pass a list with the variables for which the missing indicators should be added. Alternatively, the imputer will select and add missing indicators to all variables in the training set.

Note If `missing_only=True`, the imputer will add missing indicators only to those variables that show missing data during `fit()`. These may be a subset of the variables you indicated in `variables`.

More details in the *User Guide*.

Parameters

missing_only: bool, default=True
If missing indicators should be added to variables with missing data or to all variables.

True: indicators will be created only for those variables that showed missing data during `fit()`.

False: indicators will be created for all variables

variables: list, default=None
The list of variables to impute. If None, the imputer will find and select all variables.

Attributes

variables_:
List of variables for which the missing indicators will be created.

feature_names_in_:
List with the names of features seen during `fit`.

n_features_in_:
The number of features in the train set used in fit.

Examples

```
>>> import pandas as pd
>>> import numpy as np
>>> from feature_engine.imputation import AddMissingIndicator
>>> X = pd.DataFrame(dict(
>>>         x1 = [np.nan,1,1,0,np.nan],
>>>         x2 = ["a", np.nan, "b", np.nan, "a"],
>>>         ))
>>> ami = AddMissingIndicator()
>>> ami.fit(X)
>>> ami.transform(X)
    x1   x2  x1_na  x2_na
0  NaN    a      1      0
1  1.0  NaN      0      1
2  1.0    b      0      0
3  0.0  NaN      0      1
4  NaN    a      1      0
```

Methods

fit:	Find the variables for which the missing indicators will be created
fit_transform:	Fit to data, then transform it.
get_feature_names_out:	Get output feature names for transformation.
get_params:	Get parameters for this estimator.
set_params:	Set the parameters of this estimator.
transform:	Add the missing indicators.

`fit`(*X*, *y=None*)

Learn the variables for which the missing indicators will be created.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The training dataset.

y: pandas Series, default=None
y is not needed in this imputation. You can pass None or y.

`fit_transform`(*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to `X` and `y` with optional parameters `fit_params` and returns a transformed version of `X`.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_feature_names_out`(*input_features=None*)

Get output feature names for transformation. In other words, returns the variable names of transformed dataframe.

Parameters

input_features
[array or list, default=None] This parameter exits only for compatibility with the Scikit-learn pipeline.

- If `None`, then `feature_names_in_` is used as feature names in.
- If an array or list, then `input_features` must match `feature_names_in_`.

Returns

feature_names_out: list
Transformed feature names.

rtype
List[Union[str, int]] ..

`get_metadata_routing()`

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

`get_params(deep=True)`

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`set_params(**params)`

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform(X)`

Add the binary missing indicators.

Parameters

X
[pandas dataframe of shape = [n_samples, n_features]] The dataframe to be transformed.

Returns

X_new
[pandas dataframe of shape = [n_samples, n_features]] The dataframe containing the additional binary variables..

rtype
DataFrame ..

DropMissingData

class feature_engine.imputation.**DropMissingData**(*missing_only=True*, *threshold=None*, *variables=None*)

DropMissingData() deletes rows containing missing values. It provides similar functionality to `pandas.drop_na()`, but within the `fit` and `transform` framework.

It works for numerical and categorical variables. You can enter the list of variables for which missing values should be removed. Alternatively, the imputer will find and remove missing data in all dataframe variables.

More details in the *User Guide*.

Parameters

variables: list, default=None
The list of variables to consider for the imputation. If `None`, the imputer will check missing data in all variables in the dataframe. Alternatively, the imputer will evaluate missing data only in the variables in the list.

Note that if `missing_only=True`, missing data will be removed from variables that had missing data in the train set. These might be a subset of the variables indicated in the list.

missing_only: bool, default=True
If `True`, rows will be dropped when they show missing data in variables that had missing data during `fit()`. If `False`, rows will be dropped if there is missing data in any of the variables. This parameter only works when `threshold=None`, otherwise it is ignored.

threshold: int or float, default=None
Require that percentage of non-NA values in a row to keep it. If `threshold=1`, all variables need to have data to keep the row. If `threshold=0.5`, 50% of the variables need to have data to keep the row. If `threshold=0.01`, 10% of the variables need to have data to keep the row. If `thresh=None`, rows with NA in any of the variables will be dropped.

Attributes

variables_:
The variables for which missing data will be examined to decide if a row is dropped. The attribute `variables_` is different from the parameter `variables` when the latter is `None`, or when only a subset of the indicated variables show NA in the train set if `missing_only=True`.

feature_names_in_:
List with the names of features seen during `fit`.

n_features_in_:
The number of features in the train set used in fit.

Examples

```
>>> import pandas as pd
>>> import numpy as np
>>> from feature_engine.imputation import DropMissingData
>>> X = pd.DataFrame(dict(
>>>         x1 = [np.nan,1,1,0,np.nan],
>>>         x2 = ["a", np.nan, "b", np.nan, "a"],
>>>         ))
>>> dmd = DropMissingData()
```

(continues on next page)

(continued from previous page)

```
>>> dmd.fit(X)
>>> dmd.transform(X)
    x1 x2
2  1.0  b
```

Methods

fit:	Find the variables for which missing data should be evaluated.
fit_transform:	Fit to data, then transform it.
get_feature_names_out:	Get output feature names for transformation.
get_params:	Get parameters for this estimator.
set_params:	Set the parameters of this estimator.
return_na_data:	Returns a dataframe with the rows that contain missing data.
transform:	Remove rows with missing data.
transform_x_y:	Remove rows with missing data from X and y.

`fit`(*X*, *y=None*)

Find the variables for which missing data should be evaluated to decide if a row should be dropped.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The training data set.

y: pandas Series or dataframe, default=None
y is not needed in this imputation. You can pass None or y.

`fit_transform`(*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to `X` and `y` with optional parameters `fit_params` and returns a transformed version of `X`.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_feature_names_out`(*input_features=None*)

Get output feature names for transformation. In other words, returns the variable names of transformed dataframe.

Parameters

input_features
[array or list, default=None] This parameter exits only for compatibility with the Scikit-learn pipeline.

- If `None`, then `feature_names_in_` is used as feature names in.
- If an array or list, then `input_features` must match `feature_names_in_`.

Returns

feature_names_out: list
Transformed feature names.

rtype
`List`[`Union`[`str`, `int`]] ..

`get_metadata_routing`()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`return_na_data`(*X*)

Returns the subset of the dataframe with the rows with missing values. That is, the subset of the dataframe that would be removed with the `transform()` method. This method may be useful in production, for example if we want to store or log the removed observations, that is, rows that will not be fed into the model.

Parameters

X_na: pandas dataframe of shape = [n_samples_with_na, features]
The subset of the dataframe with the rows with missing data.

:rtype: :py:class:`~pandas.core.frame.DataFrame`

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Remove rows with missing data.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The dataframe to be transformed.

Returns

X_new: pandas dataframe
The complete case dataframe for the selected variables, of shape [n_samples - n_samples_with_na, n_features]

rtype
DataFrame ..

`transform_x_y`(*X*, *y*)

Transform, align and adjust both X and y based on the transformations applied to X, ensuring that they correspond to the same set of rows if any were removed from X.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The dataframe to transform.

y: pandas Series or Dataframe of length = n_samples
The target variable to transform. Can be multi-output.

Returns

X_new: pandas dataframe
The transformed dataframe of shape [n_samples - n_rows, n_features]. It may contain less rows than the original dataset.

y_new: pandas Series or DataFrame
The transformed target variable of length [n_samples - n_rows]. It contains as many rows as those left in X_new.

Missing data mechanisms

Data can go missing for several reasons, including:

- In surveys, respondents may choose not to answer specific questions due to privacy concerns or simply overlooking them.
- In healthcare data, not every patient might undergo a study on the efficacy of new medications due to logistical or financial constraints.
- Errors in data collection and storage can also lead to missing values.

The mechanisms that introduce missing data are known as completely at random (MCAR), missing at random (MAR) or missing not at random (NMAR).

Consequences of missing data

Missing data can significantly impact machine learning models and statistical analysis for several reasons:

- It introduces bias into machine learning model predictions or statistical tests.
- Certain machine learning models, such as those found in Scikit-learn, cannot handle datasets with missing values.

Popular machine learning algorithms like linear regression, logistic regression, support vector machine (SVM), or k-nearest neighbors (kNN) are not equipped to manage datasets containing NaN (Not a Number) or null values. Consequently, attempting to fit these models with such incomplete data will result in errors.

These factors underscore the importance of addressing missing data prior to model training, highlighting the necessity of employing data imputation techniques.

Missing Data Imputation

Missing data imputation refers to the process of estimating and replacing missing values within a dataset. It involves filling in the missing values with estimated values based on the known information in the dataset.

There are two types of missing data imputation: univariate and multivariate imputation.

Univariate data imputation

Univariate imputation addresses missing data within a variable solely based on information within that variable, without considering other variables in the dataset.

For instance, consider a dataframe with exam results of 50 college students, and 5 data points are missing. Univariate imputation fills these 5 missing values based on operations such as mean, median, or mode of the 45 observed values. Alternatively, the missing data can be filled with arbitrary predefined values, such as -1, 0, 999, -999, or 'Missing', among others.

Multivariate data imputation

In multivariate data imputation, we utilize observations from other variables in the dataset to estimate the values of missing observations. This method essentially imputes missing values by treating the imputation as a regression, using algorithms such as k-nearest neighbors or linear regression to estimate the missing values.

For example, let's say we have a dataset containing information on students' grades, ages, and IQ scores, all of which have missing values. In this scenario, we can predict the missing grade values by employing a regression model trained on existing grade data, using age and IQ as predictors. Subsequently, we can apply the same regression imputation approach to the other variables (age and IQ) in subsequent iterations.

Feature-engine currenty supports univariate imputation strategies. For multivariate imputation, check out Scikit-learn's iterative imputer.

Feature-engine's imputation methods

Feature-engine supports the following data imputation methods

- Mean-median imputation
- Arbitrary number imputation
- End tail imputation
- Random sample imputation
- Frequent category imputation
- Categorical imputation
- Complete case analysis
- Adding missing indicators

Feature-engine's imputers main characteristics

Transformer	Numerical variables	Categorical variables	Description
MeanMedianImputer()		×	Replaces missing values by the mean or median
ArbitraryNumberImputer()		x	Replaces missing values by an arbitrary value
EndTailImputer()		×	Replaces missing values by a value at the end of the distribution
CategoricalImputer()			Replaces missing values by the most frequent category or by an arbitrary value
RandomSampleImputer()			Replaces missing values by random value extractions from the variable
AddMissingIndicator()			Adds a binary variable to flag missing observations
DropMissingData()			Removes observations with missing data from the dataset

Mean-Median Imputation

Mean-median imputation replaces missing values in a numerical variable with the median or mean value of that variable.

If a variable follows a normal distribution, both the mean and the median are suitable options since they are equivalent. However, if a variable is skewed, median imputation is preferable as mean imputation can introduce bias toward the tail of the distribution.

This imputation method is suited if the data is missing completely at random (MCAR). If data is MCAR, then it is fair to assume that the missing values are close in value to the majority, that is, to the mean or median of the distribution.

Advantages:

- Fast and easy method to obtain complete data.

Limitations:

- Distorts the variance within a variable, as well as the covariance and correlation with other variables in the dataset.

Data imputed with the mean or median is commonly used to train linear regression and logistic regression models.

The *MeanMedianImputer()* implements mean-median imputation.

Arbitrary Number Imputation

Arbitrary number imputation replaces missing values in a numerical variable with an arbitrary number. Common values used for replacements are 0, 999, -999 (or other 9 combinations), or -1 (if the distribution is positive).

This imputation method is perfectly suited if the data is missing not at random (MNAR). This is because the method will flag the missing values with a predefined arbitrary value instead of replacing them with statistical estimates that make nan values look like the majority of the observations.

Advantages:

- Fast and easy way to obtain complete data.
- Flags missing values.

Limitations:

- Distorts in the variance within a variable, as well as the covariance and correlation with other variables in the dataset.
- It might hide or create outliers.
- Need to be careful not to choose an arbitrary value that is too similar to the mean or median.

Some models can be effectively trained with data that has undergone arbitrary number imputation, such as tree-based models, kNN, SVM, and ensemble models.

The *ArbitraryNumberImputer()* implements arbitrary number imputation.

End Tail Imputation

End tail imputation replaces missing values in a numerical variable with an arbitrary number located at the tail of the variable's distribution.

We can select the imputation value in one of 2 ways depending on the variable's distribution:

- If it's a normal distribution, the value can be set at the mean plus or minus 3 times the standard deviation.
- If it's a skewed distribution, the value can be set using the IQR.

This method is suitable for MNAR data. This is because this method will flag the missing value instead of replacing it with a value that is similar to the majority of observations.

Advantages:

- Fast and easy way to obtain complete datasets.
- Automates arbitrary value imputation.
- Flags missing values.

Limitations:

- Distortion of the original variance within a variable, as well as the covariance and correlation with other variables in the dataset.

- It might hide outliers.

Models like tree-based models, tree based models can be effectively trained on data imputed with end tail imputation.

The *EndTailImputer()* implements end tail imputation.

Random Sample Imputation

Random sample imputation replaces missing values in both numerical and categorical variables with a random value drawn from a distribution of that variable.

Since the replacement is drawn from the distribution of the original variable, the variance of the imputed data will be preserved. However, due to its randomness, we could obtain different imputation values on different code executions, which would lead to different machine learning model predictions. Therefore, make sure to set a proper seed during the imputation.

Random sample imputation is useful when we don't want to distort the distribution of the variable.

Advantages:

- Preserves the variance of a variable.

Limitations:

- Randomness.
- Distorts the relation with other variables.
- This imputation model is computationally more expensive than other methods.

The *RandomSampleImputer()* implements random sample imputation.

Frequent Category imputation

Frequent category imputation replaces missing values in categorical variables with the most frequent category in that variable.

Although *CategoricalImputer()* can impute both numerical and categorical variables, in practice frequent category imputation is more commonly used for categorical variable imputation.

This method is suited if the data is MCAR, as this imputation method replaces missing values with the most common observation in our variable.

Advantages:

- Fast and easy method to obtain complete data.

Limitations:

- Imputed values can distort the correlation with other variables.
- It can lead to an over-representation of the most frequent category.

Therefore, it's best to use this method if the missing values constitute a small percentage of the observations.

Tree-based models, kNN, SVM, and ensemble models can be effectively trained on data imputed with frequent category imputation.

The *CategoricalImputer()* implements frequent category imputation.

Categorical imputation

During categorical imputation, we replace missing values in a categorical variable with a specific new label, such as 'Missing' or 'NaN' for example. In essence, it consists of treating the missing observations as a category in itself.

This method is suited for MNAR data because it marks the missing values with a new label, instead of replacing them with statistical estimates that may introduce bias in our data.

Advantages:

- Fast and easy way to obtain complete data.
- Flags missing values.
- No assumption made on the data.

Limitations:

- If the proportion of missing values is little, creating an additional category might introduce noise.

The *CategoricalImputer()* implements categorical imputation.

Adding Missing Indicators

Adding missing indicators consists in adding binary variables to highlight if the values are missing. The missing indicator takes the value 0 if there is an observed value and 1 if the value was missing.

Adding missing indicators does not replace the missing data in itself. They just add the information to the data that some values were missing. Therefore, this method is never used alone. Normally, it's accompanied with other imputation methods, such as mean-median for numerical data or frequent category imputation for categorical data.

Advantages:

- Captures the importance of missing values.

Limitations:

- Expands the dimensionality of the data.

The *AddMissingIndicator()* adds missing indicators to the dataset.

Complete case analysis

Dropping missing data is the simplest method to deal with missing data. This procedure is known as complete case analysis or listwise deletion, meaning that the entire row will be excluded from analysis if any single value is missing.

This method is best suited for MCAR data and if the proportion of missing values is relatively small.

Advantages:

- Fast and easy way to obtain complete data.

Limitations:

- Reducing the sample size of available data.
- Potentially creating bias in our data, hence affecting data analysis.

The *DropMissingData()* implements complete case analysis.

Wrapping up

All Feature-engine supported data imputation methods are single imputation methods, or better said, univariate imputation methods.

There are alternative data imputation techniques that data scientists could also use, like:

Multiple Imputation: Multiple imputation generates several imputed datasets by randomly imputing missing values in each of the dataset. Suitable if the data is MAR.

Cold Deck Imputation: Cold deck imputation replaces missing values with values borrowed from a historical dataset.

Hot Deck Imputation: Hot deck imputation selects imputed values from a similar subset of observed data within the same dataset.

Multiple imputation of chained equations (MICE): MICE is a way of estimating the missing data as a regression based on the other variables in the dataset. It uses multiple rounds of imputation to improve the estimates at each iteration.

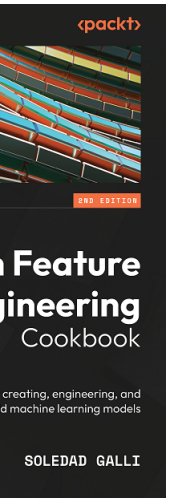
Additional resources

For tutorials about missing data imputation methods check out these resources:

Fig. 117: Feature Engineering for Machine Learning

Our book:

Both our book and courses are suitable for beginners and more advanced data scientists alike. By purchasing them you are supporting Sole, the main developer of Feature-engine.



ure Engineering

Categorical Encoding

Feature-engine's categorical encoders replace the categories of the variable with estimated or arbitrary numbers.

Summary of Feature-engine's encoders characteristics

Transformer	Regression	Classification	Multiclass	Description
`OneHotEncoder()`				Adds dummy variables to represent each category
`OrdinalEncoder()`				Replaces categories with an integer
`CountFreuencyEncoder()`				Replaces categories with their count or frequency
`MeanEncoder()`			x	Replaces categories with the targe mean value
`WoEEncoder()`	x		x	Replaces categories with the weight of the evidence
`DecisionTreeEncoder()`				Replaces categories with the predictions of a decision tree
`RareLabelEncoder()`				Groups infrequent categories into a single one

Feature-engine's categorical encoders encode only variables of type categorical or object by default. From version 1.1.0, you have the option to set the parameter `ignore_format` to True to make the transformers also accept numerical variables as input.

OneHotEncoder

class feature_engine.encoding.**OneHotEncoder**(*top_categories=None*, *drop_last=False*, *drop_last_binary=False*, *variables=None*, *ignore_format=False*)

The OneHotEncoder() replaces categorical variables by a set of binary variables representing each one of the unique categories in the variable.

The encoder has the option to create k or k-1 binary variables, where k is the number of unique categories.

The encoder has the additional option to generate binary variables only for the most popular categories, that is, the categories that are shared by the majority of the observations in the dataset. This behaviour can be specified with the parameter `top_categories`.

The encoder will encode only categorical variables by default (type 'object' or 'categorical'). You can pass a list of variables to encode. Alternatively, the encoder will find and encode all categorical variables (type 'object' or 'categorical').

With `ignore_format=True` you have the option to encode numerical variables as well. The procedure is identical, you can either enter the list of variables to encode, or the transformer will automatically select all variables.

The encoder first finds the categories to be encoded for each variable (fit). The encoder then creates one dummy variable per category for each variable (transform).

Note

New categories in the data to transform, that is, those that did not appear in the training set, will be ignored (no binary variable will be created for them). This means that observations with categories not present in the train set, will be encoded as 0 in all the binary variables.

Also Note

The original categorical variables are removed from the returned dataset when we apply the transform() method. In their place, the binary variables are returned.

More details in the *User Guide*.

Parameters

top_categories: int, default=None
If None, dummy variables will be created for each unique category of the variable. Alternatively, we can indicate in the number of most frequent categories to encode. In this case, dummy variables will be created only for those popular categories and the rest will be ignored, i.e., they will show the value 0 in all the binary variables. Note that if `top_categories` is not None, the parameter `drop_last` is ignored.

drop_last: boolean, default=False
Only used if `top_categories = None`. It indicates whether to create dummy variables for all the categories (k dummies), or if set to `True`, it will ignore the last binary variable and return k-1 dummies.

drop_last_binary: boolean, default=False
Whether to return 1 or 2 dummy variables for binary categorical variables. When a categorical variable has only 2 categories, then the second dummy variable created by one hot encoding can be completely redundant. Setting this parameter to `True`, will ensure that for every binary variable in the dataset, only 1 dummy is created.

variables: list, default=None
The list of categorical variables that will be encoded. If None, the encoder will find and transform all variables of type object or categorical by default. You can also make the transformer accept numerical variables, see the parameter `ignore_format`.

ignore_format: bool, default=False
This transformer operates only on variables of type object or categorical. To override this behaviour and allow the transformer to transform numerical variables as well, set to `True`.

If `ignore_format` is `False`, the encoder will automatically select variables of type object or categorical, or check that the variables entered by the user are of type object or categorical. If `True`, the encoder will select all variables or accept all variables entered by the user, including those cast as numeric.

In short, set to `True` when you want to encode numerical variables.

Attributes

encoder_dict_:
Dictionary with the categories for which dummy variables will be created.

variables_:
The group of variables that will be transformed.

variables_binary_:
List with binary variables identified in the data. That is, variables with only 2 categories.

feature_names_in_:
List with the names of features seen during `fit`.

n_features_in_:
The number of features in the train set used in fit.

Notes

If the variables are intended for linear models, it is recommended to encode into k-1 or top categories. If the variables are intended for tree based algorithms, it is recommended to encode into k or top n categories. If feature selection will be performed, then also encode into k or top n categories. Linear models evaluate all features during fit, while tree based models and many feature selection algorithms evaluate variables or groups of variables separately. Thus, if encoding into k-1, the last variable / category will not be examined.

References

One hot encoding of top categories was described in the following article:

[1]

Examples

```
>>> import pandas as pd
>>> from feature_engine.encoding import OneHotEncoder
>>> X = pd.DataFrame(dict(x1␣
↪= [1,2,3,4], x2 = ["a", "a", "b", "c"]))
>>> ohe = OneHotEncoder()
>>> ohe.fit(X)
>>> ohe.transform(X)
   x1  x2_a  x2_b  x2_c
0   1     1     0     0
1   2     1     0     0
2   3     0     1     0
3   4     0     0     1
```

Methods

fit:	Learn the unique categories per variable
fit_transform:	Fit to data, then transform it.
get_feature_names_out:	Get output feature names for transformation.
get_params:	Get parameters for this estimator.
set_params:	Set the parameters of this estimator.
transform:	Replace the categorical variables by the binary variables.

fit(*X*, *y=None*)

Learns the unique categories per variable. If top_categories is indicated, it will learn the most popular categories. Alternatively, it learns all unique categories per variable.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The training input samples. Can be the entire dataframe, not just seleted variables.

y: pandas series, default=None
Target. It is not needed in this encoded. You can pass y or None.

fit_transform(*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to `X` and `y` with optional parameters `fit_params` and returns a transformed version of `X`.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_feature_names_out`(*input_features=None*)

Get output feature names for transformation. In other words, returns the variable names of transformed dataframe.

Parameters

input_features
[array or list, default=None] This parameter exits only for compatibility with the Scikit-learn pipeline.

- If `None`, then `feature_names_in_` is used as feature names in.
- If an array or list, then `input_features` must match `feature_names_in_`.

Returns

feature_names_out: list
Transformed feature names.

rtype
`List`[`Union`[`str`, `int`]] ..

`get_metadata_routing`()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`inverse_transform`(*X*)

inverse_transform is not implemented for this transformer.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Replaces the categorical variables by the binary variables.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The data to transform.

Returns

X_new: pandas dataframe.
The transformed dataframe. The shape of the dataframe will be different from the original as it includes the dummy variables in place of the of the original categorical ones.

rtype
`DataFrame` ..

CountFrequencyEncoder

class feature_engine.encoding.**CountFrequencyEncoder**(*encoding_method='count'*, *variab*
missing_values='raise', *ignore_fo*
unseen='ignore')

The CountFrequencyEncoder() replaces categories by either the count or the percentage of observations per category.

For example in the variable colour, if 10 observations are blue, blue will be replaced by 10. Alternatively, if 10% of the observations are blue, blue will be replaced by 0.1.

The CountFrequencyEncoder() will encode only categorical variables by default (type 'object' or 'categorical'). You can pass a list of variables to encode. Alternatively, the encoder will find and encode all categorical variables (type 'object' or 'categorical').

With `ignore_format=True` you have the option to encode numerical variables as well. The procedure is identical, you can either enter the list of variables to encode, or the transformer will automatically select all variables.

The encoder first maps the categories to the counts or frequencies for each variable (fit). The encoder then replaces the categories with those numbers (transform).

More details in the *User Guide*.

Parameters

encoding_method: str, default='count'
Desired method of encoding.

'count': number of observations per category

'frequency': percentage of observations per category

variables: list, default=None
The list of categorical variables that will be encoded. If None, the encoder will find and transform all variables of type object or categorical by default. You can also make the transformer accept numerical variables, see the parameter `ignore_format`.

missing_values: string, default='raise'
Indicates if missing values should be ignored or raised. If `'raise'` the transformer will return an error if the the datasets to `fit` or `transform` contain missing values. If `'ignore'`, missing data will be ignored when learning parameters or performing the transformation.

ignore_format: bool, default=False
This transformer operates only on variables of type object or categorical. To override this behaviour and allow the transformer to transform numerical variables as well, set to `True`.

If `ignore_format` is `False`, the encoder will automatically select variables of type object or categorical, or check that the variables entered by the user are of type object or categorical. If `True`, the encoder will select all variables or accept all variables entered by the user, including those cast as numeric.

In short, set to `True` when you want to encode numerical variables.

unseen: string, default='ignore'
Indicates what to do when categories not present in the train set are encountered during transform. If `'raise'`, then unseen categories will raise an error. If `'ignore'`, then unseen categories will be encoded as NaN and a warning will be raised instead. If `'encode'`, unseen categories will be encoded as 0 (zero).

Attributes

encoder_dict_:
Dictionary with the count or frequency per category, per variable.

variables_:
The group of variables that will be transformed.

feature_names_in_:
List with the names of features seen during `fit`.

n_features_in_:
The number of features in the train set used in fit.

See also:

feature_engine.encoding.RareLabelEncoder
category_encoders.count.CountEncoder

Notes

NAN will be introduced when encoding categories that were not present in the training set. If this happens, try grouping infrequent categories using the RareLabelEncoder(), or set `unseen='encode'`.

There is a similar implementation in the open-source package Category encoders

Examples

```
>>> import pandas as pd
>>> from␣
↪feature_engine.encoding import CountFrequencyEncoder
>>> X = pd.DataFrame(dict(x1␣
↪= [1,2,3,4], x2 = ["c", "a", "b", "c"]))
>>> cf = CountFrequencyEncoder(encoding_method='count')
>>> cf.fit(X)
>>> cf.transform(X)
   x1  x2
0   1   2
1   2   1
2   3   1
3   4   2
```

```
>>>
↪ cf = CountFrequencyEncoder(encoding_method='frequency')
```

(continues on next page)

(continued from previous page)

```
>>> cf.fit(X)
>>> cf.transform(X)
   x1    x2
0   1  0.50
1   2  0.25
2   3  0.25
3   4  0.50
```

Methods

fit:	Learn the count or frequency per category, per variable.
fit_transform:	Fit to data, then transform it.
get_feature_names_out:	Get output feature names for transformation.
get_params:	Get parameters for this estimator.
set_params:	Set the parameters of this estimator.
inverse_transform:	Convert the data back to the original representation.
transform:	Encode the categories to numbers.

`fit`(*X*, *y=None*)

Learn the counts or frequencies which will be used to replace the categories.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The training dataset. Can be the entire dataframe, not just the variables to be transformed.

y: pandas Series, default = None
y is not needed in this encoder. You can pass y or None.

`fit_transform`(*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to `X` and `y` with optional parameters `fit_params` and returns a transformed version of `X`.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_feature_names_out`(*input_features=None*)

Get output feature names for transformation. In other words, returns the variable names of transformed dataframe.

Parameters

input_features
[array or list, default=None] This parameter exits only for compatibility with the Scikit-learn pipeline.

- If `None`, then `feature_names_in_` is used as feature names in.
- If an array or list, then `input_features` must match `feature_names_in_`.

Returns

feature_names_out: list
Transformed feature names.

rtype
`List`[`Union`[`str`, `int`]] ..

`get_metadata_routing`()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`inverse_transform`(*X*)

Convert the encoded variable back to the original values.

Parameters

X: pandas dataframe of shape = [n_samples, n_features].
The transformed dataframe.

Returns

X_tr: pandas dataframe of shape = [n_samples, n_features].
The un-transformed dataframe, with the categorical variables containing the original values.

rtype
DataFrame ..

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as Pipeline). The latter have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Replace categories with the learned parameters.

Parameters

X: pandas dataframe of shape = [n_samples, n_features].
The dataset to transform.

Returns

X_new: pandas dataframe of shape = [n_samples, n_features].
The dataframe containing the categories replaced by numbers.

rtype
DataFrame ..

OrdinalEncoder

class feature_engine.encoding.**OrdinalEncoder**(*encoding_method='ordered'*, *variables=No*
missing_values='raise', *ignore_format=Fa*
unseen='ignore')

The OrdinalEncoder() replaces categories by ordinal numbers (0, 1, 2, 3, etc). The numbers can be ordered based on the mean of the target per category, or assigned arbitrarily.

The encoder will encode only categorical variables by default (type 'object' or 'categorical'). You can pass a list of variables to encode. Alternatively, the encoder will find and encode all categorical variables (type 'object' or 'categorical').

With `ignore_format=True` you have the option to encode numerical variables as well. The procedure is identical, you can either enter the list of variables to encode, or the transformer will automatically select all variables.

The encoder first maps the categories to the numbers for each variable (fit). The encoder then transforms the categories to the mapped numbers (transform).

More details in the *User Guide*.

Parameters

encoding_method: str, default='ordered'
Desired method of encoding.

'ordered': the categories are numbered in ascending order according to the target mean value per category.

'arbitrary': categories are numbered arbitrarily.

variables: list, default=None
The list of categorical variables that will be encoded. If None, the encoder will find and transform all variables of type object or categorical by default. You can also make the transformer accept numerical variables, see the parameter `ignore_format`.

missing_values: string, default='raise'
Indicates if missing values should be ignored or raised. If `'raise'` the transformer will return an error if the the datasets to `fit` or `transform` contain missing values. If `'ignore'`, missing data will be ignored when learning parameters or performing the transformation.

ignore_format: bool, default=False
This transformer operates only on variables of type object or categorical. To override this behaviour and allow the transformer to transform numerical variables as well, set to `True`.

If `ignore_format` is `False`, the encoder will automatically select variables of type object or categorical, or check that the variables entered by the user are of type object or categorical. If `True`, the encoder will select all variables or accept all variables entered by the user, including those cast as numeric.

In short, set to `True` when you want to encode numerical variables.

unseen: string, default='ignore'
Indicates what to do when categories not present in the train set are encoun-

tered during transform. If `'raise'`, then unseen categories will raise an error. If `'ignore'`, then unseen categories will be encoded as NaN and a warning will be raised instead. If `'encode'`, unseen categories will be encoded as -1.

Attributes

encoder_dict_:
Dictionary with the ordinal number per category, per variable.

variables_:
The group of variables that will be transformed.

feature_names_in_:
List with the names of features seen during `fit`.

n_features_in_:
The number of features in the train set used in fit.

See also:

feature_engine.encoding.RareLabelEncoder
category_encoders.ordinal.OrdinalEncoder

Notes

NAN are introduced when encoding categories that were not present in the training dataset. If this happens, try grouping infrequent categories using the RareLabelEncoder().

There is a similar implementation in the the open-source package Category encoders

References

Encoding into integers ordered following target mean was discussed in the following talk at PyData London 2017:

[1]

Examples

```
>>> import pandas as pd
>>> from feature_engine.encoding import OrdinalEncoder
>>> X = pd.DataFrame(dict(x1␣
↪= [1,2,3,4], x2 = ["c", "a", "b", "c"]))
>>> y = pd.Series([0,1,1,0])
>>> od = OrdinalEncoder(encoding_method='arbitrary')
>>> od.fit(X)
>>> od.transform(X)
   x1  x2
0   1   0
1   2   1
2   3   2
3   4   0
```

You can also consider the order of the target variable:

```
>>> y = pd.Series([1,0,1,1])
>>> od = OrdinalEncoder(encoding_method='ordered')
>>> od.fit(X, y)
>>> od.transform(X)
   x1  x2
0   1   2
1   2   0
2   3   1
3   4   2
```

Methods

fit:	Find the integer to replace each category in each variable.
fit_transform:	Fit to data, then transform it.
get_feature_names_out:	Get output feature names for transformation.
get_params:	Get parameters for this estimator.
set_params:	Set the parameters of this estimator.
inverse_transform:	Convert the data back to the original representation.
transform:	Encode the categories to numbers.

`fit`(*X*, *y=None*)

Learn the numbers to be used to replace the categories in each variable.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The training input samples. Can be the entire dataframe, not just the variables to be encoded.

y: pandas series, default=None
The Target. Can be None if `encoding_method='arbitrary'`. Otherwise, y needs to be passed when fitting the transformer.

`fit_transform`(*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to `X` and `y` with optional parameters `fit_params` and returns a transformed version of `X`.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_feature_names_out`(*input_features=None*)

Get output feature names for transformation. In other words, returns the variable names of transformed dataframe.

Parameters

input_features
[array or list, default=None] This parameter exits only for compatibility with the Scikit-learn pipeline.

- If `None`, then `feature_names_in_` is used as feature names in.
- If an array or list, then `input_features` must match `feature_names_in_`.

Returns

feature_names_out: list
Transformed feature names.

rtype
`List`[`Union`[`str`, `int`]] ..

`get_metadata_routing`()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`inverse_transform`(*X*)

Convert the encoded variable back to the original values.

Parameters

X: pandas dataframe of shape = [n_samples, n_features].
The transformed dataframe.

Returns

X_tr: pandas dataframe of shape = [n_samples, n_features].
The un-transformed dataframe, with the categorical variables containing the original values.

rtype
`DataFrame` ..

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Replace categories with the learned parameters.

Parameters

X: pandas dataframe of shape = [n_samples, n_features].
The dataset to transform.

Returns

X_new: pandas dataframe of shape = [n_samples, n_features].
The dataframe containing the categories replaced by numbers.

rtype
`DataFrame` ..

MeanEncoder

class feature_engine.encoding.**MeanEncoder**(*variables=None*, *missing_values='raise'*, *ignore_format=False*, *unseen='ignore'*, *smooth*

The MeanEncoder() replaces categories by the mean value of the target for each category.

For example in the variable colour, if the mean of the target for blue, red and grey is 0.5, 0.8 and 0.1 respectively, blue is replaced by 0.5, red by 0.8 and grey by 0.1.

For rare categories, i.e., those with few observations, the mean target value might be less reliable. To mitigate poor estimates returned for rare categories, the mean target value can be determined as a mixture of the target mean value for the entire data set (also called the prior) and the mean target value for the category (the posterior), weighted by the number of observations:

$$mapping = (w_i)posterior + (1 - w_i)prior$$

where the weight is calculated as:

$$w_i = n_i t/(s + n_i t)$$

In the previous equation, t is the target variance in the entire dataset, s is the target variance within the category and n is the number of observations for the category.

The encoder will encode only categorical variables by default (type 'object' or 'categorical'). You can pass a list of variables to encode. Alternatively, the encoder will find and encode all categorical variables (type 'object' or 'categorical').

With `ignore_format=True` you have the option to encode numerical variables as well. The procedure is identical, you can either enter the list of variables to encode, or the transformer will automatically select all variables.

The encoder first maps the categories to the numbers for each variable (fit). The encoder then replaces the categories with those numbers (transform).

More details in the *User Guide*.

Parameters

variables: list, default=None
The list of categorical variables that will be encoded. If None, the encoder will find and transform all variables of type object or categorical by default. You can also make the transformer accept numerical variables, see the parameter `ignore_format`.

missing_values: string, default='raise'
Indicates if missing values should be ignored or raised. If `'raise'` the transformer will return an error if the the datasets to `fit` or `transform` contain missing values. If `'ignore'`, missing data will be ignored when learning parameters or performing the transformation.

ignore_format: bool, default=False
This transformer operates only on variables of type object or categorical. To override this behaviour and allow the transformer to transform numerical variables as well, set to `True`.

If `ignore_format` is `False`, the encoder will automatically select variables of type object or categorical, or check that the variables entered by the user are of type object or categorical. If `True`, the encoder will select all variables or accept all variables entered by the user, including those cast as numeric.

In short, set to `True` when you want to encode numerical variables.

unseen: string, default='ignore'
Indicates what to do when categories not present in the train set are encountered during transform. If `'raise'`, then unseen categories will raise an error. If `'ignore'`, then unseen categories will be encoded as NaN and a warning will be raised instead. If `'encode'`, unseen categories will be encoded with the prior.

smoothing: int, float, str, default=0.0
Smoothing factor. Should be >= 0. If 0 then no smoothing is applied, and the mean target value per category is returned without modification. If 'auto' then wi is calculated as described above and the category is encoded as the blended values of the prior and the posterior. If int or float, then the wi is calculated as ni / (ni+smoothing). Higher values lead to stronger smoothing (higher weight of prior).

Attributes

encoder_dict_:
Dictionary with the target mean value per category per variable.

variables_:
The group of variables that will be transformed.

feature_names_in_:
List with the names of features seen during `fit`.

n_features_in_:
The number of features in the train set used in fit.

See also:

feature_engine.encoding.RareLabelEncoder
category_encoders.target_encoder.TargetEncoder
category_encoders.m_estimate.MEstimateEncoder

Notes

NAN are introduced when encoding categories that were not present in the training dataset. If this happens, try grouping infrequent categories using the RareLabelEncoder().

Check also the related transformers in the the open-source package Category encoders

References

[1]

Examples

```
>>> import pandas as pd
>>> from feature_engine.encoding import MeanEncoder
>>> X = pd.DataFrame(dict(x1␣
↪= [1,2,3,4,5], x2 = ["c", "c", "c", "b", "a"]))
>>> y = pd.Series([0,1,1,1,0])
>>> me = MeanEncoder()
>>> me.fit(X,y)
>>> me.transform(X)
   x1        x2
0   1  0.666667
1   2  0.666667
2   3  0.666667
3   4  1.000000
4   5  0.000000
```

Methods

fit:	Learn the target mean value per category, per variable.
fit_transform:	Fit to data, then transform it.
get_feature_names_out:	Get output feature names for transformation.
get_params:	Get parameters for this estimator.
set_params:	Set the parameters of this estimator.
inverse_transform:	Convert the data back to the original representation.
transform:	Encode the categories to numbers.

fit(*X*, *y*)

Learn the mean value of the target for each category of the variable.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The training input samples. Can be the entire dataframe, not just the variables to be encoded.

y: pandas series
The target.

fit_transform(*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to `X` and `y` with optional parameters `fit_params` and returns a transformed version of `X`.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_feature_names_out`(*input_features=None*)

Get output feature names for transformation. In other words, returns the variable names of transformed dataframe.

Parameters

input_features
[array or list, default=None] This parameter exits only for compatibility with the Scikit-learn pipeline.

- If `None`, then `feature_names_in_` is used as feature names in.
- If an array or list, then `input_features` must match `feature_names_in_`.

Returns

feature_names_out: list
Transformed feature names.

rtype
`List`[`Union`[`str`, `int`]] ..

`get_metadata_routing`()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`inverse_transform`(*X*)

Convert the encoded variable back to the original values.

Note that if unseen was set to 'encode', then this method is not implemented.

Parameters

X: pandas dataframe of shape = [n_samples, n_features].
The transformed dataframe.

Returns

X_tr: pandas dataframe of shape = [n_samples, n_features].
The un-transformed dataframe, with the categorical variables containing the original values.

rtype
`DataFrame` ..

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Replace categories with the learned parameters.

Parameters

X: pandas dataframe of shape = [n_samples, n_features].
The dataset to transform.

Returns

X_new: pandas dataframe of shape = [n_samples, n_features].
The dataframe containing the categories replaced by numbers.

rtype
DataFrame ..

WoEEncoder

class feature_engine.encoding.**WoEEncoder**(*variables=None*, *ignore_format=False*, *unseen=fill_value=None*)

The WoEEncoder() replaces categories by the weight of evidence (WoE). The WoE was used primarily in the financial sector to create credit risk scorecards.

The encoder will encode only categorical variables by default (type 'object' or 'categorical'). You can pass a list of variables to encode. Alternatively, the encoder will find and encode all categorical variables (type 'object' or 'categorical').

With `ignore_format=True` you have the option to encode numerical variables as well. The procedure is identical, you can either enter the list of variables to encode, or the transformer will automatically select all variables.

The encoder first maps the categories to the weight of evidence for each variable (fit). The encoder then transforms the categories into the mapped numbers (transform).

This categorical encoding is exclusive for binary classification.

Note

The log(0) is not defined and the division by 0 is not defined. Thus, if any of the terms in the WoE equation are 0 for a given category, the encoder will return an error. If this happens, try grouping less frequent categories. Alternatively, you can now add a fill_value (see parameter below).

More details in the *User Guide*.

Parameters

variables: list, default=None
The list of categorical variables that will be encoded. If None, the encoder will find and transform all variables of type object or categorical by default. You can also make the transformer accept numerical variables, see the parameter `ignore_format`.

ignore_format: bool, default=False
This transformer operates only on variables of type object or categorical. To override this behaviour and allow the transformer to transform numerical variables as well, set to `True`.

If `ignore_format` is `False`, the encoder will automatically select variables of type object or categorical, or check that the variables entered by the user are

of type object or categorical. If `True`, the encoder will select all variables or accept all variables entered by the user, including those cast as numeric.

In short, set to `True` when you want to encode numerical variables.

unseen: string, default='ignore'
Indicates what to do when categories not present in the train set are encountered during transform. If `'raise'`, then unseen categories will raise an error. If `'ignore'`, then unseen categories will be encoded as NaN and a warning will be raised instead.

fill_value: int, float, default=None
When the numerator or denominator of the WoE calculation are zero, the WoE calculation is not possible. If `fill_value` is None (recommended), an error will be raised in those cases. Alternatively, fill_value will be used in place of denominators or numerators that equal zero.

Attributes

encoder_dict_:
Dictionary with the WoE per variable.

variables_:
The group of variables that will be transformed.

feature_names_in_:
List with the names of features seen during `fit`.

n_features_in_:
The number of features in the train set used in fit.

See also:

feature_engine.encoding.RareLabelEncoder
feature_engine.discretisation
category_encoders.woe.WOEEncoder

Notes

For details on the calculation of the weight of evidence visit: https://www.listendata.com/2015/03/weight-of-evidence-woe-and-information.html

NAN are introduced when encoding categories that were not present in the training dataset. If this happens, try grouping infrequent categories using the RareLabelEncoder().

There is a similar implementation in the the open-source package Category encoders

Examples

```
>>> import pandas as pd
>>> from feature_engine.encoding import WoEEncoder
>>> X = pd.DataFrame(dict(x1␣
↪= [1,2,3,4,5], x2 = ["b", "b", "b", "a", "a"]))
>>> y = pd.Series([0,1,1,1,0])
>>> woe = WoEEncoder()
>>> woe.fit(X, y)
>>> woe.transform(X)
   x1        x2
0   1  0.287682
1   2  0.287682
2   3  0.287682
3   4 -0.405465
4   5 -0.405465
```

Methods

fit:	Learn the WoE per category, per variable.
transform:	Encode the categories to numbers.
fit_transform:	Fit to data, then transform it.
get_feature_names_out:	Get output feature names for transformation.
get_params:	Get parameters for this estimator.
set_params:	Set the parameters of this estimator.
inverse_transform:	Convert the data back to the original representation.

fit(*X*, *y*)

Learn the WoE.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The training input samples. Can be the entire dataframe, not just the categorical variables.

y: pandas series.
Target, must be binary.

fit_transform(*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to `X` and `y` with optional parameters `fit_params` and returns a transformed version of `X`.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_feature_names_out`(*input_features=None*)

Get output feature names for transformation. In other words, returns the variable names of transformed dataframe.

Parameters

input_features
[array or list, default=None] This parameter exits only for compatibility with the Scikit-learn pipeline.

- If `None`, then `feature_names_in_` is used as feature names in.
- If an array or list, then `input_features` must match `feature_names_in_`.

Returns

feature_names_out: list
Transformed feature names.

rtype
`List`[`Union`[`str`, `int`]] ..

`get_metadata_routing`()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`inverse_transform`(*X*)

Convert the encoded variable back to the original values.

Parameters

X: pandas dataframe of shape = [n_samples, n_features].
The transformed dataframe.

Returns

X_tr: pandas dataframe of shape = [n_samples, n_features].
The un-transformed dataframe, with the categorical variables containing the original values.

rtype
`DataFrame` ..

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Replace categories with the learned parameters.

Parameters

X: pandas dataframe of shape = [n_samples, n_features].
The dataset to transform.

Returns

X_new: pandas dataframe of shape = [n_samples, n_features].
The dataframe containing the categories replaced by numbers.

rtype
DataFrame ..

DecisionTreeEncoder

class feature_engine.encoding.**DecisionTreeEncoder**(*encoding_method='arbitrary', cv=3 scoring='neg_mean_squared_error param_grid=None, regression=True random_state=None, variables=No ignore_format=False*)

The DecisionTreeEncoder() encodes categorical variables with predictions of a decision tree.

The encoder first fits a decision tree using a single feature and the target (fit), and then replaces the values of the original feature by the predictions of the tree (transform). The transformer will train a decision tree per every feature to encode.

The DecisionTreeEncoder() will encode only categorical variables by default (type 'object' or 'categorical'). You can pass a list of variables to encode or the encoder will find and encode all categorical variables.

With `ignore_format=True` you have the option to encode numerical variables as well. In this case, you can either enter the list of variables to encode, or the transformer will automatically select all variables.

More details in the *User Guide*.

Parameters

encoding_method: str, default='arbitrary'
The method used to encode the categories to numerical values before fitting the decision tree.

'ordered': the categories are numbered in ascending order according to the target mean value per category.

'arbitrary' : categories are numbered arbitrarily.

cv: int, cross-validation generator or an iterable, default=3
Determines the cross-validation splitting strategy. Possible inputs for cv are:

- None, to use cross_validate's default 5-fold cross validation
- int, to specify the number of folds in a (Stratified)KFold,
- **CV splitter**

  – (https://scikit-learn.org/stable/glossary.html#term-CV-splitter)

- An iterable yielding (train, test) splits as arrays of indices.

  For int/None inputs, if the estimator is a classifier and y is either binary or multiclass, StratifiedKFold is used. In all other cases, KFold is used. These splitters are instantiated with `shuffle=False` so the splits will be the same

across calls. For more details check Scikit-learn's `cross_validate`'s documentation.

scoring: str, default='neg_mean_squared_error'
Desired metric to optimise the performance for the decision tree. Comes from sklearn.metrics. See the DecisionTreeRegressor or DecisionTreeClassifier model evaluation documentation for more options: https://scikit-learn.org/stable/modules/model_evaluation.html

param_grid: dictionary, default=None
The hyperparameters for the decision tree to test with a grid search. The `param_grid` can contain any of the permitted hyperparameters for Scikit-learn's DecisionTreeRegressor() or DecisionTreeClassifier(). If None, then param_grid will optimise the 'max_depth' over `[1, 2, 3, 4]`.

regression: boolean, default=True
Indicates whether the encoder should train a regression or a classification decision tree.

random_state: int, default=None
The random_state to initialise the training of the decision tree. It is one of the parameters of the Scikit-learn's DecisionTreeRegressor() or DecisionTreeClassifier(). For reproducibility it is recommended to set the random_state to an integer.

variables: list, default=None
The list of categorical variables that will be encoded. If None, the encoder will find and transform all variables of type object or categorical by default. You can also make the transformer accept numerical variables, see the parameter `ignore_format`.

ignore_format: bool, default=False
This transformer operates only on variables of type object or categorical. To override this behaviour and allow the transformer to transform numerical variables as well, set to `True`.

If `ignore_format` is `False`, the encoder will automatically select variables of type object or categorical, or check that the variables entered by the user are of type object or categorical. If `True`, the encoder will select all variables or accept all variables entered by the user, including those cast as numeric.

In short, set to `True` when you want to encode numerical variables.

Attributes

encoder_:
sklearn Pipeline containing the ordinal encoder and the decision tree.

variables_:
The group of variables that will be transformed.

feature_names_in_:
List with the names of features seen during `fit`.

n_features_in_:
The number of features in the train set used in fit.

See also:

sklearn.ensemble.DecisionTreeRegressor
sklearn.ensemble.DecisionTreeClassifier
feature_engine.discretisation.DecisionTreeDiscretiser
feature_engine.encoding.RareLabelEncoder
feature_engine.encoding.OrdinalEncoder

Notes

The authors designed this method originally to work with numerical variables. We can replace numerical variables by the predictions of a decision tree utilising the DecisionTreeDiscretiser(). Here we extend this functionality to work also with categorical variables.

NAN are introduced when encoding categories that were not present in the training dataset. If this happens, try grouping infrequent categories using the RareLabelEncoder().

References

[1]

Examples

```
>>> import pandas as pd
>>>
 ↪ from feature_engine.encoding import DecisionTreeEncoder
>>> X = pd.DataFrame(dict(x1␣
 ↪= [1,2,3,4,5], x2 = ["b", "b", "b", "a", "a"]))
>>> y = pd.Series([2.2,4, 1.5, 3.2, 1.1])
>>> dte = DecisionTreeEncoder(cv=2)
>>> dte.fit(X, y)
>>> dte.transform(X)
   x1        x2
0   1  2.566667
1   2  2.566667
2   3  2.566667
3   4  2.150000
4   5  2.150000
```

You can also use it for classification by using `regression=False`.

```
>>> y = pd.Series([0,1,1,1,0])
>>> dte = DecisionTreeEncoder(regression=False, cv=2)
>>> dte.fit(X, y)
>>> dte.transform(X)
   x1        x2
0   1  0.666667
1   2  0.666667
2   3  0.666667
3   4  0.500000
4   5  0.500000
```

Methods

fit:	Fit a decision tree per variable.
fit_transform:	Fit to data, then transform it.
get_feature_names_out:	Get output feature names for transformation.
get_params:	Get parameters for this estimator.
set_params:	Set the parameters of this estimator.
transform:	Replace categorical variable by the predictions of the decision tre

`fit`(*X*, *y*)

Fit a decision tree per variable.

Parameters

X
[pandas dataframe of shape = [n_samples, n_features]] The training input samples. Can be the entire dataframe, not just the categorical variables.

y
[pandas series.] The target variable. Required to train the decision tree and for ordered ordinal encoding.

`fit_transform`(*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to `X` and `y` with optional parameters `fit_params` and returns a transformed version of `X`.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_feature_names_out`(*input_features=None*)

Get output feature names for transformation. In other words, returns the variable names of transformed dataframe.

Parameters

input_features
[array or list, default=None] This parameter exits only for compatibility with the Scikit-learn pipeline.

- If `None`, then `feature_names_in_` is used as feature names in.
- If an array or list, then `input_features` must match `feature_names_in_`.

Returns

feature_names_out: list
Transformed feature names.

rtype
`List`[`Union`[`str`, `int`]] ..

`get_metadata_routing`()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`inverse_transform`(*X*)

inverse_transform is not implemented for this transformer.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform(X)`

Replace categorical variables by the predictions of the decision tree.

Parameters

X
[pandas dataframe of shape = [n_samples, n_features]] The input samples.

Returns

X_new
[pandas dataframe of shape = [n_samples, n_features].] Dataframe with variables encoded with decision tree predictions.

rtype
DataFrame ..

RareLabelEncoder

`class feature_engine.encoding.RareLabelEncoder(tol=0.05, n_categories=10, max_n_cate replace_with='Rare', variables=None, missing_values='raise', ignore_format=`

The RareLabelEncoder() groups rare or infrequent categories in a new category called "Rare", or any other name entered by the user.

For example in the variable colour, if the percentage of observations for the categories magenta, cyan and burgundy are < 5 %, all those categories will be replaced by the new label "Rare".

Note

Infrequent labels can also be grouped under a user defined name, for example 'Other'. The name to replace infrequent categories is defined with the parameter `replace_with`.

The encoder will encode only categorical variables by default (type 'object' or 'categorical'). You can pass a list of variables to encode.Alternatively, the encoder will find and encode all categorical variables (type 'object' or 'categorical').

With `ignore_format=True` you have the option to encode numerical variables as well. The procedure is identical, you can either enter the list of variables to encode, or the transformer will automatically select all variables.

The encoder first finds the frequent labels for each variable (fit). The encoder then groups the infrequent labels under the new label 'Rare' or by another user defined string (transform).

More details in the *User Guide*.

Parameters

tol: float, default=0.05
The minimum frequency a label should have to be considered frequent. Categories with frequencies lower than tol will be grouped.

n_categories: int, default=10
The minimum number of categories a variable should have for the encoder to find frequent labels. If the variable contains less categories, all of them will be considered frequent.

max_n_categories: int, default=None
The maximum number of categories that should be considered frequent. If None, all categories with frequency above the tolerance (tol) will be considered frequent. If you enter 5, only the 5 most frequent categories will be retained and the rest grouped.

replace_with: string, intege or float, default='Rare'
The value that will be used to replace infrequent categories.

variables: list, default=None
The list of categorical variables that will be encoded. If None, the encoder will find and transform all variables of type object or categorical by default. You can also make the transformer accept numerical variables, see the parameter `ignore_format`.

missing_values: string, default='raise'
Indicates if missing values should be ignored or raised. If `'raise'` the transformer will return an error if the the datasets to `fit` or `transform` contain missing values. If `'ignore'`, missing data will be ignored when learning parameters or performing the transformation.

ignore_format: bool, default=False
This transformer operates only on variables of type object or categorical. To override this behaviour and allow the transformer to transform numerical variables as well, set to `True`.

If `ignore_format` is `False`, the encoder will automatically select variables of type object or categorical, or check that the variables entered by the user are of type object or categorical. If `True`, the encoder will select all variables or accept all variables entered by the user, including those cast as numeric.

In short, set to `True` when you want to encode numerical variables.

Attributes

encoder_dict_:
Dictionary with the frequent categories, i.e., those that will be kept, per variable.

variables_:
The group of variables that will be transformed.

feature_names_in_:
List with the names of features seen during `fit`.

n_features_in_:
The number of features in the train set used in fit.

Examples

```
>>> import pandas as pd
>>> from feature_engine.encoding import RareLabelEncoder
>>> X = pd.DataFrame(dict(x1␣
↪= [1,2,3,4,5,6], x2 = ["b", "b", "b", "b", "b", "a"]))
>>> rle = RareLabelEncoder(n_categories = 1, tol=0.2)
>>> rle.fit(X)
>>> rle.transform(X)
   x1    x2
0   1     b
1   2     b
2   3     b
3   4     b
4   5     b
5   6  Rare
```

Methods

fit:	Find frequent categories.
fit_transform:	Fit to data, then transform it.
get_feature_names_out:	Get output feature names for transformation.
get_params:	Get parameters for this estimator.
set_params:	Set the parameters of this estimator.
transform:	Group rare categories

fit(*X*, *y=None*)

Learn the frequent categories for each variable.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The training input samples. Can be the entire dataframe, not just selected variables

y: None
y is not required. You can pass y or None.

fit_transform(*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to `X` and `y` with optional parameters `fit_params` and returns a transformed version of `X`.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_feature_names_out`(*input_features=None*)

Get output feature names for transformation. In other words, returns the variable names of transformed dataframe.

Parameters

input_features
[array or list, default=None] This parameter exits only for compatibility with the Scikit-learn pipeline.

- If `None`, then `feature_names_in_` is used as feature names in.
- If an array or list, then `input_features` must match `feature_names_in_`.

Returns

feature_names_out: list
Transformed feature names.

rtype
`List`[`Union`[`str`, `int`]] ..

`get_metadata_routing`()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`inverse_transform`(*X*)

inverse_transform is not implemented for this transformer.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Group infrequent categories. Replace infrequent categories by the string 'Rare' or any other name provided by the user.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The input samples.

Returns

X: pandas dataframe of shape = [n_samples, n_features]
The dataframe where rare categories have been grouped.

rtype
`DataFrame` ..

StringSimilarityEncoder

class feature_engine.encoding.**StringSimilarityEncoder**(*top_categories=None*, *keywor*
missing_values='impute', *varia*
ignore_format=False)

The StringSimilarityEncoder() replaces categorical variables with a set of float variables that capture the similarity between the category names. The new variables have values between 0 and 1, where 0 indicates no similarity and 1 is an exact match between the names of the categories.

The similarity measure is a float in the range [0, 1]. It is defined as 2 * M / T, where T is the total number of elements in both categories being compared, and M is the number of matches. Note that this is 1 if the sequences are identical, and 0 if they have nothing in common.

For example, the similarity between the categories "dog" and "dig" is 0.66. T is the total number of elements in both categories, that is 6. There are 2 matches between the words, the letters d and g, so: 2 * M / T = 2 * 2 / 6 = 0.66.

This encoding is similar to one-hot encoding, in the sense that each category is encoded as a new variable. But the values, instead of 1 or 0, are the similarity between the observation's category and the dummy variable.

For example, if a variable has 3 categories, dog, dig and cat, StringSimilarityEncoder() will create 3 new variables, var_dog, var_dig and var_cat and the values would be for the observation dog: 1, 0.66 , 0. For the observation dig they would be 0.66, 1, 0. And for cat, they would be 0, 0, 1.

The encoder has the option to generate similarity variables only for the most popular categories, that is, the categories present in most observations. This behaviour can be specified with the parameter `top_categories`.

Missing values

StringSimilarityEncoder() will rreplace missing data with an empty string and then return the similarity to the remaining variables by default. Alternatively, it can be set to return an error if the variable has missing values, or to ignore them.

Unseen categories

StringSimilarityEncoder() handles unseen categories out-of-the-box by assigning a similarity measure to the other categories that were seen during `fit()`.

Categorical variables

The encoder will encode only categorical variables by default (type 'object' or 'categorical'). You can pass a list of variables to encode. Alternatively, the encoder will find and encode all categorical variables.

Numerical variables

With `ignore_format=True` you have the option to encode numerical variables as well. Encoding numerical variables with similarity measures make sense for example for variables like barcodes. In this case, you can either enter the list of variables to encode (recommended), or the transformer will automatically select all variables.

More details in the *User Guide*.

Parameters

top_categories: int, default=None
If None, dummy variables will be created for each unique category of the variable. Alternatively, we can indicate in the number of most frequent categories to encode. In this case, similarity variables will be created only for those popular categories.

missing_values: str, default='impute'
Indicates if missing values should be ignored, raised or imputed. If 'raise' the transformer will return an error if the datasets to `fit` or `transform` contain missing values. If 'ignore', missing data will be ignored when learning parameters or performing the transformation. If 'impute', the transformer will replace missing values with an empty string, '', and then return the similarity measures.

keywords: dict, default=None
Dictionary with a set of keywords to be used to create the similarity variables. The format should be: dict(feature: [keyword1, keyword2, ...]). The encoder will use these keywords to create the similarity variables. The dictionary can be defined for all the features to encode, or only for a subset of them. In this case, for the features not specified in the dictionary, the encoder will identify the categories from the data.

variables: list, default=None
The list of categorical variables that will be encoded. If None, the encoder will find and transform all variables of type object or categorical by default. You can also make the transformer accept numerical variables, see the parameter `ignore_format`.

ignore_format: bool, default=False
This transformer operates only on variables of type object or categorical. To override this behaviour and allow the transformer to transform numerical variables as well, set to `True`.

If `ignore_format` is `False`, the encoder will automatically select variables of type object or categorical, or check that the variables entered by the user are of type object or categorical. If `True`, the encoder will select all variables or accept all variables entered by the user, including those cast as numeric.

In short, set to `True` when you want to encode numerical variables.

Attributes

encoder_dict_:
Dictionary with the categories for which dummy variables will be created.

variables_:
The group of variables that will be transformed.

feature_names_in_:
List with the names of features seen during `fit`.

n_features_in_:
The number of features in the train set used in fit.

See also:

feature_engine.encoding.OneHotEncoder
dirty_cat.SimilarityEncoder

Notes

This encoder will encode unseen categories by measuring string similarity between seen and unseen categories.

No text preprocessing is applied before calculating the similarity.

The original categorical variables are removed from the returned dataset after the transformation. In their place, the binary variables are returned.

References

[1], [2]

Examples

```
>>> import pandas as pd
>>> from␣
↪feature_engine.encoding import StringSimilarityEncoder
>>> X = pd.DataFrame(dict(x1␣
↪= [1,2,3,4], x2 = ["dog", "dig", "dagger", "hi"]))
>>> sse = StringSimilarityEncoder()
>>> sse.fit(X)
>>> sse.transform(X)
   x1    x2_dog    x2_dig  x2_dagger  x2_hi
0   1  1.000000  0.666667   0.444444    0.0
1   2  0.666667  1.000000   0.444444    0.4
2   3  0.444444  0.444444   1.000000    0.0
3   4  0.000000  0.400000   0.000000    1.0
```

Methods

fit:	Learn the unique categories per variable.
fit_transform:	Fit to data, then transform it.
get_feature_names_out:	Get output feature names for transformation.
get_params:	Get parameters for this estimator.
set_params:	Set the parameters of this estimator.
transform:	Replace the categorical variables by the distance variables.

fit(*X*, *y=None*)

Learns the unique categories per variable. If top_categories is indicated, it will learn the most popular categories. Alternatively, it learns all unique categories per variable.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The training input samples. Can be the entire dataframe, not just the variables to encode.

y: pandas series, default=None
Target. It is not needed in this encoded. You can pass y or None.

`fit_transform`(*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to `X` and `y` with optional parameters `fit_params` and returns a transformed version of `X`.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_feature_names_out`(*input_features=None*)

Get output feature names for transformation. In other words, returns the variable names of transformed dataframe.

Parameters

input_features
[array or list, default=None] This parameter exits only for compatibility with the Scikit-learn pipeline.

- If `None`, then `feature_names_in_` is used as feature names in.
- If an array or list, then `input_features` must match `feature_names_in_`.

Returns

feature_names_out: list
Transformed feature names.

rtype
`List`[`Union`[`str`, `int`]] ..

`get_metadata_routing`()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`inverse_transform`(*X*)

inverse_transform is not implemented for this transformer.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Replaces the categorical variables with the similarity variables.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The data to transform.

Returns

X_new: pandas dataframe.
The transformed dataframe. The shape of the dataframe will be different from the original as it includes the similarity variables in place of the original categorical ones.

rtype
DataFrame ..

Other categorical encoding libraries

For additional categorical encoding transformations, visit the open-source package Category encoders.

Discretisation

Feature-engine's discretisation transformers transform continuous variables into discrete features. This is accomplished, in general, by sorting the variable values into continuous intervals.

Summary

Transformer	Functionality
EqualFrequencyDiscretiser()	Sorts values into intervals with similar number of observations.
EqualWidthDiscretiser()	Sorts values into intervals of equal size.
ArbitraryDiscretiser()	Sorts values into intervals predefined by the user.
DecisionTreeDiscretiser()	Replaces values by predictions of a decision tree, which are discrete.
GeometricWidthDiscretiser()	Sorts variable into geometrical intervals.

EqualFrequencyDiscretiser

class feature_engine.discretisation.**EqualFrequencyDiscretiser**(*variables=None*, *q=*
return_object=False
return_boundaries=
precision=3)

The EqualFrequencyDiscretiser() divides continuous numerical variables into contiguous equal frequency intervals, that is, intervals that contain approximately the same proportion of observations.

The EqualFrequencyDiscretiser() works only with numerical variables. A list of variables can be passed as argument. Alternatively, the discretiser will automatically select and transform all numerical variables.

The EqualFrequencyDiscretiser() first finds the boundaries for the intervals or quantiles for each variable. Then it transforms the variables, that is, it sorts the values into the intervals.

More details in the *User Guide*.

Parameters

variables: list, default=None
The list of numerical variables to transform. If None, the transformer will automatically find and select all numerical variables.

q: int, default=10
Desired number of equal frequency intervals / bins.

return_object: bool, default=False
Whether the the discrete variable should be returned as type numeric or type object. If you would like to encode the discrete variables with Feature-engine's categorical encoders, use True. Alternatively, keep the default to False.

return_boundaries: bool, default=False
Whether the output should be the interval boundaries. If True, it returns the interval boundaries. If False, it returns integers.

precision: int, default=3
The precision at which to store and display the bins labels.

Attributes

binner_dict_:
Dictionary with the interval limits per variable.

variables_:
The group of variables that will be transformed.

feature_names_in_:
List with the names of features seen during `fit`.

n_features_in_:
The number of features in the train set used in fit.

See also:

`pandas.qcut`
`sklearn.preprocessing.KBinsDiscretizer`

References

[1], [2]

Examples

```
>>> import pandas as pd
>>> import numpy as np
>>> from feature_
↪engine.discretisation import EqualFrequencyDiscretiser
>>> np.random.seed(42)
>>> X␣
↪= pd.DataFrame(dict(x = np.random.randint(1,100, 100)))
>>> efd = EqualFrequencyDiscretiser()
>>> efd.fit(X)
>>> efd.transform(X)["x"].value_counts()
8    12
```

(continues on next page)

(continued from previous page)

```
6    11
3    11
1    10
5    10
2    10
0    10
4     9
7     9
9     8
Name: x, dtype: int64
```

Methods

fit:	Find the interval limits.
fit_transform:	Fit to data, then transform it.
get_feature_names_out:	Get output feature names for transformation.
get_params:	Get parameters for this estimator.
set_params:	Set the parameters of this estimator.
transform:	Sort continuous variable values into the intervals.

`fit`(*X*, *y=None*)

Learn the limits of the equal frequency intervals.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The training dataset. Can be the entire dataframe, not just the variables to be transformed.

y: None
y is not needed in this encoder. You can pass y or None.

`fit_transform`(*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to `X` and `y` with optional parameters `fit_params` and returns a transformed version of `X`.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_feature_names_out`(*input_features=None*)

Get output feature names for transformation. In other words, returns the variable names of transformed dataframe.

Parameters

input_features
[array or list, default=None] This parameter exits only for compatibility with the Scikit-learn pipeline.

- If `None`, then `feature_names_in_` is used as feature names in.
- If an array or list, then `input_features` must match `feature_names_in_`.

Returns

feature_names_out: list
Transformed feature names.

rtype
`List`[`Union`[`str`, `int`]] ..

`get_metadata_routing`()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as Pipeline). The latter have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

transform(*X*)

Sort the variable values into the intervals.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The data to transform.

Returns

X_new: pandas dataframe of shape = [n_samples, n_features]
The transformed data with the discrete variables.

rtype
DataFrame ..

EqualWidthDiscretiser

class feature_engine.discretisation.**EqualWidthDiscretiser**(*variables=None*, *bins=10*, *return_object=False*, *return_boundaries=False*

The EqualWidthDiscretiser() divides continuous numerical variables into intervals of the same width, that is, equidistant intervals. Note that the proportion of observations per interval may vary.

The size of the interval is calculated as:

$$(max(X) - min(X))/bins$$

where bins, which is the number of intervals, is determined by the user.

The EqualWidthDiscretiser() works only with numerical variables. A list of variables can be passed as argument. Alternatively, the discretiser will automatically select all numerical variables.

The EqualWidthDiscretiser() first finds the boundaries for the intervals for each variable. Then, it transforms the variables, that is, sorts the values into the intervals.

More details in the *User Guide*.

Parameters

variables: list, default=None
The list of numerical variables to transform. If None, the transformer will automatically find and select all numerical variables.

bins: int, default=10
Desired number of equal width intervals / bins.

return_object: bool, default=False
Whether the the discrete variable should be returned as type numeric or type object. If you would like to encode the discrete variables with Feature-engine's categorical encoders, use True. Alternatively, keep the default to False.

return_boundaries: bool, default=False
Whether the output should be the interval boundaries. If True, it returns the interval boundaries. If False, it returns integers.

precision: int, default=3
The precision at which to store and display the bins labels.

Attributes

binner_dict_:
Dictionary with the interval limits per variable.

variables_:
The group of variables that will be transformed.

feature_names_in_:
List with the names of features seen during `fit`.

n_features_in_:
The number of features in the train set used in fit.

See also:

`pandas.cut`
`sklearn.preprocessing.KBinsDiscretizer`

References

[1], [2]

Examples

```
>>> import pandas as pd
>>> import numpy as np
>>> from feature_
↪engine.discretisation import EqualWidthDiscretiser
>>> np.random.seed(42)
>>> X␣
↪= pd.DataFrame(dict(x = np.random.randint(1,100, 100)))
>>> ewd = EqualWidthDiscretiser()
>>> ewd.fit(X)
>>> ewd.transform(X)["x"].value_counts()
9    15
6    15
0    13
5    11
8     9
7     8
2     8
1     7
3     7
4     7
Name: x, dtype: int64
```

Methods

fit:	Find the interval limits.
fit_transform:	Fit to data, then transform it.
get_feature_names_out:	Get output feature names for transformation.
get_params:	Get parameters for this estimator.
set_params:	Set the parameters of this estimator.
transform:	Sort continuous variable values into the intervals.

fit(*X*, *y=None*)

Learn the boundaries of the equal width intervals / bins for each variable.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The training dataset. Can be the entire dataframe, not just the variables to be transformed.

y: None
y is not needed in this encoder. You can pass y or None.

fit_transform(*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to `X` and `y` with optional parameters `fit_params` and returns a transformed version of `X`.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_feature_names_out`(*input_features=None*)

Get output feature names for transformation. In other words, returns the variable names of transformed dataframe.

Parameters

input_features
[array or list, default=None] This parameter exits only for compatibility with the Scikit-learn pipeline.

- If `None`, then `feature_names_in_` is used as feature names in.
- If an array or list, then `input_features` must match `feature_names_in_`.

Returns

feature_names_out: list
Transformed feature names.

rtype
`List`[`Union`[`str`, `int`]] ..

`get_metadata_routing`()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Sort the variable values into the intervals.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The data to transform.

Returns

X_new: pandas dataframe of shape = [n_samples, n_features]
The transformed data with the discrete variables.

rtype
`DataFrame` ..

ArbitraryDiscretiser

class feature_engine.discretisation.**ArbitraryDiscretiser**(*binning_dict*, *return_objec*
return_boundaries=False,
errors='ignore')

The ArbitraryDiscretiser() divides numerical variables into intervals which limits are determined by the user. Thus, it works only with numerical variables.

You need to enter a dictionary with variable names as keys, and a list with the limits of the intervals as values. For example the key could be the variable name 'var1' and the value the following list: [0, 10, 100, 1000]. The ArbitraryDiscretiser() will then sort var1 values into the intervals 0-10, 10-100, 100-1000, and var2 into 5-10, 10-15 and 15-20. Similar to `pandas.cut`.

More details in the *User Guide*.

Parameters

binning_dict: dict
The dictionary with the variable to interval limits pairs.

return_object: bool, default=False
Whether the the discrete variable should be returned as type numeric or type object. If you would like to encode the discrete variables with Feature-engine's categorical encoders, use True. Alternatively, keep the default to False.

return_boundaries: bool, default=False
Whether the output should be the interval boundaries. If True, it returns the interval boundaries. If False, it returns integers.

precision: int, default=3
The precision at which to store and display the bins labels.

errors: string, default='ignore'
Indicates what to do when a value is outside the limits indicated in the 'binning_dict'. If 'raise', the transformation will raise an error. If 'ignore', values outside the limits are returned as NaN and a warning will be raised instead.

Attributes

binner_dict_:
Dictionary with the interval limits per variable.

variables_:
The group of variables that will be transformed.

feature_names_in_:
List with the names of features seen during `fit`.

n_features_in_:
The number of features in the train set used in fit.

See also:

`pandas.cut`

Examples

```
>>> import pandas as pd
>>> import numpy as np
>>> from feature_
↪engine.discretisation import ArbitraryDiscretiser
>>> np.random.seed(42)
>>> X␣
↪= pd.DataFrame(dict(x = np.random.randint(1,100, 100)))
>>> bins = dict(x =  [0, 25, 50, 75, 100])
>>> ad = ArbitraryDiscretiser(binning_dict = bins)
>>> ad.fit(X)
>>> ad.transform(X)["x"].value_counts()
2    31
0    27
3    25
1    17
Name: x, dtype: int64
```

Methods

fit:	This transformer does not learn parameters.
fit_transform:	Fit to data, then transform it.
get_feature_names_out:	Get output feature names for transformation.
get_params:	Get parameters for this estimator.
set_params:	Set the parameters of this estimator.
transform:	Sort continuous variable values into the intervals.

fit(*X*, *y=None*)

This transformer does not learn any parameter.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The training dataset. Can be the entire dataframe, not just the variables to be transformed.

y: None
y is not needed in this transformer. You can pass y or None.

fit_transform(*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to `X` and `y` with optional parameters `fit_params` and returns a transformed version of `X`.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_feature_names_out`(*input_features=None*)

Get output feature names for transformation. In other words, returns the variable names of transformed dataframe.

Parameters

input_features
[array or list, default=None] This parameter exits only for compatibility with the Scikit-learn pipeline.

- If `None`, then `feature_names_in_` is used as feature names in.
- If an array or list, then `input_features` must match `feature_names_in_`.

Returns

feature_names_out: list
Transformed feature names.

rtype
`List`[`Union`[`str`, `int`]] ..

`get_metadata_routing`()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Sort the variable values into the intervals.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The data to transform.

Returns

X_new: pandas dataframe of shape = [n_samples, n_features]
The transformed data with the discrete variables.

rtype
`DataFrame` ..

DecisionTreeDiscretiser

`class` feature_engine.discretisation.**DecisionTreeDiscretiser**(*variables=None*, *cv=3*, *scoring='neg_mean_sq*, *param_grid=None*, *reg*, *random_state=None*)

The DecisionTreeDiscretiser() replaces numerical variables by discrete, i.e., finite variables, which values are the predictions of a decision tree.

The method is inspired by the following article from the winners of the KDD 2009 competition: http://www.mtome.com/Publications/CiML/CiML-v3-book.pdf

The DecisionTreeDiscretiser() trains a decision tree per variable. Then, it transforms the variables, with predictions of the decision tree.

The DecisionTreeDiscretiser() works only with numerical variables. A list of variables to transform can be indicated. Alternatively, the discretiser will automatically select all numerical variables.

More details in the *User Guide*.

Parameters

variables: list, default=None
The list of numerical variables to transform. If None, the transformer will automatically find and select all numerical variables.

cv: int, cross-validation generator or an iterable, default=3
Determines the cross-validation splitting strategy. Possible inputs for cv are:

- None, to use cross_validate's default 5-fold cross validation
- int, to specify the number of folds in a (Stratified)KFold,
- **CV splitter**
  - (https://scikit-learn.org/stable/glossary.html#term-CV-splitter)
- An iterable yielding (train, test) splits as arrays of indices.

For int/None inputs, if the estimator is a classifier and y is either binary or multiclass, StratifiedKFold is used. In all other cases, KFold is used. These splitters are instantiated with `shuffle=False` so the splits will be the same across calls. For more details check Scikit-learn's `cross_validate`'s documentation.

scoring: str, default='neg_mean_squared_error'
Desired metric to optimise the performance of the tree. Comes from sklearn.metrics. See the DecisionTreeRegressor or DecisionTreeClassifier model evaluation documentation for more options: https://scikit-learn.org/stable/modules/model_evaluation.html

param_grid: dictionary, default=None
The hyperparameters for the decision tree to test with a grid search. The `param_grid` can contain any of the permitted hyperparameters for Scikit-learn's DecisionTreeRegressor() or DecisionTreeClassifier(). If None, then param_grid will optimise the 'max_depth' over `[1, 2, 3, 4]`.

regression: boolean, default=True
Indicates whether the discretiser should train a regression or a classification decision tree.

random_state
[int, default=None] The random_state to initialise the training of the decision tree. It is one of the parameters of the Scikit-learn's DecisionTreeRegressor() or DecisionTreeClassifier(). For reproducibility it is recommended to set the random_state to an integer.

Attributes

binner_dict_:
Dictionary containing the fitted tree per variable.

scores_dict_:
Dictionary with the score of the best decision tree per variable.

variables_:
The group of variables that will be transformed.

feature_names_in_:
List with the names of features seen during `fit`.

n_features_in_:
The number of features in the train set used in fit.

See also:

sklearn.tree.DecisionTreeClassifier
sklearn.tree.DecisionTreeRegressor

References

[1]

Examples

```
>>> import pandas as pd
>>> import numpy as np
>>> from feature_
↪engine.discretisation import DecisionTreeDiscretiser
>>> np.random.seed(42)
>>>␣
↪X = pd.DataFrame(dict(x= np.random.randint(1,100, 100)))
>>> y_reg = pd.Series(np.random.randn(100))
>>> dtd = DecisionTreeDiscretiser(random_state=42)
>>> dtd.fit(X, y_reg)
>>> dtd.transform(X)["x"].value_counts()
-0.090091    90
0.479454    10
Name: x, dtype: int64
```

You can also apply this for classification problems adjusting the scoring metric.

```
>>> y_clf = pd.Series(np.random.randint(0,2,100))
>>> dtd = DecisionTreeDiscretiser(regression=False,
↪ scoring="f1", random_state=42)
>>> dtd.fit(X, y_clf)
>>> dtd.transform(X)["x"].value_counts()
0.480769    52
0.687500    48
Name: x, dtype: int64
```

Methods

fit:	Fit a decision tree per variable.
fit_transform:	Fit to data, then transform it.
get_feature_names_out:	Get output feature names for transformation.
get_params:	Get parameters for this estimator.
set_params:	Set the parameters of this estimator.
transform:	Replace continuous variable values by the predictions of the deci

`fit`(*X*, *y*)

Fit one decision tree per variable to discretize with cross-validation and grid-search for hyperparameters.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The training dataset. Can be the entire dataframe, not just the variables to be transformed.

y: pandas series.
Target variable. Required to train the decision tree.

`fit_transform`(*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to `X` and `y` with optional parameters `fit_params` and returns a transformed version of `X`.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_feature_names_out`(*input_features=None*)

Get output feature names for transformation. In other words, returns the variable names of transformed dataframe.

Parameters

input_features
[array or list, default=None] This parameter exits only for compatibility with the Scikit-learn pipeline.

- If `None`, then `feature_names_in_` is used as feature names in.
- If an array or list, then `input_features` must match `feature_names_in_`.

Returns

feature_names_out: list
Transformed feature names.

rtype
`List`[`Union`[`str`, `int`]] ..

`get_metadata_routing`()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Replaces original variable values with the predictions of the tree. The decision tree predictions are finite, aka, discrete.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The input samples.

Returns

X_new: pandas dataframe of shape = [n_samples, n_features]
The dataframe with transformed variables.

rtype
`DataFrame` ..

GeometricWidthDiscretiser

`class` feature_engine.discretisation.`GeometricWidthDiscretiser`(*variables=None*, *bin*
return_object=False
return_boundaries=
precision=7)

The `GeometricWidthDiscretiser()` divides continuous numerical variables into intervals of increasing width. The width of each succeeding interval is larger than the previous interval by a constant amount (cw).

The constant amount is calculated as:

$$cw = (Max - Min)^{1/n}$$

were Max and Min are the variable's maximum and minimum value, and n is the number of intervals.

The sizes of the intervals themselves are calculated with a geometric progression:

$$a_{i+1} = a_i cw$$

Thus, the first interval's width equals cw, the second interval's width equals 2 * cw, and so on.

Note that the proportion of observations per interval may vary.

This discretisation technique is great when the distribution of the variable is right skewed.

Note: The width of some bins might be very small. Thus, to allow this transformer to work properly, it might help to increase the precision value, that is, the number of decimal values allowed to define each bin. If the variable has a narrow range or you are sorting into several bins, allow greater precision (i.e., if precision = 3, then 0.001; if precision = 7, then 0.0001).

The *GeometricWidthDiscretiser()* works only with numerical variables. A list of variables to discretise can be indicated, or the discretiser will automatically select all numerical variables in the train set.

More details in the *User Guide*.

Parameters

variables: list, default=None
The list of numerical variables to transform. If None, the transformer will automatically find and select all numerical variables.

bins: int, default=10
Desired number of intervals / bins.

return_object: bool, default=False
Whether the the discrete variable should be returned as type numeric or type object. If you would like to encode the discrete variables with Feature-engine's categorical encoders, use True. Alternatively, keep the default to False.

return_boundaries: bool, default=False
Whether the output should be the interval boundaries. If True, it returns the interval boundaries. If False, it returns integers.

precision: int, default=3
The precision at which to store and display the bins labels.

Attributes

binner_dict_:
Dictionary with the interval limits per variable.

variables_:
The group of variables that will be transformed.

feature_names_in_:
List with the names of features seen during `fit`.

n_features_in_:
The number of features in the train set used in fit.

References

[1], [2], [3]

Methods

fit:	Find the interval limits.
fit_transform:	Fit to data, then transform it.
get_feature_names_out:	Get output feature names for transformation.
get_params:	Get parameters for this estimator.
set_params:	Set the parameters of this estimator.
transform:	Sort continuous variable values into the intervals.

`fit`(*X*, *y=None*)

Learn the boundaries of the geometric width intervals / bins for each variable.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The training dataset. Can be the entire dataframe, not just the variables to be transformed.

y: None
y is not needed in this encoder. You can pass y or None.

`fit_transform`(*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to `X` and `y` with optional parameters `fit_params` and returns a transformed version of `X`.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_feature_names_out`(*input_features=None*)

Get output feature names for transformation. In other words, returns the variable names of transformed dataframe.

Parameters

input_features
[array or list, default=None] This parameter exits only for compatibility with the Scikit-learn pipeline.

- If `None`, then `feature_names_in_` is used as feature names in.
- If an array or list, then `input_features` must match `feature_names_in_`.

Returns

feature_names_out: list
Transformed feature names.

rtype
`List`[`Union`[`str`, `int`]] ..

`get_metadata_routing`()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Sort the variable values into the intervals.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The data to transform.

Returns

X_new: pandas dataframe of shape = [n_samples, n_features]
The transformed data with the discrete variables.

rtype
DataFrame ..

Additional transformers for discretisation

For discretisation using K-means, check Scikit-learn's KBinsDiscretizer.

Outlier Handling

Feature-engine's outlier transformers cap maximum or minimum values of a variable at an arbitrary or derived value. The OutlierTrimmer removes outliers from the dataset.

Transformer	Description
Winsorizer()	Caps variables at automatically determined extreme values
ArbitraryOutlierCapper()	Caps variables at values determined by the user
OutlierTrimmer()	Removes outliers from the dataframe

Winsorizer

class feature_engine.outliers.**Winsorizer**(*capping_method='gaussian'*, *tail='right'*, *fold=3*, *add_indicators=False*, *variables=None*, *missing_values='raise'*)

The Winsorizer() caps maximum and/or minimum values of a variable at automatically determined values, and optionally adds indicators.

The extreme values beyond which an observation is considered an outlier are determined using:

- a Gaussian approximation
- the inter-quantile range proximity rule (IQR)
- MAD-median rule (MAD)
- percentiles

Gaussian limits:

- right tail: mean + 3* std
- left tail: mean - 3* std

IQR limits:

- right tail: 75th quantile + 3* IQR
- left tail: 25th quantile - 3* IQR

where IQR is the inter-quartile range: 75th quantile - 25th quantile.

MAD limits:

- right tail: median + 3* MAD
- left tail: median - 3* MAD

where MAD is the median absoulte deviation from the median.

percentiles:

- right tail: 95th percentile
- left tail: 5th percentile

You can select how far out to cap the maximum or minimum values with the parameter `'fold'`.

If `capping_method='gaussian'` fold gives the value to multiply the std.

If `capping_method='iqr'` fold is the value to multiply the IQR.

If `capping_method='mad'` fold is the value to multiply the MAD.

If `capping_method='quantiles'`, fold is the percentile on each tail that should be censored. For example, if fold=0.05, the limits will be the 5th and 95th percentiles. If fold=0.1, the limits will be the 10th and 90th percentiles.

The Winsorizer() works only with numerical variables. A list of variables can be indicated. Alternatively, the Winsorizer() will select and cap all numerical variables in the train set.

The transformer first finds the values at one or both tails of the distributions (fit). The transformer then caps the variables (transform).

More details in the *User Guide*.

Parameters

capping_method: str, default='gaussian'
Desired outlier detection method. Can be 'gaussian', 'iqr', 'mad', 'quantiles'.

The transformer will find the maximum and / or minimum values beyond which a data point will be considered an outlier using: **'gaussian'**: the Gaussian approximation. **'iqr'**: the IQR proximity rule. **'quantiles'**: the percentiles. **'mad'**: the Gaussian approximation but using robust statistics.

tail: str, default='right'
Whether to look for outliers on the right, left or both tails of the distribution. Can take 'left', 'right' or 'both'.

fold: int or float, default=0.05 if `quantile`, or 3 otherwise.
The factor used to multiply the std, MAD or IQR to calculate the maximum or minimum allowed values. Recommended values are 2 or 3 for the gaussian approximation, 1.5 or 3 for the IQR proximity rule and 3 or 3.5 for MAD rule.

If `capping_method='quantile'`, then `'fold'` indicates the percentile. So if `fold=0.05`, the limits will be the 95th and 5th percentiles.

Note: Outliers will be removed up to a maximum of the 20th percentiles on both sides. Thus, when `capping_method='quantile'`, then `'fold'` takes values between 0 and 0.20.

add_indicators: bool, default=False
Whether to add indicator variables to flag the capped outliers. If 'True', binary variables will be added to flag outliers on the left and right tails of the distribution. One binary variable per tail, per variable.

variables: list, default=None
The list of numerical variables to transform. If None, the transformer will automatically find and select all numerical variables.

missing_values: string, default='raise'
Indicates if missing values should be ignored or raised. If `'raise'` the transformer will return an error if the the datasets to `fit` or `transform` contain missing values. If `'ignore'`, missing data will be ignored when learning parameters or performing the transformation.

Attributes

right_tail_caps_:
Dictionary with the maximum values beyond which a value will be considered an outlier.

left_tail_caps_:
Dictionary with the minimum values beyond which a value will be considered an outlier.

variables_:
The group of variables that will be transformed.

feature_names_in_:
List with the names of features seen during `fit`.

n_features_in_:
The number of features in the train set used in fit.

Examples

```
>>> import numpy as np
>>> import pandas as pd
>>> from feature_engine.outliers import Winsorizer
>>> np.random.seed(42)
>>>
 ↪ X = pd.DataFrame(dict(x = np.random.normal(size = 10)))
>>> wz␣
 ↪= Winsorizer(capping_method='mad', tail='both', fold=3)
>>> wz.fit(X)
>>> wz.transform(X)
          x
0  0.496714
1 -0.138264
2  0.647689
3  1.523030
4 -0.234153
5 -0.234137
6  1.579213
7  0.767435
8 -0.469474
9  0.542560
```

```
>>> import numpy as np
>>> import pandas as pd
>>> from feature_engine.outliers import Winsorizer
>>> np.random.seed(42)
>>>
 ↪ X = pd.DataFrame(dict(x = np.random.normal(size = 10)))
>>> wz␣
 ↪= Winsorizer(capping_method='mad', tail='both', fold=3)
>>> wz.fit(X)
>>> wz.transform(X)
          x
0  0.496714
1 -0.138264
2  0.647689
3  1.523030
4 -0.234153
5 -0.234137
6  1.579213
7  0.767435
8 -0.469474
9  0.542560
```

Methods

fit:	Learn the values that will replace the outliers.
fit_transform:	Fit to data, then transform it.
get_feature_names_out:	Get output feature names for transformation.
get_params:	Get parameters for this estimator.
set_params:	Set the parameters of this estimator.
transform:	Cap the variables.

`fit`(*X*, *y=None*)

Learn the values that should be used to replace outliers.

Parameters

X
[pandas dataframe of shape = [n_samples, n_features]] The training input samples.

y
[pandas Series, default=None] y is not needed in this transformer. You can pass y or None.

`fit_transform`(*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to `X` and `y` with optional parameters `fit_params` and returns a transformed version of `X`.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_feature_names_out`(*input_features=None*)

Get output feature names for transformation. In other words, returns the variable names of transformed dataframe.

Parameters

input_features
[array or list, default=None] This parameter exits only for compatibility with the Scikit-learn pipeline.

- If `None`, then `feature_names_in_` is used as feature names in.
- If an array or list, then `input_features` must match `feature_names_in_`.

Returns

feature_names_out: list
Transformed feature names.

rtype
`List`[`Union`[`str`, `int`]] ..

`get_metadata_routing`()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Cap the variable values. Optionally, add outlier indicators.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The data to be transformed.

Returns

X_new: pandas dataframe of shape = [n_samples, n_features + n_ind]
The dataframe with the capped variables and indicators. The number of output variables depends on the values for 'tail' and 'add_indicators': if passing 'add_indicators=False', will be equal to 'n_features', otherwise, will have an additional indicator column per processed feature for each tail.

rtype
DataFrame ..

ArbitraryOutlierCapper

`class` feature_engine.outliers.**ArbitraryOutlierCapper**(*max_capping_dict=None*, *min_capping_dict=None*, *missing_values='raise'*)

The ArbitraryOutlierCapper() caps the maximum or minimum values of a variable at an arbitrary value indicated by the user.

You must provide the maximum or minimum values that will be used to cap each variable in a dictionary containing the features as keys and the capping values as values.

More details in the *User Guide*.

Parameters

max_capping_dict: dictionary, default=None
Dictionary containing the user specified capping values for the right tail of the distribution of each variable to cap (maximum values).

min_capping_dict: dictionary, default=None
Dictionary containing user specified capping values for the eft tail of the distribution of each variable to cap (minimum values).

missing_values: string, default='raise'
Indicates if missing values should be ignored or raised. If `'raise'` the transformer will return an error if the the datasets to `fit` or `transform` contain missing values. If `'ignore'`, missing data will be ignored when learning parameters or performing the transformation.

Attributes

right_tail_caps_:
Dictionary with the maximum values beyond which a value will be considered an outlier.

left_tail_caps_:
Dictionary with the minimum values beyond which a value will be considered an outlier.

variables_:
The group of variables that will be transformed.

feature_names_in_:
List with the names of features seen during `fit`.

n_features_in_:
The number of features in the train set used in fit.

Examples

```
>>> import pandas as pd
>>> from␣
↪feature_engine.outliers import ArbitraryOutlierCapper
>>> X = pd.DataFrame(dict(x1 = [1,2,3,4,5,6,7,8,9,10]))
>>> aoc␣
↪= ArbitraryOutlierCapper(max_capping_dict=dict(x1 =  8),
>>>    ␣
↪                          min_capping_dict=dict(x1 = 2))
>>> aoc.fit(X)
>>> aoc.transform(X)
   x1
0   2
1   2
2   3
3   4
4   5
5   6
6   7
7   8
8   8
9   8
```

Methods

fit:	This transformer does not learn parameters.
fit_transform:	Fit to data, then transform it.
get_feature_names_out:	Get output feature names for transformation.
get_params:	Get parameters for this estimator.
set_params:	Set the parameters of this estimator.
transform:	Cap the variables.

`fit`(*X*, *y=None*)

This transformer does not learn any parameter.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The training input samples.

y: pandas Series, default=None
y is not needed in this transformer. You can pass y or None.

`fit_transform`(*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to `X` and `y` with optional parameters `fit_params` and returns a transformed version of `X`.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_feature_names_out`(*input_features=None*)

Get output feature names for transformation. In other words, returns the variable names of transformed dataframe.

Parameters

input_features
[array or list, default=None] This parameter exits only for compatibility with the Scikit-learn pipeline.

- If `None`, then `feature_names_in_` is used as feature names in.
- If an array or list, then `input_features` must match `feature_names_in_`.

Returns

feature_names_out: list
Transformed feature names.

rtype
List[Union[str, int]] ..

`get_metadata_routing()`

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

`get_params(deep=True)`

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`set_params(**params)`

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform(X)`

Cap the variable values.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The data to be transformed.

Returns

X_new: pandas dataframe of shape = [n_samples, n_features]
The dataframe with the capped variables.

rtype
DataFrame ..

OutlierTrimmer

class feature_engine.outliers.**OutlierTrimmer**(*capping_method='gaussian'*, *tail='right'*, *f*
variables=None, *missing_values='raise'*)

The OutlierTrimmer() removes observations with outliers from the dataset.

The OutlierTrimmer() first calculates the maximum and /or minimum values beyond which a value will be considered an outlier, and thus removed.

The extreme values beyond which an observation is considered an outlier are determined using:

- a Gaussian approximation
- the inter-quantile range proximity rule (IQR)
- MAD-median rule (MAD)
- percentiles

Gaussian limits:

- right tail: mean + 3* std
- left tail: mean - 3* std

IQR limits:

- right tail: 75th quantile + 3* IQR
- left tail: 25th quantile - 3* IQR

where IQR is the inter-quartile range: 75th quantile - 25th quantile.

MAD limits:

- right tail: median + 3* MAD
- left tail: median - 3* MAD

where MAD is the median absoulte deviation from the median.

percentiles:

- right tail: 95th percentile
- left tail: 5th percentile

You can select how far out to cap the maximum or minimum values with the parameter `'fold'`.

If `capping_method='gaussian'` fold gives the value to multiply the std.

If `capping_method='iqr'` fold is the value to multiply the IQR.

If `capping_method='mad'` fold is the value to multiply the MAD.

If `capping_method='quantiles'`, fold is the percentile on each tail that should be censored. For example, if fold=0.05, the limits will be the 5th and 95th percentiles. If fold=0.1, the limits will be the 10th and 90th percentiles.

The OutlierTrimmer() works only with numerical variables. A list of variables can be indicated. Alternatively, it will select all numerical variables.

The transformer first finds the values at one or both tails of the distributions (fit). The transformer then removes observations with outliers from the dataframe (transform).

More details in the *User Guide*.

Parameters

capping_method: str, default='gaussian'
Desired outlier detection method. Can be 'gaussian', 'iqr', 'mad', 'quantiles'.

The transformer will find the maximum and / or minimum values beyond which a data point will be considered an outlier using: **'gaussian'**: the Gaussian approximation. **'iqr'**: the IQR proximity rule. **'quantiles'**: the percentiles. **'mad'**: the Gaussian approximation but using robust statistics.

tail: str, default='right'
Whether to look for outliers on the right, left or both tails of the distribution. Can take 'left', 'right' or 'both'.

fold: int or float, default=0.05 if `quantile`, or 3 otherwise.
The factor used to multiply the std, MAD or IQR to calculate the maximum or minimum allowed values. Recommended values are 2 or 3 for the gaussian approximation, 1.5 or 3 for the IQR proximity rule and 3 or 3.5 for MAD rule.

If `capping_method='quantile'`, then `'fold'` indicates the percentile. So if `fold=0.05`, the limits will be the 95th and 5th percentiles.

Note: Outliers will be removed up to a maximum of the 20th percentiles on both sides. Thus, when `capping_method='quantile'`, then `'fold'` takes values between 0 and 0.20.

variables: list, default=None
The list of numerical variables to transform. If None, the transformer will automatically find and select all numerical variables.

missing_values: string, default='raise'
Indicates if missing values should be ignored or raised. If `'raise'` the transformer will return an error if the the datasets to `fit` or `transform` contain missing values. If `'ignore'`, missing data will be ignored when learning parameters or performing the transformation.

Attributes

right_tail_caps_:
Dictionary with the maximum values beyond which a value will be considered an outlier.

left_tail_caps_:
Dictionary with the minimum values beyond which a value will be considered an outlier.

variables_:
The group of variables that will be transformed.

feature_names_in_:
List with the names of features seen during `fit`.

n_features_in_:
The number of features in the train set used in fit.

Examples

```
>>> import pandas as pd
>>> from feature_engine.outliers import OutlierTrimmer
>>> X = pd.DataFrame(dict(x = [0.49671,
>>>                            -0.1382,
>>>                             0.64768,
>>>                             1.52302,
>>>                            -0.2341,
>>>                            -17.2341,
>>>                             1.57921,
>>>                             0.76743,
>>>                            -0.4694,
>>>                             0.54256]))
>>> ot = OutlierTrimmer(capping_
↪method='gaussian', tail='left', fold=3)
>>> ot.fit(X)
>>> ot.transform(X)
          x
0   0.49671
1  -0.13820
2   0.64768
3   1.52302
4  -0.23410
5 -17.23410
6   1.57921
7   0.76743
8  -0.46940
9   0.54256
```

```
>>> import pandas as pd
>>> from feature_engine.outliers import OutlierTrimmer
>>> X = pd.DataFrame(dict(x = [0.49671,
>>>                            -0.1382,
>>>                             0.64768,
>>>                             1.52302,
>>>                            -0.2341,
>>>                            -17.2341,
>>>                             1.57921,
>>>                             0.76743,
>>>                            -0.4694,
>>>                             0.54256]))
>>> ot = OutlierTrimmer(capping_
↪method='mad', tail='left', fold=3)
```

(continues on next page)

(continued from previous page)

```
>>> ot.fit(X)
>>> ot.transform(X)
         x
0  0.49671
1 -0.13820
2  0.64768
3  1.52302
4 -0.23410
6  1.57921
7  0.76743
8 -0.46940
9  0.54256
```

Methods

fit:	Find maximum and minimum values.
fit_transform:	Fit to data, then transform it.
get_feature_names_out:	Get output feature names for transformation.
get_params:	Get parameters for this estimator.
set_params:	Set the parameters of this estimator.
transform:	Remove outliers.
transform_x_y:	Remove rows with outliers from X set and y.

`fit`(*X*, *y=None*)

Learn the values that should be used to replace outliers.

Parameters

X

[pandas dataframe of shape = [n_samples, n_features]] The training input samples.

y

[pandas Series, default=None] y is not needed in this transformer. You can pass y or None.

`fit_transform`(*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to `X` and `y` with optional parameters `fit_params` and returns a transformed version of `X`.

Parameters

X

[array-like of shape (n_samples, n_features)] Input samples.

y

[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_feature_names_out`(*input_features=None*)

Get output feature names for transformation. In other words, returns the variable names of transformed dataframe.

Parameters

input_features
[array or list, default=None] This parameter exits only for compatibility with the Scikit-learn pipeline.

- If `None`, then `feature_names_in_` is used as feature names in.
- If an array or list, then `input_features` must match `feature_names_in_`.

Returns

feature_names_out: list
Transformed feature names.

rtype
`List`[`Union`[`str`, `int`]] ..

`get_metadata_routing`()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Remove observations with outliers from the dataframe.

Parameters

X
[pandas dataframe of shape = [n_samples, n_features]] The data to be transformed.

Returns

X_new: pandas dataframe of shape = [n_samples, n_features]
The dataframe without outlier observations.

rtype
`DataFrame` ..

`transform_x_y`(*X*, *y*)

Transform, align and adjust both X and y based on the transformations applied to X, ensuring that they correspond to the same set of rows if any were removed from X.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The dataframe to transform.

y: pandas Series or Dataframe of length = n_samples
The target variable to transform. Can be multi-output.

Returns

X_new: pandas dataframe
The transformed dataframe of shape [n_samples - n_rows, n_features]. It may contain less rows than the original dataset.

y_new: pandas Series or DataFrame
The transformed target variable of length [n_samples - n_rows]. It contains as many rows as those left in X_new.

Variance Stabilizing Transformations

Feature-engine's variable transformers transform numerical variables with various mathematical transformations.

LogTransformer

class feature_engine.transformation.**LogTransformer**(*variables=None*, *base='e'*)

The LogTransformer() applies the natural logarithm or the base 10 logarithm to numerical variables. The natural logarithm is the logarithm in base e.

The LogTransformer() only works with positive values. If the variable contains a zero or a negative value the transformer will return an error.

A list of variables can be passed as an argument. Alternatively, the transformer will automatically select and transform all variables of type numeric.

More details in the *User Guide*.

Parameters

variables: list, default=None
The list of numerical variables to transform. If None, the transformer will automatically find and select all numerical variables.

base: string, default='e'
Indicates if the natural or base 10 logarithm should be applied. Can take values 'e' or '10'.

Attributes

variables_:
The group of variables that will be transformed.

feature_names_in_:
List with the names of features seen during `fit`.

n_features_in_:
The number of features in the train set used in fit.

Examples

```
>>> import numpy as np
>>> import pandas as pd
>>>␣
↪from feature_engine.transformation import LogTransformer
>>> np.random.seed(42)
>>> X =␣
↪pd.DataFrame(dict(x = np.random.lognormal(size = 100)))
>>> lt = LogTransformer()
>>> lt.fit(X)
>>> X = lt.transform(X)
>>> X.head()
          x
0  0.496714
1 -0.138264
2  0.647689
3  1.523030
4 -0.234153
```

Methods

fit:	This transformer does not learn parameters.
fit_transform:	Fit to data, then transform it.
get_feature_names_out:	Get output feature names for transformation.
get_params:	Get parameters for this estimator.
set_params:	Set the parameters of this estimator.
inverse_transform:	Convert the data back to the original representation.
transform:	Transform the variables using the logarithm.

fit(*X*, *y=None*)

This transformer does not learn parameters.

Selects the numerical variables and determines whether the logarithm can be applied on the selected variables, i.e., it checks that the variables are positive.

Parameters

X: Pandas DataFrame of shape = [n_samples, n_features].
The training input samples. Can be the entire dataframe, not just the variables to transform.

y: pandas Series, default=None
It is not needed in this transformer. You can pass y or None.

fit_transform(*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to `X` and `y` with optional parameters `fit_params` and returns a transformed version of `X`.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_feature_names_out`(*input_features=None*)

Get output feature names for transformation. In other words, returns the variable names of transformed dataframe.

Parameters

input_features
[array or list, default=None] This parameter exits only for compatibility with the Scikit-learn pipeline.

- If `None`, then `feature_names_in_` is used as feature names in.
- If an array or list, then `input_features` must match `feature_names_in_`.

Returns

feature_names_out: list
Transformed feature names.

rtype
`List`[`Union`[`str`, `int`]] ..

`get_metadata_routing`()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`inverse_transform`(*X*)

Convert the data back to the original representation.

Parameters

X: Pandas DataFrame of shape = [n_samples, n_features]
The data to be transformed.

Returns

X_tr: pandas dataframe
The dataframe with the transformed variables.

rtype
`DataFrame` ..

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Transform the variables with the logarithm.

Parameters

X: Pandas DataFrame of shape = [n_samples, n_features]
The data to be transformed.

Returns

X_new: pandas dataframe
The dataframe with the transformed variables.

rtype
DataFrame ..

LogCpTransformer

class feature_engine.transformation.**LogCpTransformer**(*variables=None*, *base='e'*, *C='a*

The LogCpTransformer() applies the transformation log(x + C), where C is a positive constant, to the input variable. It applies the natural logarithm or the base 10 logarithm, where the natural logarithm is logarithm in base e.

The logarithm can only be applied to numerical non-negative values. If the variable contains a zero or a negative value after adding a constant C, the transformer will return an error.

A list of variables can be passed as an argument. Alternatively, the transformer will automatically select and transform all variables of type numeric.

More details in the *User Guide*.

Parameters

variables: list, default=None
The list of numerical variables to transform. If None, the transformer will find and select all numerical variables. If C is a dictionary, then this parameter is ignored and the variables to transform are selected from the dictionary keys.

base: string, default='e'
Indicates if the natural or base 10 logarithm should be applied. Can take values 'e' or '10'.

C: "auto", int or dict, default="auto"
The constant C to add to the variable before the logarithm, i.e., log(x + C).

- If int, then log(x + C)
- If "auto", then C = abs(min(x)) + 1
- If dict, dictionary mapping the constant C to apply to each variable.

Note, when C is a dictionary, the parameter `variables` is ignored.

Attributes

variables_:
The group of variables that will be transformed.

C_:
The constant C to add to each variable. If C = "auto" a dictionary with C = abs(min(variable)) + 1.

feature_names_in_:
List with the names of features seen during `fit`.

n_features_in_:
The number of features in the train set used in fit.

Examples

```
>>> import numpy as np
>>> import pandas as pd
>>> from␣
↪feature_engine.transformation import LogCpTransformer
>>> np.random.seed(42)
>>> X =␣
↪pd.DataFrame(dict(x = np.random.lognormal(size = 100)))
>>> lct = LogCpTransformer()
>>> lct.fit(X)
>>> X = lct.transform(X)
>>> X.head()
          x
0  0.944097
1  0.586701
2  1.043204
3  1.707159
4  0.541405
```

Methods

fit:	Learn the constant C.
fit_transform:	Fit to data, then transform it.
get_feature_names_out:	Get output feature names for transformation.
get_params:	Get parameters for this estimator.
set_params:	Set the parameters of this estimator.
inverse_transform:	Convert the data back to the original representation.
transform:	Transform the variables with the logarithm of x plus C.

fit(*X*, *y=None*)

Learn the constant C to add to the variable before the logarithm transformation if C="auto".

Select the numerical variables or check that the variables entered by the user are numerical. Then check that the selected variables are positive after addition of C.

Parameters

X: Pandas DataFrame of shape = [n_samples, n_features].
The training input samples. Can be the entire dataframe, not just the variables to transform.

y: pandas Series, default=None
It is not needed in this transformer. You can pass y or None.

`fit_transform`(*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to `X` and `y` with optional parameters `fit_params` and returns a transformed version of `X`.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_feature_names_out`(*input_features=None*)

Get output feature names for transformation. In other words, returns the variable names of transformed dataframe.

Parameters

input_features
[array or list, default=None] This parameter exits only for compatibility with the Scikit-learn pipeline.

- If `None`, then `feature_names_in_` is used as feature names in.
- If an array or list, then `input_features` must match `feature_names_in_`.

Returns

feature_names_out: list
Transformed feature names.

rtype
`List`[`Union`[`str`, `int`]] ..

`get_metadata_routing`()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`inverse_transform`(*X*)

Convert the data back to the original representation.

Parameters

X: Pandas DataFrame of shape = [n_samples, n_features]
The data to be transformed.

Returns

X_tr: Pandas dataframe
The dataframe with the transformed variables.

rtype
`DataFrame` ..

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Transform the variables with the logarithm of x plus a constant C.

Parameters

X: Pandas DataFrame of shape = [n_samples, n_features]
The data to be transformed.

Returns

X_new: pandas dataframe
The dataframe with the transformed variables.

rtype
DataFrame ..

ReciprocalTransformer

class feature_engine.transformation.**ReciprocalTransformer**(*variables=None*)

The ReciprocalTransformer() applies the reciprocal transformation 1 / x to numerical variables.

The ReciprocalTransformer() only works with numerical variables with non-zero values. If a variable contains the value 0, the transformer will raise an error.

A list of variables can be passed as an argument. Alternatively, the transformer will automatically select and transform all numerical variables.

More details in the *User Guide*.

Parameters

variables: list, default=None
The list of numerical variables to transform. If None, the transformer will automatically find and select all numerical variables.

Attributes

variables_:
The group of variables that will be transformed.

feature_names_in_:
List with the names of features seen during `fit`.

n_features_in_:
The number of features in the train set used in fit.

Examples

```
>>> import numpy as np
>>> import pandas as pd
>>> from feature_
↪engine.transformation import ReciprocalTransformer
>>> np.random.seed(42)
>>> X = pd.DataFrame(dict(x␣
↪= 10 - np.random.exponential(size = 100)))
>>> rt = ReciprocalTransformer()
>>> rt.fit(X)
>>> X = rt.transform(X)
>>> X.head()
          x
0  0.104924
1  0.143064
2  0.115164
3  0.110047
4  0.101726
```

Methods

fit:	This transformer does not learn parameters.
fit_transform:	Fit to data, then transform it.
get_feature_names_out:	Get output feature names for transformation.
get_params:	Get parameters for this estimator.
set_params:	Set the parameters of this estimator.
inverse_transform:	Convert the data back to the original representation.
transform:	Apply the reciprocal 1 / x transformation.

fit(*X*, *y=None*)

This transformer does not learn parameters.

Parameters

X: Pandas DataFrame of shape = [n_samples, n_features].
The training input samples. Can be the entire dataframe, not just the variables to transform.

y: pandas Series, default=None
It is not needed in this transformer. You can pass y or None.

fit_transform(*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to `X` and `y` with optional parameters `fit_params` and returns a transformed version of `X`.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_feature_names_out`(*input_features=None*)

Get output feature names for transformation. In other words, returns the variable names of transformed dataframe.

Parameters

input_features
[array or list, default=None] This parameter exits only for compatibility with the Scikit-learn pipeline.

- If `None`, then `feature_names_in_` is used as feature names in.
- If an array or list, then `input_features` must match `feature_names_in_`.

Returns

feature_names_out: list
Transformed feature names.

rtype
`List`[`Union`[`str`, `int`]] ..

`get_metadata_routing`()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`inverse_transform`(*X*)

Convert the data back to the original representation.

Parameters

X: Pandas DataFrame of shape = [n_samples, n_features]
The data to be transformed.

Returns

X_tr: pandas dataframe
The dataframe with the transformed variables.

rtype
`DataFrame` ..

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Apply the reciprocal 1 / x transformation.

Parameters

X: Pandas DataFrame of shape = [n_samples, n_features]
The data to be transformed.

Returns

X_new: pandas dataframe
The dataframe with the transformed variables.

rtype
DataFrame ..

ArcsinTransformer

class feature_engine.transformation.**ArcsinTransformer**(*variables=None*)

The ArcsinTransformer() applies the arcsin transformation to numerical variables.

The arcsin transformation, also called arcsin square root transformation, or angular transformation, takes the form of arcsin(sqrt(x)) where x is a real number between 0 and 1.

The arcsin square root transformation helps in dealing with probabilities, percents, and proportions. It aims to stabilize the variance of the variable and return more evenly distributed (Gaussian looking) values.

The ArcsinTransformer() only works with numerical variables which values are between 0 and 1. If a variable contains values outside of this range, the transformer will raise an error.

A list of variables can be passed as an argument. Alternatively, the transformer will automatically select and transform all numerical variables.

More details in the *User Guide*.

Parameters

variables: list, default=None
The list of numerical variables to transform. If None, the transformer will automatically find and select all numerical variables.

Attributes

variables_:
The group of variables that will be transformed.

feature_names_in_:
List with the names of features seen during `fit`.

n_features_in_:
The number of features in the train set used in fit.

Examples

```
>>> import numpy as np
>>> import pandas as pd
>>> from␣
↪feature_engine.transformation import ArcsinTransformer
>>> np.random.seed(42)
>>> X =␣
↪pd.DataFrame(dict(x = np.random.beta(1, 1, size = 100)))
>>> ast = ArcsinTransformer()
>>> ast.fit(X)
>>> X = ast.transform(X)
>>> X.head()
          x
0  0.785437
1  0.253389
2  0.144664
3  0.783236
4  0.650777
```

Methods

fit:	This transformer does not learn parameters.
fit_transform:	Fit to data, then transform it.
get_feature_names_out:	Get output feature names for transformation.
get_params:	Get parameters for this estimator.
set_params:	Set the parameters of this estimator.
inverse_transform:	Convert the data back to the original representation.
transform:	Apply the arcsin transformation.

`fit`(*X*, *y=None*)

This transformer does not learn parameters.

Parameters

X: Pandas DataFrame of shape = [n_samples, n_features].
The training input samples. Can be the entire dataframe, not just the variables to transform.

y: pandas Series, default=None
It is not needed in this transformer. You can pass y or None.

`fit_transform`(*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to `X` and `y` with optional parameters `fit_params` and returns a transformed version of `X`.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_feature_names_out`(*input_features=None*)

Get output feature names for transformation. In other words, returns the variable names of transformed dataframe.

Parameters

input_features
[array or list, default=None] This parameter exits only for compatibility with the Scikit-learn pipeline.

- If `None`, then `feature_names_in_` is used as feature names in.
- If an array or list, then `input_features` must match `feature_names_in_`.

Returns

feature_names_out: list
Transformed feature names.

rtype
`List`[`Union`[`str`, `int`]] ..

`get_metadata_routing`()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`inverse_transform`(*X*)

Convert the data back to the original representation.

Parameters

X: Pandas DataFrame of shape = [n_samples, n_features]
The data to be transformed.

Returns

X_tr: pandas dataframe
The dataframe with the transformed variables.

rtype
`DataFrame` ..

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Apply the arcsin transformation.

Parameters

X: Pandas DataFrame of shape = [n_samples, n_features]
The data to be transformed.

Returns

X_new: pandas dataframe
The dataframe with the transformed variables.

rtype
DataFrame ..

PowerTransformer

class feature_engine.transformation.**PowerTransformer**(*variables=None*, *exp=0.5*)

The PowerTransformer() applies power or exponential transformations to numerical variables.

The PowerTransformer() works only with numerical variables.

A list of variables can be passed as an argument. Alternatively, the transformer will automatically select and transform all numerical variables.

More details in the *User Guide*.

Parameters

variables: list, default=None
The list of numerical variables to transform. If None, the transformer will automatically find and select all numerical variables.

exp: float or int, default=0.5
The power (or exponent).

Attributes

variables_:
The group of variables that will be transformed.

feature_names_in_:
List with the names of features seen during `fit`.

n_features_in_:
The number of features in the train set used in fit.

Examples

```
>>> import numpy as np
>>> import pandas as pd
>>> from␣
↪feature_engine.transformation import PowerTransformer
>>> np.random.seed(42)
>>> X =␣
↪pd.DataFrame(dict(x = np.random.lognormal(size = 100)))
>>> pt = PowerTransformer()
>>> pt.fit(X)
```

(continues on next page)

(continued from previous page)

```
>>> X = pt.transform(X)
>>> X.head()
          x
0  1.281918
1  0.933203
2  1.382432
3  2.141518
4  0.889517
```

Methods

fit:	This transformer does not learn parameters.
fit_transform:	Fit to data, then transform it.
get_feature_names_out:	Get output feature names for transformation.
get_params:	Get parameters for this estimator.
set_params:	Set the parameters of this estimator.
inverse_transform:	Convert the data back to the original representation.
transform:	Apply the power transformation to the variables.

`fit`(*X*, *y=None*)

This transformer does not learn parameters.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The training input samples. Can be the entire dataframe, not just the variables to transform.

y: pandas Series, default=None
It is not needed in this transformer. You can pass y or None.

`fit_transform`(*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to `X` and `y` with optional parameters `fit_params` and returns a transformed version of `X`.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_feature_names_out`(*input_features=None*)

Get output feature names for transformation. In other words, returns the variable names of transformed dataframe.

Parameters

input_features
[array or list, default=None] This parameter exits only for compatibility with the Scikit-learn pipeline.

- If `None`, then `feature_names_in_` is used as feature names in.
- If an array or list, then `input_features` must match `feature_names_in_`.

Returns

feature_names_out: list
Transformed feature names.

rtype
`List`[`Union`[`str`, `int`]] ..

`get_metadata_routing`()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`inverse_transform`(*X*)

Convert the data back to the original representation.

Parameters

X: Pandas DataFrame of shape = [n_samples, n_features]
The data to be transformed.

Returns

X_tr: pandas Dataframe
The dataframe with the power transformed variables.

rtype
DataFrame ..

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as Pipeline). The latter have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Apply the power transformation to the variables.

Parameters

X: Pandas DataFrame of shape = [n_samples, n_features]
The data to be transformed.

Returns

X_new: pandas Dataframe
The dataframe with the power transformed variables.

rtype
DataFrame ..

BoxCoxTransformer

class feature_engine.transformation.**BoxCoxTransformer**(*variables=None*)

The BoxCoxTransformer() applies the BoxCox transformation to numerical variables.

The Box-Cox transformation is defined as:

- T(Y)=(Y exp()1)/ if !=0
- log(Y) otherwise

where Y is the response variable and is the transformation parameter. varies, typically from -5 to 5. In the transformation, all values of are considered and the optimal value for a given variable is selected.

The BoxCox transformation implemented by this transformer is that of SciPy.stats: https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.boxcox.html

The BoxCoxTransformer() works only with numerical positive variables (>=0).

A list of variables can be passed as an argument. Alternatively, the transformer will automatically select and transform all numerical variables.

More details in the *User Guide*.

Parameters

variables: list, default=None
The list of numerical variables to transform. If None, the transformer will automatically find and select all numerical variables.

Attributes

lambda_dict_:
Dictionary with the best BoxCox exponent per variable.

variables_:
The group of variables that will be transformed.

feature_names_in_:
List with the names of features seen during `fit`.

n_features_in_:
The number of features in the train set used in fit.

References

[1]

Examples

```
>>> import numpy as np
>>> import pandas as pd
>>> from␣
↪feature_engine.transformation import BoxCoxTransformer
>>> np.random.seed(42)
>>> X =␣
↪pd.DataFrame(dict(x = np.random.lognormal(size = 100)))
>>> bct = BoxCoxTransformer()
>>> bct.fit(X)
>>> X = bct.transform(X)
>>> X.head()
          x
0  0.505485
1 -0.137595
2  0.662654
3  1.607518
4 -0.232237
```

Methods

fit:	Learn the optimal lambda for the BoxCox transformation.
fit_transform:	Fit to data, then transform it.
get_feature_names_out:	Get output feature names for transformation.
get_params:	Get parameters for this estimator.
set_params:	Set the parameters of this estimator.
inverse_transform:	Convert the data back to the original representation.
transform:	Apply the BoxCox transformation.

fit(*X*, *y=None*)

Learn the optimal lambda for the BoxCox transformation.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The training input samples. Can be the entire dataframe, not just the variables to transform.

y: pandas Series, default=None
It is not needed in this transformer. You can pass y or None.

fit_transform(*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to `X` and `y` with optional parameters `fit_params` and returns a transformed version of `X`.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_feature_names_out`(*input_features=None*)

Get output feature names for transformation. In other words, returns the variable names of transformed dataframe.

Parameters

input_features
[array or list, default=None] This parameter exits only for compatibility with the Scikit-learn pipeline.

- If `None`, then `feature_names_in_` is used as feature names in.
- If an array or list, then `input_features` must match `feature_names_in_`.

Returns

feature_names_out: list
Transformed feature names.

rtype
`List`[`Union`[`str`, `int`]] ..

`get_metadata_routing`()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`inverse_transform`(*X*)

Convert the data back to the original representation.

Parameters

X: Pandas DataFrame of shape = [n_samples, n_features]
The data to be inverse transformed.

Returns

X_new: pandas dataframe
The dataframe with the original variables.

rtype
`DataFrame` ..

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Apply the BoxCox transformation.

Parameters

X: Pandas DataFrame of shape = [n_samples, n_features]
The data to be transformed.

Returns

X_new: pandas dataframe
The dataframe with the transformed variables.

rtype
DataFrame ..

YeoJohnsonTransformer

class feature_engine.transformation.**YeoJohnsonTransformer**(*variables=None*)

The YeoJohnsonTransformer() applies the Yeo-Johnson transformation to the numerical variables.

The Yeo-Johnson transformation implemented by this transformer is that of SciPy.stats: https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.yeojohnson.html

The YeoJohnsonTransformer() works only with numerical variables.

A list of variables can be passed as an argument. Alternatively, the transformer will automatically select and transform all numerical variables.

More details in the *User Guide*.

Parameters

variables: list, default=None
The list of numerical variables to transform. If None, the transformer will automatically find and select all numerical variables.

Attributes

lambda_dict_
Dictionary containing the best lambda for the Yeo-Johnson per variable.

variables_:
The group of variables that will be transformed.

feature_names_in_:
List with the names of features seen during `fit`.

n_features_in_:
The number of features in the train set used in fit.

References

[1], [2]

Examples

```
>>> import numpy as np
>>> import pandas as pd
>>> from feature_
↪engine.transformation import YeoJohnsonTransformer
>>> np.random.seed(42)
>>> X = pd.DataFrame(dict(x␣
↪= np.random.lognormal(size = 100) - 10))
>>> yjt = YeoJohnsonTransformer()
>>> yjt.fit(X)
>>> X = yjt.transform(X)
>>> X.head()
                x
0 -267042.906453
1 -444357.138990
2 -221626.115742
3  -23647.632651
4 -467264.993249
```

Methods

fit:	Learn the optimal lambda for the Yeo-Johnson transformation.
fit_transform:	Fit to data, then transform it.
get_feature_names_out:	Get output feature names for transformation.
get_params:	Get parameters for this estimator.
set_params:	Set the parameters of this estimator.
inverse_transform:	Convert the data back to the original representation.
transform:	Apply the Yeo-Johnson transformation.

fit(*X*, *y=None*)

Learn the optimal lambda for the Yeo-Johnson transformation.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The training input samples. Can be the entire dataframe, not just the variables to transform.

y: pandas Series, default=None
It is not needed in this transformer. You can pass y or None.

fit_transform(*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to `X` and `y` with optional parameters `fit_params` and returns a transformed version of `X`.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_feature_names_out`(*input_features=None*)

Get output feature names for transformation. In other words, returns the variable names of transformed dataframe.

Parameters

input_features
[array or list, default=None] This parameter exits only for compatibility with the Scikit-learn pipeline.

- If `None`, then `feature_names_in_` is used as feature names in.
- If an array or list, then `input_features` must match `feature_names_in_`.

Returns

feature_names_out: list
Transformed feature names.

rtype
`List`[`Union`[`str`, `int`]] ..

`get_metadata_routing`()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`inverse_transform`(*X*)

Convert the data back to the original representation.

Parameters

X: Pandas DataFrame of shape = [n_samples, n_features]
The data to be transformed.

Returns

X_tr: pandas dataframe
The dataframe with the transformed variables.

rtype
`DataFrame` ..

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Apply the Yeo-Johnson transformation.

Parameters

X: Pandas DataFrame of shape = [n_samples, n_features]
The data to be transformed.

Returns

X: pandas dataframe
The dataframe with the transformed variables.

rtype
DataFrame ..

Transformers in other Libraries

These and additional transformations can be obtained with the following Scikit-learn classes:

- FunctionTransformer
- PowerTransformer

Note that Scikit-klearn classes return Numpy arrays and are applied to the entire dataset.

10.3.2 Creation

Feature Creation

Feature-engine's creation transformers create and add new features to the dataframe by either combining or transforming existing features.

MathFeatures

class feature_engine.creation.**MathFeatures**(*variables*, *func*, *new_variables_names=None*, *missing_values='raise'*, *drop_original=False*)

MathFeatures(() applies functions across multiple features returning one or more additional features as a result. It uses `pandas.agg()` to create the features, setting `axis=1`.

For supported aggregation functions, see pandas documentation.

Note that if some of the variables have missing data and `missing_values='ignore'`, the value will be ignored in the computation. To be clear, if variables A, B and C, have values 10, 20 and NA, and we perform the sum, the result will be A + B = 30.

More details in the *User Guide*.

Parameters

variables: list
The list of input variables. Variables must be numerical and there must be at least 2 different variables in the list.

func: function, string, list
Functions to use for aggregating the data. Same functionality as parameter `func` in `pandas.agg()`. If a function, it must either work when passed a DataFrame or when passed to DataFrame.apply. Accepted combinations are:

- function
- string function name
- list of functions and/or function names, e.g. [np.sum, 'mean']

Each function will result in a new variable that will be added to the transformed dataset.

new_variables_names: list, default=None
Names of the new variables. If passing a list with names (recommended), enter one name per function. If None, the transformer will assign arbitrary names, starting with the function and followed by the variables separated by _.

missing_values: string, default='raise'
Indicates if missing values should be ignored or raised. If `'raise'` the transformer will return an error if the the datasets to `fit` or `transform` contain missing values. If `'ignore'`, missing data will be ignored when learning parameters or performing the transformation.

drop_original: bool, default=False
If True, the original variables to transform will be dropped from the dataframe.

Attributes

variables_:
The group of variables that will be transformed.

feature_names_in_:
List with the names of features seen during `fit`.

n_features_in_:
The number of features in the train set used in fit.

Notes

Although the transformer allows us to combine any features with any functions, we recommend using it to create features based on domain knowledge. Typical examples in finance are:

- Sum debt across financial products, i.e., credit cards, to obtain the total debt.
- Take the average payments to various financial products.
- Find the minimum payment done at any one month.

In insurance, we can sum the damage to various parts of a car to obtain the total damage.

Examples

```
>>> import pandas as pd
>>> from feature_engine.creation import MathFeatures
>>> X = pd.DataFrame(dict(x1 = [1,2,3], x2 = [4,5,6]))
>>>␣
↪mf = MathFeatures(variables = ["x1","x2"], func = "sum")
>>> mf.fit(X)
>>> mf.transform(X)
   x1  x2  sum_x1_x2
0   1   4          5
1   2   5          7
2   3   6          9
```

```
>>> mf␣
↪= MathFeatures(variables = ["x1","x2"], func = "prod")
>>> mf.fit(X)
>>> mf.transform(X)
   x1  x2  prod_x1_x2
0   1   4           4
1   2   5          10
2   3   6          18
```

```
>>> mf␣
↪= MathFeatures(variables = ["x1","x2"], func = "mean")
>>> mf.fit(X)
>>> mf.transform(X))
   x1  x2  mean_x1_x2
0   1   4         2.5
1   2   5         3.5
2   3   6         4.5
```

Methods

fit:	This transformer does not learn parameters.
fit_transform:	Fit to data, then transform it.
get_feature_names_out:	Get output feature names for transformation.
get_params:	Get parameters for this estimator.
set_params:	Set the parameters of this estimator.
transform:	Create new features.

fit(*X*, *y=None*)

This transformer does not learn parameters.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The training input samples.

y: pandas Series, or np.array. Defaults to None.
It is not needed in this transformer. You can pass y or None.

`fit_transform`(*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to `X` and `y` with optional parameters `fit_params` and returns a transformed version of `X`.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_feature_names_out`(*input_features=None*)

Get output feature names for transformation. In other words, returns the variable names of transformed dataframe.

Parameters

input_features
[array or list, default=None] This parameter exits only for compatibility with the Scikit-learn pipeline.

- If `None`, then `feature_names_in_` is used as feature names in.
- If an array or list, then `input_features` must match `feature_names_in_`.

Returns

feature_names_out: list
Transformed feature names.

rtype
`List`[`Union`[`str`, `int`]] ..

`get_metadata_routing`()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A MetadataRequest encapsulating routing information.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as Pipeline). The latter have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Create and add new variables.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The data to transform.

Returns

X_new: Pandas dataframe, shape = [n_samples, n_features + n_operations]
The input dataframe plus the new variables.

rtype
DataFrame ..

RelativeFeatures

class feature_engine.creation.**RelativeFeatures**(*variables*, *reference*, *func*, *fill_value=No*
missing_values='ignore', *drop_original=*

RelativeFeatures() applies basic mathematical operations between a group of variables and one or more reference features. It adds the resulting features to the dataframe.

In other words, RelativeFeatures() adds, subtracts, multiplies, performs the division, true division, floor division, module or exponentiation of a group of features to / by a group of reference variables. The features resulting from these functions are added to the dataframe.

This transformer works only with numerical variables. It uses the pandas methods `pd.DataFrme.add`, `pd.DataFrme.sub`, `pd.DataFrme.mul`, `pd.DataFrme.div`, `pd.DataFrme.truediv`, `pd.DataFrme.floordiv`, `pd.DataFrme.mod` and `pd.DataFrme.pow`. Find out more in pandas documentation.

More details in the *User Guide*.

Parameters

variables: list
The list of numerical variables to combine with the reference variables.

reference: list
The list of reference variables that will be added, subtracted, multiplied, used as denominator for division and module, or exponent for the exponentiation.

func: list
The list of functions to be used in the transformation. The list can contain one or more of the following strings: 'add', 'mul','sub', 'div', truediv, 'floordiv', 'mod', 'pow'.

fill_value: int, float, default=None
When dividing by zero, this value is used in place of infinity. If None, then an error will be raised when dividing by zero.

missing_values: string, default='raise'
Indicates if missing values should be ignored or raised. If `'raise'` the transformer will return an error if the the datasets to `fit` or `transform` contain missing values. If `'ignore'`, missing data will be ignored when learning parameters or performing the transformation.

drop_original: bool, default=False
If True, the original variables to transform will be dropped from the dataframe.

Attributes

variables_:
The group of variables that will be transformed.

feature_names_in_:
List with the names of features seen during `fit`.

n_features_in_:
The number of features in the train set used in fit.

Notes

Although the transformer allows us to combine any feature with any function, we recommend its use to create domain knowledge variables. Typical examples within the financial sector are:

- Ratio between income and debt to create the debt_to_income_ratio.
- Subtraction of rent from income to obtain the disposable_income.

Examples

```
>>> import pandas as pd
>>> from feature_engine.creation import RelativeFeatures
>>> X = pd.DataFrame(dict(x1␣
↪= [1,2,3], x2 = [4,5,6], x3 = [3,4,5]))
>>> rf = RelativeFeatures(variables = ["x1","x2"],
>>>                       reference = ["x3"],
>>>                       func = ["div"])
>>> rf.fit(X)
>>> rf.transform(X)
   x1  x2  x3  x1_div_x3  x2_div_x3
0   1   4   3   0.333333   1.333333
1   2   5   4   0.500000   1.250000
2   3   6   5   0.600000   1.200000
```

Methods

fit:	This transformer does not learn parameters.
fit_transform:	Fit to data, then transform it.
get_feature_names_out:	Get output feature names for transformation.
get_params:	Get parameters for this estimator.
set_params:	Set the parameters of this estimator.
transform:	Create new features.

fit(*X*, *y=None*)

This transformer does not learn parameters.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The training input samples.

y: pandas Series, or np.array. Defaults to None.
It is not needed in this transformer. You can pass y or None.

fit_transform(*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to `X` and `y` with optional parameters `fit_params` and returns a transformed version of `X`.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_feature_names_out`(*input_features=None*)

Get output feature names for transformation. In other words, returns the variable names of transformed dataframe.

Parameters

input_features
[array or list, default=None] This parameter exits only for compatibility with the Scikit-learn pipeline.

- If `None`, then `feature_names_in_` is used as feature names in.
- If an array or list, then `input_features` must match `feature_names_in_`.

Returns

feature_names_out: list
Transformed feature names.

rtype
`List`[`Union`[`str`, `int`]] ..

`get_metadata_routing`()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Add new features.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The data to transform.

Returns

X_new: Pandas dataframe
The input dataframe plus the new variables.

rtype
`DataFrame` ..

CyclicalFeatures

class feature_engine.creation.**CyclicalFeatures**(*variables=None*, *max_values=None*, *drop_original=False*)

CyclicalFeatures() applies cyclical transformations to numerical variables, returning 2 new features per variable, according to:

- var_sin = sin(variable * (2. * pi / max_value))
- var_cos = cos(variable * (2. * pi / max_value))

where max_value is the maximum value in the variable, and pi is 3.14…

CyclicalFeatures() works only with numerical variables. A list of variables to transform can be passed as an argument. Alternatively, the transformer will automatically select and transform all numerical variables.

Missing data should be imputed before using this transformer.

More details in the *User Guide*.

Parameters

variables: list, default=None
The list of numerical variables to transform. If None, the transformer will automatically find and select all numerical variables.

max_values: dict, default=None
A dictionary with the maximum value of each variable to transform. Useful when the maximum value is not present in the dataset. If None, the transformer will automatically find the maximum value of each variable.

drop_original: bool, default=False
If True, the original variables to transform will be dropped from the dataframe.

Attributes

max_values_:
The feature's maximum values.

variables_:
The group of variables that will be transformed.

feature_names_in_:
List with the names of features seen during `fit`.

n_features_in_:
The number of features in the train set used in fit.

References

Debaditya Chakraborty & Hazem Elzarka (2019), Advanced machine learning techniques for building performance simulation: a comparative analysis, Journal of Building Performance Simulation, 12:2, 193-207

Examples

```
>>> import pandas as pd
>>> from feature_engine.creation import CyclicalFeatures
>>> X = pd.DataFrame(dict(x= [1,4,3,3,4,2,1,2]))
>>> cf = CyclicalFeatures()
>>> cf.fit(X)
>>> cf.transform(X)
   x         x_sin         x_cos
0  1  1.000000e+00  6.123234e-17
1  4 -2.449294e-16  1.000000e+00
2  3 -1.000000e+00 -1.836970e-16
3  3 -1.000000e+00 -1.836970e-16
4  4 -2.449294e-16  1.000000e+00
5  2  1.224647e-16 -1.000000e+00
6  1  1.000000e+00  6.123234e-17
7  2  1.224647e-16 -1.000000e+00
```

Methods

fit:	Learns the variable's maximum values.
fit_transform:	Fit to data, then transform it.
get_feature_names_out:	Get output feature names for transformation.
get_params:	Get parameters for this estimator.
set_params:	Set the parameters of this estimator.
transform:	Create new features.

fit(*X*, *y=None*)

Learns the maximum value of each variable.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The training input samples. Can be the entire dataframe, not just the variables to transform.

y: pandas Series, default=None
It is not needed in this transformer. You can pass y or None.

fit_transform(*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to `X` and `y` with optional parameters `fit_params` and returns a transformed version of `X`.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_feature_names_out`(*input_features=None*)

Get output feature names for transformation. In other words, returns the variable names of transformed dataframe.

Parameters

input_features
[array or list, default=None] This parameter exits only for compatibility with the Scikit-learn pipeline.

- If `None`, then `feature_names_in_` is used as feature names in.
- If an array or list, then `input_features` must match `feature_names_in_`.

Returns

feature_names_out: list
Transformed feature names.

rtype
`List`[`Union`[`str`, `int`]] ..

`get_metadata_routing`()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Creates new features using the cyclical transformations.

Parameters

X: Pandas DataFrame of shame = [n_samples, n_features]
The data to be transformed.

Returns

X_new: Pandas dataframe.
The original dataframe plus the additional features.

Transformers in other Libraries

Check also the following transformer from Scikit-learn:

- PolynomialFeatures
- SplineTransformer

Datetime Features

Feature-engine's datetime transformers are able to extract a wide variety of datetime features from existing datetime or object-like data.

DatetimeFeatures

class feature_engine.datetime.**DatetimeFeatures**(*variables=None*, *features_to_extract=N*
drop_original=True, *missing_values='r*
dayfirst=False, *yearfirst=False*, *utc=No*
format=None)

DatetimeFeatures extracts date and time features from datetime variables, adding new columns to the dataset. DatetimeFeatures can extract datetime information from existing datetime or object-like variables or from the dataframe index.

DatetimeFeatures uses `pandas.to_datetime` to convert object variables to datetime and pandas.dt to extract the features from datetime.

The transformer supports the extraction of the following features:

- "month"
- "quarter"
- "semester"
- "year"
- "week"
- "day_of_week"
- "day_of_month"
- "day_of_year"
- "weekend"
- "month_start"
- "month_end"
- "quarter_start"
- "quarter_end"
- "year_start"
- "year_end"
- "leap_year"
- "days_in_month"
- "hour"
- "minute"
- "second"

More details in the *User Guide*.

Parameters

variables: str, list, default=None
List with the variables from which date and time information will be extracted. If None, the transformer will find and select all datetime variables, including variables of type object that can be converted to datetime. If "index", the transformer will extract datetime features from the index of the dataframe.

features_to_extract: list, default=None
The list of date features to extract. If None, the following features will be extracted: "month", "year", "day_of_week", "day_of_month", "hour", "minute" and "second". If "all", all supported features will be extracted. Alternatively, you can pass a list with the names of the features you want to extract.

drop_original: bool, default="True"
If True, the original datetime variables will be dropped from the dataframe.

missing_values: string, default='raise'
Indicates if missing values should be ignored or raised. If 'raise' the transformer will return an error if the the datasets to `fit` or `transform` contain missing values. If 'ignore', missing data will be ignored when performing the feature extraction. Missing data is only evaluated in the variables that will be used to derive the date and time features. If features are derived from the dataframe index, missing data will be checked in the index.

dayfirst: bool, default="False"
Specify a date parse order if arg is str or is list-like. If True, parses dates with the day first, e.g. 10/11/12 is parsed as 2012-11-10. Same as in `pandas.to_datetime`.

yearfirst: bool, default="False"
Specify a date parse order if arg is str or is list-like. Same as in `pandas.to_datetime`.

- If True parses dates with the year first, e.g. 10/11/12 is parsed as 2010-11-12.
- If both dayfirst and yearfirst are True, yearfirst is preceded.

utc: bool, default=None
Return UTC DatetimeIndex if True (converting any tz-aware datetime.datetime objects as well). Same as in `pandas.to_datetime`.

format: str, default None
The strftime to parse time, e.g. "%d/%m/%Y". Check pandas `to_datetime()` for more information on choices. If you have variables with different formats pass "mixed", to infer the format for each element individually. This is risky, and you should probably use it along with dayfirst, according to pandas' documentation.

Attributes

variables_:
List of variables from which date and time features will be extracted. If None, features will be extracted from the dataframe index.

features_to_extract_:
The date and time features that will be extracted from each variable or the

index.

feature_names_in_:
List with the names of features seen during `fit`.

n_features_in_:
The number of features in the train set used in fit.

See also:

`pandas.to_datetime`
`pandas.dt`

Examples

```
>>> import pandas as pd
>>> from feature_engine.datetime import DatetimeFeatures
>>> X = pd.DataFrame(dict(date␣
↪= ["2022-09-18", "2022-10-27", "2022-12-24"]))
>>> dtf = DatetimeFeatures(features_
↪to_extract = ["year", "month", "day_of_month"])
>>> dtf.fit(X)
>>> dtf.transform(X)
    date_year  date_month  date_day_of_month
0        2022           9                 18
1        2022          10                 27
2        2022          12                 24
```

Methods

fit:	This transformer does not learn parameters.
fit_transform:	Fit to data, then transform it.
get_feature_names_out:	Get output feature names for transformation.
get_params:	Get parameters for this estimator.
set_params:	Set the parameters of this estimator.
transform:	Add the date and time features.

`fit`(*X*, *y=None*)

This transformer does not learn any parameter.

Finds datetime variables or checks that the variables selected by the user can be converted to datetime.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The training input samples. Can be the entire dataframe, not just the variables to transform.

y: pandas Series, default=None
It is not needed in this transformer. You can pass y or None.

`fit_transform`(*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to `X` and `y` with optional parameters `fit_params` and returns a transformed version of `X`.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_feature_names_out`(*input_features=None*)

Get output feature names for transformation. In other words, returns the variable names of transformed dataframe.

Parameters

input_features
[array or list, default=None] This parameter exits only for compatibility with the Scikit-learn pipeline.

- If `None`, then `feature_names_in_` is used as feature names in.
- If an array or list, then `input_features` must match `feature_names_in_`.

Returns

feature_names_out: list
Transformed feature names.

rtype
`List`[`Union`[`str`, `int`]] ..

`get_metadata_routing`()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Extract the date and time features and add them to the dataframe.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The data to transform.

Returns

X_new: Pandas dataframe, shape = [n_samples, n_features x n_df_features]
The dataframe with the original variables plus the new variables.

rtype
`DataFrame` ..

DatetimeSubtraction

class feature_engine.datetime.**DatetimeSubtraction**(*variables=None*, *reference=None*, *new_variables_names=None*, *outpu*
missing_values='ignore', *drop_orig*
dayfirst=False, *yearfirst=False*, *utc=*
format=None)

DatetimeSubtraction() applies datetime subtraction between a group of datetime variables and one or more datetime features, adding the resulting variables to the dataframe.

DatetimeSubtraction() works with variables cast as datetime or object. It subtracts the variables listed in the parameter `reference` from those listed in the parameter `variables`.

More details in the *User Guide*.

Parameters

variables: list
The list of datetime variables that the reference variables will be subtracted from (left side of the subtraction operation).

reference: list
The list of datetime reference variables that will be subtracted from `variables` (right side of the subtraction operation).

new_variables_names: list, default=None
Names of the new variables. You have the option to pass a list with the names you'd like to assing to the new variables. If `None`, the transformer will assign arbitrary names.

output_unit: string, default='D'
The string representation of the output unit of the datetime differences. The default is `D` for day. This parameter is passed to `numpy.timedelta64`. Other possible values are `Y` for year, `M` for month, `W` for week, `h` for hour, `m` for minute, `s` for second, `ms` for millisecond, `us` or `s` for microsecond, `ns` for nanosecond, `ps` for picosecond, `fs` for femtosecond and `as` for attosecond.

missing_values: string, default='raise'
Indicates if missing values should be ignored or raised. If `'raise'` the transformer will return an error if the the datasets to `fit` or `transform` contain missing values. If `'ignore'`, missing data will be ignored when learning parameters or performing the transformation.

drop_original: bool, default="False"
If `True`, the variables listed in `variables` and `reference` will be dropped from the dataframe after the computation of the new features.

dayfirst: bool, default="False"
Specify a date parse order if arg is str or is list-like. If True, parses dates with the day first, e.g. 10/11/12 is parsed as 2012-11-10. Same as in `pandas.to_datetime`.

yearfirst: bool, default="False"
Specify a date parse order if arg is str or is list-like. Same as in `pandas.to_datetime`.

- If True parses dates with the year first, e.g. 10/11/12 is parsed as 2010-11-12.
- If both dayfirst and yearfirst are True, yearfirst is preceded.

utc: bool, default=None
Return UTC DatetimeIndex if True (converting any tz-aware datetime.datetime objects as well). Same as in `pandas.to_datetime`.

format: str, default None
The strftime to parse time, e.g. "%d/%m/%Y". Check pandas `to_datetime()` for more information on choices. If you have variables with different formats pass "mixed", to infer the format for each element individually. This is risky, and you should probably use it along with dayfirst, according to pandas' documentation.

Attributes

variables_:
The list with datetime variables from which the variables in `reference` will be substracted. It is created after the transformer corroborates that the variables in `variables` are, or can be parsed to datetime.

reference_:
The list with the datetime variables that will be subtracted from `variables_`. It is created after the transformer corroborates that the variables in `reference` are, or can be parsed to datetime.

feature_names_in_:
List with the names of features seen during `fit`.

n_features_in_:
The number of features in the train set used in fit.

Examples

```
>>> import pandas as pd
>>>
↪ from feature_engine.datetime import DatetimeSubtraction
>>> X = pd.DataFrame({
>>>␣
↪    "date1": ["2022-09-18", "2022-10-27", "2022-12-24"],
>>> ␣
↪   "date2": ["2022-08-18", "2022-08-27", "2022-06-24"]})
>>> dtf = DatetimeSubtraction(variables=[
↪"date1"], reference=["date2"])
>>> dtf.fit(X)
>>> dtf.transform(X)
        date1       date2  date1_sub_date2
0  2022-09-18  2022-08-18             31.0
1  2022-10-27  2022-08-27             61.0
2  2022-12-24  2022-06-24            183.0
```

Methods

fit:	This transformer does not learn parameters.
fit_transform:	Fit to data, then transform it.
get_feature_names_out:	Get output feature names for transformation.
get_params:	Get parameters for this estimator.
set_params:	Set the parameters of this estimator.
transform:	Create new features.

`fit`(*X*, *y=None*)

This transformer does not learn any parameter.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The training input samples. Can be the entire dataframe, not just the variables to transform.

y: pandas Series, or np.array. Default=None.
It is not needed in this transformer. You can pass y or None.

`fit_transform`(*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to `X` and `y` with optional parameters `fit_params` and returns a transformed version of `X`.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_feature_names_out`(*input_features=None*)

Get output feature names for transformation. In other words, returns the variable names of transformed dataframe.

Parameters

input_features
[array or list, default=None] This parameter exits only for compatibility with the Scikit-learn pipeline.

- If `None`, then `feature_names_in_` is used as feature names in.
- If an array or list, then `input_features` must match `feature_names_in_`.

Returns

feature_names_out: list
Transformed feature names.

rtype
`List`[`Union`[`str`, `int`]] ..

`get_metadata_routing`()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Add new features.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The data to transform.

Returns

X_new: Pandas dataframe
The input dataframe plus the new variables.

rtype
DataFrame ..

10.3.3 Selection

Feature Selection

Feature-engine's feature selection transformers are used to drop subsets of variables with low predictive value. Feature-engine hosts selection algorithms that are, in general, not available in other libraries. These algorithms have been gathered from data science competitions or used in the industry.

Feature-engine's transformers select features based on different strategies. Some algorithms remove constant or quasi-constant features. Some algorithms remove duplicated or correlated variables. Some algorithms select features based on a machine learning model performance. Some transformers implement selection procedures used in finance. And some transformers support functionality that has been developed in the industry or in data science competitions.

In the following tables you find the algorithms that belong to each category.

Selection based on feature characteristics

Transformer	Categorical variables	Allows NA	Description
DropFeatures()			Drops arbitrary features determined by user
DropConstantFeatures()			Drops constant and quasi-constant features
DropDuplicateFeatures()			Drops features that are duplicated
DropCorrelatedFeatures()			Drops features that are correlated
SmartCorrelatedSelection()			From a correlated feature group drops the less useful features

Selection based on a machine learning model

Transformer	Categorical variables	Allows NA	Description
SelectBySingleFeaturePerformance()	×	×	Selects features based on single feature model performance
RecursiveFeatureElimination()	×	×	Removes features recursively by evaluating model performance
RecursiveFeatureAddition()	×	×	Adds features recursively by evaluating model performance

Selection methods commonly used in finance

Transformer	Categorical variables	Allows NA	Description
DropHighPSIFeatures()	×		Drops features with high Population Stability Index
SelectByInformationValue()		x	Drops features with low information value

Alternative feature selection methods

Transformer	Categorical variables	Allows NA	Description
SelectByShuffling()	×	×	Selects features if shuffling their values causes a drop in model performance
SelectByTargetMeanPerformance()		×	Using the target mean as performance proxy, selects high performing features
ProbeFeatureSelection()	×	×	Selects features who importance is greater than those of random variables

DropFeatures

class feature_engine.selection.**DropFeatures**(*features_to_drop*)

DropFeatures() drops a list of variables indicated by the user from the dataframe.

More details in the *User Guide*.

Parameters

features_to_drop: str or list
Variable(s) to be dropped from the dataframe

Attributes

features_to_drop_:
The features that will be dropped.

feature_names_in_:
List with the names of features seen during `fit`.

n_features_in_:
The number of features in the train set used in fit.

Examples

```
>>> import pandas as pd
>>> from feature_engine.selection import DropFeatures
>>> X = pd.DataFrame(dict(x1 = [1,2,3,4],
>>>                       x2 = ["a", "a", "b", "c"],
>>> ␣
↪                      x3 = [True, False, False, True]))
>>> df = DropFeatures(features_to_drop=["x2"])
>>> df.fit_transform(X)
   x1     x3
0   1   True
1   2  False
2   3  False
3   4   True
```

Methods

fit:	This transformer does not learn any parameter.
fit_transform:	Fit to data, then transform it.
get_feature_names_out:	Get output feature names for transformation.
get_support:	Get a mask, or integer index, of the features selected.
get_params:	Get parameters for this estimator.
set_params:	Set the parameters of this estimator.
transform:	Drops indicated features.

fit(*X*, *y=None*)

This transformer does not learn any parameter.

Parameters

X
[pandas dataframe of shape = [n_samples, n_features]] The input dataframe

y
[pandas Series, default = None] y is not needed for this transformer. You can pass y or None.

fit_transform(*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to `X` and `y` with optional parameters `fit_params` and returns a transformed version of `X`.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_feature_names_out`(*input_features=None*)

Get output feature names for transformation. In other words, returns the variable names of transformed dataframe.

Parameters

input_features
[array or list, default=None] This parameter exits only for compatibility with the Scikit-learn pipeline.

- If `None`, then `feature_names_in_` is used as feature names in.
- If an array or list, then `input_features` must match `feature_names_in_`.

Returns

feature_names_out: list
Transformed feature names.

rtype
`List`[`Union`[`str`, `int`]] ..

`get_metadata_routing`()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`get_support`(*indices=False*)

Get a mask, or integer index, of the features selected.

Parameters

indices
[bool, default=False] If True, the return value will be an array of integers, rather than a boolean mask.

Returns

support
[array] An index that selects the retained features from a feature vector. If `indices` is False, this is a boolean array of shape [# input features], in which an element is True if its corresponding feature is selected for retention. If `indices` is True, this is an integer array of shape [# output features] whose values are indices into the input feature vector.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Return dataframe with selected features.

Parameters

X: pandas dataframe of shape = [n_samples, n_features].
The input dataframe.

Returns

X_new: pandas dataframe of shape = [n_samples, n_selected_features]
Pandas dataframe with the selected features.

rtype
DataFrame ..

DropConstantFeatures

class feature_engine.selection.**DropConstantFeatures**(*variables=None*, *tol=1*, *missing_*
confirm_variables=False)

DropConstantFeatures() drops constant and quasi-constant variables from a dataframe. Constant variables show the same value in all the observations in the dataset. Quasi-constant variables show the same value in almost all the observations in the dataset.

This transformer works with numerical and categorical variables. The user can indicate a list of variables to examine. Alternatively, the transformer will evaluate all the variables in the dataset.

The transformer will first identify and store the constant and quasi-constant variables. Next, the transformer will drop these variables from a dataframe.

More details in the *User Guide*.

Parameters

variables: list, default=None
The list of variables to evaluate. If None, the transformer will evaluate all variables in the dataset.

tol: float,int, default=1
Threshold to detect constant/quasi-constant features. Variables showing the same value in a percentage of observations greater than tol will be considered constant / quasi-constant and dropped. If tol=1, the transformer removes constant variables. Else, it will remove quasi-constant variables. For example, if tol=0.98, the transformer will remove variables that show the same value in 98% of the observations.

missing_values: str, default=raises
Whether the missing values should be raised as error, ignored or included as an additional value of the variable. Takes values 'raise', 'ignore', 'include'.

confirm_variables: bool, default=False
If set to True, variables that are not present in the input dataframe will be removed from the list of variables. Only used when passing a variable list to the parameter `variables`. See parameter variables for more details.

Attributes

features_to_drop_:
List with constant and quasi-constant features.

variables_:
The variables that will be considered for the feature selection procedure.:

feature_names_in_:
List with the names of features seen during `fit`.

n_features_in_:
The number of features in the train set used in fit.

See also:

`sklearn.feature_selection.VarianceThreshold`

Notes

This transformer is a similar concept to the VarianceThreshold from Scikit-learn, but it evaluates number of unique values instead of variance.

Examples

```
>>> import pandas as pd
>>> from␣
↪feature_engine.selection import DropConstantFeatures
>>> X = pd.DataFrame(dict(x1 = [1,1,1,1],
>>>                       x2 = ["a", "a", "b", "c"],
>>>                       x3 = [True, False, False, True]))
>>> dcf = DropConstantFeatures()
>>> dcf.fit_transform(X)
    x2     x3
0  a   True
1  a  False
2  b  False
3  c   True
```

Additionally, you can set the Threshold for quasi-constant features:

```
>>> X = pd.DataFrame(dict(x1 = [1,1,1,1],
>>>                       x2 = ["a", "a", "b", "c"],
>>>
↪                      x3 = [True, False, False, False]))
>>> dcf = DropConstantFeatures(tol = 0.75)
>>> dcf.fit_transform(X)
    x2
0  a
1  a
2  b
3  c
```

Methods

fit:	Find constant and quasi-constant features.
fit_transform:	Fit to data, then transform it.
get_feature_names_out:	Get output feature names for transformation.
get_params:	Get parameters for this estimator.
set_params:	Set the parameters of this estimator.
get_support:	Get a mask, or integer index, of the features selected.
transform:	Remove constant and quasi-constant features.

`fit`(*X*, *y=None*)

Find constant and quasi-constant features.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The input dataframe.

y: None
y is not needed for this transformer. You can pass y or None.

`fit_transform`(*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to `X` and `y` with optional parameters `fit_params` and returns a transformed version of `X`.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_feature_names_out`(*input_features=None*)

Get output feature names for transformation. In other words, returns the variable names of transformed dataframe.

Parameters

input_features
[array or list, default=None] This parameter exits only for compatibility with the Scikit-learn pipeline.

- If `None`, then `feature_names_in_` is used as feature names in.
- If an array or list, then `input_features` must match `feature_names_in_`.

Returns

feature_names_out: list
Transformed feature names.

rtype
`List`[`Union`[`str`, `int`]] ..

`get_metadata_routing`()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`get_support`(*indices=False*)

Get a mask, or integer index, of the features selected.

Parameters

indices
[bool, default=False] If True, the return value will be an array of integers, rather than a boolean mask.

Returns

support
[array] An index that selects the retained features from a feature vector. If `indices` is False, this is a boolean array of shape [# input features], in which an element is True if its corresponding feature is selected for retention. If `indices` is True, this is an integer array of shape [# output features] whose values are indices into the input feature vector.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Return dataframe with selected features.

Parameters

X: pandas dataframe of shape = [n_samples, n_features].
The input dataframe.

Returns

X_new: pandas dataframe of shape = [n_samples, n_selected_features]
Pandas dataframe with the selected features.

rtype
`DataFrame` ..

DropDuplicateFeatures

`class feature_engine.selection.DropDuplicateFeatures`(*variables=None*, *missing_values*
confirm_variables=False)

DropDuplicateFeatures() finds and removes duplicated features in a dataframe.

Duplicated features are identical features, regardless of the variable or column name. If they show the same values for every observation, then they are considered duplicated.

This transformer works with numerical and categorical variables. The user can indicate a list of variables to examine. Alternatively, the transformer will evaluate all the variables in the dataset.

The transformer will first identify and store the duplicated variables. Next, the transformer will drop these variables from a dataframe.

More details in the *User Guide*.

Parameters

variables: list, default=None
The list of variables to evaluate. If None, the transformer will evaluate all variables in the dataset.

missing_values: str, default=ignore
Whether the missing values should be raised as error or ignored when determining correlation. Takes values 'raise' and 'ignore'.

confirm_variables: bool, default=False
If set to True, variables that are not present in the input dataframe will be removed from the list of variables. Only used when passing a variable list to the parameter `variables`. See parameter variables for more details.

Attributes

features_to_drop_:
Set with the duplicated features that will be dropped.

duplicated_feature_sets_:
Groups of duplicated features. Each list is a group of duplicated features.

variables_:
The variables that will be considered for the feature selection procedure.

feature_names_in_:
List with the names of features seen during `fit`.

n_features_in_:
The number of features in the train set used in fit.

Examples

```
>>> import pandas as pd
>>> from␣
↪feature_engine.selection import DropDuplicateFeatures
>>> X = pd.DataFrame(dict(x1 = [1,1,1,1],
>>>                       x2 = [1,1,1,1],
>>>                       x3 = [True, False, False, False]))
>>> ddf = DropDuplicateFeatures()
>>> ddf.fit_transform(X)
   x1     x3
0   1   True
1   1  False
2   1  False
3   1  False
```

Methods

fit:	Find duplicated features.
fit_transform:	Fit to data, then transform it.
get_feature_names_out:	Get output feature names for transformation.
get_params:	Get parameters for this estimator.
set_params:	Set the parameters of this estimator.
get_support:	Get a mask, or integer index, of the features selected.
transform:	Remove duplicated features.

`fit`(*X*, *y=None*)

Find duplicated features.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The input dataframe.

y: None
y is not needed for this transformer. You can pass y or None.

`fit_transform`(*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to `X` and `y` with optional parameters `fit_params` and returns a transformed version of `X`.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_feature_names_out`(*input_features=None*)

Get output feature names for transformation. In other words, returns the variable names of transformed dataframe.

Parameters

input_features
[array or list, default=None] This parameter exits only for compatibility with the Scikit-learn pipeline.

- If `None`, then `feature_names_in_` is used as feature names in.
- If an array or list, then `input_features` must match `feature_names_in_`.

Returns

feature_names_out: list
Transformed feature names.

rtype
`List`[`Union`[`str`, `int`]] ..

`get_metadata_routing`()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`get_support`(*indices=False*)

Get a mask, or integer index, of the features selected.

Parameters

indices
[bool, default=False] If True, the return value will be an array of integers, rather than a boolean mask.

Returns

support
[array] An index that selects the retained features from a feature vector. If `indices` is False, this is a boolean array of shape [# input features], in which an element is True if its corresponding feature is selected for retention. If `indices` is True, this is an integer array of shape [# output features] whose values are indices into the input feature vector.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Return dataframe with selected features.

Parameters

X: pandas dataframe of shape = [n_samples, n_features].
The input dataframe.

Returns

X_new: pandas dataframe of shape = [n_samples, n_selected_features]
Pandas dataframe with the selected features.

rtype
`DataFrame` ..

DropCorrelatedFeatures

`class feature_engine.selection.`**`DropCorrelatedFeatures`**(*variables=None*, *method='pea*
threshold=0.8, *missing_values*
confirm_variables=False)

DropCorrelatedFeatures() finds and removes correlated features. Correlation is calculated with `pandas.corr()`. Features are removed on first found, first removed basis, without any further insight.

DropCorrelatedFeatures() works only with numerical variables. Categorical variables will need to be encoded to numerical or will be excluded from the analysis.

To make the selector deterministic, features are sorted alphabetically before examining correlation.

More details in the *User Guide*.

Parameters

variables: str or list, default=None
The list of variables to evaluate. If None, the transformer will evaluate all numerical features in the dataset.

method: string or callable, default='pearson'
Can take 'pearson', 'spearman', 'kendall' or callable. It refers to the correlation method to be used to identify the correlated features.

- 'pearson': standard correlation coefficient
- 'kendall': Kendall Tau correlation coefficient
- 'spearman': Spearman rank correlation
- callable: callable with input two 1d ndarrays and returning a float.

For more details on this parameter visit the `pandas.corr()` documentation.

threshold: float, default=0.8
The correlation threshold above which a feature will be deemed correlated with another one and removed from the dataset.

missing_values: str, default=ignore
Whether the missing values should be raised as error or ignored when determining correlation. Takes values 'raise' and 'ignore'.

confirm_variables: bool, default=False
If set to True, variables that are not present in the input dataframe will be removed from the list of variables. Only used when passing a variable list to the parameter `variables`. See parameter variables for more details.

Attributes

features_to_drop_:
Set with the correlated features that will be dropped.

correlated_feature_sets_:
Groups of correlated features. Each list is a group of correlated features.

correlated_feature_dict_: dict
Dictionary containing the correlated feature groups. The key is the feature against which all other features were evaluated. The values are the features correlated with the key. Key + values should be the same as the set found in `correlated_feature_groups`. We introduced this attribute in version 1.17.0 because from the set, it is not easy to see which feature will be retained and which ones will be removed. The key is retained, the values will be dropped.

variables_:
The variables that will be considered for the feature selection procedure.

feature_names_in_:
List with the names of features seen during `fit`.

n_features_in_:
The number of features in the train set used in fit.

See also:

`pandas.corr`
`feature_engine.selection.SmartCorrelationSelection`

Notes

If you want to select from each group of correlated features those that are perhaps more predictive or more complete, check Feature-engine's SmartCorrelationSelection.

Examples

```
>>> import pandas as pd
>>> from␣
↪feature_engine.selection import DropCorrelatedFeatures
>>> X = pd.DataFrame(dict(x1␣
↪= [1,2,1,1], x2 = [2,4,3,1], x3 = [1, 0, 0, 1]))
>>> dcf = DropCorrelatedFeatures(threshold=0.7)
>>> dcf.fit_transform(X)
    x1  x3
0   1   1
1   2   0
2   1   0
3   1   1
```

Methods

fit:	Find correlated features.
fit_transform:	Fit to data, then transform it.
get_feature_names_out:	Get output feature names for transformation.
get_params:	Get parameters for this estimator.
set_params:	Set the parameters of this estimator.
get_support:	Get a mask, or integer index, of the features selected.
transform:	Remove correlated features.

`fit`(*X*, *y=None*)

Find the correlated features.

Parameters

X
[pandas dataframe of shape = [n_samples, n_features]] The training dataset.

y
[pandas series. Default = None] y is not needed in this transformer. You can pass y or None.

`fit_transform`(*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to `X` and `y` with optional parameters `fit_params` and returns a transformed version of `X`.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_feature_names_out`(*input_features=None*)

Get output feature names for transformation. In other words, returns the variable names of transformed dataframe.

Parameters

input_features
[array or list, default=None] This parameter exits only for compatibility with the Scikit-learn pipeline.

- If `None`, then `feature_names_in_` is used as feature names in.
- If an array or list, then `input_features` must match `feature_names_in_`.

Returns

feature_names_out: list
Transformed feature names.

rtype
`List`[`Union`[`str`, `int`]] ..

`get_metadata_routing`()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`get_support`(*indices=False*)

Get a mask, or integer index, of the features selected.

Parameters

indices
[bool, default=False] If True, the return value will be an array of integers, rather than a boolean mask.

Returns

support
[array] An index that selects the retained features from a feature vector. If `indices` is False, this is a boolean array of shape [# input features], in which an element is True if its corresponding feature is selected for retention. If `indices` is True, this is an integer array of shape [# output features] whose values are indices into the input feature vector.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Return dataframe with selected features.

Parameters

X: pandas dataframe of shape = [n_samples, n_features].
The input dataframe.

Returns

X_new: pandas dataframe of shape = [n_samples, n_selected_features]
Pandas dataframe with the selected features.

rtype
DataFrame ..

SmartCorrelatedSelection

`class` feature_engine.selection.`SmartCorrelatedSelection`(*variables=None*, *method='p*
threshold=0.8, *missing_valu*
*selection_method='missing_*
estimator=None, *scoring='r*
confirm_variables=False)

SmartCorrelatedSelection() finds groups of correlated features and then selects, from each group, a feature following certain criteria:

- Feature with the least missing values.
- Feature with the highest cardinality (greatest number of unique values).
- Feature with the highest variance.
- Feature with the highest importance according to an estimator.

SmartCorrelatedSelection() returns a dataframe containing from each group of correlated features, the selected variable, plus all the features that were not correlated to any other.

Correlation is calculated with `pandas.corr()`.

SmartCorrelatedSelection() works only with numerical variables. Categorical variables will need to be encoded to numerical or will be excluded from the analysis.

More details in the *User Guide*.

Parameters

variables: str or list, default=None
The list of variables to evaluate. If None, the transformer will evaluate all numerical features in the dataset.

method: string or callable, default='pearson'
Can take 'pearson', 'spearman', 'kendall' or callable. It refers to the correlation method to be used to identify the correlated features.

- 'pearson': standard correlation coefficient
- 'kendall': Kendall Tau correlation coefficient
- 'spearman': Spearman rank correlation
- callable: callable with input two 1d ndarrays and returning a float.

For more details on this parameter visit the `pandas.corr()` documentation.

threshold: float, default=0.8
The correlation threshold above which a feature will be deemed correlated with another one and removed from the dataset.

missing_values: str, default=ignore
Whether the missing values should be raised as error or ignored when determining correlation. Takes values 'raise' and 'ignore'.

selection_method: str, default= "missing_values"
Takes the values "missing_values", "cardinality", "variance" and "model_performance".

"missing_values": keeps the feature from the correlated group with the least missing observations.

"cardinality": keeps the feature from the correlated group with the highest cardinality.

"variance": keeps the feature from the correlated group with the highest variance.

"model_performance": trains a machine learning model using each of the features in a correlated group and retains the feature with the highest importance.

estimator: object
A Scikit-learn estimator for regression or classification.

scoring: str, default='roc_auc'
Metric to evaluate the performance of the estimator. Comes from sklearn.metrics. See the model evaluation documentation for more options: https://scikit-learn.org/stable/modules/model_evaluation.html

cv: int, cross-validation generator or an iterable, default=3
Determines the cross-validation splitting strategy. Possible inputs for cv are:

- None, to use cross_validate's default 5-fold cross validation
- int, to specify the number of folds in a (Stratified)KFold,
- CV splitter: (https://scikit-learn.org/stable/glossary.html#term-CV-splitter)
- An iterable yielding (train, test) splits as arrays of indices.

For int/None inputs, if the estimator is a classifier and y is either binary or multiclass, StratifiedKFold is used. In all other cases, KFold is used. These splitters are instantiated with `shuffle=False` so the splits will be the same across calls. For more details check Scikit-learn's `cross_validate`'s documentation.

confirm_variables: bool, default=False
If set to True, variables that are not present in the input dataframe will be removed from the list of variables. Only used when passing a variable list to the parameter `variables`. See parameter variables for more details.

Attributes

correlated_feature_sets_:
Groups of correlated features. Each list is a group of correlated features.

correlated_feature_dict_: dict
Dictionary containing the correlated feature groups. The key is the feature against which all other features were evaluated. The values are the features correlated with the key. Key + values should be the same as the set found in `correlated_feature_groups`. We introduced this attribute in version 1.17.0 because from the set, it is not easy to see which feature will be retained and which ones will be removed. The key is retained, the values will be dropped.

features_to_drop_:
The correlated features to remove from the dataset.

variables_:
The variables that will be considered for the feature selection procedure.

feature_names_in_:
List with the names of features seen during `fit`.

n_features_in_:
The number of features in the train set used in fit.

See also:

pandas.corr
feature_engine.selection.DropCorrelatedFeatures

Notes

For brute-force correlation selection, check Feature-engine's DropCorrelatedFeatures().

Examples

```
>>> import pandas as pd
>>> from␣
↪feature_engine.selection import SmartCorrelatedSelection
>>> X = pd.DataFrame(dict(x1 = [1,2,1,1],
>>>                   x2 = [2,4,3,1],
>>>                   x3 = [1, 0, 0, 0]))
>>> scs = SmartCorrelatedSelection(threshold=0.7)
>>> scs.fit_transform(X)
   x2  x3
0   2   1
1   4   0
```

(continues on next page)

(continued from previous page)

```
2   3   0
3   1   0
```

It is also possible to use alternative selection methods. Here, we select those features with the higher variance:

```
>>> X = pd.DataFrame(dict(x1 = [2,4,3,1],
>>>                   x2 = [1000,2000,1500,500],
>>>                   x3 = [1, 0, 0, 0]))
>>> scs = SmartCorrelatedSelection(threshold=0.
↪7, selection_method="variance")
>>> scs.fit_transform(X)
     x2  x3
0  1000   1
1  2000   0
2  1500   0
3   500   0
```

Methods

fit:	Find best feature from each correlated group.
fit_transform:	Fit to data, then transform it.
get_feature_names_out:	Get output feature names for transformation.
get_params:	Get parameters for this estimator.
set_params:	Set the parameters of this estimator.
get_support:	Get a mask, or integer index, of the features selected.
transform:	Return selected features.

fit(*X*, *y=None*)

Find the correlated feature groups. Determine which feature should be selected from each group.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The training dataset.

y: pandas series. Default = None
y is needed if selection_method == 'model_performance'.

fit_transform(*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to `X` and `y` with optional parameters `fit_params` and returns a transformed version of `X`.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_feature_names_out`(*input_features=None*)

Get output feature names for transformation. In other words, returns the variable names of transformed dataframe.

Parameters

input_features
[array or list, default=None] This parameter exits only for compatibility with the Scikit-learn pipeline.

- If `None`, then `feature_names_in_` is used as feature names in.
- If an array or list, then `input_features` must match `feature_names_in_`.

Returns

feature_names_out: list
Transformed feature names.

rtype
`List`[`Union`[`str`, `int`]] ..

`get_metadata_routing`()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`get_support`(*indices=False*)

Get a mask, or integer index, of the features selected.

Parameters

indices
[bool, default=False] If True, the return value will be an array of integers, rather than a boolean mask.

Returns

support
[array] An index that selects the retained features from a feature vector. If `indices` is False, this is a boolean array of shape [# input features], in which an element is True if its corresponding feature is selected for retention. If `indices` is True, this is an integer array of shape [# output features] whose values are indices into the input feature vector.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Return dataframe with selected features.

Parameters

X: pandas dataframe of shape = [n_samples, n_features].
The input dataframe.

Returns

X_new: pandas dataframe of shape = [n_samples, n_selected_features]
Pandas dataframe with the selected features.

rtype
DataFrame ..

SelectBySingleFeaturePerformance

class feature_engine.selection.**SelectBySingleFeaturePerformance**(*estimator*, *scoring*, *cv=3*, *threshold=*, *variables=None*, *confirm_variables*

SelectBySingleFeaturePerformance() selects features based on the performance of a machine learning model trained utilising a single feature. In other words, it trains a machine learning model for every single feature, then determines each model's performance. If the performance of the model is greater than a user specified threshold, then the feature is retained, otherwise removed.

The models are trained on each individual features using cross-validation. The performance metric to evaluate and the machine learning model to train are specified by the user.

More details in the *User Guide*.

Parameters

estimator: object
A Scikit-learn estimator for regression or classification.

variables: str or list, default=None
The list of variables to evaluate. If None, the transformer will evaluate all numerical features in the dataset.

scoring: str, default='roc_auc'
Metric to evaluate the performance of the estimator. Comes from sklearn.metrics. See the model evaluation documentation for more options: https://scikit-learn.org/stable/modules/model_evaluation.html

threshold: float, int, default = 0.01
The value that defines whether a feature will be selected. Note that for metrics like the roc-auc, r2, and the accuracy, the threshold will be a float between 0 and 1. For metrics like the mean squared error and the root mean squared error, the threshold can take any number. The threshold must be defined by the user. With bigger thresholds, fewer features will be selected.

cv: int, cross-validation generator or an iterable, default=3
Determines the cross-validation splitting strategy. Possible inputs for cv are:

- None, to use cross_validate's default 5-fold cross validation
- int, to specify the number of folds in a (Stratified)KFold,
- CV splitter: (https://scikit-learn.org/stable/glossary.html#term-CV-splitter)
- An iterable yielding (train, test) splits as arrays of indices.

For int/None inputs, if the estimator is a classifier and y is either binary or multiclass, StratifiedKFold is used. In all other cases, KFold is used. These splitters are instantiated with `shuffle=False` so the splits will be the same across calls. For more details check Scikit-learn's `cross_validate`'s documentation.

confirm_variables: bool, default=False
If set to True, variables that are not present in the input dataframe will be removed from the list of variables. Only used when passing a variable list to the parameter `variables`. See parameter variables for more details.

Attributes

features_to_drop_:
List with the features that will be removed.

feature_performance_:
Dictionary with the single feature model performance per feature.

variables_:
The variables that will be considered for the feature selection procedure.

feature_names_in_:
List with the names of features seen during `fit`.

n_features_in_:
The number of features in the train set used in fit.

References

Selection based on single feature performance was used in Credit Risk modelling as discussed in the following talk at PyData London 2017:

[1]

Examples

```
>>> import pandas as pd
>>> from sklearn.ensemble import RandomForestClassifier
>>> from feature_
↪engine.selection import SelectBySingleFeaturePerformance
>>> X =␣
↪pd.DataFrame(dict(x1 = [1000,2000,1000,1000,2000,3000],
>>>                    x2 = [2,4,3,1,2,2],
>>>                    x3 = [1,1,1,0,0,0],
>>>                    x4 = [1,2,1,1,0,1],
>>>                    x5 = [1,1,1,1,1,1]))
>>> y = pd.Series([1,0,0,1,1,0])
>>> sfp = SelectBySingleFeaturePerformance(
>>>     ␣
↪                RandomForestClassifier(random_state=42),
>>>                    cv=2)
>>> sfp.fit_transform(X, y)
    x2  x3
```

(continues on next page)

(continued from previous page)

```
0   2   1
1   4   1
2   3   1
3   1   0
4   2   0
5   2   0
```

Methods

fit:	Find the important features.
fit_transform:	Fit to data, then transform it.
get_feature_names_out:	Get output feature names for transformation.
get_params:	Get parameters for this estimator.
set_params:	Set the parameters of this estimator.
get_support:	Get a mask, or integer index, of the features selected.
transform:	Reduce X to the selected features.

`fit`(*X*, *y*)

Determines model performance based on single features. Selects features whose performance is above the threshold.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The input dataframe

y: array-like of shape (n_samples)
Target variable. Required to train the estimator.

`fit_transform`(*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to `X` and `y` with optional parameters `fit_params` and returns a transformed version of `X`.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_feature_names_out`(*input_features=None*)

Get output feature names for transformation. In other words, returns the variable names of transformed dataframe.

Parameters

input_features
[array or list, default=None] This parameter exits only for compatibility with the Scikit-learn pipeline.

- If `None`, then `feature_names_in_` is used as feature names in.
- If an array or list, then `input_features` must match `feature_names_in_`.

Returns

feature_names_out: list
Transformed feature names.

rtype
`List`[`Union`[`str`, `int`]] ..

`get_metadata_routing`()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`get_support`(*indices=False*)

Get a mask, or integer index, of the features selected.

Parameters

indices
[bool, default=False] If True, the return value will be an array of integers, rather than a boolean mask.

Returns

support
[array] An index that selects the retained features from a feature vector. If `indices` is False, this is a boolean array of shape [# input features], in which an element is True if its corresponding feature is selected for retention. If `indices` is True, this is an integer array of shape [# output features] whose values are indices into the input feature vector.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Return dataframe with selected features.

Parameters

X: pandas dataframe of shape = [n_samples, n_features].
The input dataframe.

Returns

X_new: pandas dataframe of shape = [n_samples, n_selected_features]
Pandas dataframe with the selected features.

rtype
`DataFrame` ..

RecursiveFeatureElimination

class feature_engine.selection.**RecursiveFeatureElimination**(*estimator*, *scoring='roc_*
threshold=0.01, *variabl*
confirm_variables=Fals

RecursiveFeatureElimination() selects features following a recursive elimination process.

The process is as follows:

1. Train an estimator using all the features.
2. Rank the features according to their importance derived from the estimator.
3. Remove the least important feature and fit a new estimator.
4. Calculate the performance of the new estimator.
5. Calculate the performance difference between the new and original estimator.
6. If the performance drop is below the threshold the feature is removed.
7. Repeat steps 3-6 until all features have been evaluated.

Model training and performance evaluation are done with cross-validation.

More details in the *User Guide*.

Parameters

estimator: object
A Scikit-learn estimator for regression or classification. The estimator must have either a `feature_importances` or a `coef_` attribute after fitting.

variables: str or list, default=None
The list of variables to evaluate. If None, the transformer will evaluate all numerical features in the dataset.

scoring: str, default='roc_auc'
Metric to evaluate the performance of the estimator. Comes from sklearn.metrics. See the model evaluation documentation for more options: https://scikit-learn.org/stable/modules/model_evaluation.html

threshold: float, int, default = 0.01
The value that defines whether a feature will be selected. Note that for metrics like the roc-auc, r2, and the accuracy, the threshold will be a float between 0 and 1. For metrics like the mean squared error and the root mean squared error, the threshold can take any number. The threshold must be defined by the user. With bigger thresholds, fewer features will be selected.

cv: int, cross-validation generator or an iterable, default=3
Determines the cross-validation splitting strategy. Possible inputs for cv are:

- None, to use cross_validate's default 5-fold cross validation
- int, to specify the number of folds in a (Stratified)KFold,
- CV splitter: (https://scikit-learn.org/stable/glossary.html#term-CV-splitter)
- An iterable yielding (train, test) splits as arrays of indices.

For int/None inputs, if the estimator is a classifier and y is either binary or multiclass, StratifiedKFold is used. In all other cases, KFold is used. These splitters are instantiated with `shuffle=False` so the splits will be the same across calls. For more details check Scikit-learn's `cross_validate`'s documentation.

confirm_variables: bool, default=False
If set to True, variables that are not present in the input dataframe will be removed from the list of variables. Only used when passing a variable list to the parameter `variables`. See parameter variables for more details.

Attributes

initial_model_performance_:
The model's performance when trained with the original dataset.

feature_importances_:
Pandas Series with the feature importance (comes from step 2)

performance_drifts_:
Dictionary with the performance drift per examined feature (comes from step 5).

features_to_drop_:
List with the features that will be removed.

variables_:
The variables that will be considered for the feature selection procedure.

feature_names_in_:
List with the names of features seen during `fit`.

n_features_in_:
The number of features in the train set used in fit.

Examples

```
>>> import pandas as pd
>>> from sklearn.ensemble import RandomForestClassifier
>>> from feature_
↪engine.selection import RecursiveFeatureElimination
>>> X =␣
↪pd.DataFrame(dict(x1 = [1000,2000,1000,1000,2000,3000],
>>>                    x2 = [2,4,3,1,2,2],
>>>                    x3 = [1,1,1,0,0,0],
>>>                    x4 = [1,2,1,1,0,1],
>>>                    x5 = [1,1,1,1,1,1]))
>>> y = pd.Series([1,0,0,1,1,0])
>
↪>
↪>
↪␣
↪rfe␣
↪=␣
↪RecursiveFeatureElimination(RandomForestClassifier(random_
↪state=2), cv=2)
```

(continues on next page)

(continued from previous page)

```
>>> rfe.fit_transform(X, y)
   x2
0   2
1   4
2   3
3   1
4   2
5   2
```

Methods

fit:	Find the important features.
fit_transform:	Fit to data, then transform it.
get_feature_names_out:	Get output feature names for transformation.
get_params:	Get parameters for this estimator.
set_params:	Set the parameters of this estimator.
get_support:	Get a mask, or integer index, of the features selected.
transform:	Reduce X to the selected features.

`fit`(*X*, *y*)

Find the important features. Note that the selector trains various models at each round of selection, so it might take a while.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The input dataframe

y: array-like of shape (n_samples)
Target variable. Required to train the estimator.

`fit_transform`(*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to `X` and `y` with optional parameters `fit_params` and returns a transformed version of `X`.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_feature_names_out`(*input_features=None*)

Get output feature names for transformation. In other words, returns the variable names of transformed dataframe.

Parameters

input_features
[array or list, default=None] This parameter exits only for compatibility with the Scikit-learn pipeline.

- If `None`, then `feature_names_in_` is used as feature names in.
- If an array or list, then `input_features` must match `feature_names_in_`.

Returns

feature_names_out: list
Transformed feature names.

rtype
`List`[`Union`[`str`, `int`]] ..

`get_metadata_routing`()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`get_support`(*indices=False*)

Get a mask, or integer index, of the features selected.

Parameters

indices
[bool, default=False] If True, the return value will be an array of integers, rather than a boolean mask.

Returns

support
[array] An index that selects the retained features from a feature vector. If `indices` is False, this is a boolean array of shape [# input features], in which an element is True if its corresponding feature is selected for retention. If `indices` is True, this is an integer array of shape [# output features] whose values are indices into the input feature vector.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Return dataframe with selected features.

Parameters

X: pandas dataframe of shape = [n_samples, n_features].
The input dataframe.

Returns

X_new: pandas dataframe of shape = [n_samples, n_selected_features]
Pandas dataframe with the selected features.

rtype
`DataFrame` ..

RecursiveFeatureAddition

class feature_engine.selection.**RecursiveFeatureAddition**(*estimator*, *scoring='roc_auc'*, *threshold=0.01*, *variables=None*, *confirm_variables=False*)

RecursiveFeatureAddition() selects features following a recursive addition process.

The process is as follows:

1. Train an estimator using all the features.
2. Rank the features according to their importance derived from the estimator.
3. Train an estimator with the most important feature and determine performance.
4. Add the second most important feature and train a new estimator.
5. Calculate the difference in performance between estimators.
6. If the performance increases beyond the threshold, the feature is kept.
7. Repeat steps 4-6 until all features have been evaluated.

Model training and performance calculation are done with cross-validation.

More details in the *User Guide*.

Parameters

estimator: object
A Scikit-learn estimator for regression or classification. The estimator must have either a `feature_importances` or a `coef_` attribute after fitting.

variables: str or list, default=None
The list of variables to evaluate. If None, the transformer will evaluate all numerical features in the dataset.

scoring: str, default='roc_auc'
Metric to evaluate the performance of the estimator. Comes from sklearn.metrics. See the model evaluation documentation for more options: https://scikit-learn.org/stable/modules/model_evaluation.html

threshold: float, int, default = 0.01
The value that defines whether a feature will be selected. Note that for metrics like the roc-auc, r2, and the accuracy, the threshold will be a float between 0 and 1. For metrics like the mean squared error and the root mean squared error, the threshold can take any number. The threshold must be defined by the user. With bigger thresholds, fewer features will be selected.

cv: int, cross-validation generator or an iterable, default=3
Determines the cross-validation splitting strategy. Possible inputs for cv are:

- None, to use cross_validate's default 5-fold cross validation
- int, to specify the number of folds in a (Stratified)KFold,
- CV splitter: (https://scikit-learn.org/stable/glossary.html#term-CV-splitter)
- An iterable yielding (train, test) splits as arrays of indices.

For int/None inputs, if the estimator is a classifier and y is either binary or multiclass, StratifiedKFold is used. In all other cases, KFold is used. These splitters are instantiated with `shuffle=False` so the splits will be the same across calls. For more details check Scikit-learn's `cross_validate`'s documentation.

confirm_variables: bool, default=False
If set to True, variables that are not present in the input dataframe will be removed from the list of variables. Only used when passing a variable list to the parameter `variables`. See parameter variables for more details.

Attributes

initial_model_performance_:
The model's performance when trained with the original dataset.

feature_importances_:
Pandas Series with the feature importance (comes from step 2)

performance_drifts_:
Dictionary with the performance drift per examined feature (comes from step 5).

features_to_drop_:
List with the features that will be removed.

variables_:
The variables that will be considered for the feature selection procedure.

feature_names_in_:
List with the names of features seen during `fit`.

n_features_in_:
The number of features in the train set used in fit.

Examples

```
>>> import pandas as pd
>>> from sklearn.ensemble import RandomForestClassifier
>>> from␣
↪feature_engine.selection import RecursiveFeatureAddition
>>> X =␣
↪pd.DataFrame(dict(x1 = [1000,2000,1000,1000,2000,3000],
>>>                    x2 = [2,4,3,1,2,2],
>>>                    x3 = [1,1,1,0,0,0],
>>>                    x4 = [1,2,1,1,0,1],
>>>                    x5 = [1,1,1,1,1,1]))
>>> y = pd.Series([1,0,0,1,1,0])
>>> rfa =␣
↪RecursiveFeatureAddition(RandomForestClassifier(random_
↪state=42), cv=2)
>>> rfa.fit_transform(X, y)
   x2  x4
0   2   1
1   4   2
2   3   1
```

(continues on next page)

(continued from previous page)

```
3   1   1
4   2   0
5   2   1
```

Methods

fit:	Find the important features.
fit_transform:	Fit to data, then transform it.
get_feature_names_out:	Get output feature names for transformation.
get_params:	Get parameters for this estimator.
set_params:	Set the parameters of this estimator.
get_support:	Get a mask, or integer index, of the features selected.
transform:	Reduce X to the selected features.

`fit`(*X*, *y*)

Find the important features. Note that the selector trains various models at each round of selection, so it might take a while.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The input dataframe

y: array-like of shape (n_samples)
Target variable. Required to train the estimator.

`fit_transform`(*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to `X` and `y` with optional parameters `fit_params` and returns a transformed version of `X`.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_feature_names_out`(*input_features=None*)

Get output feature names for transformation. In other words, returns the variable names of transformed dataframe.

Parameters

input_features
[array or list, default=None] This parameter exits only for compatibility with the Scikit-learn pipeline.

- If `None`, then `feature_names_in_` is used as feature names in.
- If an array or list, then `input_features` must match `feature_names_in_`.

Returns

feature_names_out: list
Transformed feature names.

rtype
`List`[`Union`[`str`, `int`]] ..

`get_metadata_routing`()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`get_support`(*indices=False*)

Get a mask, or integer index, of the features selected.

Parameters

indices
[bool, default=False] If True, the return value will be an array of integers, rather than a boolean mask.

Returns

support
[array] An index that selects the retained features from a feature vector. If `indices` is False, this is a boolean array of shape [# input features], in which an element is True if its corresponding feature is selected for retention. If `indices` is True, this is an integer array of shape [# output features] whose values are indices into the input feature vector.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Return dataframe with selected features.

Parameters

X: pandas dataframe of shape = [n_samples, n_features].
The input dataframe.

Returns

X_new: pandas dataframe of shape = [n_samples, n_selected_features]
Pandas dataframe with the selected features.

rtype
`DataFrame` ..

DropHighPSIFeatures

class feature_engine.selection.**DropHighPSIFeatures**(*split_col=None*, *split_frac=0.5*, *split_distinct=False*, *cut_off=None*, *threshold=0.25*, *bins=10*, *strategy='equal_frequency'*, *min_pct_empty_bins=0.0001*, *missing_values='raise'*, *variables=*, *confirm_variables=False*, *p_value=*

DropHighPSIFeatures() drops features which Population Stability Index (PSI) is above a given threshold.

The PSI is used to compare distributions. Higher PSI values mean greater changes in a feature's distribution. Therefore, a feature with high PSI can be considered unstable.

To compute the PSI, DropHighPSIFeatures() splits the dataset in two: a basis and a test set. Then, it compares the distribution of each feature between those sets.

To determine the PSI, continuous features are sorted into discrete intervals, and then, the number of observations per interval are compared between the 2 distributions.

The PSI is calculated as:

PSI = sum ((test_i - basis_i) x ln(test_i/basis_i))

where `basis` and `test` are the 2 datasets, `i` refers to each interval, and then, `test_i` and `basis_i` are the number of observations in interval i in each data set.

The PSI has traditionally been used to assess changes in distributions of continuous variables.

In version 1.7, we extended the functionality of DropHighPSIFeatures() to calculate the PSI for categorical features as well. In this case, `i` is each unique category, and `test_i` and `basis_i` are the number of observations in category i.

Threshold

Different thresholds can be used to assess the magnitude of the distribution shift according to the PSI value. The most commonly used thresholds are:

- Below 10%, the variable has not experienced a significant shift.
- Above 25%, the variable has experienced a major shift.
- Between those two values, the shift is intermediate.

Data split

To compute the PSI, DropHighPSIFeatures() splits the dataset in two: a basis and a test set. Then, it compares the distribution of each feature between those sets.

There are various options to split a dataset:

First, you can indicate which variable should be used to guide the data split. This variable can be of any data type. If you do not enter a variable name,

DropHighPSIFeatures() will use the dataframe index.

Next, you need to specify how that variable (or the index) should be used to split the data. You can specify a proportion of observations to be put in each data set, or alternatively, provide a cut-off value.

If you specify a proportion through the `split_frac` parameter, the data will be sorted to accommodate that proportion. If `split_frac` is 0.5, 50% of the observations will go to either basis or test sets. If `split_frac` is 0.6, 60% of the samples will go to the basis data set and the remaining 40% to the test set.

If `split_distinct` is True, the data will be sorted considering unique values in the selected variables. Check the parameter below for more details.

If you define a numeric cut-off value or a specific date using the `cut_off` parameter, the observations with value <= cut-off will go to the basis data set and the remaining ones to the test set. If the variable used to guide the split is categorical, its values are sorted alphabetically and cut accordingly.

If you pass a list of values in the `cut-off`, the observations with the values in the list, will go to the basis set, and the remaining ones to the test set.

More details in the *User Guide*.

Parameters

split_col: string or int, default=None.
The variable that will be used to split the dataset into the basis and test sets. If None, the dataframe index will be used. `split_col` can be a numerical, categorical or datetime variable. If `split_col` is a categorical variable, and the splitting criteria is given by `split_frac`, it will be assumed that the labels of the variable are sorted alphabetically.

split_frac: float, default=0.5.
The proportion of observations in each of the basis and test dataframes. If `split_frac` is 0.6, 60% of the observations will be put in the basis data set.

If `split_distinct` is True, the indicated fraction may not be achieved exactly. See parameter `split_distinct` for more details.

If `cut_off` is not None, `split_frac` will be ignored and the data split based on the `cut_off` value.

split_distinct: boolean, default=False.
If True, `split_frac` is applied to the vector of unique values in `split_col` instead of being applied to the whole vector of values. For example, if the values in `split_col` are [1, 1, 1, 1, 2, 2, 3, 4] and `split_frac` is 0.5, we have the following:

- **`split_distinct=False` splits the vector in two equally sized parts:** [1, 1, 1, 1] and [2, 2, 3, 4]. This involves that 2 dataframes with 4 observations each are used for the PSI calculations.
- **`split_distinct=True` computes the vector of unique values in `split_col`** ([1, 2, 3, 4]) and splits that vector in two equal parts: [1, 2] and [3, 4]. The number of observations in the two dataframes used for the PSI calculations is respectively 6 ([1, 1, 1, 1, 2, 2]) and 2 ([3, 4]).

cut_off: int, float, date or list, default=None
Threshold to split the dataset based on the `split_col` variable. If int, float

or date, observations where the `split_col` values are <= threshold will go to the basis data set and the rest to the test set. If `cut_off` is a list, the observations where the `split_col` values are within the list will go to the basis data set and the remaining observations to the test set. If `cut_off` is not None, this parameter will be used to split the data and `split_frac` will be ignored.

switch: boolean, default=False.
If True, the order of the 2 dataframes used to determine the PSI (basis and test) will be switched. This is important because the PSI is not symmetric, i.e., PSI(a, b) != PSI(b, a)).

threshold: float, str, default = 0.25.
The threshold to drop a feature. If the PSI for a feature is >= threshold, the feature will be dropped. The most common threshold values are 0.25 (large shift) and 0.10 (medium shift). If 'auto', the threshold will be calculated based on the size of the basis and test dataset and the number of bins as:

threshold = 2(q, B1) × (1/N + 1/M)

where:

- q = quantile of the distribution (or 1 - p-value),
- B = number of bins/categories,
- N = size of basis dataset,
- M = size of test dataset.

See formula (5.2) from reference [1].

bins: int, default = 10
Number of bins or intervals. For continuous features with good value spread, 10 bins is commonly used. For features with lower cardinality or highly skewed distributions, lower values may be required.

strategy: string, default='equal_frequency'
If the intervals into which the features should be discretized are of equal size or equal number of observations. Takes values "equal_width" for equally spaced bins or "equal_frequency" for bins based on quantiles, that is, bins with similar number of observations.

min_pct_empty_bins: float, default = 0.0001
Value to add to empty bins or intervals. If after sorting the variable values into bins, a bin is empty, the PSI cannot be determined. By adding a small number to empty bins, we can avoid this issue. Note, that if the value added is too large, it may disturb the PSI calculation.

missing_values: str, default='raise'
Whether to perform the PSI feature selection on a dataframe with missing values. Takes values 'raise' or 'ignore'. If 'ignore', missing values will be dropped when determining the PSI for that particular feature. If 'raise' the transformer will raise an error and features will not be selected.

p_value: float, default = 0.001
The p-value to test the null hypothesis that there is no feature drift. In that case, the PSI-value approximates a random variable that follows a chi-square distribution. See [1] for details. This parameter is used only if `threshold` is set to 'auto'.

variables: int, str, list, default = None
The list of variables to evaluate. If `None`, the transformer will evaluate all numerical variables in the dataset. If `"all"` the transformer will evaluate all categorical and numerical variables in the dataset. Alternatively, the transformer will evaluate the variables indicated in the list or string.

confirm_variables: bool, default=False
If set to True, variables that are not present in the input dataframe will be removed from the list of variables. Only used when passing a variable list to the parameter `variables`. See parameter variables for more details.

Attributes

features_to_drop_:
List with the features that will be dropped.

variables_:
The variables that will be considered for the feature selection procedure.

psi_values_:
Dictionary containing the PSI value per feature.

cut_off_:
Value used to split the dataframe into basis and test. This value is computed when not given as parameter.

feature_names_in_:
List with the names of features seen during `fit`.

n_features_in_:
The number of features in the train set used in fit.

See also:

feature_engine.discretisation.EqualFrequencyDiscretiser
feature_engine.discretisation.EqualWidthDiscretiser

References

[1]

Examples

```
>>> import pandas as pd
>>>␣
↪from feature_engine.selection import DropHighPSIFeatures
>>> X = pd.DataFrame(dict(
>>>
↪          x1 = [1,1,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
>>>          x2␣
↪= [32,87,6,32,11,44,8,7,9,0,32,87,6,32,11,44,8,7,9,0],
>>>          ))
>>> psi = DropHighPSIFeatures()
>>> psi.fit_transform(X)
    x2
```

(continues on next page)

(continued from previous page)

```
0   32
1   87
2    6
3   32
4   11
5   44
6    8
7    7
8    9
9    0
10  32
```

Methods

fit:	Find features with high PSI values.
fit_transform:	Fit to data, then transform it.
get_feature_names_out:	Get output feature names for transformation.
get_params:	Get parameters for this estimator.
set_params:	Set the parameters of this estimator.
get_support:	Get a mask, or integer index, of the features selected.
transform:	Remove features with high PSI values.

`fit`(*X*, *y=None*)

Find features with high PSI values.

Parameters

X

[pandas dataframe of shape = [n_samples, n_features]] The training dataset.

y

[pandas series. Default = None] y is not needed in this transformer. You can pass y or None.

`fit_transform`(*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to `X` and `y` with optional parameters `fit_params` and returns a transformed version of `X`.

Parameters

X

[array-like of shape (n_samples, n_features)] Input samples.

y

[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**

[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_feature_names_out`(*input_features=None*)

Get output feature names for transformation. In other words, returns the variable names of transformed dataframe.

Parameters

input_features
[array or list, default=None] This parameter exits only for compatibility with the Scikit-learn pipeline.

- If `None`, then `feature_names_in_` is used as feature names in.
- If an array or list, then `input_features` must match `feature_names_in_`.

Returns

feature_names_out: list
Transformed feature names.

rtype
`List`[`Union`[`str`, `int`]] ..

`get_metadata_routing`()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`get_support`(*indices=False*)

Get a mask, or integer index, of the features selected.

Parameters

indices
[bool, default=False] If True, the return value will be an array of integers, rather than a boolean mask.

Returns

support
[array] An index that selects the retained features from a feature vector. If `indices` is False, this is a boolean array of shape [# input features], in which an element is True if its corresponding feature is selected for retention. If `indices` is True, this is an integer array of shape [# output features] whose values are indices into the input feature vector.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Return dataframe with selected features.

Parameters

X: pandas dataframe of shape = [n_samples, n_features].
The input dataframe.

Returns

X_new: pandas dataframe of shape = [n_samples, n_selected_features]
Pandas dataframe with the selected features.

rtype
`DataFrame` ..

SelectByInformationValue

class feature_engine.selection.**SelectByInformationValue**(*variables=None*, *bins=5*, *strategy='equal_width'*, *thre*
confirm_variables=False)

SelectByInformationValue() selects features based on their information value (IV). The IV is calculated as:

$$IV = (fractionofpositivecases - fractionofnegativecases) * WoE$$

where:

- **the fraction of positive cases is the proportion of observations of class 1,** from the total class 1 observations.
- **the fraction of negative cases is the proportion of observations of class 0,** from the total class 0 observations.
  - WoE is the weight of the evidence.

SelectByInformationValue() is only suitable to select features for binary classification.

SelectByInformationValue() can determine the IV for numerical and categorical variables. For numerical variables, it first sorts the variables into intervals, and then determines the IV.

You can pass a list of variables to examine. Alternatively, the transformer will examine all variables.

The IV allows you to assess each variable's independent contribution to the target variable. The transformer selects those variables whose IV is higher than the threshold.

More details in the *User Guide*.

Parameters

variables: list, default=None
The list of variables to evaluate. If None, the transformer will evaluate all variables in the dataset (except datetime).

bins: int, default = 5
If the dataset contains numerical variables, the number of bins into which the values will be sorted.

strategy: str, default = 'equal_width'
Whether the bins should be of equal width ('equal_width') or equal frequency ('equal_frequency').

threshold: float, int, default = 0.2.
The threshold to drop a feature. If the IV for a feature is < threshold, the feature will be dropped.

confirm_variables: bool, default=False
If set to True, variables that are not present in the input dataframe will be removed from the list of variables. Only used when passing a variable list to the parameter `variables`. See parameter variables for more details.

Attributes

variables_:
The group of variables that will be transformed.

information_values_:
A dictionary with the information values for each feature.

features_to_drop_:
List with the features that will be removed.

feature_names_in_:
List with the names of features seen during `fit`.

n_features_in_:
The number of features in the train set used in fit.

See also:

feature_engine.encoding.WoEEncoder
feature_engine.discretisation.EqualWidthDiscretiser
feature_engine.discretisation.EqualFrequencyDiscretiser

References

[1], [2]

Examples

```
>>> import pandas as pd
>>> from␣
↪feature_engine.selection import SelectByInformationValue
>>> X = pd.DataFrame(dict(x1 = [1,1,1,1,1,1],
>>>                       x2 = [3,2,2,3,3,2],
>>>                       x3 = ["a","b","c","a","c","b"]))
>>> y = pd.Series([1,1,1,0,0,0])
>>> iv = SelectByInformationValue()
>>> iv.fit_transform(X, y)
    x2
0   3
1   2
2   2
3   3
4   3
5   2
```

Methods

fit:	Find features with high information value.
fit_transform:	Fit to data, then transform it.
get_feature_names_out:	Get output feature names for transformation.
get_params:	Get parameters for this estimator.
set_params:	Set the parameters of this estimator.
get_support:	Get a mask, or integer index, of the features selected.
transform:	Remove features with low information value.

`fit`(*X*, *y*)

Learn the information value. Find features with IV above the threshold.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The training input samples.

y: pandas series of shape = [n_samples,]
Target, must be binary.

`fit_transform`(*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to `X` and `y` with optional parameters `fit_params` and returns a transformed version of `X`.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_feature_names_out`(*input_features=None*)

Get output feature names for transformation. In other words, returns the variable names of transformed dataframe.

Parameters

input_features
[array or list, default=None] This parameter exits only for compatibility with the Scikit-learn pipeline.

- If `None`, then `feature_names_in_` is used as feature names in.
- If an array or list, then `input_features` must match `feature_names_in_`.

Returns

feature_names_out: list
Transformed feature names.

rtype
`List`[`Union`[`str`, `int`]] ..

`get_metadata_routing`()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`get_support`(*indices=False*)

Get a mask, or integer index, of the features selected.

Parameters

indices
[bool, default=False] If True, the return value will be an array of integers, rather than a boolean mask.

Returns

support
[array] An index that selects the retained features from a feature vector. If `indices` is False, this is a boolean array of shape [# input features], in which an element is True if its corresponding feature is selected for retention. If `indices` is True, this is an integer array of shape [# output features] whose values are indices into the input feature vector.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Return dataframe with selected features.

Parameters

X: pandas dataframe of shape = [n_samples, n_features].
The input dataframe.

Returns

X_new: pandas dataframe of shape = [n_samples, n_selected_features]
Pandas dataframe with the selected features.

rtype
`DataFrame` ..

SelectByShuffling

`class feature_engine.selection.SelectByShuffling`(*estimator*, *scoring='roc_auc'*, *cv=3*, *threshold=None*, *variables=None*, *random_state=None*, *confirm_variab*

SelectByShuffling() selects features by determining the drop in machine learning model performance when each feature's values are randomly shuffled.

If the variables are important, a random permutation of their values will decrease dramatically the machine learning model performance. Contrarily, the

permutation of the values should have little to no effect on the model performance metric we are assessing if the feature is not predictive.

The SelectByShuffling() first trains a machine learning model utilising all features. Next, it shuffles the values of 1 feature, obtains a prediction with the pre-trained model, and determines the performance drop (if any). If the drop in performance is bigger than a threshold then the feature is retained, otherwise removed. It continues until all features have been shuffled and examined.

The user can determine the model for which performance drop after feature shuffling should be assessed. The user also determines the threshold in performance under which a feature will be removed, and the performance metric to evaluate.

Model training and performance calculation are done with cross-validation.

More details in the *User Guide*.

Parameters

estimator: object
A Scikit-learn estimator for regression or classification.

variables: str or list, default=None
The list of variables to evaluate. If None, the transformer will evaluate all numerical features in the dataset.

scoring: str, default='roc_auc'
Metric to evaluate the performance of the estimator. Comes from sklearn.metrics. See the model evaluation documentation for more options: https://scikit-learn.org/stable/modules/model_evaluation.html

threshold: float, int, default = 0.01
The value that defines whether a feature will be selected. Note that for metrics like the roc-auc, r2, and the accuracy, the threshold will be a float between 0 and 1. For metrics like the mean squared error and the root mean squared error, the threshold can take any number. The threshold must be defined by the user. With bigger thresholds, fewer features will be selected.

cv: int, cross-validation generator or an iterable, default=3
Determines the cross-validation splitting strategy. Possible inputs for cv are:

- None, to use cross_validate's default 5-fold cross validation
- int, to specify the number of folds in a (Stratified)KFold,
- CV splitter: (https://scikit-learn.org/stable/glossary.html#term-CV-splitter)
- An iterable yielding (train, test) splits as arrays of indices.

For int/None inputs, if the estimator is a classifier and y is either binary or multiclass, StratifiedKFold is used. In all other cases, KFold is used. These splitters are instantiated with `shuffle=False` so the splits will be the same across calls. For more details check Scikit-learn's `cross_validate`'s documentation.

random_state: int, default=None
Controls the randomness when shuffling features.

confirm_variables: bool, default=False
If set to True, variables that are not present in the input dataframe will be

removed from the list of variables. Only used when passing a variable list to the parameter `variables`. See parameter variables for more details.

Attributes

initial_model_performance_:
The model's performance when trained with the original dataset.

performance_drifts_:
Dictionary with the performance drift per shuffled feature.

features_to_drop_:
List with the features that will be removed.

variables_:
The variables that will be considered for the feature selection procedure.

feature_names_in_:
List with the names of features seen during `fit`.

n_features_in_:
The number of features in the train set used in fit.

See also:

`sklearn.inspection.permutation_importance`

Notes

This transformer is a similar concept to the `permutation_importance` from Scikit-learn. The function in Scikit-learn is used to evaluate feature importance instead of to select features.

Examples

```
>>> import pandas as pd
>>> from sklearn.ensemble import RandomForestClassifier
>>> from feature_engine.selection import SelectByShuffling
>>> X =␣
↪pd.DataFrame(dict(x1 = [1000,2000,1000,1000,2000,3000],
>>>                    x2 = [2,4,3,1,2,2],
>>>                    x3 = [1,1,1,0,0,0],
>>>                    x4 = [1,2,1,1,0,1],
>>>                    x5 = [1,1,1,1,1,1]))
>>> y = pd.Series([1,0,0,1,1,0])
>>> sbs = SelectByShuffling(
>>>         RandomForestClassifier(random_state=42),
>>>         cv=2,
>>>         random_state=42,
>>>       )
>>> sbs.fit_transform(X, y)
   x2  x4  x5
0   2   1   1
```

(continues on next page)

(continued from previous page)

```
1   4   2   1
2   3   1   1
3   1   1   1
4   2   0   1
5   2   1   1
```

Methods

fit:	Find the important features.
fit_transform:	Fit to data, then transform it.
get_feature_names_out:	Get output feature names for transformation.
get_params:	Get parameters for this estimator.
set_params:	Set the parameters of this estimator.
get_support:	Get a mask, or integer index, of the features selected.
transform:	Reduce X to the selected features.

`fit`(*X*, *y*, *sample_weight=None*)

Find the important features.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The input dataframe.

y: array-like of shape (n_samples)
Target variable. Required to train the estimator.

sample_weight
[array-like of shape (n_samples,), default=None] Sample weights. If None, then samples are equally weighted.

`fit_transform`(*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to `X` and `y` with optional parameters `fit_params` and returns a transformed version of `X`.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_feature_names_out`(*input_features=None*)

Get output feature names for transformation. In other words, returns the variable names of transformed dataframe.

Parameters

input_features
[array or list, default=None] This parameter exits only for compatibility with the Scikit-learn pipeline.

- If `None`, then `feature_names_in_` is used as feature names in.
- If an array or list, then `input_features` must match `feature_names_in_`.

Returns

feature_names_out: list
Transformed feature names.

rtype
`List`[`Union`[`str`, `int`]] ..

`get_metadata_routing`()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`get_support`(*indices=False*)

Get a mask, or integer index, of the features selected.

Parameters

indices
[bool, default=False] If True, the return value will be an array of integers, rather than a boolean mask.

Returns

support
[array] An index that selects the retained features from a feature vector. If `indices` is False, this is a boolean array of shape [# input features], in which an element is True if its corresponding feature is selected for retention. If `indices` is True, this is an integer array of shape [# output features] whose values are indices into the input feature vector.

set_fit_request(*, *sample_weight: Union[bool, None, str] = '$UNCHANGED$'*) → *SelectByShu*

Request metadata passed to the `fit` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `fit` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `fit`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

Parameters

sample_weight
[str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED] Metadata routing for `sample_weight` parameter in `fit`.

Returns

self
[object] The updated object.

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as Pipeline). The latter have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

transform(*X*)

Return dataframe with selected features.

Parameters

X: pandas dataframe of shape = [n_samples, n_features].
The input dataframe.

Returns

X_new: pandas dataframe of shape = [n_samples, n_selected_features]
Pandas dataframe with the selected features.

rtype
DataFrame ..

SelectByTargetMeanPerformance

class feature_engine.selection.**SelectByTargetMeanPerformance**(*variables=None*, *bins*
strategy='equal_width
scoring='roc_auc', *cv*
threshold=None, *regr*
confirm_variables=Fa

SelectByTargetMeanPerformance() uses the mean value of the target per category or per interval(if the variable is numerical), as proxy for target estimation. With this proxy, the selector determines the performance of each feature based on a metric of choice, and then selects the features based on this performance value.

SelectByTargetMeanPerformance() can evaluate numerical and categorical variables, without much prior manipulation. In other words, you don't need to encode the categorical variables or transform the numerical variables to assess their importance if you use this transformer.

SelectByTargetMeanPerformance() requires that the dataset is complete, without missing data.

SelectByTargetMeanPerformance() determines the performance of each variable with cross-validation. More specifically:

For each categorical variable:

1. Determines the mean target value per category in the training folds.
2. Replaces the categories by the target mean values in the test folds.
3. Determines the performance of the transformed variables in the test folds.

For each numerical variable:

1. Discretises the variable into intervals of equal width or equal frequency.
2. Determines the mean value of the target per interval in the training folds.
3. Replaces the intervals by the target mean values in the test fold.
4. Determines the performance of the transformed variable in the test fold.

Finally, it selects the features which performance is bigger than the indicated threshold. If the threshold if left to None, it selects features which performance is bigger than the mean performance of all features.

All the steps are performed with cross-validation. That means, that intervals and target mean values per interval or category are determined in a certain portion of the data, and evaluated in a left-out sample. The performance metric per variable is the average across the cross-validation folds.

More details in the *User Guide*.

Parameters

variables: list, default=None
The list of variables to evaluate. If None, the transformer will evaluate all variables in the dataset (except datetime).

bins: int, default = 5
If the dataset contains numerical variables, the number of bins into which the values will be sorted.

strategy: str, default = 'equal_width'
Whether the bins should be of equal width ('equal_width') or equal frequency ('equal_frequency').

scoring: str, default='roc_auc'
Metric to evaluate the performance of the estimator. Comes from sklearn.metrics. See the model evaluation documentation for more options: https://scikit-learn.org/stable/modules/model_evaluation.html

threshold: float, int, default = 0.01
The value that defines whether a feature will be selected. Note that for metrics like the roc-auc, r2, and the accuracy, the threshold will be a float between 0 and 1. For metrics like the mean squared error and the root mean squared error, the threshold can take any number. The threshold must be defined by the user. With bigger thresholds, fewer features will be selected.

cv: int, cross-validation generator or an iterable, default=3
Determines the cross-validation splitting strategy. Possible inputs for cv are:

- None, to use cross_validate's default 5-fold cross validation
- int, to specify the number of folds in a (Stratified)KFold,
- CV splitter: (https://scikit-learn.org/stable/glossary.html#term-CV-splitter)
- An iterable yielding (train, test) splits as arrays of indices.

  For int/None inputs, if the estimator is a classifier and y is either binary or multiclass, StratifiedKFold is used. In all other cases, KFold is used. These splitters are instantiated with `shuffle=False` so the splits will be the same across calls. For more details check Scikit-learn's `cross_validate`'s documentation.

regression: boolean, default=True
Indicates whether the target is one for regression or a classification.

confirm_variables: bool, default=False
If set to True, variables that are not present in the input dataframe will be removed from the list of variables. Only used when passing a variable list to the parameter `variables`. See parameter variables for more details.

Attributes

variables_:
The variables that will be considered for the feature selection procedure.

feature_performance_:
Dictionary with the performance of each feature.

features_to_drop_:
List with the features that will be removed.

feature_names_in_:
List with the names of features seen during `fit`.

n_features_in_:
The number of features in the train set used in fit.

See also:

feature_engine.encoding.MeanEncoder
feature_engine.discretisation.EqualWidthDiscretiser
feature_engine.discretisation.EqualFrequencyDiscretiser

Notes

Replacing the categories or intervals by the target mean is the equivalent to target mean encoding.

References

[1]

Examples

```
>>> from sklearn.ensemble import RandomForestClassifier
>>> from feature_
↪engine.selection import SelectByTargetMeanPerformance
>>> X =␣
↪pd.DataFrame(dict(x1 = [1000,2000,1000,1000,2000,3000],
>>>                    x2 = [1,1,1,0,0,0],
>>>                    x3 = [1,2,1,1,0,1],
>>>                    x4 = [1,1,1,1,1,1]))
>>> y = pd.Series([1,0,0,1,1,0])
>>> tmp = SelectByTargetMeanPerformance(bins␣
↪= 3, cv=2,scoring='accuracy')
>>> tmp.fit_transform(X, y)
   x2  x3  x4
0   1   1   1
1   1   2   1
2   1   1   1
3   0   1   1
4   0   0   1
5   0   1   1
```

This transformer also works with Categorical examples:

```
>>> X = pd.DataFrame(dict(x1 = ["a","b","a","a","b","b"],
>>>                x2 = ["a","a","a","b","b","b"]))
>>> y = pd.Series([1,0,0,1,1,0])
>>> tmp = SelectByTargetMeanPerformance(bins␣
↪= 3, cv=2,scoring='accuracy')
>>> tmp.fit_transform(X, y)
  x2
0  a
1  a
2  a
3  b
4  b
5  b
```

Methods

fit:	Find the important features.
fit_transform:	Fit to data, then transform it.
get_feature_names_out:	Get output feature names for transformation.
get_params:	Get parameters for this estimator.
set_params:	Set the parameters of this estimator.
get_support:	Get a mask, or integer index, of the features selected.
transform:	Reduce X to the selected features.

`fit`(*X*, *y*)

Find the important features.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The input dataframe.

y: array-like of shape (n_samples)
Target variable. Required to train the estimator.

`fit_transform`(*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to `X` and `y` with optional parameters `fit_params` and returns a transformed version of `X`.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_feature_names_out`(*input_features=None*)

Get output feature names for transformation. In other words, returns the variable names of transformed dataframe.

Parameters

input_features
[array or list, default=None] This parameter exits only for compatibility with the Scikit-learn pipeline.

- If `None`, then `feature_names_in_` is used as feature names in.
- If an array or list, then `input_features` must match `feature_names_in_`.

Returns

feature_names_out: list
Transformed feature names.

rtype
List[Union[str, int]] ..

`get_metadata_routing()`

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

`get_params(deep=True)`

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`get_support(indices=False)`

Get a mask, or integer index, of the features selected.

Parameters

indices
[bool, default=False] If True, the return value will be an array of integers, rather than a boolean mask.

Returns

support
[array] An index that selects the retained features from a feature vector. If `indices` is False, this is a boolean array of shape [# input features], in which an element is True if its corresponding feature is selected for retention. If `indices` is True, this is an integer array of shape [# output features] whose values are indices into the input feature vector.

`set_params(**params)`

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Return dataframe with selected features.

Parameters

X: pandas dataframe of shape = [n_samples, n_features].
The input dataframe.

Returns

X_new: pandas dataframe of shape = [n_samples, n_selected_features]
Pandas dataframe with the selected features.

rtype
`DataFrame` ..

ProbeFeatureSelection

`class` feature_engine.selection.`ProbeFeatureSelection`(*estimator*, *variables=None*, *scor*
n_probes=1, *distribution='norm*
random_state=0, *confirm_varia*

ProbeFeatureSelection() generates one or more probe features based on the user-selected distribution. The distribution options are 'normal', 'binomial', 'uniform', or 'all'. 'all' creates at least one distribution for each of the three aforementioned distributions.

Using cross validation, the class fits a Scikit-learn estimator to the provided dataset's variables and the probe features.

The class derives the feature importance for each variable and probe feature. In the case of there being more than one probe feature, ProbeFeatureSelection() calculates the average feature importance of all the probe features.

The variables that have a feature importance less than the feature importance or average feature importance of the probe feature(s) are dropped from the dataset.

More details in the *User Guide*.

Parameters

estimator: object
A Scikit-learn estimator for regression or classification. The estimator must have either a `feature_importances` or a `coef_` attribute after fitting.

variables: str or list, default=None
The list of variables to evaluate. If None, the transformer will evaluate all numerical features in the dataset.

scoring: str, default='roc_auc'
Metric to evaluate the performance of the estimator. Comes from sklearn.metrics. See the model evaluation documentation for more options: https://scikit-learn.org/stable/modules/model_evaluation.html

n_probes: int, default=1
Number of probe features to be created. If distribution is 'all', n_probes must be a multiple of 3.

distribution: str, default='normal'
The distribution used to create the probe features. The options are 'normal', 'binomial', 'uniform', and 'all'. 'all' creates at least 1 or more probe features comprised of each distribution type, i.e., normal, binomial, and uniform. The remaining options create `n_probes` features of the selected distribution.

cv: int, cross-validation generator or an iterable, default=3
Determines the cross-validation splitting strategy. Possible inputs for cv are:

- None, to use cross_validate's default 5-fold cross validation
- int, to specify the number of folds in a (Stratified)KFold,
- CV splitter: (https://scikit-learn.org/stable/glossary.html#term-CV-splitter)
- An iterable yielding (train, test) splits as arrays of indices.

For int/None inputs, if the estimator is a classifier and y is either binary or multiclass, StratifiedKFold is used. In all other cases, KFold is used. These splitters are instantiated with `shuffle=False` so the splits will be the same across calls. For more details check Scikit-learn's `cross_validate`'s documentation.

Attributes

probe_features_:
A dataframe comprised of the pseudo-randomly generated features based on the selected distribution.

feature_importances_:
Pandas Series with the feature importance.

features_to_drop_:
List with the features that will be removed.

variables_:
The variables that will be considered for the feature selection procedure.

feature_names_in_:
List with the names of features seen during `fit`.

n_features_in_:
The number of features in the train set used in fit.

References

[1]

Examples

```
>>> from sklearn.datasets import load_breast_cancer
>>> from sklearn.linear_model import LogisticRegression
>>> from␣
↪feature_engine.selection import ProbeFeatureSelection
>>> X,
↪ y = load_breast_cancer(return_X_y=True, as_frame=True)
>>> sel = ProbeFeatureSelection(
>>>     estimator=LogisticRegression(),
>>>     scoring="roc_auc",
>>>     n_probes=3,
>>>     distribution="normal",
>>>     cv=3,
>>>     random_state=150,
>>> )
>>> X_tr = sel.fit_transform(X, y)
print(X.shape, X_tr.shape)
```

Methods

fit:	Find the important features.
fit_transform:	Fit to data, then transform it.
get_feature_names_out:	Get output feature names for transformation.
get_params:	Get parameters for this estimator.
set_params:	Set the parameters of this estimator.
get_support:	Get a mask, or integer index, of the features selected.
transform:	Reduce X to the selected features.

fit(*X*, *y*)

Find the important features.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
y: array-like of shape (n_samples)
Target variable. Required to train the estimator.

fit_transform(*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to `X` and `y` with optional parameters `fit_params` and returns a transformed version of `X`.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_feature_names_out`(*input_features=None*)

Get output feature names for transformation. In other words, returns the variable names of transformed dataframe.

Parameters

input_features
[array or list, default=None] This parameter exits only for compatibility with the Scikit-learn pipeline.

- If `None`, then `feature_names_in_` is used as feature names in.
- If an array or list, then `input_features` must match `feature_names_in_`.

Returns

feature_names_out: list
Transformed feature names.

rtype
`List`[`Union`[`str`, `int`]] ..

`get_metadata_routing`()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`get_support`(*indices=False*)

Get a mask, or integer index, of the features selected.

Parameters

indices
[bool, default=False] If True, the return value will be an array of integers, rather than a boolean mask.

Returns

support
[array] An index that selects the retained features from a feature vector. If `indices` is False, this is a boolean array of shape [# input features], in which an element is True if its corresponding feature is selected for retention. If `indices` is True, this is an integer array of shape [# output features] whose values are indices into the input feature vector.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Return dataframe with selected features.

Parameters

X: pandas dataframe of shape = [n_samples, n_features].
The input dataframe.

Returns

X_new: pandas dataframe of shape = [n_samples, n_selected_features]
Pandas dataframe with the selected features.

rtype
DataFrame ..

Other Feature Selection Libraries

For additional feature selection algorithms visit the following open-source libraries:

- Scikit-learn selection
- MLXtend selection

Scikit-learn hosts multiple filter and embedded methods that select features based on statistical tests or machine learning model derived importance. MLXtend hosts greedy (wrapper) feature selection methods.

10.3.4 Time series

Time Series Features

Feature-engine's time series transformers derive features from time series data.

Forecasting Features

Feature-engine's time series forecasting transformers create and add new features to the dataframe by lagging features or calculating statistics over windows of time in the past.

LagFeatures

class feature_engine.timeseries.forecasting.**LagFeatures**(*variables=None*, *periods=1*, *fill_value=None*, *sort_index*, *missing_values='raise'*, *drop_original=False*, *drop_*

LagFeatures adds lag features to the dataframe. A lag feature is a feature with information about a prior time step.

LagFeatures has the same functionality as pandas `shift()` with the exception that only one of `periods` or `freq` can be indicated at a time. LagFeatures builds on top of pandas `shift()` in that multiple lags can be created at the same time and the features with names will be concatenated to the original dataframe.

To be compatible with LagFeatures, the dataframe's index must have unique values and no NaN.

LagFeatures works only with numerical variables. You can pass a list of variables to lag. Alternatively, LagFeatures will automatically select and lag all numerical variables found in the training set.

More details in the *User Guide*.

Parameters

variables: list, default=None
The list of numerical variables to transform. If None, the transformer will automatically find and select all numerical variables.

periods: int, list of ints, default=1
Number of periods to shift. Can be a positive integer or list of positive integers. If list, features will be created for each one of the periods in the list. If the parameter `freq` is specified, `periods` will be ignored.

freq: str, list of str, default=None
Offset to use from the tseries module or time rule. See parameter `freq` in pandas `shift()`. It is the same functionality. If freq is a list, lag features will be created for each one of the frequency values in the list. If freq is not None, then this parameter overrides the parameter `periods`.

fill_value: object, optional
The scalar value to use for newly introduced missing values. The default depends on the dtype of the variable. For numeric data, np.nan is used. For datetime, timedelta, or period data, NaT is used. For extension dtypes, self.dtype.na_value is used.

sort_index: bool, default=True
Whether to order the index of the dataframe before creating the lag features.

missing_values: string, default='raise'
Indicates if missing values should be ignored or raised. If `'raise'` the transformer will return an error if the the datasets to `fit` or `transform` contain missing values. If `'ignore'`, missing data will be ignored when learning parameters or performing the transformation.

drop_original: bool, default=False
If True, the original variables to transform will be dropped from the dataframe.

drop_na: bool, default=False.
Whether the NAN introduced in the lag features should be removed.

Attributes

variables_:
The group of variables that will be lagged.

feature_names_in_:
List with the names of features seen during `fit`.

n_features_in_:
The number of features in the train set used in fit.

See also:

`pandas.shift`

Examples

```
>>> import pandas as pd
>>> from␣
↪feature_engine.timeseries.forecasting import LagFeatures
>>> X = pd.DataFrame(dict(date = ["2022-09-18",
>>>                               "2022-09-19",
>>>                               "2022-09-20",
>>>                               "2022-09-21",
>>>                               "2022-09-22"],
>>>                       x1 = [1,2,3,4,5],
>>>                       x2 = [6,7,8,9,10]
>>>                      ))
>>> lf = LagFeatures(periods=[1,2])
>>> lf.fit_transform(X)
      ␣
↪    date  x1  x2  x1_lag_1  x2_lag_1  x1_lag_2  x2_lag_2
0  2022-
↪09-18   1   6       NaN       NaN       NaN       NaN
1  2022-
↪09-19   2   7       1.0       6.0       NaN       NaN
2  2022-
↪09-20   3   8       2.0       7.0       1.0       6.0
3  2022-
↪09-21   4   9       3.0       8.0       2.0       7.0
4  2022-
↪09-22   5  10       4.0       9.0       3.0       8.0
```

Methods

fit:	This transformer does not learn parameters.
fit_transform:	Fit to data, then transform it.
get_feature_names_out:	Get output feature names for transformation.
get_params:	Get parameters for this estimator.
set_params:	Set the parameters of this estimator.
transform:	Add lag features.
transform_x_y:	Remove rows with missing data from X and y.

fit(*X*, *y=None*)

This transformer does not learn parameters.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The training dataset.

y: pandas Series, default=None
y is not needed in this transformer. You can pass None or y.

fit_transform(*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to `X` and `y` with optional parameters `fit_params` and returns a transformed version of `X`.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_feature_names_out`(*input_features=None*)

Get output feature names for transformation. In other words, returns the variable names of transformed dataframe.

Parameters

input_features
[array or list, default=None] This parameter exits only for compatibility with the Scikit-learn pipeline.

- If `None`, then `feature_names_in_` is used as feature names in.
- If an array or list, then `input_features` must match `feature_names_in_`.

Returns

feature_names_out: list
Transformed feature names.

rtype
`List`[`Union`[`str`, `int`]] ..

`get_metadata_routing`()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Adds lag features.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The data to transform.

Returns

X_new: Pandas dataframe, shape = [n_samples, n_features + lag_features]
The dataframe with the original plus the new variables.

rtype
`DataFrame` ..

`transform_x_y`(*X*, *y*)

Transform, align and adjust both X and y based on the transformations applied

to X, ensuring that they correspond to the same set of rows if any were removed from X.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The dataframe to transform.

y: pandas Series or Dataframe of length = n_samples
The target variable to transform. Can be multi-output.

Returns

X_new: pandas dataframe
The transformed dataframe of shape [n_samples - n_rows, n_features]. It may contain less rows than the original dataset.

y_new: pandas Series or DataFrame
The transformed target variable of length [n_samples - n_rows]. It contains as many rows as those left in X_new.

WindowFeatures

class feature_engine.timeseries.forecasting.**WindowFeatures**(*variables=None*, *windo*
min_periods=None, *fun*
periods=1, *freq=None*,
sort_index=True,
missing_values='raise',
drop_original=False, *d*

WindowFeatures adds new features to a dataframe based on window operations. Window operations are operations that perform an aggregation over a sliding partition of past values. A window feature is, in other words, a feature created after computing statistics (e.g., mean, min, max, etc.) using a window over the past data. For example, the mean value of the previous 3 months of data is a window feature. The maximum value of the previous three rows of data is another window feature.

WindowFeatures uses pandas functions `rolling()`, `agg()` and `shift()`. With `rolling()`, it creates rolling windows. With `agg()` it applies multiple functions within those windows. With `shift()` it allocates the values to the correct rows.

For supported aggregation functions, see Rolling Window Functions.

With pandas `rolling()` we can perform rolling operations over 1 window size at a time. WindowFeatures builds on top of pandas `rolling()` in that new features can be derived from multiple window sizes, and the created features will be automatically concatenated to the original dataframe.

To be compatible with WindowFeatures, the dataframe's index must have unique values and no missing data.

WindowFeatures works only with numerical variables. You can pass a list of variables to use as input for the windows. Alternatively, WindowFeatures will automatically select all numerical variables in the training set.

More details in the *User Guide*.

Parameters

variables: list, default=None
The list of numerical variables to transform. If None, the transformer will automatically find and select all numerical variables.

window: int, offset, BaseIndexer subclass, or list, default=3
Size of the moving window. If an integer, the fixed number of observations used for each window. If an offset (recommended), the time period of each window. It can also take a function. See parameter `windows` in pandas `rolling()` documentation for more details.

In addition to pandas normal input values, `window` can also take a list with the above specified values, in which case, features will be created for each one of the windows specified in the list.

min_periods: int, default None.
Minimum number of observations in the window required to have a value; otherwise, the result is np.nan. See parameter `min_periods` in pandas `rolling()` documentation for more details.

functions: string or list of strings, default = 'mean'
The functions to apply within the window. Valid functions can be found here.

periods: int, list of ints, default=1
Number of periods to shift. Can be a positive integer. See param `periods` in pandas `shift()`.

freq: str, list of str, default=None
Offset to use from the tseries module or time rule. See parameter `freq` in pandas `shift()`.

sort_index: bool, default=True
Whether to order the index of the dataframe before creating the features.

missing_values: string, default='raise'
Indicates if missing values should be ignored or raised. If `'raise'` the transformer will return an error if the the datasets to `fit` or `transform` contain missing values. If `'ignore'`, missing data will be ignored when learning parameters or performing the transformation.

drop_original: bool, default=False
If True, the original variables to transform will be dropped from the dataframe.

drop_na: bool, default=False.
Whether the NAN introduced in the lag features should be removed.

Attributes

variables_:
The group of variables that will be used to create the window features.

feature_names_in_:
List with the names of features seen during `fit`.

n_features_in_:
The number of features in the train set used in fit.

See also:

`pandas.rolling`
`pandas.aggregate`
`pandas.shift`

Examples

```
>>> import pandas as pd
>>> from feature_
↪engine.timeseries.forecasting import WindowFeatures
>>> X = pd.DataFrame(dict(date = ["2022-09-18",
>>>                               "2022-09-19",
>>>                               "2022-09-20",
>>>                               "2022-09-21",
>>>                               "2022-09-22"],
>>>                       x1 = [1,2,3,4,5],
>>>                       x2 = [6,7,8,9,10]
>>>                      ))
>>> wf = WindowFeatures(window = 2)
>>> wf.fit_transform(X)
         date  x1  x2  x1_window_2_mean  x2_window_2_mean
0  2022-09-18   1   6               NaN               NaN
1  2022-09-19   2   7               NaN               NaN
2  2022-09-20   3   8               1.5               6.5
3  2022-09-21   4   9               2.5               7.5
4  2022-09-22   5  10               3.5               8.5
```

Methods

fit:	This transformer does not learn parameters.
transform:	Add window features.
transform_x_y:	Remove rows with missing data from X and y.
fit_transform:	Fit to data, then transform it.
get_feature_names_out:	Get output feature names for transformation.
get_params:	Get parameters for this estimator.
set_params:	Set the parameters of this estimator.

fit(*X*, *y=None*)

This transformer does not learn parameters.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The training dataset.

y: pandas Series, default=None
y is not needed in this transformer. You can pass None or y.

`fit_transform`(*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to `X` and `y` with optional parameters `fit_params` and returns a transformed version of `X`.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_feature_names_out`(*input_features=None*)

Get output feature names for transformation. In other words, returns the variable names of transformed dataframe.

Parameters

input_features
[array or list, default=None] This parameter exits only for compatibility with the Scikit-learn pipeline.

- If `None`, then `feature_names_in_` is used as feature names in.
- If an array or list, then `input_features` must match `feature_names_in_`.

Returns

feature_names_out: list
Transformed feature names.

rtype
`List`[`Union`[`str`, `int`]] ..

`get_metadata_routing`()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Adds window features.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The data to transform.

Returns

X_new: Pandas dataframe, shape = [n_samples, n_features + window_features]
The dataframe with the original plus the new variables.

rtype
`DataFrame` ..

`transform_x_y`(*X*, *y*)

Transform, align and adjust both X and y based on the transformations applied

to X, ensuring that they correspond to the same set of rows if any were removed from X.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The dataframe to transform.

y: pandas Series or Dataframe of length = n_samples
The target variable to transform. Can be multi-output.

Returns

X_new: pandas dataframe
The transformed dataframe of shape [n_samples - n_rows, n_features]. It may contain less rows than the original dataset.

y_new: pandas Series or DataFrame
The transformed target variable of length [n_samples - n_rows]. It contains as many rows as those left in X_new.

ExpandingWindowFeatures

class feature_engine.timeseries.forecasting.**ExpandingWindowFeatures**(*variables=N*
min_periods
functions='n
periods=1, f
sort_index=
missing_valu
drop_origina
drop_na=Fa

ExpandingWindowFeatures adds new features to a dataframe based on expanding window operations. Expanding window operations are operations that perform an aggregation over an expanding window of all past values relative to the value of interest. An expanding window feature is, in other words, a feature created after computing statistics (e.g., mean, min, max, etc.) using a window over all the past data. For example, the mean value of all months prior to the month of interest is an expanding window feature.

ExpandingWindowFeatures uses the pandas' functions `expanding()`, `agg()` and `shift()`. With `expanding()`, it creates expanding windows. With `agg()` it applies multiple functions within those windows. With 'shift()' it allocates the values to the correct rows.

For supported aggregation functions, see Expanding Window Functions.

To be compatible with ExpandingWindowFeatures, the dataframe's index must have unique values and no NaN.

ExpandingWindowFeatures works only with numerical variables. You can pass a list of variables to use as input for the expanding window. Alternatively, ExpandingWindowFeatures will automatically select all numerical variables in the training set.

More details in the *User Guide*.

Parameters

variables: list, default=None
The list of numerical variables to transform. If None, the transformer will automatically find and select all numerical variables.

min_periods: int, default None.
Minimum number of observations in window required to have a value; otherwise, result is np.nan. See parameter `min_periods` in the pandas `expanding()` documentation for more details.

functions: str, list of str, default = 'mean'
The functions to apply within the window. Valid functions can be found here.

periods: int, list of ints, default=1
Number of periods to shift. Can be a positive integer. See param `periods` in pandas `shift`.

freq: str, list of str, default=None
Offset to use from the tseries module or time rule. See parameter `freq` in pandas `shift()`.

sort_index: bool, default=True
Whether to order the index of the dataframe before creating the expanding window feature.

missing_values: string, default='raise'
Indicates if missing values should be ignored or raised. If `'raise'` the transformer will return an error if the the datasets to `fit` or `transform` contain missing values. If `'ignore'`, missing data will be ignored when learning parameters or performing the transformation.

drop_original: bool, default=False
If True, the original variables to transform will be dropped from the dataframe.

drop_na: bool, default=False.
Whether the NAN introduced in the created features should be removed.

Attributes

variables_:
The group of variables that will be used to create the expanding window features.

feature_names_in_:
List with the names of features seen during `fit`.

n_features_in_:
The number of features in the train set used in fit.

See also:

`pandas.expanding`
`pandas.aggregate`
`pandas.shift`

Examples

```
>>> import pandas as pd
>>> from feature_engine.
↪timeseries.forecasting import ExpandingWindowFeatures
>>> X = pd.DataFrame(dict(date = ["2022-09-18",
>>>                               "2022-09-19",
>>>                               "2022-09-20",
>>>                               "2022-09-21",
>>>                               "2022-09-22"],
>>>                       x1 = [1,2,3,4,5],
>>>                       x2 = [6,7,8,9,10]
>>>                      ))
>>> ewf = ExpandingWindowFeatures()
>>> ewf.fit_transform(X)
  ␣
↪      date  x1  x2  x1_expanding_mean  x2_expanding_mean
0 ␣
↪2022-09-18   1   6                NaN                NaN
1 ␣
↪2022-09-19   2   7                1.0                6.0
2 ␣
↪2022-09-20   3   8                1.5                6.5
3 ␣
↪2022-09-21   4   9                2.0                7.0
4 ␣
↪2022-09-22   5  10                2.5                7.5
```

Methods

fit:	This transformer does not learn parameters.
transform:	Add expanding window features.
transform_x_y:	Remove rows with missing data from X and y.
fit_transform:	Fit to data, then transform it.
get_feature_names_out:	Get output feature names for transformation.
get_params:	Get parameters for this estimator.
set_params:	Set the parameters of this estimator.

fit(*X*, *y=None*)

This transformer does not learn parameters.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The training dataset.

y: pandas Series, default=None
y is not needed in this transformer. You can pass None or y.

fit_transform(*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to `X` and `y` with optional parameters `fit_params` and returns a transformed version of `X`.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_feature_names_out`(*input_features=None*)

Get output feature names for transformation. In other words, returns the variable names of transformed dataframe.

Parameters

input_features
[array or list, default=None] This parameter exits only for compatibility with the Scikit-learn pipeline.

- If `None`, then `feature_names_in_` is used as feature names in.
- If an array or list, then `input_features` must match `feature_names_in_`.

Returns

feature_names_out: list
Transformed feature names.

rtype
`List`[`Union`[`str`, `int`]] ..

`get_metadata_routing`()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Adds expanding window features.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The data to transform.

Returns

X_new: Pandas dataframe, shape = [n_samples, n_features + window_features]
The dataframe with the original plus the new variables.

rtype
`DataFrame` ..

`transform_x_y`(*X*, *y*)

Transform, align and adjust both X and y based on the transformations applied

to X, ensuring that they correspond to the same set of rows if any were removed from X.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The dataframe to transform.

y: pandas Series or Dataframe of length = n_samples
The target variable to transform. Can be multi-output.

Returns

X_new: pandas dataframe
The transformed dataframe of shape [n_samples - n_rows, n_features]. It may contain less rows than the original dataset.

y_new: pandas Series or DataFrame
The transformed target variable of length [n_samples - n_rows]. It contains as many rows as those left in X_new.

10.3.5 Other

Preprocessing

Feature-engine's preprocessing transformers apply general data pre-processing and transformation procedures.

MatchCategories

class feature_engine.preprocessing.**MatchCategories**(*variables=None*, *ignore_format=F*
missing_values='raise')

MatchCategories() ensures that categorical variables are encoded as pandas `'categorical'` dtype, instead of generic python `'object'` or other dtypes.

Under the hood, `'categorical'` dtype is a representation that maps each category to an integer, thus providing a more memory-efficient object structure than, e.g., 'str', and allowing faster grouping, mapping, and similar operations on the resulting object.

MatchCategories() remembers the encodings or levels that represent each category, and can thus can be used to ensure that the correct encoding gets applied when passing categorical data to modeling packages that support this dtype, or to prevent unseen categories from reaching a further transformer or estimator in a pipeline, for example.

More details in the *User Guide*.

Parameters

variables: list, default=None
The list of categorical variables that will be encoded. If None, the encoder will find and transform all variables of type object or categorical by default. You

can also make the transformer accept numerical variables, see the parameter `ignore_format`.

ignore_format: bool, default=False
This transformer operates only on variables of type object or categorical. To override this behaviour and allow the transformer to transform numerical variables as well, set to `True`.

If `ignore_format` is `False`, the encoder will automatically select variables of type object or categorical, or check that the variables entered by the user are of type object or categorical. If `True`, the encoder will select all variables or accept all variables entered by the user, including those cast as numeric.

In short, set to `True` when you want to encode numerical variables.

missing_values: string, default='raise'
Indicates if missing values should be ignored or raised. If `'raise'` the transformer will return an error if the the datasets to `fit` or `transform` contain missing values. If `'ignore'`, missing data will be ignored when learning parameters or performing the transformation.

Attributes

category_dict_:
Dictionary with the category encodings assigned to each variable.

variables_:
The group of variables that will be transformed.

feature_names_in_:
List with the names of features seen during `fit`.

n_features_in_:
The number of features in the train set used in fit.

Examples

```
>>> import pandas as pd
>>>␣
↪from feature_engine.preprocessing import MatchCategories
>>> X_train␣
↪= pd.DataFrame(dict(x1 = ["a","b","c"], x2 = [4,5,6]))
>>> X_test = pd.
↪DataFrame(dict(x1 = ["c","b","a","d"], x2 = [5,6,4,7]))
>>> mc = MatchCategories(missing_values="ignore")
>>> mc.fit(X_train)
>>> mc.transform(X_train)
  x1  x2
0  a   4
1  b   5
2  c   6
>>> mc.transform(X_test)
    x1  x2
0    c   5
1    b   6
```

(continues on next page)

(continued from previous page)

```
2    a   4
3  NaN   7
```

Methods

fit:	Learn the encodings or levels to use for each variable.
fit_transform:	Fit to the data. Then transform it.
get_feature_names_out:	Get output feature names for transformation.
get_params:	Get parameters for this estimator.
set_params:	Set the parameters of this estimator.
transform:	Enforce the type of categorical variables as dtype `categorical`.

`fit`(*X*, *y=None*)

Learn the encodings or levels to use for representing categorical variables.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The training dataset. Can be the entire dataframe, not just the variables to be transformed.

y: pandas Series, default = None
y is not needed in this encoder. You can pass y or None.

`fit_transform`(*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to `X` and `y` with optional parameters `fit_params` and returns a transformed version of `X`.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_feature_names_out`(*input_features=None*)

Get output feature names for transformation. In other words, returns the variable names of transformed dataframe.

Parameters

input_features
[array or list, default=None] This parameter exits only for compatibility with the Scikit-learn pipeline.

- If `None`, then `feature_names_in_` is used as feature names in.
- If an array or list, then `input_features` must match `feature_names_in_`.

Returns

feature_names_out: list
Transformed feature names.

rtype
`List`[`Union`[`str`, `int`]] ..

`get_metadata_routing`()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`inverse_transform`(*X*)

Convert the encoded variable back to the original values.

Parameters

X: pandas dataframe of shape = [n_samples, n_features].
The transformed dataframe.

Returns

X_tr: pandas dataframe of shape = [n_samples, n_features].
The un-transformed dataframe, with the categorical variables containing the original values.

rtype
DataFrame ..

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as Pipeline). The latter have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Encode categorical variables as pandas categorical dtype.

Parameters

X: pandas dataframe of shape = [n_samples, n_features].
The dataset to encode.

Returns

X_new: pandas dataframe of shape = [n_samples, n_features].
The dataframe with the variables encoded as pandas categorical dtype.

rtype
DataFrame ..

MatchVariables

`class feature_engine.preprocessing.MatchVariables`(*fill_value=nan*, *missing_values='ra*
match_dtypes=False, *verbose=True*

MatchVariables() ensures that the same variables observed in the train set are present in the test set. If the dataset to transform contains variables that were not present in the train set, they are dropped. If the dataset to transform lacks variables that were present in the train set, these variables are added to the dataframe with a value determined by the user (np.nan by default).

```
train = pd.DataFrame({
    "Name": ["tom", "nick", "krish", "jack"],
    "City
↪": ["London", "Manchester", "Liverpool", "Bristol"],
    "Age": [20, 21, 19, 18],
    "Marks": [0.9, 0.8, 0.7, 0.6],
})

test = pd.DataFrame({
    "Name": ["tom", "sam", "nick"],
    "Age": [20, 22, 23],
    "Marks": [0.9, 0.7, 0.6],
    "Hobbies": ["tennis", "rugby", "football"]
})

match_columns = MatchVariables()

match_columns.fit(train)

df_transformed = match_columns.transform(test)
```

Note that in the returned dataframe, the variable "Hobbies" was removed and the variable "City" was added with np.nan:

```
df_transformed

    Name    City  Age  Marks
0    tom  np.nan   20    0.9
1    sam  np.nan   22    0.7
2   nick  np.nan   23    0.6
```

The order of the variables in the transformed dataset is also adjusted to match that observed in the train set.

More details in the *User Guide*.

Parameters

fill_value: integer, float or string. Default=np.nan
The values for the variables that will be added to the transformed dataset.

missing_values: string, default='ignore'
Indicates if missing values should be ignored or raised. If 'raise' the transformer will return an error if the the datasets to `fit` or `transform` contain missing values. If 'ignore', missing data will be ignored when learning parameters or performing the transformation.

match_dtypes: bool, default=False
Indicates whether the dtypes observed in the train set should be applied to variables in the test set.

verbose: bool, default=True
If True, the transformer will print out the names of the variables that are added and / or removed from the dataset.

Attributes

feature_names_in_:
The variables present in the train set, in the order observed during fit.

n_features_in_:
The number of features in the train set used in fit.

dtype_dict_:
If `match_dtypes` is set to `True`, then this attribute will exist, and it will contain a dictionary of variables and their corresponding dtypes.

Examples

```
>>> import pandas as pd
>>>
↪ from feature_engine.preprocessing import MatchVariables
>>> X_train␣
↪= pd.DataFrame(dict(x1 = ["a","b","c"], x2 = [4,5,6]))
>>> X_test = pd.DataFrame(dict(x1 = ["c","b","a","d"],
>>>                            x2 = [5,6,4,7],
>>>                            x3 = [1,1,1,1]))
>>> mv = MatchVariables(missing_values="ignore")
>>> mv.fit(X_train)
>>> mv.transform(X_train)
x1  x2
0  a   4
1  b   5
2  c   6
>>> mv.transform(X_test)
The following␣
↪variables are dropped from the DataFrame: ['x3']
  x1  x2
0  c   5
1  b   6
2  a   4
3  d   7
```

```
>>> import pandas as pd
>>>
↪ from feature_engine.preprocessing import MatchVariables
>>> X_train = pd.DataFrame(dict(x1 = ["a","b","c"],
>>>␣
↪                            x2 = [4,5,6], x3 = [1,1,1]))
>>> X_test = pd.
↪DataFrame(dict(x1 = ["c","b","a","d"], x2 = [5,6,4,7]))
>>> mv = MatchVariables(missing_values="ignore")
>>> mv.fit(X_train)
>>> mv.transform(X_train)
  x1  x2  x3
0  a   4   1
1  b   5   1
```

(continues on next page)

(continued from previous page)

```
2  c   6   1
>>> mv.transform(X_test)
The following variables are added to the DataFrame: ['x3']
  x1  x2  x3
0  c   5 NaN
1  b   6 NaN
2  a   4 NaN
3  d   7 NaN
```

Methods

fit:	Identify the variable names in the train set.
fit_transform:	Fit to the data. Then transform it.
get_feature_names_out:	Get output feature names for transformation.
get_params:	Get parameters for this estimator.
set_params:	Set the parameters of this estimator.
transform:	Add or delete variables to match those observed in the train set.

`fit`(*X*, *y=None*)

Learns and stores the names of the variables in the training dataset.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The input dataframe.

y: None
y is not needed for this transformer. You can pass y or None.

`fit_transform`(*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to `X` and `y` with optional parameters `fit_params` and returns a transformed version of `X`.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_feature_names_out`(*input_features=None*)

Get output feature names for transformation. In other words, returns the variable names of transformed dataframe.

Parameters

input_features
[array or list, default=None] This parameter exits only for compatibility with the Scikit-learn pipeline.

- If `None`, then `feature_names_in_` is used as feature names in.
- If an array or list, then `input_features` must match `feature_names_in_`.

Returns

feature_names_out: list
Transformed feature names.

rtype
`List`[`Union`[`str`, `int`]] ..

`get_metadata_routing`()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A `MetadataRequest` encapsulating routing information.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form

<component>__<parameter> so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Drops variables that were not seen in the train set and adds variables that were in the train set but not in the data to transform. In other words, it returns a dataframe with matching columns.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The data to transform.

Returns

X_new: Pandas dataframe, shape = [n_samples, n_features]
The dataframe with variables that match those observed in the train set.

rtype
DataFrame ..

Scikit-learn Wrapper

Feature-engine's Scikit-learn wrappers wrap Scikit-learn transformers allowing their implementation only on a selected subset of features.

SklearnTransformerWrapper

class feature_engine.wrappers.**SklearnTransformerWrapper**(*transformer*, *variables=Non*

Wrapper to apply Scikit-learn transformers to a selected group of variables. It supports the following transformers:

- Binarizer and KBinsDiscretizer (only when encoding=Ordinal)
- FunctionTransformer, PowerTransformer and QuantileTransformer
- SimpleImputer, IterativeImputer and KNNImputer (only when add_indicators=False)
- OrdinalEncoder and OneHotEncoder (only when sparse is False)
- MaxAbsScaler, MinMaxScaler, StandardScaler, RobustScaler, Normalizer

- All selection transformers including VarianceThreshold
- PolynomialFeautures

More details in the *User Guide*.

Parameters

transformer: sklearn transformer
The desired Scikit-learn transformer.

variables: list, default=None
The list of variables to be transformed. If None, the wrapper will select all variables of type numeric for all transformers, except the SimpleImputer, OrdinalEncoder and OneHotEncoder, in which case, it will select all variables in the dataset.

Attributes

transformer_:
The fitted Scikit-learn transformer.

variables_:
The group of variables that will be transformed.

features_to_drop_:
The variables that will be dropped. Only present when using selection transformers

feature_names_in_:
List with the names of features seen during `fit`.

n_features_in_:
The number of features in the train set used in fit.

See also:

`sklearn.compose.ColumnTransformer`

Notes

This transformer offers similar functionality to the ColumnTransformer from Scikit-learn, but it allows entering the transformations directly into a Pipeline and returns pandas dataframes.

Examples

```
>>> import pandas as pd
>>> from␣
↪feature_engine.wrappers import SklearnTransformerWrapper
>>> from sklearn.preprocessing import StandardScaler
>>> X = pd.DataFrame(dict(x1␣
↪= ["a","b","c"], x2 = [1,2,3], x3 = [4,5,6]))
>>> skw = SklearnTransformerWrapper(StandardScaler())
```

(continues on next page)

(continued from previous page)

```
>>> skw.fit(X)
>>> skw.transform(X)
  x1        x2        x3
0  a -1.224745 -1.224745
1  b  0.000000  0.000000
2  c  1.224745  1.224745
```

```
>>> import pandas as pd
>>> from␣
↪feature_engine.wrappers import SklearnTransformerWrapper
>>> from sklearn.preprocessing import OneHotEncoder
>>> X = pd.DataFrame(dict(x1␣
↪= ["a","b","c"], x2 = [1,2,3], x3 = [4,5,6]))
>>> skw = SklearnTransformerWrapper(
>>>    ␣
↪ OneHotEncoder(sparse_output = False), variables = "x1")
>>> skw.fit(X)
>>> skw.transform(X)
   x2  x3  x1_a  x1_b  x1_c
0   1   4   1.0   0.0   0.0
1   2   5   0.0   1.0   0.0
2   3   6   0.0   0.0   1.0
```

```
>>> import pandas as pd
>>> from␣
↪feature_engine.wrappers import SklearnTransformerWrapper
>>> from sklearn.preprocessing import PolynomialFeatures
>>> X = pd.DataFrame(dict(x1␣
↪= ["a","b","c"], x2 = [1,2,3], x3 = [4,5,6]))
>>> skw␣
↪= SklearnTransformerWrapper(PolynomialFeatures(include_
↪bias = False))
>>> skw.fit(X)
>>> skw.transform(X)
  x1   x2   x3  x2^2  x2 x3  x3^2
0  a  1.0  4.0   1.0    4.0  16.0
1  b  2.0  5.0   4.0   10.0  25.0
2  c  3.0  6.0   9.0   18.0  36.0
```

Methods

fit:	Fit Scikit-learn transformer.
fit_transform:	Fit to data, then transform it.
get_feature_names_out:	Get output feature names for transformation.
get_params:	Get parameters for this estimator.
set_params:	Set the parameters of this estimator.
inverse_transform:	Convert the data back to the original representation.
transform:	Transform data with the Scikit-learn transformer.

fit(*X*, *y=None*)

Fits the Scikit-learn transformer to the selected variables.

Parameters

X: Pandas DataFrame
The dataset to fit the transformer.

y: pandas Series, default=None
The target variable.

`fit_transform`(*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to `X` and `y` with optional parameters `fit_params` and returns a transformed version of `X`.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_feature_names_out`(*input_features=None*)

Get output feature names for transformation.

input_features: list, default=None
If `None`, then the names of all the variables in the transformed dataset is returned. For those transformers that create and add new features to the dataset, like the OneHotEncoder or the PolynomialFeatures, you have the option to pass a list with the input features to obtain the newly created variables. For all other transformers, this parameter will be ignored.

Returns

feature_names_out: list
The feature names.

rtype
`List` ..

get_metadata_routing()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRequest] A MetadataRequest encapsulating routing information.

get_params(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

inverse_transform(*X*)

Convert the transformed variables back to the original values. Only implemented for the following Scikit-learn transformers:

PowerTransformer, QuantileTransformer, OrdinalEncoder, MaxAbsScaler, MinMaxScaler, StandardScaler, RobustScaler.

If you would like this method implemented for additional transformers, please check if they have the inverse_transform method in Scikit-learn and then raise an issue in our repo.

Parameters

X: pandas dataframe of shape = [n_samples, n_features].
The transformed dataframe.

Returns

X_tr: pandas dataframe of shape = [n_samples, n_features].
The dataframe with the original values.

rtype
DataFrame ..

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as Pipeline). The latter have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform(X)`

Apply the transformation to the dataframe. Only the selected variables will be modified.

If the Scikit-learn transformer is the OneHotEncoder or the PolynomialFeatures, the new features will be concatenated to the input dataset.

If the Scikit-learn transformer is for feature selection, the non-selected features will be dropped from the dataframe.

For all other transformers, the original variables will be replaced by the transformed ones.

Parameters

X: Pandas DataFrame
The data to transform.

Returns

X_new: Pandas DataFrame
The transformed dataset.

rtype
DataFrame ..

Other wrappers

The *SklearnTransformerWrapper()* offers a similar function to the ColumnTransformer class available in Scikit-learn. They differ in the implementation to select the variables.

10.3.6 Pipeline

Pipeline

Feature-engine's Pipeline is equivalent to Scikit-learn's pipeline, and in addition, it accepts the method `transform_x_y`, to adjust both X and y, in those cases where rows are removed from X.

Pipeline

class feature_engine.pipeline.**Pipeline**(*steps*, *, *memory=None*, *verbose=False*)

A sequence of data transformers with an optional final predictor.

`Pipeline` allows you to sequentially apply a list of transformers to preprocess the data and, if desired, conclude the sequence with a final `predictor` for predictive modeling.

Intermediate steps of the pipeline must be 'transforms', that is, they must implement `fit` and `transform` methods. The final `estimator` only needs to implement `fit`. The transformers in the pipeline can be cached using `memory` argument.

This pipeline allows intermediate transformers to remove rows from the dataset. It will automatically adjust the target variable to match the remaining observations.

The purpose of the pipeline is to assemble several steps that can be cross-validated together while setting different parameters. For this, it enables setting parameters of the various steps using their names and the parameter name separated by a `'__'`, as in the example below. A step's estimator may be replaced entirely by setting the parameter with its name to another estimator, or a transformer removed by setting it to `'passthrough'` or `None`.

More details in the *User Guide*.

Parameters

steps
[list of tuples] List of (name of step, estimator) tuples that are to be chained in sequential order. To be compatible with the scikit-learn API, all steps must define `fit`. All non-last steps must also define `transform`. See Combining Estimators for more details.

memory
[str or object with the joblib.Memory interface, default=None] Used to cache the fitted transformers of the pipeline. The last step will never be cached, even if it is a transformer. By default, no caching is performed. If a string is given, it is the path to the caching directory. Enabling caching triggers a clone of the transformers before fitting. Therefore, the transformer instance given to the pipeline cannot be inspected directly. Use the attribute `named_steps` or `steps` to inspect estimators within the pipeline. Caching the transformers is advantageous when fitting is time consuming.

verbose
[bool, default=False] If True, the time elapsed while fitting each step will be printed as it is completed.

Attributes

named_steps
[Bunch] Access the steps by name.

*classes_*
[ndarray of shape (n_classes,)] The classes labels.

*n_features_in_*
[int] Number of features seen during first step `fit` method.

*feature_names_in_*
[ndarray of shape (`n_features_in_`,)] Names of features seen during first step `fit` method.

Examples

```
>>> from sklearn.svm import SVC
>>> from sklearn.preprocessing import StandardScaler
>>> from sklearn.datasets import make_classification
>>> from sklearn.model_selection import train_test_split
>>> from feature_engine.pipeline import Pipeline
>>> X, y = make_classification(random_state=0)
>>> X_
↪train, X_test, y_train, y_test = train_test_split(X, y,
...                                                ␣
↪                                                  random_state=0)
>>> pipe =␣
↪Pipeline([('scaler', StandardScaler()), ('svc', SVC())])
>>> # The pipeline can be used as any other estimator
>>> # and avoids leaking the test set into the train set
>>> pipe.fit(X_train, y_train).score(X_test, y_test)
0.88
>>> # An estimator's parameter can be set using '__' syntax
>>> pipe.set_params(svc_
↪_C=10).fit(X_train, y_train).score(X_test, y_test)
0.76
```

property classes_

The classes labels. Only exist if the last step is a classifier.

decision_function(*X*, ***params*)

Transform the data, and apply `decision_function` with the final estimator.

Call `transform` of each transformer in the pipeline. The transformed data are finally passed to the final estimator that calls `decision_function` method. Only valid if the final estimator implements `decision_function`.

Parameters

X
[iterable] Data to predict on. Must fulfill input requirements of first step of the pipeline.

****params**
[dict of string -> object] Parameters requested and accepted by steps. Each step must have requested certain metadata for these parameters to be forwarded to them.

New in version 1.4: Only available if `enable_metadata_routing=True`. See Metadata Routing User Guide for more details.

Returns

y_score
[ndarray of shape (n_samples, n_classes)] Result of calling `decision_function` on the final estimator.

property feature_names_in_

Names of features seen during first step `fit` method.

fit(*X*, *y=None*, ***params*)

Fit the model.

Fit all the transformers one after the other and transform the data, then fit the transformed data using the final estimator.

Parameters

X
[iterable] Training data. Must fulfill input requirements of first step of the pipeline.

y
[iterable, default=None] Training targets. Must fulfill label requirements for all steps of the pipeline.

****params**
[dict of str -> object]

- If `enable_metadata_routing=False` (default):

  Parameters passed to the `fit` method of each step, where each parameter name is prefixed such that parameter `p` for step `s` has key `s__p`.

- If `enable_metadata_routing=True`:

  Parameters requested and accepted by steps. Each step must have requested certain metadata for these parameters to be forwarded to them.

  Changed in version 1.4: Parameters are now passed to the `transform` method of the intermediate steps as well, if requested, and if `enable_metadata_routing=True` is set via `set_config()`.

  See Metadata Routing User Guide for more details.

Returns

self
[Pipeline] This estimator.

`fit_predict`(*X*, *y=None*, ***params*)

Transform the data, and apply `fit_predict` with the final estimator.

Call `fit_transform` of each transformer in the pipeline. The transformed data are finally passed to the final estimator that calls `fit_predict` method. Only valid if the final estimator implements `fit_predict`.

Parameters

X
[iterable] Training data. Must fulfill input requirements of first step of the pipeline.

y
[iterable, default=None] Training targets. Must fulfill label requirements for all steps of the pipeline.

****params**
[dict of str -> object]

- If `enable_metadata_routing=False` (default):

  Parameters to the `predict` called at the end of all transformations in the pipeline.

- If `enable_metadata_routing=True`:

  Parameters requested and accepted by steps. Each step must have requested certain metadata for these parameters to be forwarded to them.

  New in version 0.20.

  Changed in version 1.4: Parameters are now passed to the `transform` method of the intermediate steps as well, if requested, and if `enable_metadata_routing=True`.

  See Metadata Routing User Guide for more details.

  Note that while this may be used to return uncertainties from some models with `return_std` or `return_cov`, uncertainties that are generated by the transformations in the pipeline are not propagated to the final estimator.

Returns

y_pred
[ndarray] Result of calling `fit_predict` on the final estimator.

`fit_transform`(*X*, *y=None*, ***params*)

Fit the model and transform with the final transformer.

Fit all the transformers one after the other and sequentially transform the data. Only valid if last step of the pipeline has method `transform`.

Parameters

X
[iterable] Training data. Must fulfill input requirements of first step of the pipeline.

y
[iterable, default=None] Training targets. Must fulfill label requirements for all steps of the pipeline.

****params**
[dict of str -> object]

- If `enable_metadata_routing=False` (default):

  Parameters passed to the `fit` method of each step, where each parameter name is prefixed such that parameter `p` for step `s` has key `s__p`.

- If `enable_metadata_routing=True`:

  Parameters requested and accepted by steps. Each step must have requested certain metadata for these parameters to be forwarded to them.

  Changed in version 1.4: Parameters are now passed to the `transform` method of the intermediate steps as well, if requested, and if `enable_metadata_routing=True`.

  See Metadata Routing User Guide for more details.

Returns

Xt
[array-like of shape (n_samples, n_transformed_features)] Transformed samples.

`get_feature_names_out`(*input_features=None*)

Get output feature names for transformation.

Transform input features using the pipeline.

Parameters

input_features
[array-like of str or None, default=None] Input features.

Returns

feature_names_out
[ndarray of str objects] Transformed feature names.

`get_metadata_routing`()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns

routing
[MetadataRouter] A `MetadataRouter` encapsulating routing information.

`get_params`(*deep=True*)

Get parameters for this estimator.

Returns the parameters given in the constructor as well as the estimators contained within the `steps` of the `Pipeline`.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[mapping of string to any] Parameter names mapped to their values.

`inverse_transform`(*Xt*, ***params*)

Apply `inverse_transform` for each step in a reverse order.

All estimators in the pipeline must support `inverse_transform`.

Parameters

Xt
[array-like of shape (n_samples, n_transformed_features)] Data samples, where `n_samples` is the number of samples and `n_features` is the number of features. Must fulfill input requirements of last step of pipeline's `inverse_transform` method.

****params**
[dict of str -> object] Parameters requested and accepted by steps. Each step must have requested certain metadata for these parameters to be forwarded to them.

New in version 1.4: Only available if `enable_metadata_routing=True`. See Metadata Routing User Guide for more details.

Returns

Xt
[ndarray of shape (n_samples, n_features)] Inverse transformed data, that is, data in the original feature space.

`property n_features_in_`

Number of features seen during first step `fit` method.

`property named_steps`

Access the steps by name.

Read-only attribute to access any step by given name. Keys are steps names and values are the steps objects.

`predict`(*X*, ***params*)

Transform the data, and apply `predict` with the final estimator.

Call `transform` of each transformer in the pipeline. The transformed data are finally passed to the final estimator that calls `predict` method. Only valid if the final estimator implements `predict`.

Parameters

X
[iterable] Data to predict on. Must fulfill input requirements of first step of the pipeline.

****params**
[dict of str -> object]

- If `enable_metadata_routing=False` (default):

  Parameters to the `predict` called at the end of all transformations in the pipeline.

- If `enable_metadata_routing=True`:

  Parameters requested and accepted by steps. Each step must have requested certain metadata for these parameters to be forwarded to them.

  New in version 0.20.

  Changed in version 1.4: Parameters are now passed to the `transform` method of the intermediate steps as well, if requested, and if `enable_metadata_routing=True` is set via `set_config()`.

  See Metadata Routing User Guide for more details.

  Note that while this may be used to return uncertainties from some models with `return_std` or `return_cov`, uncertainties that are generated by the transformations in the pipeline are not propagated to the final estimator.

Returns

y_pred
[ndarray] Result of calling `predict` on the final estimator.

`predict_log_proba`(*X*, ***params*)

Transform the data, and apply `predict_log_proba` with the final estimator.

Call `transform` of each transformer in the pipeline. The transformed data are finally passed to the final estimator that calls `predict_log_proba` method. Only valid if the final estimator implements `predict_log_proba`.

Parameters

X
[iterable] Data to predict on. Must fulfill input requirements of first step of the pipeline.

****params**
[dict of str -> object]

- If `enable_metadata_routing=False` (default):

  Parameters to the `predict_log_proba` called at the end of all transformations in the pipeline.

- If `enable_metadata_routing=True`:

  Parameters requested and accepted by steps. Each step must have requested certain metadata for these parameters to be forwarded to them.

  New in version 0.20.

  Changed in version 1.4: Parameters are now passed to the `transform` method of the intermediate steps as well, if requested, and if `enable_metadata_routing=True`.

  See Metadata Routing User Guide for more details.

Returns

y_log_proba
[ndarray of shape (n_samples, n_classes)] Result of calling `predict_log_proba` on the final estimator.

`predict_proba`(*X*, ***params*)

Transform the data, and apply `predict_proba` with the final estimator.

Call `transform` of each transformer in the pipeline. The transformed data are finally passed to the final estimator that calls `predict_proba` method. Only valid if the final estimator implements `predict_proba`.

Parameters

X
[iterable] Data to predict on. Must fulfill input requirements of first step of the pipeline.

****params**
[dict of str -> object]

- If `enable_metadata_routing=False` (default):

  Parameters to the `predict_proba` called at the end of all transformations in the pipeline.

- If `enable_metadata_routing=True`:

  Parameters requested and accepted by steps. Each step must have requested certain metadata for these parameters to be forwarded to them.

  New in version 0.20.

  Changed in version 1.4: Parameters are now passed to the `transform` method of the intermediate steps as well, if requested, and if `enable_metadata_routing=True`.

  See Metadata Routing User Guide for more details.

Returns

y_proba
[ndarray of shape (n_samples, n_classes)] Result of calling `predict_proba` on the final estimator.

`score`(*X*, *y=None*, *sample_weight=None*, ***params*)

Transform the data, and apply `score` with the final estimator.

Call `transform` of each transformer in the pipeline. The transformed data are finally passed to the final estimator that calls `score` method. Only valid if the final estimator implements `score`.

Parameters

X
[iterable] Data to predict on. Must fulfill input requirements of first step of the pipeline.

y
[iterable, default=None] Targets used for scoring. Must fulfill label requirements for all steps of the pipeline.

sample_weight
[array-like, default=None] If not None, this argument is passed as `sample_weight` keyword argument to the `score` method of the final estimator.

****params**
[dict of str -> object] Parameters requested and accepted by steps. Each step must have requested certain metadata for these parameters to be forwarded to them.

New in version 1.4: Only available if `enable_metadata_routing=True`. See Metadata Routing User Guide for more details.

Returns

score
[float] Result of calling `score` on the final estimator.

`score_samples`(*X*)

Transform the data, and apply `score_samples` with the final estimator.

Call `transform` of each transformer in the pipeline. The transformed data are finally passed to the final estimator that calls `score_samples` method. Only valid if the final estimator implements `score_samples`.

Parameters

X
[iterable] Data to predict on. Must fulfill input requirements of first step of the pipeline.

Returns

y_score
[ndarray of shape (n_samples,)] Result of calling `score_samples` on the final estimator.

`set_params`(***kwargs*)

Set the parameters of this estimator.

Valid parameter keys can be listed with `get_params()`. Note that you can directly set the parameters of the estimators contained in `steps`.

Parameters

****kwargs**
[dict] Parameters of this estimator or parameters of estimators contained in `steps`. Parameters of the steps may be set using its name and the parameter name separated by a '__'.

Returns

self
[object] Pipeline class instance.

`set_score_request`(*, *sample_weight: Union[bool, None, str] = '$UNCHANGED$'*) → *Pipeline*

Request metadata passed to the `score` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `score` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `score`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

Parameters

sample_weight
[str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED] Metadata routing for `sample_weight` parameter in `score`.

Returns

self
[object] The updated object.

`transform`(*X*, ***params*)

Transform the data, and apply `transform` with the final estimator.

Call `transform` of each transformer in the pipeline. The transformed data are finally passed to the final estimator that calls `transform` method. Only valid if the final estimator implements `transform`.

This also works where final estimator is `None` in which case all prior transformations are applied.

Parameters

X
[iterable] Data to transform. Must fulfill input requirements of first step of the pipeline.

****params**
[dict of str -> object] Parameters requested and accepted by steps. Each step must have requested certain metadata for these parameters to be forwarded to them.

New in version 1.4: Only available if `enable_metadata_routing=True`. See Metadata Routing User Guide for more details.

Returns

Xt
[ndarray of shape (n_samples, n_transformed_features)] Transformed data.

`transform_x_y`(*X*, *y*, ***params*)

Fit the model and transform with the final estimator.

Fit all the transformers one after the other and sequentially transform the data and the target. Only valid if the final estimator either implements `fit_transform` or `fit` and `transform`.

Parameters

X
[iterable] Training data. Must fulfill input requirements of first step of the pipeline.

y
[iterable] Training targets. Must fulfill label requirements for all steps of the pipeline.

****params**
[dict of str -> object]

- If `enable_metadata_routing=False` (default):

  Parameters passed to the `fit` method of each step, where each parameter name is prefixed such that parameter `p` for step `s` has key `s__p`.

- If `enable_metadata_routing=True`:

  Parameters requested and accepted by steps. Each step must have requested certain metadata for these parameters to be forwarded to them.

  Changed in version 1.4: Parameters are now passed to the `transform` method of the intermediate steps as well, if requested, and if `enable_metadata_routing=True`.

  See Metadata Routing User Guide for more details.

Returns

Xt
[ndarray of shape (n_samples - n_rows, n_transformed_features)] Transformed samples.

yt
[ndarray of length (n_samples - n_rows)] Transformed target.

make_pipeline

`feature_engine.pipeline.`**`make_pipeline`**`(`*steps*, *memory=None*, *verbose=False*`)`

Construct a Pipeline from the given estimators.

This is a shorthand for the `Pipeline` constructor; it does not require, and does not permit, naming the estimators. Instead, their names will be set to the lowercase of their types automatically.

More details in the *User Guide*.

Parameters

***steps**
[list of Estimator objects] List of the scikit-learn estimators that are chained together.

memory
[str or object with the joblib.Memory interface, default=None] Used to cache the fitted transformers of the pipeline. The last step will never be cached, even if it is a transformer. By default, no caching is performed. If a string is given, it is the path to the caching directory. Enabling caching triggers a clone of the transformers before fitting. Therefore, the transformer instance given to the pipeline cannot be inspected directly. Use the attribute `named_steps` or `steps` to inspect estimators within the pipeline. Caching the transformers is advantageous when fitting is time consuming.

verbose
[bool, default=False] If True, the time elapsed while fitting each step will be printed as it is completed.

Returns

p
[Pipeline] Returns a scikit-learn *Pipeline* object.

See also:

Pipeline
Class for creating a pipeline of transforms with a final estimator.

Examples

```
>>> from sklearn.naive_bayes import GaussianNB
>>> from sklearn.preprocessing import StandardScaler
>>> from feature_engine.pipeline import make_pipeline
>>>␣
 ↪make_pipeline(StandardScaler(), GaussianNB(priors=None))
Pipeline(steps=[('standardscaler', StandardScaler()),
                ('gaussiannb', GaussianNB())])
```

10.3.7 Datasets

Datasets

We are starting to build a library of functions that allow you and us to quickly load datasets to demonstrate and test the functionality of Feature-engine (and, why not, other Python libraries).

At the moment, we support the following functions:

load_titanic

feature_engine.datasets.**load_titanic**(*return_X_y_frame=False*, *predictors_only=False*, *handle_missing=False*, *cabin=None*)

The load_titanic() function returns the well-known titanic dataset.

Note that you need to have an internet connection for this function to work, as we are calling the dataset stored in openML which can be downloaded from here.

Parameters

return_X_y_frame: bool, default=False
If `True`, it returns a DataFrame (X) with the predictors and a Series (y) with the target variable. If `False`, it returns a single DataFrame with predictors and target.

predictors_only: bool, default=False
If `False`, it returns all the variables from the original Titanic Dataset. If `True`, it reurns only relevant predictors.

handle_missing: bool, default=False
If `False`, it returns the original dataset with missing values. If `True`, missing data is replaced with the string "Missing" in categorical variables and the mean in numerical variables.

cabin: str, default=None
If `None`, it returns the variable cabin as in the original data. If 'drop', it removes the variable from the data. If 'letter_only' it returns just the first letter of the cabin, without the number.

Examples

```
>>> from feature_engine.datasets import load_titanic
>>>
↪ data = load_titanic(predictors_only=True, cabin="drop")
>>> print(data.head())
   pclass  survived␣
↪     sex      age  sibsp  parch      fare embarked
0       1   ␣
↪     1  female  29.0000      0      0  211.3375        S
1       1   ␣
↪     1    male   0.9167      1      2  151.5500        S
2       1   ␣
↪     0  female   2.0000      1      2  151.5500        S
3       1   ␣
↪     0    male  30.0000      1      2  151.5500        S
4       1   ␣
↪     0  female  25.0000      1      2  151.5500        S
```

10.3.8 Tools

Variable handling functions

This set of functions find variables of a specific type in a dataframe, or check that a list of variables is of a specified data type.

The `find` functions take a dataframe as an argument and returns a list with the names of the variables of the desired type.

The `check` functions check that the list of variables are all of the desired data type.

The `retain` functions select the variables in a list if they fulfill a condition.

These functions are used under-the-hood by all Feature-engine transformers to select the variables that they will modify.

find_all_variables

feature_engine.variable_handling.**find_all_variables**(*X*, *exclude_datetime=False*)

Returns a list with the names of all the variables in the dataframe. It has the option to exlcude variables that can be parsed as datetime.

More details in the *User Guide*.

Parameters

X
[pandas dataframe of shape = [n_samples, n_features]] The dataset

exclude_datetime: bool, default=False
Whether to exclude datetime variables.

Returns

variables: List
The names of the variables.

Examples

```
>>> import pandas as pd
>>> from feature_
↪engine.variable_handling import find_all_variables
>>> X = pd.DataFrame({
>>>     "var_num": [1, 2, 3],
>>>     "var_cat": ["A", "B", "C"],
>>>     "var_
↪date": pd.date_range("2020-02-24", periods=3, freq="T")
>>> })
>>> vars_all = find_all_variables(X)
>>> vars_all
['var_num', 'var_cat', 'var_date']
```

Return type
List[Union[str, int]]

find_categorical_variables

feature_engine.variable_handling.**find_categorical_variables**(*X*)

Returns a list with the names of all the categorical variables in a dataframe. Note that variables cast as object that can be parsed to datetime will be excluded.

More details in the *User Guide*.

Parameters

X
[pandas dataframe of shape = [n_samples, n_features]] The dataset

Returns

variables: List
The names of the categorical variables.

Examples

```
>>> import pandas as pd
>>> from feature_engine.
↪variable_handling import find_categorical_variables
>>> X = pd.DataFrame({
>>>     "var_num": [1, 2, 3],
>>>     "var_cat": ["A", "B", "C"],
>>>     "var_
↪date": pd.date_range("2020-02-24", periods=3, freq="T")
>>> })
>>> var_ = find_categorical_variables(X)
>>> var_
['var_cat']
```

Return type
List[Union[str, int]]

find_datetime_variables

feature_engine.variable_handling.**find_datetime_variables**(*X*)

Returns a list with the names of the variables that are or can be parsed as datetime.

Note that this function will select variables cast as object if they can be cast as datetime as well.

More details in the *User Guide*.

Parameters

X
[pandas dataframe of shape = [n_samples, n_features]] The dataset.

Returns

variables: List
The names of the datetime variables.

Examples

```
>>> import pandas as pd
>>> from feature_
↪engine.variable_handling import find_datetime_variables
>>> X = pd.DataFrame({
>>>     "var_num": [1, 2, 3],
>>>     "var_cat": ["A", "B", "C"],
>>>     "var_
↪date": pd.date_range("2020-02-24", periods=3, freq="T")
>>> })
>>> var_date = find_datetime_variables(X)
>>> var_date
['var_date']
```

Return type
List[Union[str, int]]

find_numerical_variables

feature_engine.variable_handling.**find_numerical_variables**(*X*)

Returns a list with the names of all the numerical variables in a dataframe.

More details in the *User Guide*.

Parameters

X
[pandas dataframe of shape = [n_samples, n_features]] The dataset.

Returns

variables: List
The names of the numerical variables.

Examples

```
>>> import pandas as pd
>>> from feature_
↪engine.variable_handling import find_numerical_variables
>>> X = pd.DataFrame({
>>>     "var_num": [1, 2, 3],
>>>     "var_cat": ["A", "B", "C"],
>>>     "var_
↪date": pd.date_range("2020-02-24", periods=3, freq="T")
>>> })
>>> var_ = find_numerical_variables(X)
>>> var_
['var_num']
```

Return type
List[Union[str, int]]

find_categorical_and_numerical_variables

feature_engine.variable_handling.**find_categorical_and_numerical_variables**(*X*, *va*

Find numerical and categorical variables in a dataframe or from a list.

The function returns two lists; the first one with the names of the variables of type object or categorical and the second list with the names of the numerical variables.

More details in the *User Guide*.

Parameters

X
[pandas dataframe of shape = [n_samples, n_features]] The dataset

variables
[list, default=None] If `None`, the function will find all categorical and numerical variables in X. Alternatively, it will find categorical and numerical variables in X, selecting from the given list.

Returns

variables: tuple
Tupe containing a list with the categorical variables, and a List with the numerical variables.

Examples

```
>>> import pandas as pd
>>> from feature_engine.variable_handling import (
>>>   find_categorical_and_numerical_variables
>>>)
>>> X = pd.DataFrame({
>>>     "var_num": [1, 2, 3],
>>>     "var_cat": ["A", "B", "C"],
>>>     "var_
↪date": pd.date_range("2020-02-24", periods=3, freq="T")
>>> })
>>> var_cat,
↪ var_num = find_categorical_and_numerical_variables(X)
>>> var_cat, var_num
(['var_cat'], ['var_num'])
```

Return type
Tuple[List[Union[str, int]], List[Union[str, int]]]

check_all_variables

feature_engine.variable_handling.**check_all_variables**(*X*, *variables*)

Checks that the variables in the list are in the dataframe.

More details in the *User Guide*.

Parameters

X
[pandas dataframe of shape = [n_samples, n_features]] The dataset

variables
[list] The list with the names of the variables to check.

Returns

variables: List
The names of the variables.

Examples

```
>>> import pandas as pd
>>> from feature_
↪engine.variable_handling import check_all_variables
>>> X = pd.DataFrame({
>>>     "var_num": [1, 2, 3],
>>>     "var_cat": ["A", "B", "C"],
>>>     "var_
↪date": pd.date_range("2020-02-24", periods=3, freq="T")
>>> })
>>> vars_all = check_
↪all_variables(X, ['var_num', 'var_cat', 'var_date'])
>>> vars_all
['var_num', 'var_cat', 'var_date']
```

Return type
List[Union[str, int]]

check_categorical_variables

feature_engine.variable_handling.**check_categorical_variables**(*X*, *variables*)

Checks that the variables in the list are of type object or categorical.

More details in the *User Guide*.

Parameters

X
[pandas dataframe of shape = [n_samples, n_features]] The dataset

variables
[list] The list with the names of the variables to check.

Returns

variables: List
The names of the categorical variables.

Examples

```
>>> import pandas as pd
>>> from feature_engine.
↪variable_handling import check_categorical_variables
>>> X = pd.DataFrame({
>>>     "var_num": [1, 2, 3],
>>>     "var_cat": ["A", "B", "C"],
>>>     "var_
↪date": pd.date_range("2020-02-24", periods=3, freq="T")
>>> })
>>> var_ = check_categorical_variables(X, "var_cat")
>>> var_
['var_cat']
```

Return type
List[Union[str, int]]

check_datetime_variables

feature_engine.variable_handling.**check_datetime_variables**(*X*, *variables*)

Checks that the variables in the list are or can be parsed as datetime.

More details in the *User Guide*.

Parameters

X
[pandas dataframe of shape = [n_samples, n_features]] The dataset

variables
[list] The list with the names of the variables to check.

Returns

variables: List
The names of the datetime variables.

Examples

```
>>> import pandas as pd
>>> from feature_
↪engine.variable_handling import check_datetime_variables
>>> X = pd.DataFrame({
>>>     "var_num": [1, 2, 3],
>>>     "var_cat": ["A", "B", "C"],
>>>     "var_
↪date": pd.date_range("2020-02-24", periods=3, freq="T")
>>> })
>>> var_date = check_datetime_variables(X, "var_date")
>>> var_date
['var_date']
```

Return type
List[Union[str, int]]

check_numerical_variables

feature_engine.variable_handling.**check_numerical_variables**(*X*, *variables*)

Checks that the variables in the list are of type numerical.

More details in the *User Guide*.

Parameters

X
[pandas dataframe of shape = [n_samples, n_features]] The dataset.

variables
[List] The list with the names of the variables to check.

Returns

variables: List
The names of the numerical variables.

Examples

```
>>> import pandas as pd
>>> from feature_engine.
↪variable_handling import check_numerical_variables
>>> X = pd.DataFrame({
>>>     "var_num": [1, 2, 3],
>>>     "var_cat": ["A", "B", "C"],
>>>     "var_
↪date": pd.date_range("2020-02-24", periods=3, freq="T")
>>> })
>>> var_
```

(continues on next page)

(continued from previous page)

```
↪ = check_numerical_variables(X, variables=["var_num"])
>>> var_
['var_num']
```

Return type
List[Union[str, int]]

retain_variables_if_in_df

feature_engine.variable_handling.**retain_variables_if_in_df**(*X*, *variables*)

Returns the subset of variables in the list that are present in the dataframe.

More details in the *User Guide*.

Parameters

X: pandas dataframe of shape = [n_samples, n_features]
The dataset.

variables: string, int or list of strings or int.
The names of the variables to check.

Returns

variables_in_df: List.
The subset of `variables` that is present `X`.

Examples

```
>>> import pandas as pd
>>> from feature_engine.
↪variable_handling import retain_variables_if_in_df
>>> X = pd.DataFrame({
>>>     "var_num": [1, 2, 3],
>>>     "var_cat": ["A", "B", "C"],
>>>     "var_
↪date": pd.date_range("2020-02-24", periods=3, freq="T")
>>> })
>>> vars_in_df = retain_variables_
↪if_in_df(X, ['var_num', 'var_cat', 'var_other'])
>>> vars_in_df
['var_num', 'var_cat']
```

10.4 Resources

Here you find learning resources to know more about Feature-engine and feature engineering and selection in general.

We have gathered online courses, books, blogs, videos, podcasts, jupyter notebook and kaggle kernels, so you can follow the resource with the way of learning that you like the most.

10.4.1 Courses

You can learn more about how to use Feature-engine and, feature engineering and feature selection in general in the following online courses:

Fig. 120: Feature Engineering for Machine Learning

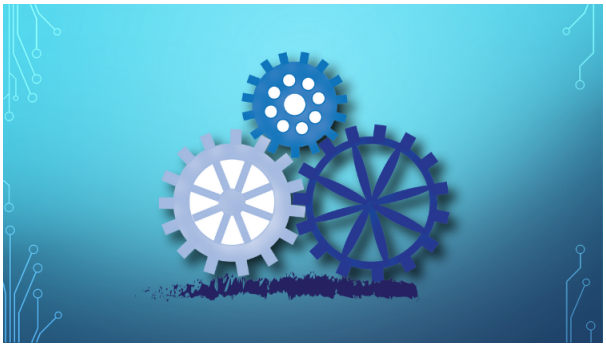


Fig. 121: Feature Selection for Machine Learning

Fig. 122: Forecasting with Machine Learning

Fig. 123: Feature Engineering for Time Series Forecasting

Fig. 124: Interpreting Machine Learning Models

10.4.2 Books

You can learn more about how to use Feature-engine and feature engineering in general in the following books:

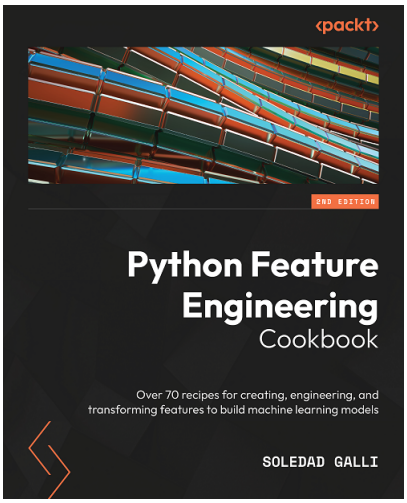


Fig. 126: Python Feature Engineering Cookbook

10.4.3 Blogs, Videos and More

Here you find articles and videos about Feature-engine and feature engineering and selection in general.

Blogs

Feature engineering

- Feature-engine: A new open-source Python package for feature engineering.
- Practical Code Implementations of Feature Engineering for Machine Learning with Python.
- Streamlining Feature Engineering Pipelines with Feature-engine.
- Feature Engineering for Machine Learning: A comprehensive Overview.
- Variance Stabilizing Transformations.

Feature selection

- Feature selection in machine learning with Python.
- Recursive feature elimination with Python.
- Population Stability Index and feature selection in Python
- Feature Selection for Machine Learning: A comprehensive Overview.

Videos

- Streamlining Feature Engineering Pipes with Feature-engine, Data Science Festival, 2020.
- Optimising Feature Engineering Pipelines with Feature-engine, Pydata Cambridge 2020, from minute 51:43.
- Feature selection in machine learning - DataTalks.Club, DataTalks.Club 2022.
- Feature engineering for time series forecasting DataTalks.Club 2022.

Podcasts

- Build Composable And Reusable Feature Engineering Pipelines with Feature-Engine.

En Español

Empezamos una nueva website con blogs en español. Chequeala:

- Entrena en Datos.

Ademas, te pueden interesar:

- Ingeniería de variables, MachinLenin, charla con video online.

More resources will be added as they appear online. If you know of a good resource, let us know.

10.4.4 Tutorials

How To

Check our jupyter notebooks showcasing the functionality of each Feature-engine transformer.

Kaggle Kernels

We also prepared Kaggle kernels with demos mixing data exploration, feature engineering, feature creation, feature selection and hyperparameter optimization of entire pipelines.

- Feature selection for bank customer satisfaction prediction
- Feature engineering and selection for house price prediction
- Feature creation for wine quality prediction
- Feature engineering and model stacking for house price modelling
- Feature engineering with Feature-engine and Randomized search
- Feature engineering with Feature-engine and Grid search

Video tutorials

You can find some videos on how to use Feature-engine in the Feature-engine playlist in Train in Data's YouTube channel. The list is a bit short at the moment, apologies.

10.5 Contribute

Feature-engine is an open source project, originally designed to support the online course Feature Engineering for Machine Learning, but has now gained popularity and supports transformations beyond those taught in the course.

Feature-engine is currently supported by a growing community and we will be delighted to accept contributions, large or small, that you wish to make to the project.

Contributing to open-source is a great way to learn and improve coding skills, and also a fun thing to do. If you've never contributed to an open source project, we hope to make it easy for you with the following guidelines.

Read more about "Why Contribute to Open-Source".

10.5.1 Ways to contribute

There are many ways to contribute to Feature-engine:

- Create a new transformer
- Enhance functionality of current transformers
- Fix a bug
- If you find a bug, let us know by creating an issue on Github.
- If you would like additional functionality or a new feature, create an issue on Github.
- Add a Jupyter notebook to our Jupyter notebooks example gallery.
- Improve our documentation, i.e., fix typos, improve grammar, or add more code examples.
- Write a blog, tweet, or share our project with others.
- Use Feature-engine in your lectures if you teach.
- Sponsor us.

With plenty of ways to get involved, we would be happy for you to support the project. You only need to abide by the principles of openness, respect, and consideration of others, as described in the Python Software Foundation Code of Conduct and you are ready to go!.

Read more about "Ways to Contribute to Open Source".

10.5.2 Getting in touch

We prefer to handle most contributions through the github repository. You can also join our Gitter community.

1. Open issues.
2. Gitter community.

10.5.3 Contributing Guide

Contribute Code

Contributing code to Feature-engine is fun and easy. If you want to make a code contribution, you can check the issue tracker for already requested and wanted functionality. Alternatively, you can create a new issue with functionality you would like to see included in Feature-engine and then work it through.

Contributing workflow

A typical contributing workflow goes like this:

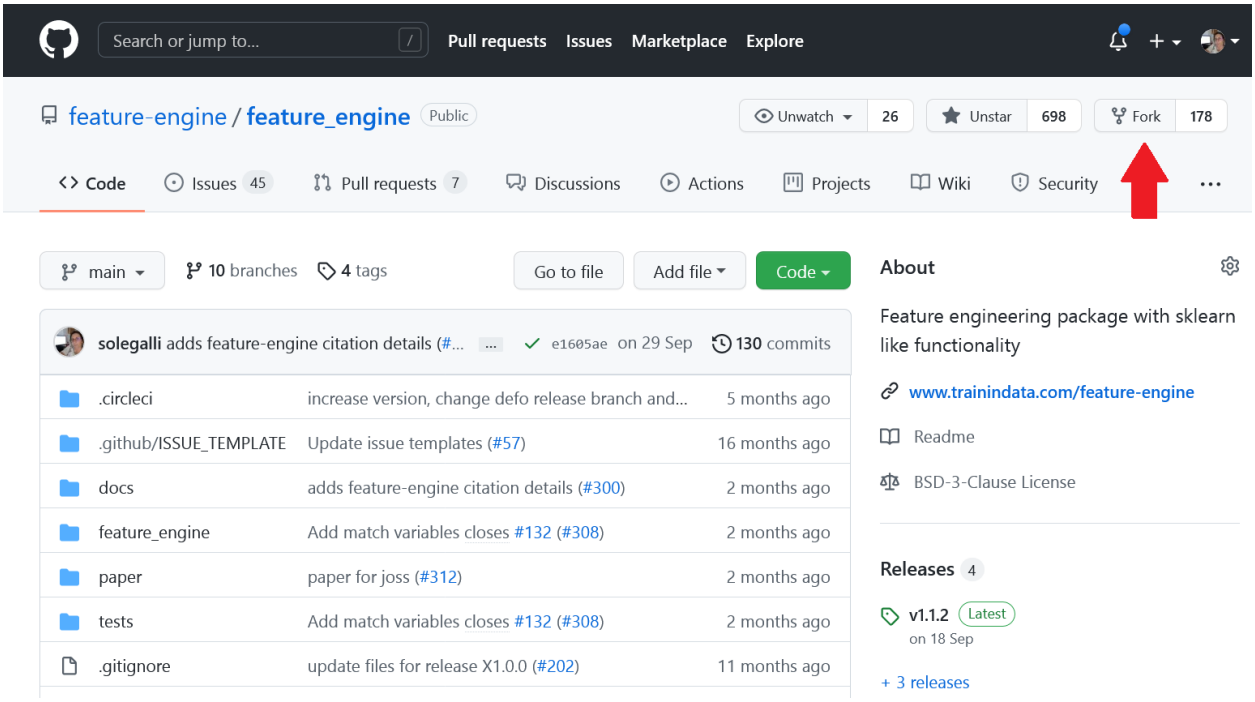
1. **Suggest** new functionality or **pick up** an issue from the issue tracker.
2. **Mention** in the issue that you are "working on it".
3. **Fork** the repository into your GitHub account.
4. **Clone** your fork into your local computer.
5. Set up the **development environment**.
6. **Create** a new branch with the name of your feature
7. **Code** the feature, the tests and update or add the documentation.
8. **Commit** the changes to your fork.
9. Make a **Pull Request (PR)** with your changes from your fork to the main repo.
10. **Test** the code.
11. **Review** the code with us.
12. Make the **changes** and commit them to your fork, using the same branch created in 5.
13. When it is ready, we will **merge** your contribution into Feature-engine's source code base.

    To avoid extra work or duplication, it is important that we communicate right from the beginning, so we can build together a clear understanding of how you would like to get involved and what is needed to complete the task. This is particularly important for big code additions.

    In the rest of the document, we will describe steps 3 to 13 in more detail.

Fork the Repository

When you fork the repository, you create a copy of Feature-engine's source code into your Github account, which you can edit. To fork Feature-engine's repository, click the **fork** button in the upper right corner of Feature-engine's Repository.



Clone the Repository

Once you forked the repository, clone the fork to your local machine.

1. Clone your fork into your local machine:

```
$ git␣
↪clone https://github.com/<YOURUSERNAME>/feature_engine
```

2. Set up an `upstream` remote from where you can pull the latest code changes occurring in the main Feature-engine repository:

```
$ git remote add upstream␣
↪https://github.com/feature-engine/feature_engine.git
```

3. Check that the remote was set correctly:

```
$ git remote -v
```

You should see both your fork (origin) and the main repository (upstream) linked to your local copy:

```
origin    https:/
↪/github.com/YOUR_USERNAME/feature_engine.git (fetch)
origin    https:/
↪/github.com/YOUR_USERNAMEfeature_engine.git (push)
upstream  https:/
↪/github.com/feature-engine/feature_engine.git (fetch)
upstream  https:/
↪/github.com/feature-engine/feature_engine.git (push)
```

Keep in mind that Feature-engine is being actively developed, so you may need to update your fork regularly. See below how to **Keep your fork up to date**.

Set up the Development Environment

After creating a local copy of the repo, you need to set up the development environment. Setting up a development environment will ensure that you have all the libraries you need for the development, no more and no less. These libraries include Feature-engine dependencies, like Pandas, NumPy and Scikit-learn and also software development libraries like pytest, mypy, flake8, isort and black.

It is optional but highly advisable that you create a virtual environment. A virtual environment is a "separate space", where you can install Feature-engine's dependencies. To create a virtual environment, use any virtual environment tool of your choice. Some examples include:

1. venv
2. conda environments

In the previous links, you find details on how to create the environments. We provide some guidelines below.

venv

If you use venv, from the windows cmd or Mac terminal, create and activate the environment like this:

```
python -m venv /path/to/new/virtual/environment
```

For example I would do:

```
python -m venv Documents/Repositories/envs/featureengine
```

where "featureengine" will be the name of the environment and "Documents/Repositories/envs" the location where the environment will be created.

Then, to activate the environment, run:

```
Documents/Repositories/envs/featureengine/Scripts/activate
```

Note for windows users: you may need to use \ instead of /.

conda

If you are using anaconda, from your conda prompt, create and activate the environment like this:

```
conda create --name myenv
```

where "myenv" will be the name of the environment, so you probably want to change that to something more meaningful.

Then, to activate the environment, run:

```
conda activate myenv
```

Install dependencies

Now, you are ready to install all dependencies, that is, all the Python libraries used by Feature-engine. First, navigate to your clone of Feature-engine:

```
$ cd feature_engine
```

Now, install Feature_engine in developer mode:

```
$ pip install -e .
```

Don't forget the `.` after the `-e`. This will add Feature-engine to your PYTHONPATH so your code edits are automatically picked up, and there is no need to re-install the package after each code change. This will also install Feature'engine's dependencies.

Finally, install the additional dependencies for tests and documentation:

```
$ pip install -r test_requirements.txt
$ pip install -r docs/requirements.txt
```

Make sure that your local main branch is up to date with the remote main branch:

```
$ git pull --rebase upstream main
```

If you just cloned your fork, your local main branch should be up to date. If you cloned your fork a time ago, probably the main repository had some code changes. To sync your fork main branch to the main repository, read below the section **Keep your fork up to date**.

Create a branch

It is important to create a new branch, different from main, where you will code your changes. It is advisable, almost never to work on the main branch.

Create a new branch where you will develop your feature:

```
$ git checkout -b myfeaturebranch
```

where "myfeaturebranch" is the name you choose for your branch.

There are 3 things to keep in mind when creating a feature branch:

1. Give the branch a name that identifies the feature you are going to build.
2. Make sure you checked out your branch from the main branch.
3. Make sure your local main branch was updated with the upstream main branch.

Code your feature

Now, you are ready to make your code changes. When you develop a new feature, fix a bug, or make any code contribution, there are a few things to consider:

1. Make regular code commits to your branch, locally.
2. Give clear messages to your commits, indicating which changes were made at each commit (use present tense).
3. Try and push regularly to your fork, so that you don't lose your changes.

Commit

Make small changes and commit immediately. This way it is easier to track what was changed. To commit changes do the following:

```
$ git add .
$ git commit -m "my commit message"
```

and make sure to include an informative but succinct commit message in the present tense, for example "fix style in imputation error message".

The previous commands will commit all files that have changes. If you want to commit just 1 or 2 files, you can do so as follows:

```
$ git add file1.py file2.py
$ git commit -m "my commit message"
```

It is important that you commit only the files relevant to your feature, and not others that may have been accidentally changed, for example through code styling (more on this in **Test the Code** below).

After making a few commits, push your changes to your fork:

```
$ git push origin myfeaturebranch
```

This will automatically create a branch in your remote fork called "myfeature-branch" containing all your changes.

Make a Pull Request

After pushing the first changes, go to your fork in Github. You will see the branch you just pushed and next to it a button to create a PR (Pull Request). Go ahead and create a PR from your feature branch to Feature_engine's **main branch**. In the PR message, describe what the overall aim of the PR is, and if it resolves an issue, link the issue in the message. This will notify us of your changes.

Don't worry, you can continue making changes and committing more code to the branch. You basically need to repeat these steps as often as you need:

```
$ git add .
$ git commit -m "my commit message"
$ git push origin myfeaturebranch
```

Once you think your code is ready to review, leave a message in the PR saying "please review" or something similar.

Create Docstrings

If you are coding an entire new class, make sure you follow our *guidelines to create the docstrings*.

Test the Code

The code you submit must pass any test you add plus all current tests in the library. The tests are triggered automatically when you first make a PR, and then any time you commit new changes to the PR. It is important that the tests pass when you ask us for review.

We have tests for:

1. Functionality, using pytest
2. Code style, using flake8
3. Typehints, using mypy
4. Documentation, using sphinx.
5. Coverage using coverage

In the following paragraphs, we will take you through how to test each of the above.

Test functionality

We use pytest to create and run our tests. If you set up the development environment as we described previously, you should have pytest installed. Alternatively, run from the windows cmd or mac terminal:

```
$ pip install pytest
```

You can now run the tests from your command line interface. Make sure you are within the feature-engine folder. Then run:

```
$ pytest
```

These command will run all the test scripts within the test folder. It will take a few minutes.

Alternatively, you can run a specific script as follows:

```
$ pytest tests/test_encoding/test_onehot_encoder.py
```

The previous command will just run the tests for the one hot encoder.

It will be faster if you just test the code you created, in which case you would do:

```
$ pytest␣
↪tests/test_my_new_feature_folder/test_my_new_feature.py
```

where test_my_new_feature.py is the name of your test script, and it is located in the test_my_new_feature_folder.

If you are using Pycharm, this is even easier:

1. In your project directory (where you have all the files and scripts), right click with the mouse on the folder "tests".
2. Select "Run pytest in tests".

   This will run all tests.

   To run your specific tests:

1. Locate your test file
2. Right click with the mouse on the test file.
3. Select "Run pytest in tests".

   Sweet, isn't it?

   With the above procedure you can also "click" on your individual test script and run only those tests.

Code coverage

We use coverage to test the extend of coverage of our tests. To evaluate the code coverage, you need to run pytest with coverage:

```
$ coverage run -m pytest
```

And then, you can visualize the report like this:

```
$ coverage report
```

If you see that there is coverage missing in any of the classes you are working with, try to add tests to increase coverage. We aim for 97%.

Test Code Style

We follow PEP8 and we keep our code lines up to 88 characters. Before testing the code style, make sure to automatically fix anything that might not abide by PEP8 with **black** and **isort**.

If you set up the development environment as we described previously, you should have these libraries installed. Alternatively, run from the windows cmd or mac terminal:

```
$ pip install black
$ pip install isort
```

Then, you can sort the imports alphabetically by running:

```
$ isort my_new_script.py
```

You can fix code style by running:

```
$ black my_new_script.py
```

You need to run isort and black on both code files and test files.

Black and isort may make changes to your file. Don't forget to commit those changes:

```
$ git add my_new_script.py
$ git commit -m "fix code style"
$ git push origin my_feature_branch
```

Now, you can go ahead and test that your scripts pass the code styling tests. To do so, execute from the command line:

```
$ flake8 my_new_script.py
```

If the flake8 test pass, you are good to go. Alternatively, you will get an error, indicating which line of code is not following the coding convention.

Test Typehint

We use Typehint. To test typehinting we use **mypy**.

If you set up the development environment as we described previously, you should have mypy installed. Alternatively, run from the windows cmd or mac terminal:

```
$ pip install mypy
```

now, you test typehint by running:

```
$ mypy feature_engine
```

A few things to notice:

- We use typehint only on the code base and not on the tests.
- You need to run mypy on the entire module and not just your script.

  Otherwise, you will most likely get an error.

Test the docs

If after running pytest, black and mypy you do not get errors, you are only left with testing that the documentation builds correctly.

To do this, first make sure you have all the documentation dependencies installed. If you set up the environment as we described previously, they should be installed. Alternatively, from the windows cmd or mac terminal, run:

```
$ pip install -r docs/requirements.txt
```

Make sure you are within the feature_engine module when you run the previous command.

Now, you can go ahead and build the documentation:

```
$ sphinx-build -b html docs build
```

This will trigger the building of the docs, which will be stored in html format in the "build" folder within the repository. You can open those up with your browser. But the important thing is that you do not get any red warning during the build process.

Using tox

In Feature-engine, we use tox to run all our tests automatically. If you want to run all the tests using tox locally:

1. Install tox in your development environment:

```
$ pip install tox
```

2. Make sure you are in the repository folder, alternatively:

```
$ cd feature_engine
```

3. Run the tests in tox:

```
$ tox
```

Just writing `tox`, will trigger automatically the functionality tests, code styling tests, typehint tests and documentation test. These will test the entire Feature-engine ecosystem and not just your new scripts, so it will be more time consuming.

If the tests pass, the code is in optimal condition :)

A few things to note:

Tox runs our tests in Python versions 3.6, 3.7, 3.8 and 3.9. However, it will only be able to run the tests in the version you have installed locally. All others will fail. This is OK. As long as the tests in the Python version you have installed pass, you are good to go.

Tox may modify some local files that are not relevant to your feature. Please **DO NOT** add those files to your PR.

If you want to know more about tox check this link. If you want to know why we prefer tox, this article will tell you everything ;)

Review Process

Once your contribution contains the new code, the tests and the documentation, you can request a review by mentioning that in a comment in the Pull Request. Likely, there will be some back and forth until the final submission. We will work together to get the code in the final shape.

If you feel you would like to clarify something before the first draft is done, or if you can't get some tests to pass, do not hesitate to mention that in a comment, and we will try to help.

We aim to review PRs within a week. If for some reason we can't, we will let you know through the PR as well.

Once the submission is reviewed and provided the continuous integration tests have passed and the code is up to date with Feature-engine's main branch, we will be ready to "Squash and Merge" your contribution into the `main` branch of Feature-engine. "Squash and Merge" combines all of your commits into a single commit which helps keep the history of the repository clean and tidy.

Once your contribution has been merged into main, you will be listed as a Feature-engine contributor :)

Merge Pull Requests

Only Core contributors have write access to the repository, can review and merge pull requests. Some preferences for commit messages when merging in pull requests:

- Make sure to use the "Squash and Merge" option in order to create a Git history that is understandable.
- Keep the title of the commit short and descriptive; be sure it links all related issues.

Releases

After a few features have been added to the main branch by yourself and other contributors, we will merge main into a release branch, e.g. 1.2.X, to release a new version of Feature-engine to PyPI and conda-forge.

Keep your Fork up to Date

When you're collaborating using forks, it's important to update your fork to capture changes that have been made by other collaborators.

If your feature takes a few weeks or months to develop, it may happen that new code changes are made to Feature_engine's main branch by other contributors. Some of the files that are changed maybe the same files you are working on. Thus, it is really important that you pull and rebase the upstream main branch into your feature branch. To keep your branches up to date:

1. Check out your local main branch:

```
$ git checkout main
```

If your feature branch has uncommitted changes, it will ask you to commit or stage those first. Refer to the commit guidelines we described above.

2. Pull and rebase the upstream main branch on your local main branch:

```
$ git pull --rebase upstream main
```

Your main should be a copy of the upstream main after this. If was is not, there may appear some conflicting files. You will need to resolve these conflicts and continue the rebase.

3. Pull the changes to your fork:

```
$ git push -f origin main
```

The previous command will update your fork (remote) so that your fork's main branch is in sync with Feature-engine's main. Now, you need to rebase main onto your feature branch.

4. Check out your feature branch:

```
$ git checkout myfeaturebranch
```

5. Rebase main onto it:

```
$ git rebase main
```

Again, if conflicts arise, try and resolve them and continue the rebase.

Now you are good to go to continue developing your feature.

Contribute Docs

If you contribute a new transformer, or enhance the functionality of a current transformer, most likely, you would have to add or update the documentation as well.

This is Feature-engine's documentation ecosystem:

- Feature-engine documentation is built using Sphinx and is hosted on Read the Docs.
- We use the pydata sphinx theme.
- We follow PEP 257 for doscstring conventions and use numpydoc docstring style.
- All documentation files are located within the docs folder in the repository.

To learn more about Sphinx check the Sphinx Quickstart documentation.

Documents organisation

Feature-engine has just adopted Scikit-learn's documentation style, were we offer API documentation, as well as, a User Guide with examples on how to use the different transformers.

The API documentation is built directly from the docstrings from each transformer. If you are adding a new transformer, you need to reference it in a new rst file placed within the api_doc folder.

If you would like to add additional examples, you need to update the rst files located in the user_guide folder.

Docstrings

The quickest way to get started with writing the transformer docstrings, is too look at the docstrings of some of the classes we already have in Feature-engine. Then simply copy and paste those docstrings and edit the bits that you need. If you copy and paste, make sure to delete irrelevant parameters and methods.

Link a new transformer

If you coded a new transformer from scratch, you need to update the following files to make sure users can find information on how to use the class correctly:

Add the name of your transformer in these files:

- Readme.
- main index.
- api index.
- user guide index.

Add an rst file with the name of your transformer in these folders:

- api_doc folder.
- user_guide folder.

That's it!

Expand the User Guide

You can add more examples or more details to our current User Guide examples. First, find the relevant rst file for the transformer you would like to work with. Feel free to add more details on the description of the method, expand the code showcasing other parameters or whatever you see fit.

We normally run the code on jupyter notebooks, and then copy and paste the code and the output in the rst files.

Build the documentation

To build the documentation, make sure you have properly installed Sphinx and the required dependencies. If you set up the development environment as we described in the *contribute code guide*, you should have those installed already.

Alternatively, first activate your environment. Then navigate to the root folder of Feature-engine. And now install the requirements for the documentation:

```
$ pip install -r docs/requirements.txt
```

To build the documentation (and test if it is working properly) run:

```
$ sphinx-build -b html docs build
```

This command tells sphinx that the documentation files are within the "docs" folder, and the html files should be placed in the "build" folder.

If everything worked fine, you can open the html files located in build using your browser. Alternatively, you need to troubleshoot through the error messages returned by sphinx.

Good luck and get in touch if stuck!

Contribute Jupyter notebooks

We created a collection of Jupyter notebooks that showcase the main functionality of Feature-engine's transformers. We link these notebooks throughout the main documentation to offer users more examples and details about transformers and how to use them.

Note that the Jupyter notebooks are hosted in a separate Github repository.

Here are some guidelines on how to add a new notebook or update an existing one. The contribution workflow is the same we use for the main source code base.

Jupyter contribution workflow

1. Fork the Github repository.
2. Clone your fork into your local computer: `git clone https://github.com/<YOURUSERNAME>/feature-engine-examples.git`.
3. Navigate into the project directory: `cd feature-engine-examples`.
4. If you haven't done so yet, install feature-engine: `pip install feature_engine`.
5. Create a feature branch with a meaningful name: `git checkout -b mynotebookbranch`.
6. Develop your notebook
7. Add the changes to your copy of the fork: `git add .`, `git commit -m "a meaningful commit message"`, `git pull origin mynotebookbranch`.
8. Go to your fork on Github and make a PR to this repo
9. Done

   The review process for notebooks is usually much faster than for the main source code base.

Jupyter creation guidelines

If you want to add a new Jupyter notebook, there are a few things to note:

- Make sure that the dataset you use is publicly available and with a clear license that it is free to use
- Do not upload datasets to the repository
- Add instructions on how to obtain and prepare the data for the demo
- Throughout the notebook, add guidelines on what you are going to do next, and what is the conclusion of the output

That's it! Fairly straightforward.

We look forward to your contribution :)

Other ways to contribute

A common misconception about contributing to open source is that you need to contribute code. Equally important to the code contributions are contributions to the documentation or to the gallery of examples on how to use the project, which we already discussed.

However, a good project does no-one any good if people don't know about it. So spreading the word about Feature-engine is extremely important. You will do the project a big favor if you help us spread the word about Feature-engine.

Here are some examples of how you could do that:

Spread the word

1. Star Feature-engine's Repository.
2. Share Feature-engine or any of our *blogs and videos* on social media.
3. Write a blog about Feature-engine.
4. Give a talk about Feature-engine or mention it in one of your talks.
5. If you teach, use Feature-engine in your lectures.
6. Share Feature-engine with your colleagues.

   If you write a blog or give a talk that is publicly available, and it features Feature-engine, let us know so we link it to the project or share it on social.

   If you teach Feature-engine, give a shout-out on Twiter or LinkedIn and tag the maintainers.

   If you have other ideas on how to spread the word, let us know or update this page straightaway.

   We are also happy to talk about the project in talks and podcasts. If you host a meetup or a podcast channel, do get in touch.

Sponsor us

Empower Sole, the main developer of Feature-engine, to assemble a team of paid contributors to accelerate the development of Feature-engine.

Currently, Sole and our contributors dedicate their free time voluntarily to advancing the project. You can help us reach a funding milestone, so that we can gather on a group of 2-3 contributors who will commit regular hours each week to enhance documentation and expand Feature-engine's functionality at a faster pace. Your contribution will play a vital role in propelling Feature-engine to new heights, ensuring it remains a valuable resource for the data science community.

If you don't have a Github account, you can also sponsor us here.

Code of Conduct

Feature-engine is an open source Python project. We follow the Python Software Foundation Code of Conduct. All interactions among members of the Feature-engine community must meet those guidelines. This includes (but is not limited to) interactions through the mailing list, GitHub and StackOverflow.

Everyone is expected to be open, considerate, and respectful of others no matter what their position is within the project. We show gratitude for any contribution, big or small. We welcome feedback and participation. We want to make Feature-engine a nice, welcoming and safe place for you to do your first contribution to open source, and why not the second, the third and so on :).

10.6 About

In this section you will find information about the Feature-engine's origin, main developers, roadmap and overall vision for the package. You will also find information about how to cite Feature-engine and our main sponsors.

10.6.1 About

History

Data scientists spend a huge amount of time on data pre-processing and transformation. It would be great (we thought back in the day) to gather the most frequently used data pre-processing techniques and transformations in a library, from which we could pick and choose the transformation that we need, and use it just like we would use any other sklearn class. This was the original vision for Feature-engine.

Feature-engine is an open source Python package originally designed to support the online course Feature Engineering for Machine Learning, but has now gained popularity and supports transformations beyond those taught in the course. It was launched in 2017, and since then, several releases have appeared and a growing international community is beginning to lead the development.

Governance

The decision making process and governance structure of Feature-engine is laid out in the **governance document***.

Core contributors

The following people are currently core contributors to Feature-engine's development and maintenance:

Former core contributors

The following people are former core contributors to Feature-engine's development and maintenance:

Contributors

A growing international community is beginning to lead Feature-engine's development. You can learn more about Feature-engine's Contributors in the GitHub contributors page.

Citing Feature-engine

If you use Feature-engine in a scientific publication, you can cite the following paper: Galli, S., (2021). Feature-engine: A Python package for feature engineering for machine learning. Journal of Open Source Software, 6(65), 3642.

Bibtex entry:

```
@article{Galli2021,
doi = {10.21105/joss.03642},
url = {https://doi.org/10.21105/joss.03642},
year = {2021},
publisher = {The Open Journal},
volume = {6},
number = {65},
```

(continues on next page)

(continued from previous page)

```
pages = {3642},
author = {Soledad Galli},
title = {Feature-engine: A Python␣
↪package for feature engineering for machine learning},
journal = {Journal of Open Source Software}
}
```

You can also find a DOI (digital object identifier) for every version of Feature-engine on zenodo.org; use the BibTeX on this site to reference specific versions of the software.

Artwork

High quality PNG and SVG logos are available in the docs/images/ source directory of the repository.

10.6.2 Governance

The purpose of this document is to formalize the governance process used by the Feature-engine project and clarify how decisions are made and how the community works together. This is the first version of our governance policy and will be updated as our community grows and more of us take on different roles.

Roles and Responsibilities

Contributors

Contributors are community members who contribute in various ways to the project. Anyone can become a contributor, and contributions can be of various forms, not just code. To see how you can help check the ***Contribute page***.

Core Contributors

Core Contributors are community members who are dedicated to the continued development of the project through ongoing engagement with the community. Core Contributors are expected to review code contributions, can approve and merge pull requests, can decide on the fate of pull requests, and can be involved in deciding major changes to the Feature-engine API. Core Contributors determine who can join as a Core Contributor.

Founder and Leadership

Feature-engine was founded by Soledad Galli who at the time was solely responsible for the initial prototypes, documentation, and dissemination of the project. In the tradition of Python, Sole is referred to as the "benevolent dictator for life" (BDFL) of the project, or simply, "the founder". From a governance perspective, the BDFL has a special role in that they provide vision, thought leadership, and high-level direction for the project's members and contributors. The BDFL has the authority to make all final decisions for the Feature-engine Project. However, in practice, the BDFL, chooses to defer that authority to the consensus of the community discussion channels and the Core Contributors. The BDFL can also propose and vote for new Core Contributors.

Join the community

Feature-engine is currently looking to expand the team of Core Contributors, if you are interested, please get in touch.

If you want to Contribute to the project in any other way, get in touch using our Github issues page or through Gitter:

1. Github issues.
2. Gitter community.

10.6.3 Roadmap

This document provides general directions on what the core contributors would like to see developed in Feature-engine. As resources are limited, we can't promise when or if the transformers listed here will be included in the library. We welcome all the help we can get to support this vision. If you are interested in contributing, please get in touch.

Purpose

Feature-engine's mission is to simplify and streamline the implementation of end-to-end feature engineering pipelines. It aims to help users both during the research phase and while putting a model in production.

Feature-engine makes data engineering easy by allowing the selection of feature subsets directly within its transformers. It also interlaces well with exploratory data analysis (EDA) by returning dataframes for easy data exploration.

Feature-engine's transformers preserve Scikit-learn functionality with the methods fit() and transform() and can be integrated into a Pipeline to simplify putting the model in production.

Feature-engine was designed to be used in real settings. Each transformer has a concrete aim, and is tailored to certain variables and certain data. Transformers raise errors and warnings to support the user to use a suitable transformation given the data. These errors help avoid inadvertedly incorporating missing values to the dataframe at unwanted stages of the development.

Vision

At the moment, Feature-engine's functionality is tailored to tabular data, with numerical, categorical, or datetime variables. We started supporting the creation of features for time series forecasting in 2022.

But we would like to extend Feature-engine's functionality to work with text and time series, as well as, expand its current functionality for tabular data.

In the following figure we show the overall structure and vision for Feature-engine:

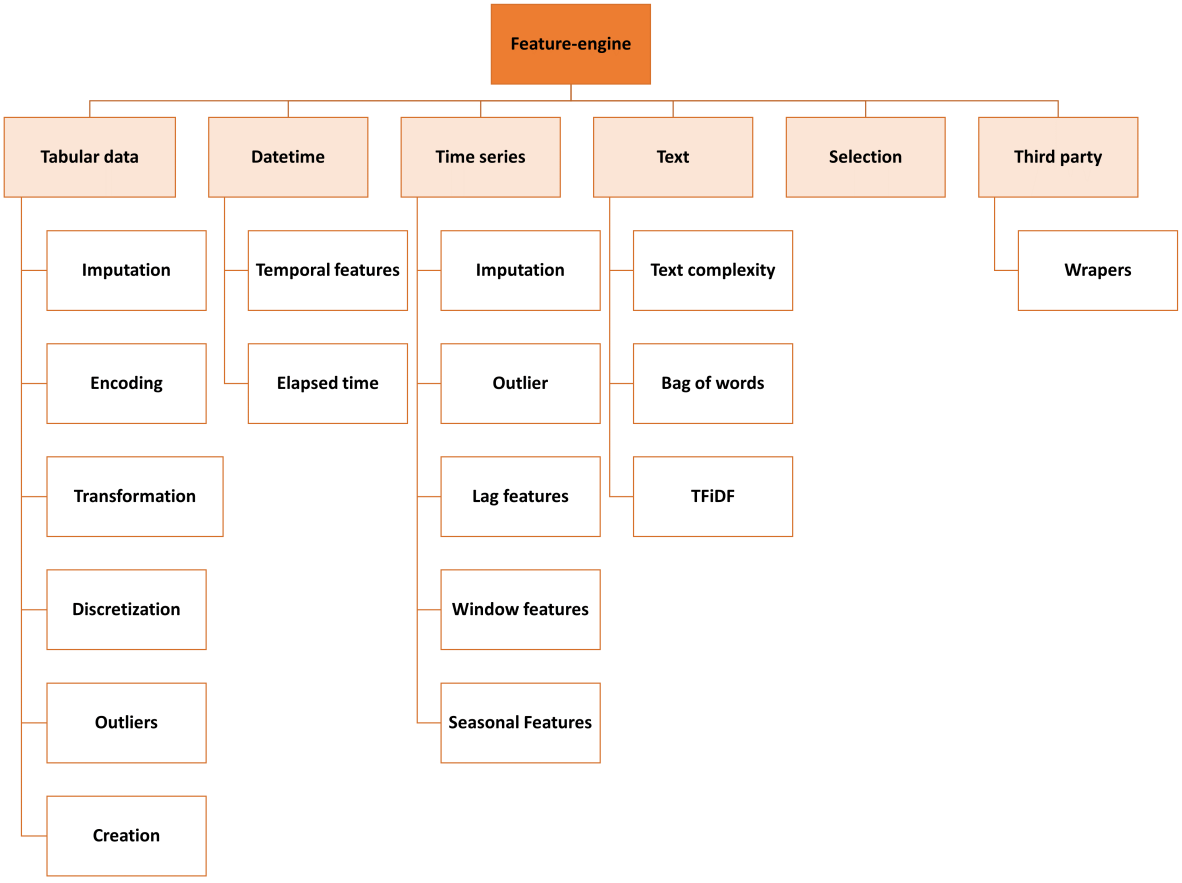


Fig. 128: Feature-engine structure

Current functionality

Most of the functionality for tabular data is already included in the package. We expand and update this arm of the library, based on user feedback and suggestions, and our own research in the field. In grey, the transformers that are not yet included in the package:

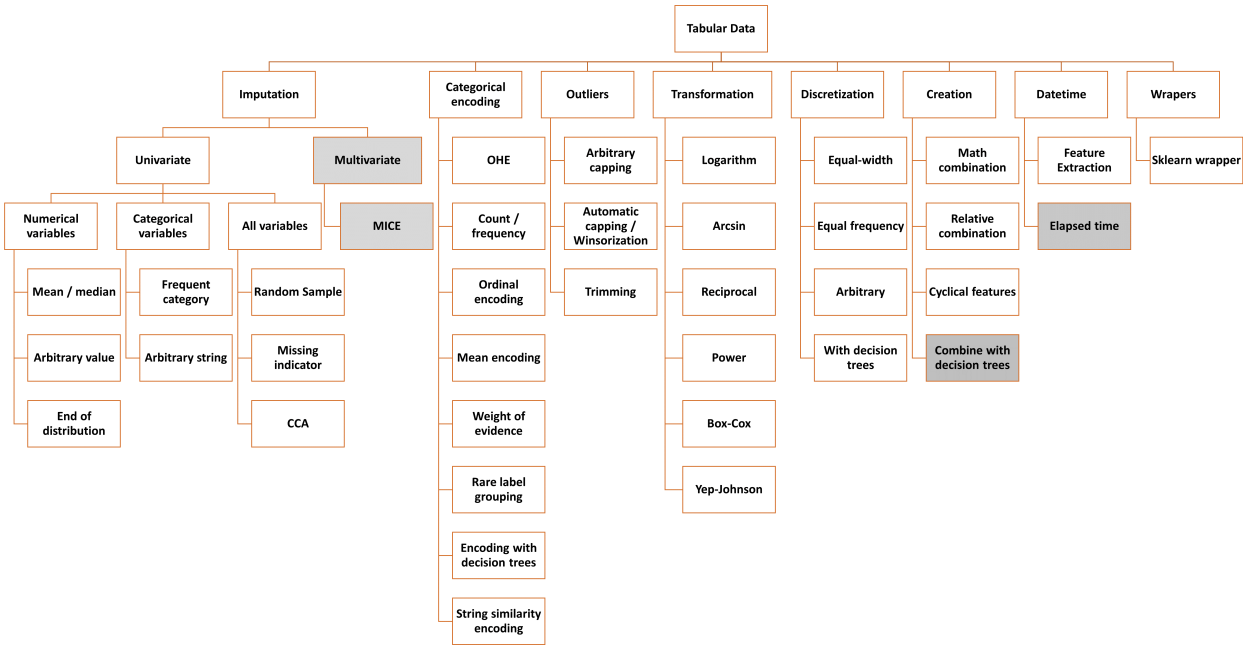


Fig. 129: Transformers for tabular data

The current transformations supported by Feature-engine return features that are easy to interpret, and the effects of the transformations are clear and easy to understand. The original aim of Feature-engine was to provide technology that is suitable to create models that will be used in real settings, and return understandable variables.

Having said this, more and more, users are requesting features to combine or transform variables in ways that would return features that are not human readable, in an attempt to improve model performance and perhaps have an edge in data science competitions. We are currently contemplating the incorporation of this functionality to the package.

Wanted functionality

We would also like to add a module that returns straightforward features from simple text variables, to capture text complexity, like for example counting the number of words, unique words, lexical complexity, number of paragraphs and sentences. We would also consider integrating the Bag of Words and TFiDF from sklearn with a wrapper that returns a dataframe ready to use to train machine learning models. Below we show more detail into these new modules.

In addition, we would like to expand our module for time series forecasting features. The transformations we are considering are shown in this image:

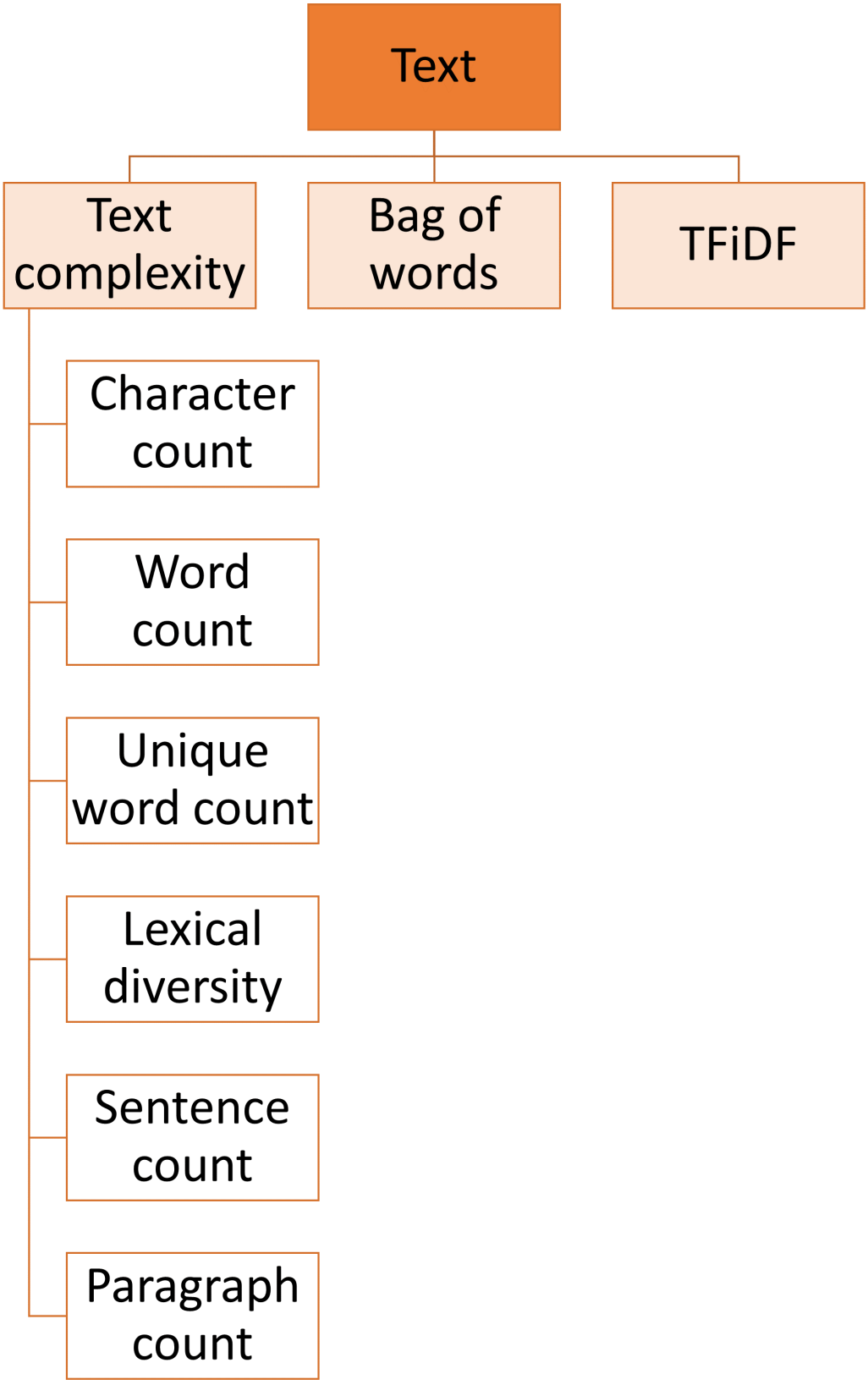


Fig. 130: New models wanted: datetime and text

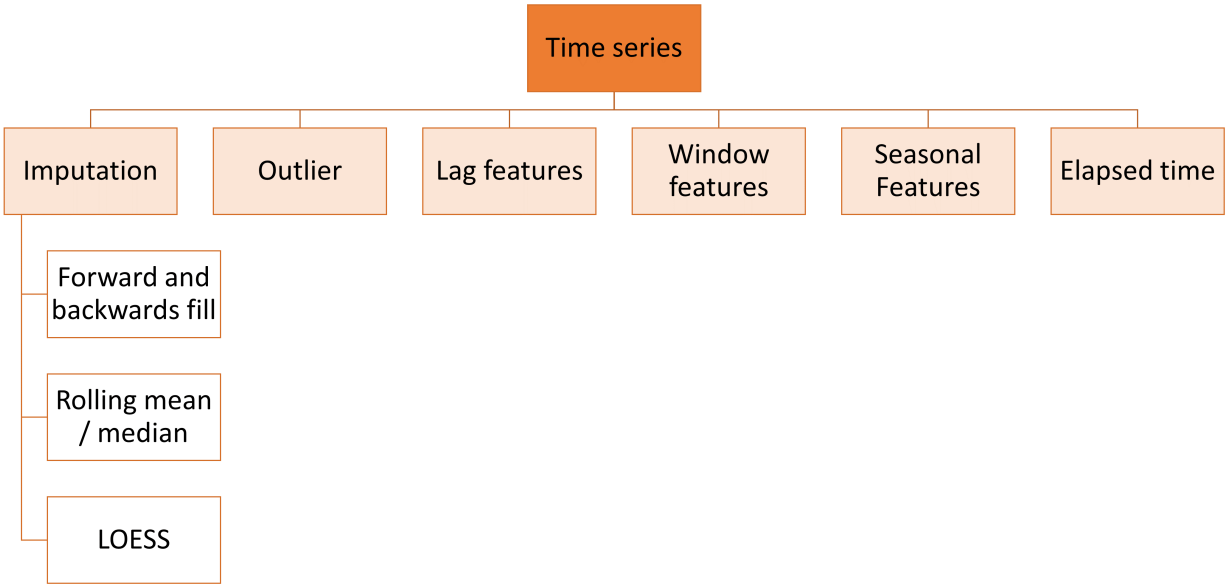


Fig. 131: Time series module and the transformations envisioned

Goals

Our main goals are:

- Continue maintaining a high-quality, well-documented collection of canonical tools for data processing.
- Expand the documentation with more examples about Feature-engine's functionality.
- Expand the documentation with more detail on how to contribute to the package.
- Expand the library's functionality as per the precedent paragraphs.

For more fine-grained goals and current and lined-up issues please visit the issues section in our repo.

10.7 What's new

Find out what's new in each new version release.

10.7.1 Version 1.7.X

Version 1.7.0

Deployed: 24th March 2024

Contributors

- dlaprins
- Gleb Levitski
- Chris Samiullah
- Morgan Sell
- Darigov Research
- Soledad Galli

There are a few big additions in this new release. First, we introduce a new `Pipeline` that supports transformers that remove rows from the dataset during the data transformation. From now on, you can use `DropMissingData`, `OutlierTrimmer`, `LagFeatures` and `WindowFeatures` as part of a feature engineering pipeline that will transform your variables, re-align the target to the remaining rows if necessary, and then fit a model. All in one go!

In addition, transformers that remove rows from the dataset, like `DropMissingData`, `OutlierTrimmer`, `LagFeatures` and `WindowFeatures`, can now adjust the target value to the remaining rows through the new method `transform_x_y`.

The third big improvement consists in a massive speed optimization of our correlation transformers, which now find and remove correlated features at doble the speed, and also, let you easily identify the feature against which the correlated ones were determined.

Other than that, we did a lot of work to catch up with the latest developments of Scikit-learn and pandas, to ensure that our transformers keep on being compatible. That being said, we are a small team and maintenance is hard for us, so we've deprecated support of earlier releases of these libraries.

Read on to find out more what we've been up to!

New functionality

- We now have a `Pipeline()` and `make_pipeline` that support transformers that remove rows from the dataset (Soledad Galli)
- `DropMissingData`, `OutlierTrimmer`, `LagFeatures`, `ExpandingWindowFeatures` and `WindowFeatures` have the method `transform_x_y` to remove rows from the data and then adjust the target variable (Soledad Galli)

Enhancements

- `DropCorrelatedFeatures()` and `SmartCorrelationSelection` have a new attribute to indicate which feature will be retained from each correlated group (Soledad Galli, dlaprins)
- `DropCorrelatedFeatures()` and `SmartCorrelationSelection` are twice as fast and can sort variables based on variance, cardinality or alphabetically before the search (Soledad Galli, dlaprins)
- `LagFeatures` can now impute the introduced nan values (Soledad Galli)

Bug fixes

- `DropCorrelatedFeatures()` and `SmartCorrelationSelection` are now deterministic (Soledad Galli, Gleb Levitski, dlaprins)

  In addition to these bug fixes, we fixed other pandas, and scikit-learn new version and deprecation related bugs.

Code improvements

- Improves logic to select the variables to examine in all feature selection transformers (Soledad Galli)
- Add circleCI tests for python 3.11 and 3.12 (Soledad Galli, Chris Samiullah)

Documentation

- Improve user guide for `DropCorrelatedFeatures()` and `SmartCorrelationSelection` (Soledad Galli)
- Improve user guide for DropMissingData()`(`Soledad Galli)
- Improve user guide for OutlierTrimmer()`(`Soledad Galli)
- Improve user guide for `LagFeatures`, `ExpandingWindowFeatures` and WindowFeatures`(`Soledad Galli)
- Add user guide for `Pipeline` (Soledad Galli)
- Improve feature creation user guide index (Soledad Galli and Morgan Sell)
- Make one click copyable code in Readme (Darigov Research)

Deprecations

- We remove support for Python 3.8 (Soledad Galli)
- We bump the dependencies on pandas and Scikit-learn to their latest versions (Soledad Galli)

10.7.2 Version 1.6.X

Version 1.6.2

Deployed: 18th September 2023

Contributors

- Giorgio Segalla
- David Cortes
- Kyle Gilde
- Darigov Research
- Soledad Galli

New functionality

- `MatchVariables()` can now also match the **dtypes** of the variables (Kyle Gilde)
- `DatetimeFeatures()` and `DatetimeSubtraction()` can now specify the format of the datetime variables (Soledad Galli)
- Add `inverse_transform` method to `YeoJohnsonTransformer()` (Giorgio Segalla)

Bug fixes

This bugs were introduced by the latest releases of pandas, Scikit-learn and Scipy.

- Fix failing test for `YeoJohnsonTransformer()` (Soledad Galli)
- Fix failing test for `RareLabelEncoder()` (Soledad Galli)
- Fix failing test for `DatetimeFeatures()` (Soledad Galli)
- Fix failing test for many encoders: removed `downcast=infer` as it will be deprecated (Soledad Galli)
- Fix version related failing style checks (Soledad Galli)
- Fix version related failing type checks (Soledad Galli)
- Fix version related failing doc checks (Soledad Galli)
- Fix future warning categorical imputation (Soledad Galli)

Code improvements

- Routine in `DatetimeFeatures()` does not enter into our check for `utc=True` when working with different timezones any more (Soledad Galli)
- Improve performance in `OneHotEncoder()` (Soledad Galli)
- Add check for dupicated variable names in dataframe (David Cortes)

Documentation

- Fix various typos in user guide (Soledad Galli)
- Update readthedocs.yml file (Soledad Galli)
- Add link to license in Readme (Darigov Research)

Version 1.6.1

Deployed: 8th June 2023

Contributors

- dlaprins
- Claudio Salvatore Arcidiacono
- Morgan Sell
- Gleb Levitski
- Soledad Galli

In this release, we make Feature-engine compatible with pandas 2.0, extend the functionality of some transformers, and we fix bugs introduced in the previous release.

Thank you so much to all contributors, Gleb Levitski and Claudio Salvatore Arcidiacono for helping with review and to those of you who created issues flagging bugs or requesting new functionality.

New functionality

- The Population Stability Index can now be used to evaluate categorical variables (dlaprins and Claudio Salvatore Arcidiacono)
- `RelativeFeatures` has the option to add a constant to avoid dividing by zero (Morgan Sell and Soledad Galli)
- `SelectByShuffling` now accepts sample weights (Soledad Galli)
- `WoEEncoder` now let's you know which variables fail in the encoding (Soledad Galli)
- `WoEEncoder` has the option to add a constant to avoid dividing by zero (Soledad Galli)

Bug fixes

- Fixed various bugs in `RareLabelEncoder()` (Soledad Galli)
- Renamed `transform` method in base classes to `check_transform_input_and_state`, which fixed bugs raised when `set_output(transform="pandas")` in various classes (Soledad Galli and Claudio Salvatore Arcidiacono)

Code improvements

- Made code base compatible with pandas 2.0 (Claudio Salvatore Arcidiacono)
- Moved docstrings of selection transformers to docstrings module (Soledad Galli)

Version 1.6.0

Deployed: 16th March 2023

Contributors

- Gleb Levitski
- Morgan Sell
- Alfonso Tobar
- Nodar Okroshiashvili
- Luís Seabra
- Kyle Gilde
- Soledad Galli

In this release, we make Feature-engine transformers compatible with the `set_output` API from Scikit-learn, which was released in version 1.2.0. We also make Feature-engine compatible with the newest direction of pandas, in removing the `inplace` functionality that our transformers use under the hood.

We introduce a major change: most of the **categorical encoders can now encode variables even if they have missing data**.

We are also releasing **3 brand new transformers**: One for discretization, one for feature selection and one for operations between datetime variables.

We also made a major improvement in the performance of the `DropDuplicateFeatures` and some smaller bug fixes here and there.

We'd like to thank all contributors for fixing bugs and expanding the functionality and documentation of Feature-engine.

Thank you so much to all contributors and to those of you who created issues flagging bugs or requesting new functionality.

New transformers

- **ProbeFeatureSelection**: introduces random features and selects variables whose importance is greater than the random ones (Morgan Sell and Soledad Galli)
- **DatetimeSubtraction**: creates new features by subtracting datetime variables (Kyle Gilde and Soledad Galli)
- **GeometricWidthDiscretiser**: sorts continuous variables into intervals determined by geometric progression (Gleb Levitski)

New functionality

- Allow categorical encoders to encode variables with NaN (Soledad Galli)
- Make transformers compatible with new `set_output` functionality from sklearn (Soledad Galli)
- The `ArbitraryDiscretiser()` now includes the lowest limits in the intervals (Soledad Galli)

New modules

- New **Datasets** module with functions to load specific datasets (Alfonso Tobar)
- New **variable_handling** module with functions to automatically select numerical, categorical, or datetime variables (Soledad Galli)

Bug fixes

- Fixed bug in `DropFeatures()` (Luís Seabra)
- Fixed bug in `RecursiveFeatureElimination()` caused when only 1 feature remained in data (Soledad Galli)

Documentation

- Add example code snippets to the selection module API docs (Alfonso Tobar)
- Add example code snippets to the outlier module API docs (Alfonso Tobar)
- Add example code snippets to the transformation module API docs (Alfonso Tobar)
- Add example code snippets to the time series module API docs (Alfonso Tobar)
- Add example code snippets to the preprocessing module API docs (Alfonso Tobar)
- Add example code snippets to the wrapper module API docs (Alfonso Tobar)
- Updated documentation using new Dataset module (Alfonso Tobar and Soledad Galli)
- Reorganized Readme badges (Gleb Levitski)
- New Jupyter notebooks for `GeometricWidthDiscretiser` (Gleb Levitski)
- Fixed typos (Gleb Levitski)
- Remove examples using the boston house dataset (Soledad Galli)
- Update sponsor page and contribute page (Soledad Galli)

Deprecations

- The class `PRatioEncoder` is no longer supported and was removed from the API (Soledad Galli)

Code improvements

- Massive improvement in the performance (speed) of `DropDuplicateFeatures()` (Nodar Okroshiashvili)
- Remove `inplace` and other issues related to pandas new direction (Luís Seabra)
- Move most docstrings to dedicated docstrings module (Soledad Galli)
- Unnest tests for encoders (Soledad Galli)

10.7.3 Version 1.5.X

Version 1.5.2

Deployed: 21th November 2022

Contributors

- Gleb Levitski
- Alfonso Tobar
- pxn39
- Soledad Galli

In this release, we expand the functionality of existing classes and the documentation.

New functionality

- The `StringSimilarityEncoder` can now create similarity variables based on keywords entered by the user (Gleb Levitski)
- The `Winsorizer` and `OutlierTrimmer` now automatically adjust the value of the `fold` parameter based on the `capping_method` (pxn39)

Bug fixes

- Type checks errors raised by newer versions (Gleb Levitski)

Documentation

- Add example code snippets to the categorical encoding API docs (Alfonso Tobar)
- Add example code snippets to the imputation module API docs (Alfonso Tobar)
- Add example code snippets to the discretisation module API docs (Alfonso Tobar)
- Add example code snippets to the creation module API docs (Alfonso Tobar)
- Add example code snippets to the datetime module API docs (Alfonso Tobar)
- Update the user guide docs for the forecasting feature transformers (Soledad Galli)
- Update the user guide docs for datetime features and cyclical features (Soledad Galli)
- Fix badges in README (Gleb Levitski)

Version 1.5.0

Deployed: 17th October 2022

Contributors

- Gleb Levitski
- David Cortes
- Alfonso Tobar
- Morgan Sell
- Soledad Galli

In this release, we fix a bug that made the `get_feature_names_out` not compatible with the Scikit-learn pipeline.

In addition, thanks to Gleb Levitski, we've got a new encoder to replace categories by string similarity variables. Gleb Levitski also made a number of code enhancements to various transformers across the library, making a lot of new functionality available.

Finally, we'd like to thank Alfonso Tobar, David Cortes and Morgan Sell for creating new transformers, fixing bugs and expanding the functionality of Feature-engine.

Thank you so much to all contributors and to those of you who created issues flagging bugs or requesting new functionality.

New transformers

- **StringSimilarityEncoder**: encodes categorical variables based on string similarity (Gleb Levitski)
- **MatchCategories**: matches the categories in train and test set when of type pandas categorical (David Cortes)
- **SelectByInformationValue**: selects features based on the information value (Morgan Sell and Soledad Galli)

New functionality

- The `MeanEncoder` can now implement smoothing during the encoding to handle high cardinality (Gleb Levitski)
- The `MeanEncoder` can now encode unseen categories (Gleb Levitski)
- The `OrdinalEncoder` can now encode unseen categories (Soledad Galli)
- The `CountFrequencyEncoder` can now encode unseen categories (David Cortes)
- All outlier transformers can now detect outliers based on the MAD rule (Gleb Levitski)
- Add automatic calculation of PSI threshold in `DropHighPSIFeatures` (Gleb Levitski)
- All feature selection transformers now have the method `get_support()` (Soledad Galli)

Bug fixes

- `get_feature_names_out` is now compatible with the Scikit-learn pipeline in all transformers (Soledad Galli)
- The `inverse_transform` method in encoders now correctly handles unseen categories or raises not implemented errors (Soledad Galli)
- Fixes output of `SklearnTransformerWrapper` for `OneHotEncoder` and `PolynomialFeatures` (Alfonso Tobar)

Documentation

- Add more resources to documentation (Soledad Galli)
- User guide for `StringSimilarityEncoder` (Gleb Levitski)
- New Jupyter notebook for `StringSimilarityEncoder` (Gleb Levitski)
- User guide for SelectByInformationValue (Morgan Sell and Soledad Galli)

Deprecations

- Parameter `errors` in encoders is now replaced by `unseen` (Soledad Galli)
- The classes `MathematicalCombination`, `CombineWithFeatureReference` and `CyclicalTransformer` are removed (Soledad Galli)
- We are deprecating `PRatioEncoder` in version 1.5 and it will be removed in version 1.6 (Soledad Galli)

Code improvements

- Adds code coverage test (Soledad Galli)
- Changes logic of encoding unseen categories to work with inverse_transform (Soledad Galli)
- Increases code coverage for encoders (Soledad Galli)
- Removes CategoricalInitExpandedMixin (Soledad Galli)
- Removes checks for encoding dictionaries in all encoders (Soledad Galli)
- Refactors creation module (Soledad Galli)
- Refactors docstring module (Soledad Galli)
- Refactors variable handling module (Soledad Galli)
- Refactors numerical dictionary checks (Soledad Galli)
- Refactors base transformers module (Soledad Galli)
- Makes dataframe checks more performant (Soledad Galli)
- Replaces pd.concat by pd.group in all target based encoders (Soledad Galli)

10.7.4 Version 1.4.X

Version 1.4.1

Deployed: 13th June 2022

Contributors

- Sangam
- Soledad Galli

Enhancements

- The `BoxCoxTransformer` has now `inverse_transform` functionality (Sangam)
- Transformers now check for duplicated variable names entered in the init `variables` parameter (Soledad Galli)

Bug fixes

- Fix test on Python 3.10 in CircleCI (Sangam and Soledad Galli)
- Fix numpy typing error related to numpy newest version (Sangam and Soledad Galli)

Documentation

- Update python versions in setup file (Soledad Galli)

Version 1.4.0

Deployed: 9th June 2022

Contributors

- tomtom-95
- Fernando Barbosa
- Sangam
- Swati A Firangi
- Mohamed Emad
- Brice
- Soledad Galli

In this release, we fix a major bug that was preventing you guys from using the new module introduced in version 1.3: timeseries. We basically, forgot to add the __init__ file and for some reason, we did not pick up this error from our development environments. Thank you Brice for reporting this very important issue.

In addition, we updated our code base to work with the latest release of Scikit-learn (1.1.1) and pandas (1.4.2), which means that like Scikit-learn, we no longer support Python 3.7.

We are delaying the complete deprecation of `MathematicalCombination`, `CombineWithFeatureReference`, and `CyclicalTransformer` to our next release (1.5), as this release is a bit short notice, to give you more time to adapt your code bases.

In addition, we've added a new transformer, a number of new badges and made some enhancements to our code base.

I am very happy to announce that for this release, we had a number of contributions from **first time contributors**. Thank you so much for your support!

Thank you so much to all contributors to this release for making it possible!

New transformers

- **ArcsinTransformer**: transforms variables with the arcsin transformation: arcsin(sqrt(x)) (tomtom-95)

Bug fixes

- The `SklearnTransformerWrapper` now accepts categorical variables when used with the FunctionTransformer (Fernando Barbosa)
- Added init file to allow import of time series module (Soledad Galli)

Documentation

- Add Yeo-Johnson article as reference (Sangam)
- Add first timers friendly badge (Swati A Firangi)
- Fixed source of logo in readme (Mohamed Emad)

Deprecations

- We are extending the complete deprecation of `MathematicalCombination`, `CombineWithFeatureReference` and `CyclicalTransformer` to version 1.5

Code improvements

- Improved message returned when y is not what expected (following sklearn 1.1.1) (Soledad Galli)
- Introduced check for some selectors to ensure user passes more than 1 variable (Soledad Galli)

For developers

- We broke down base categorical classes into MixIns (Soledad Galli)
- Accommodated lack of future pandas support for sets as indexers (Soledad Galli)

10.7.5 Version 1.3.X

Version 1.3.0

Deployed: 5th May 2022

Contributors

- Morgan Sell
- Kishan Manani
- Gilles Verbockhaven
- Noah Green
- Ben Reiniger
- Edoardo Argiolas
- Alejandro Giacometti
- Tim Vink
- Soledad Galli

In this release, we add the **get_feature_names_out** functionality to all our transformers! You asked for it, we delivered :)

In addition, we introduce a **new module** for **time series forecasting**. This module will host transformers that create features suitable for, well…, time series forecasting. We created three new transformers: **LagFeatures**, **WindowFeatures** and **ExpandingWindowFeatures**. We had the extraordinary support from Kishan Manani who is an experienced forecaster, and Morgan Sell who helped us draft the new classes. Thank you both for the incredible work!

We also improved the functionality of our feature creation classes. To do this, we are deprecating our former classes, `MathematicalCombination` and `CombineWithFeatureReference`, which are a bit of a mouthful, for the new classes `MathFeatures` and `RelativeFeatures`.

We are also renaming the class `CyclicalTransformer` to `CyclicalFeatures`.

We've also enhanced the functionality of the `SelectByTargetMeanPerformance` and `SklearnTransformerWrapper`.

In addition, we've had some bug reports and bug fixes that we list below, and a number of enhancements to our current classes.

Thank you so much to all contributors to this release for making this massive release possible!

New modules

- **timeseries-forecasting: this module hosts transformers that create features suitable for time seri forecasting (Morgan Sell, Kishan Manani and Soledad Galli)**
  - LagFeatures
  - WindowFeatures
  - ExpandingWindowFeatures

New transformers

- **LagFeatures**: adds lag versions of the features (Morgan Sell, Kishan Manani and Soledad Galli)
- **WindowFeatures**: creates features from operations on past time windows (Morgan Sell, Kishan Manani and Soledad Galli)
- **ExpandingWindowFeatures**: creates features from operations on all past data (Kishan Manani)
- **MathFeatures**: replaces `MathematicalCombination` and expands its functionality (Soledad Galli)
- **RelativeFeatures**: replaces `CombineWithFeatureReference` and expands its functionality (Soledad Galli)
- **CyclicalFeatures**: new name for `CyclicalTransformer` with same functionality (Soledad Galli)

New functionality

- All our transformers have now the `get_feature_names_out` functionality to obtain the names of the output features (Alejandro Giacometti, Morgan Sell and Soledad Galli)
- SelectByTargetMeanPerformance now uses cross-validation and supports all possible performance metrics for classification and regression (Morgan Sell and Soledad Galli)

Enhancements

- All our feature selection transformers can now check that the variables were not dropped in a previous selection step (Gilles Verbockhaven)
- The `DecisionTreeDiscretiser` and the `DecisionTreeEncoder` now check that the user enters a target suitable for regression or classification (Morgan Sell)
- The `DecisionTreeDiscretiser` and the `DecisionTreeEncoder` now accept all sklearn cross-validation constructors (Soledad Galli)
- The `SklearnTransformerWrapper` now implements the method `inverse_transform` (Soledad Galli)

- The `SklearnTransformerWrapper` now supports additional transformers, for example, PolynomialFeatures (Soledad Galli)
- The `CategoricalImputer()` now let's you know which variables have more than one mode (Soledad Galli)
- The `DatetimeFeatures()` now can extract features from the dataframe index (Edoardo Argiolas)
- Transformers that take y now check that X and y match (Noah Green and Ben Reiniger)

Bug fixes

- The `SklearnTransformerWrapper` now works with cross-validation when using the one hot encoder (Noah Green)
- The `SelectByShuffling` now evaluates the initial performance and the performance after shuffling in the same data parts (Gilles Verbockhaven)
- **Discretisers**: when setting `return_boundaries=True` the interval limits are now returned as strings and the variables as object data type (Soledad Galli)
- `DecisionTreeEncoder` now enforces passing y to `fit()` (Soledad Galli)
- `DropMissingData` can now take a string in the `variables` parameter (Soledad Galli)
- `DropFeatures` now accepts a string as input of the features_to_drop parameter (Noah Green)
- Categorical encoders now work correctly with numpy arrays as inputs (Noah Green and Ben Reiniger)

Documentation

- Improved user guide for `SelectByTargetMeanPerformance` with lots of tips for troubleshooting (Soledad Galli)
- Added guides on how to use `MathFeatures` and `RelativeFeatures` (Soledad Galli)
- Expanded user guide on how to use `CyclicalFeatures` with explanation and demos of what these features are (Soledad Galli)
- Added a Jupyter notebook with a demo of the `CyclicalFeatures` class (Soledad Galli)
- We now display all available methods in the documentation methods summary (Soledad Galli)
- Fixes typo in `ArbitraryNumberImputer` documentation (Tim Vink)

Deprecations

- We are deprecating `MathematicalCombination`, `CombineWithFeatureReference` and `CyclicalTransformer` in version 1.3 and they will be removed in version 1.4
- Feature-engine does not longer work with Python 3.6 due to dependence on latest versions of Scikit-learn
- In `MatchColumns` the attribute `input_features_` was replaced by `feature_names_in_` to adopt Scikit-learn convention

Code improvements

- **Imputers**: removed looping over every variable to replace NaN. Now passing imputer dictionary to `pd.fillna()` (Soledad Galli)
- `AddMissingIndicators`: removed looping over every variable to add missing indicators. Now using `pd.isna()` (Soledad Galli)
- `CategoricalImputer` now captures all modes in one go, without looping over variables (Soledad Galli)
- Removed workaround to import docstrings for `transform()` method in various transformers (Soledad Galli)

For developers

- Created functions and docstrings for common descriptions of methods and attributes (Soledad Galli)
- We introduce the use of common tests that are applied to all transformers (Soledad Galli)

Experimental

New experimental, currently private module: **prediction**, that hosts classes that are used by the `SelectByTargetMeanPerformance` feature selection transformer. The estimators in this module have functionality that exceed that required by the selector, in that, they can output estimates of the target by taking the average across a group of variables.

- New private module, **prediction** with a regression and a classification estimator (Morgan Sell and Soledad Galli)
- `TargetMeanRegressor`: estimates the target based on the average target mean value per class or interval, across variables (Morgan Sell and Soledad Galli)
- `TargetMeanClassifier`: estimates the target based on the average target mean value per class or interval, across variables (Morgan Sell and Soledad Galli)

10.7.6 Version 1.2.X

Version 1.2.0

Deployed: 4th January 2022

Contributors

- Edoardo Argiolas
- gverbock
- Thibault Blanc
- David Cortes
- Morgan Sell
- Kevin Kurek
- Soledad Galli

In this big release, we add 3 new transformers, we expand the functionality of existing classes, we add information about citing Feature-engine and we expand the documentation with a new look, extended user guide with examples, and more details on how to contribute to the project.

Thank you so much to the contributors for making this massive release possible!

Thank you to reviewers Nicolas Galli and Chris Samiullah for useful advice on various PRs.

New transformers

- **DatetimeFeatures**: extracts date and time features from datetime variables (Edoardo Argiolas)
- **DropHishPSIFeatures**: finds and drops features with high population stability index (gverbock)
- **Matchvariables**: ensures that the same variables observed in the train set are present in the test set (Thibault Blanc)

Enhancements

- The **Winsorizer** can now add binary variable indicators to flag outlier values (David Cortes)
- The **DropMissingData** now allows to drop rows based on % of missing data (Kevin Kurek)
- **Categorical encoders** can now either raise a warning or an error when encoding categories not seen in the train set (Morgan Sell)
- The **ArbitraryDiscretiser** can now either raise a warning or an error when values fall outside the limits entered by the user (Morgan Sell)

- **CombineWithReferenceFeature** and **MathematicalCombination** have now the option to drop the original input variables after the feature creation (Edoardo Argiolas)

Bug fixes

- All **Encoders** are now able to exclude datetime variables cast as object or categorical when searching for categorical variables automatically (Edoardo Argiolas)
- All transformers will now raise an error when users pass an empty list to the variables parameter (Edoardo Argiolas)
- All transformers now check the variable type when user passes a single variable to the variables parameter (Edoardo Argiolas)

Documentation

- We changed the template to pydata (Soledad Galli)
- We split the information about transformers into a user guide and an API (Soledad Galli)
- The API documentation shows how to use the transformers (Soledad Galli)
- The user guide expands the API docs with plenty of examples and tips on when and how to use the transformers (Soledad Galli)
- We expanded the contribute section with plenty of details on how to make a contribution and how to check your code is top notch (Soledad Galli)
- You can now sponsor Feature-engine (Soledad Galli)
- You can now cite our JOSS article when using Feature-engine (Soledad Galli)
- We added plenty of examples on how to use the new class DropHighPSIFeatures (gverbock)
- We included various examples on how to extract date and time features using the new DatetimeFeatures class (Edoardo Argiolas)
- We included examples on how to use the new class MatchVariables (Thibault Blanc)
- We added a Jupyter notebook with a demo of the new DatetimeFeatures class (Edoardo Argiolas)
- We added a Jupyter notebook with a demo of the new DropHighPSIFeatures class (Soledad Galli)

10.7.7 Version 1.1.X

Version 1.1.2

Deployed: 31th August 2021

Contributors

- Soledad Galli

  This small release fixes a Bug in how the OneHotEncoder handles binary categorical variables when the parameter `drop_last_binary` is set to True. It also ensures that the values in the `OneHotEncoder.encoder_dict_` are lists of categories and not arrays. These bugs were introduced in v1.1.0.

Bug fix

- **OneHotEncoder**: drop_last_binary now outputs 1 dummy variable per binary variable when set to true

Version 1.1.1

Deployed: 6th August 2021

Contributors

- Miguel Trema Marrufo
- Nicolas Galli
- Soledad Galli

  In this release, we add a new transformer, expand the functionality of 2 other transformers and migrate the repo to its own organisation!

Mayor changes

- Feature-engine is now hosted in its own Github organisation

New transformer

- **LogCpTransformer**: applies the logarithm transformation after adding a constant (**Miguel Trema Marrufo**)

Minor changes

- Expands functionality of `DropCorrelatedFeatures` and `SmartCorrelationSelectionFeature` to accept callables as a correlation function (**Miguel Trema Marrufo**)
- Adds `inverse_transform` to all transformers from the transformation module (**Nicolas Galli**).

Documentation

- Migrates main repo to Feature-engine's Github organisation
- Migrates example jupyter notebooks to separate repo
- Adds Roadmap

Version 1.1.0

Deployed: 22st June 2021

Contributors

- Hector Patino
- Andrew Tan
- Shubhmay Potdar
- Agustin Firpo
- Indy Navarro Vidal
- Ashok Kumar
- Chris Samiullah
- Soledad Galli

In this release, we enforce compatibility with Scikit-learn by adding the check_estimator tests to **all transformers** in the package.

In order to pass the tests, we needed to modify some of the internal functionality of Feature-engine transformers and create new attributes. We tried not to break backwards compatibility as much as possible.

Mayor changes

- Most transformers have now the additional attribute `variables_` containing the variables that will be modified. The former attribute `variables` is retained. `variables_` will almost always be identical to `variables` except when the transformer is initialised with `variables=None`.
- The parameter `transformer` in the SklearnTransformerWrapper and the parameter `estimator` in the SelectBySingleFeaturePerformance, SelectByShuffling, RecursiveFeatureElimination and RecursiveFeatureAddition

need a compulsory entry, and cannot be left blank when initialising the transformers.

- Categorical encoders support now variables cast as `category` as well as `object` (**Shubhmay Potdar and Soledad Galli**)
- Categorical encoders have now the parameter `ignore_format` to allow the transformer to work with any variable type, and not just object or categorical.
- `CategoricalImputer` has now the parameter `ignore_format` to allow the transformer to work with any variable type, and not just object or categorical.
- All transformers have now the new attribute `n_features_in` with captures the number of features in the dataset used to train the transformer (during fit()).

Minor changes

- Feature selection transformers support now all cross-validation schemes in the `cv` parameter, and not just an integer. That is, you can initialize the transformer with LOOCV, or StratifiedCV for example.
- The OneHotEncoder includes additional functionality to return just 1 dummy variable for categorical variables that contain only 2 categories. In the new attribute `variables_binary_` you can identify the original binary variables.
- MathematicalCombinator now supports use of dataframes with null values (**Agustin Firpo**).

New transformer

- **CyclicalTransformer**: applies a cyclical transformation to numerical variables (**Hector Patino**)

Code improvement

- Tests from check_estimator added to all transformers
- Test for compatibility with Python 3.9 added to circleCI (**Chris Samiullah and Soledad Galli**)
- Automatic black8 and linting added to tox
- Additional code fixes (**Andrew Tan and Indy Navarro Vidal**).

Documentation

- Additional comparison tables for imputers and encoders.
- Updates Readme with new badges and resources.
- Expanded SklearnWrapper demos in Jupyter notebooks.
- Expanded outlier transformer demos in Jupyter notebooks (**Ashok Kumar**)
- Expanded Pipeline demos in Jupyter notebooks.

Community

- Created Gitter community to support users and foster knowledge exchange

Version 1.0.2

Deployed: 22th January 2021

Contributors

- Nicolas Galli
- Pradumna Suryawanshi
- Elamraoui Sohayb
- Soledad Galli

New transformers

- **CombineWithReferenceFeatures**: applies mathematical operations between a group of variables and reference variables (**by Nicolas Galli**)
- **DropMissingData**: removes missing observations from a dataset (**Pradumna Suryawanshi**)

Bug Fix

- Fix bugs in SelectByTargetMeanPerformance.
- Fix documentation and jupyter notebook typos.

Tutorials

- **Creation**: updated "how to" examples on how to combine variables into new features (**by Elamraoui Sohayb and Nicolas Galli**)
- **Kaggle Kernels**: include links to Kaggle kernels

Version 1.0.1

Deployed: 11th January 2021

Bug Fix

- Fix use of r2 in SelectBySingleFeaturePerformance and SelectByTargetMeanPerformance.
- Fix documentation not showing properly in readthedocs.

Version 1.0.0

Deployed: 31st December 2020

Contributors

- Ashok Kumar
- Christopher Samiullah
- Nicolas Galli
- Nodar Okroshiashvili
- Pradumna Suryawanshi
- Sana Ben Driss
- Tejash Shah
- Tung Lee
- Soledad Galli

In this version, we made a major overhaul of the package, with code quality improvement throughout the code base, unification of attributes and methods, addition of new transformers and extended documentation. Read below for more details.

New transformers for Feature Selection

We included a whole new module with multiple transformers to select features.

- **DropConstantFeatures**: removes constant and quasi-constant features from a dataframe (**by Tejash Shah**)
- **DropDuplicateFeatures**: removes duplicated features from a dataset (**by Tejash Shah and Soledad Galli**)
- **DropCorrelatedFeatures**: removes features that are correlated (**by Nicolas Galli**)
- **SmartCorrelationSelection**: selects feature from group of correlated features based on certain criteria (**by Soledad Galli**)
- **ShuffleFeaturesSelector**: selects features by drop in machine learning model performance after feature's values are randomly shuffled (**by Sana Ben Driss**)
- **SelectBySingleFeaturePerformance**: selects features based on a ML model performance trained on individual features (**by Nicolas Galli**)

- **SelectByTargetMeanPerformance**: selects features encoding the categories or intervals with the target mean and using that as proxy for performance (**by Tung Lee and Soledad Galli**)
- **RecursiveFeatureElimination**: selects features recursively, evaluating the drop in ML performance, from the least to the most important feature (**by Sana Ben Driss**)
- **RecursiveFeatureAddition**: selects features recursively, evaluating the increase in ML performance, from the most to the least important feature (**by Sana Ben Driss**)

Renaming of Modules

Feature-engine transformers have been sorted into submodules to smooth the development of the package and shorten import syntax for users.

- **Module imputation**: missing data imputers are now imported from `feature_engine.imputation` instead of `feature_engine.missing_data_imputation`.
- **Module encoding**: categorical variable encoders are now imported from `feature_engine.encoding` instead of `feature_engine_categorical_encoders`.
- **Module discretisation**: discretisation transformers are now imported from `feature_engine.discretisation` instead of `feature_engine.discretisers`.
- **Module transformation**: transformers are now imported from `feature_engine.transformation` instead of `feature_engine.variable_transformers`.
- **Module outliers**: transformers to remove or censor outliers are now imported from `feature_engine.outliers` instead of `feature_engine.outlier_removers`.
- **Module selection**: new module hosts transformers to select or remove variables from a dataset.
- **Module creation**: new module hosts transformers that combine variables into new features using mathematical or other operations.

Renaming of Classes

We shortened the name of categorical encoders, and also renamed other classes to simplify import syntax.

- **Encoders**: the word `Categorical` was removed from the classes name. Now, instead of `MeanCategoricalEncoder`, the class is called `MeanEncoder`. Instead of `RareLabelCategoricalEncoder` it is `RareLabelEncoder` and so on. Please check the encoders documentation for more details.
- **Imputers**: the `CategoricalVariableImputer` is now called `CategoricalImputer`.
- **Discretisers**: the `UserInputDiscretiser` is now called `ArbitraryDiscretiser`.

- **Creation**: the `MathematicalCombinator` is not called `MathematicalCombination`.
- **WoEEncoder and PRatioEncoder**: the `WoEEncoder` now applies only encoding with the weight of evidence. To apply encoding by probability ratios, use a different transformer: the `PRatioEncoder` (**by Nicolas Galli**).

Renaming of Parameters

We renamed a few parameters to unify the nomenclature across the Package.

- **EndTailImputer**: the parameter `distribution` is now called `imputation_method` to unify convention among imputers. To impute using the IQR, we now need to pass `imputation_method="iqr"` instead of `imputation_method="skewed"`.
- **AddMissingIndicator**: the parameter `missing_only` now takes the boolean values `True` or `False`.
- **Winzoriser and OutlierTrimmer**: the parameter `distribution` is now called `capping_method` to unify names across Feature-engine transformers.

Tutorials

- **Imputation**: updated "how to" examples of missing data imputation (**by Pradumna Suryawanshi**)
- **Encoders**: new and updated "how to" examples of categorical encoding (**by Ashok Kumar**)
- **Discretisation**: new and updated "how to" examples of discretisation (**by Ashok Kumar**)
- **Variable transformation**: updated "how to" examples on how to apply mathematical transformations to variables (**by Pradumna Suryawanshi**)

For Contributors and Developers

Code Architecture

- **Submodules**: transformers have been grouped within relevant submodules and modules.
- **Individual tests**: testing classes have been subdivided into individual tests
- **Code Style**: we adopted the use of flake8 for linting and PEP8 style checks, and black for automatic re-styling of code.
- **Type hint**: we rolled out the use of type hint throughout classes and functions (**by Nodar Okroshiashvili, Soledad Galli and Chris Samiullah**)

Documentation

- Switched fully to numpydoc and away from Napoleon
- Included more detail about methods, parameters, returns and raises, as per numpydoc docstring style (**by Nodar Okroshiashvili, Soledad Galli**)
- Linked documentation to github repository
- Improved layout

Other Changes

- **Updated documentation**: documentation reflects the current use of Feature-engine transformers
- **Typo fixes**: Thank you to all who contributed to typo fixes (Tim Vink, Github user @piecot)

10.7.8 Version 0.6.X

Version 0.6.1

Deployed: Friday, September 18, 2020

Contributors: Soledad Galli

Minor Changes:

- **Updated docs**: updated and expanded Contributing guidelines, added Governance, updated references to Feature-engine online.
- **Updated Readme**: updated and expanded readme.

Version 0.6.0

Deployed: Friday, August 14, 2020

Contributors:

- Michał Gromiec
- Surya Krishnamurthy
- Gleb Levitskiy
- Karthik Kothareddy
- Richard Cornelius Suwandi
- Chris Samiullah
- Soledad Galli

Major Changes:

- **New Transformer**: the `MathematicalCombinator` allows you combine multiple features into new variables by performing mathematical operations like sum, product, mean, standard deviation, or finding the minimum and maximum values (by Michał Gromiec).
- **New Transformer**: the `DropFeatures` allows you remove specified variables from a dataset (by Karthik Kothareddy).
- **New Transformer**: the `DecisionTreeCategoricalEncoder` encodes categorical variables with a decision tree (by Surya Krishnamurthy).
- **Bug fix**: the `SklearnTransformerWrapper` can now automatically select numerical or numerical and categorical variables depending on the Scikit-learn transformer the user implements (by Michał Gromiec).
- **Bug fix**: the `SklearnTransformerWrapper` can now wrap Scikit-learn's OneHotEncoder and concatenate the binary features back to the original dataframe (by Michał Gromiec).
- **Added functionality**: the `ArbitraryNumberImputer` can now take a dictionary of variable, arbitrary number pairs, to impute different variables with different numbers (by Michał Gromiec).
- **Added functionality**: the `CategoricalVariableImputer` can now replace missing data in categorical variables by a string defined by the user (by Gleb Levitskiy).
- **Added functionality**: the `RareLabelEnoder` now allows the user to determine the maximum number of categories that the variable should have when grouping infrequent values (by Surya Krishnamurthy).

Minor Changes:

- **Improved docs**: fixed typos and tidy Readme.md (by Richard Cornelius Suwandi)
- **Improved engineering practices**: added Manifest.in to include md and licenses in tar ball in pypi (by Chris Samiullah)
- **Improved engineering practices**: updated circleci yaml and created release branch for orchestrated release of new versions with significant changes (by Soledad Galli and Chris Samiullah)
- **Improved engineering practices**: added test for doc build in circleci yaml (by Soledad Galli and Chris Samiullah)
- **Transformer fix**: removed parameter return_object from the RareLabelEncoder as it was not working as intended(by Karthik Kothareddy and Soledad Galli)

Version 0.5.0

- Deployed: Friday, July 10, 2020
- Contributors: Soledad Galli

Major Changes:

- **Bug fix: fixed error in weight of evidence formula in the `WoERatioCategoricalEncoder`. The c formula, that is np.log(p(1) / p(0)) is preserved, and can be obtained by setting the `encoding_method` to 'log_ratio'. If `encoding_method` is set to 'woe', now the correct formula v operate.**
  - **Added functionality**: most categorical encoders have the option `inverse_transform`, to obtain the original value of the variable from the transformed dataset.
  - **Added functionality**: the `'Winsorizer`, `OutlierTrimmer` and `ArbitraryOutlierCapper` have now the option to ignore missing values, and obtain the parameters from the original variable distribution, or raise an error if the dataframe contains na, by setting the parameter `missing_values` to `raise` or `ignore`.
  - **New Transformer**: the `UserInputDiscretiser` allows users to discretise numerical variables into arbitrarily defined buckets.

Version 0.4.3

- Deployed: Friday, May 15, 2020
- Contributors: Soledad Galli, Christopher Samiullah

Major Changes:

- **New Transformer**: the `'SklearnTransformerWrapper`` allows you to use most Scikit-learn transformers just on a subset of features. Works with the SimpleImputer, the OrdinalEncoder and most scalers.

Minor Changes:

- **Added functionality**: the `'EqualFrequencyDiscretiser`` and `EqualWidthDiscretiser` now have the ability to return interval boundaries as well as integers, to identify the bins. To return boundareis set the parameter `return_boundaries=True`.
- **Improved docs**: added contibuting section, where you can find information on how to participate in the development of Feature-engine's code base, and more.

Version 0.4.0

- Deployed: Monday, April 04, 2020
- Contributors: Soledad Galli, Christopher Samiullah

Major Changes:

- **Deprecated**: the `FrequentCategoryImputer` was integrated into the class `CategoricalVariableImputer`. To perform frequent category imputation now use: `CategoricalVariableImputer(imputation_method='frequent')`
- **Renamed**: the `AddNaNBinaryImputer` is now called `AddMissingIndicator`.
- **New**: the `OutlierTrimmer` was introduced into the package and allows you to remove outliers from the dataset

Minor Changes:

- **Improved**: the `EndTailImputer` now has the additional option to place outliers at a factor of the maximum value.
- **Improved**: the `FrequentCategoryImputer` has now the functionality to return numerical variables cast as object, in case you want to operate with them as if they were categorical. Set `return_object=True`.
- **Improved**: the `RareLabelEncoder` now allows the user to define the name for the label that will replace rare categories.
- **Improved**: All feature engine transformers (except missing data imputers) check that the data sets do not contain missing values.
- **Improved**: the `LogTransformer` will raise an error if a variable has zero or negative values.
- **Improved**: the `ReciprocalTransformer` now works with variables of type integer.
- **Improved**: the `ReciprocalTransformer` will raise an error if the variable contains the value zero.
- **Improved**: the `BoxCoxTransformer` will raise an error if the variable contains negative values.
- **Improved**: the `OutlierCapper` now finds and removes outliers based of percentiles.
- **Improved**: Feature-engine is now compatible with latest releases of Pandas and Scikit-learn.

Version 0.3.0

- Deployed: Monday, August 05, 2019
- Contributors: Soledad Galli.

Major Changes:

- **New**: the `RandomSampleImputer` now has the option to set one seed for batch imputation or set a seed observation per observations based on 1 or more additional numerical variables for that observation, which can be combined with multiplication or addition.
- **New**: the `YeoJohnsonTransfomer` has been included to perform Yeo-Johnson transformation of numerical variables.
- **Renamed**: the `ExponentialTransformer` is now called `PowerTransformer`.
- **Improved**: the `DecisionTreeDiscretiser` now allows to provide a grid of parameters to tune the decision trees which is done with a GridSearchCV under the hood.
- **New**: Extended documentation for all Feature-engine's transformers.
- **New**: *Quickstart* guide to jump on straight onto how to use Feature-engine.
- **New**: *Changelog* to track what is new in Feature-engine.
- **Updated**: new `Jupyter notebooks` with examples on how to use Feature-engine's transformers.

Minor Changes:

- **Unified**: dictionary attributes in transformers, which contain the transformation mappings, now end with _, for example `binner_dict_`.

10.8 Other versions

Web-based documentation is available for versions listed below:

- Feature-engine 1.6

10.9 Sponsor us

Support Feature-engine financially through Github Sponsors and help further our mission to democratize machine learning and programming tools through open-source.

More details about how we use donations in the sponsors page.

10.10 Sponsors

Feature-engine is a community driven project, however institutional, private and individual support help to assure its sustainability. The project would like to thank the following sponsors:

BIBLIOGRAPHY

[1] Niculescu-Mizil, et al. “Winning the KDD Cup Orange Challenge with Ensemble Selection”. JMLR: Workshop and Conference Proceedings 7: 23-34. KDD 2009 http://proceedings.mlr.press/v7/niculescu09/niculescu09.pdf

[1] Galli S. “Machine Learning in Financial Risk Assessment”. https://www.youtube.com/watch?v=KHGGlozsRtA

[1] Micci-Barreca D. “A Preprocessing Scheme for High-Cardinality Categorical Attributes in Classification and Prediction Problems”. ACM SIGKDD Explorations Newsletter, 2001. https://dl.acm.org/citation.cfm?id=507538

[1] Niculescu-Mizil, et al. “Winning the KDD Cup Orange Challenge with Ensemble Selection”. JMLR: Workshop and Conference Proceedings 7: 23-34. KDD 2009 http://proceedings.mlr.press/v7/niculescu09/niculescu09.pdf

[1] Cerda P, Varoquaux G, Kégl B. “Similarity encoding for learning with dirty categorical variables”. Machine Learning, Springer Verlag, 2018.

[2] Cerda P, Varoquaux G. “Encoding high-cardinality string categorical variables”. IEEE Transactions on Knowledge & Data Engineering, 2020.

[1] Kotsiantis and Pintelas, “Data preprocessing for supervised leaning,” International Journal of Computer Science, vol. 1, pp. 111 117, 2006.

[2] Dong. “Beating Kaggle the easy way”. Master Thesis. https://www.ke.tu-darmstadt.de/lehre/arbeiten/studien/2015/Dong_Ying.pdf

[1] Kotsiantis and Pintelas, “Data preprocessing for supervised leaning,” International Journal of Computer Science, vol. 1, pp. 111 117, 2006.

[2] Dong. “Beating Kaggle the easy way”. Master Thesis. https://www.ke.tu-darmstadt.de/lehre/arbeiten/studien/2015/Dong_Ying.pdf

[1] Niculescu-Mizil, et al. “Winning the KDD Cup Orange Challenge with Ensemble Selection”. JMLR: Workshop and Conference Proceedings 7: 23-34. KDD 2009 http://proceedings.mlr.press/v7/niculescu09/niculescu09.pdf

[1] J. Reiser, “Classification Systems”, https://www.slideshare.net/johnjreiser/classification-systems

[2] Geometric Interval Classification http://wiki.gis.com/wiki/index.php/Geometric_Interval_Classification

[3] Geometric progression https://en.wikipedia.org/wiki/Geometric_progression

[1] Box and Cox. “An Analysis of Transformations”. Read at a RESEARCH MEETING, 1964. https://rss.onlinelibrary.wiley.com/doi/abs/10.1111/j.2517-6161.1964.tb00553.x

[1] Yeo, In-Kwon and Johnson, Richard (2000). A new family of power transformations to improve normality or symmetry. Biometrika, 87, 954-959.

[2] Weisberg S. “Yeo-Johnson Power Transformations”. https://www.stat.umn.edu/arc/yjpower.pdf

[1] Galli S. “Machine Learning in Financial Risk Assessment”. https://www.youtube.com/watch?v=KHGGlozsRtA

[1] Yurdakul B. “Statistical properties of population stability index”. Western Michigan University, 2018. https://scholarworks.wmich.edu/dissertations/3208/

[1] Weight of evidence and information value explained https://www.listendata.com/2015/03/weight-of-evidence-woe-and-information.html

[2] WoE and IV for continuous variables https://www.listendata.com/2019/08/WOE-IV-Continuous-Dependent.html

[1] Miller, et al. “Predicting customer behaviour: The University of Melbourne’s KDD Cup report”. JMLR Workshop and Conference Proceeding. KDD 2009 http://proceedings.mlr.press/v7/miller09/miller09.pdf

[1] Stoppiglia, et al. “Ranking a Random Feature for Variable and Feature Selection”. JMLR: 1399-1414, 2003 https://jmlr.org/papers/volume3/stoppiglia03a/stoppiglia03a.pdf

INDEX

G

L

M

N

O

P

R

S

T

W

Y