# feature_engine Documentation

***Release 1.0.2***

**Feature-engine Developers**

**Jan 23, 2021**

# TABLE OF CONTENTS

Fig. 1: Feature-engine rocks!

Feature-engine is a Python library with multiple transformers to engineer features for 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 variable encoding
- Discretisation
- Variable transformation
- Outlier capping or removal
- Variable creation
- Variable selection

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. That is, one object with the entire sequence of variable transformations to leave the raw data ready to be consumed by a machine learning algorithm, and the machine learning model at the back. Check the **quickstart** for an example.

**Would you like to know more about what is unique about Feature-engine?**

This article provides a nice summary:

- Feature-engine: A new open source Python package for feature engineering.

# CHAPTER ONE

# INSTALLATION

Feature-engine is a Python 3 package and works well with 3.6 or later. Earlier versions have not been tested. 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 TWO

# FEATURE-ENGINE FEATURES IN THE FOLLOWING RESOURCES

- Website.
- Feature Engineering for Machine Learning, Online Course.
- Python Feature Engineering Cookbook.
- Feature-engine: A new open-source Python package for feature engineering.
- Practical Code Implementations of Feature Engineering for Machine Learning with Python.

En Español:

- Ingeniería de variables para machine learning, Curso Online.
- Ingeniería de variables, MachinLenin, charla online.

More resources in the **Learning Resources** sections on the navigation panel on the left.

CHAPTER

THREE

# FEATURE-ENGINE'S TRANSFORMERS

## 3.1 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 value
- *EndTailImputer*: replaces missing data in numerical variables by numbers at the distribution tails
- *CategoricalImputer*: replaces missing data in categorical variables with the string 'Missing' or by the most frequent category
- *RandomSampleImputer*: replaces missing data with random samples of the variable
- *AddMissingIndicator*: adds a binary missing indicator to flag observations with missing data
- *DropMissingData*: removes rows containing NA values from dataframe

## 3.2 Categorical Variable Encoders: Encoders

- *OneHotEncoder*: performs one hot encoding, optional: of popular categories
- *CountFrequencyEncoder*: replaces categories by 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
- *PRatioEncoder*: replaces categories by a ratio of probabilities
- *DecisionTreeEncoder*: replaces categories by predictions of a decision tree
- *RareLabelEncoder*: groups infrequent categories

## 3.3 Numerical Variable Transformation: Transformers

- *LogTransformer*: performs logarithmic transformation of numerical variables
- *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

## 3.4 Variable Discretisation: Discretisers

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

## 3.5 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

## 3.6 Scikit-learn Wrapper:

- *SklearnTransformerWrapper*: executes Scikit-learn various transformers only on the selected subset of features

## 3.7 Mathematical Combination:

- *MathematicalCombination*: creates new variables by combining features with mathematical operations
- *CombineWithReferenceFeature*: creates variables with reference features through mathematical operations

## 3.8 Feature Selection:

- *DropFeatures*: drops a 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 feature from correlated group

- *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

# CHAPTER FOUR

# GETTING HELP

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

1. The docs (you’re here!).
2. Stack Overflow. If you ask a question, please tag it with “feature-engine”.
3. If you are enrolled in the Feature Engineering for Machine Learning course in Udemy , post a question in a relevant section.
4. Join our mailing list.
5. Ask a question in the repo by filing an issue.

# CHAPTER FIVE

# FOUND A BUG OR HAVE A SUGGESTION?

Check if there's already an open issue on the topic. If not, open a new issue with your bug report, suggestion or new feature request.

CHAPTER

SIX

# CONTRIBUTING

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

Feature-engine is a welcoming and inclusive project and it would be great 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 "Contributing" link on the left of this page.

# CHAPTER SEVEN

# 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.

## 7.1 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.

### 7.1.1 Installation

Feature-engine is a Python 3 package and works well with 3.6 or later. Earlier versions have not been tested. 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
```

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:

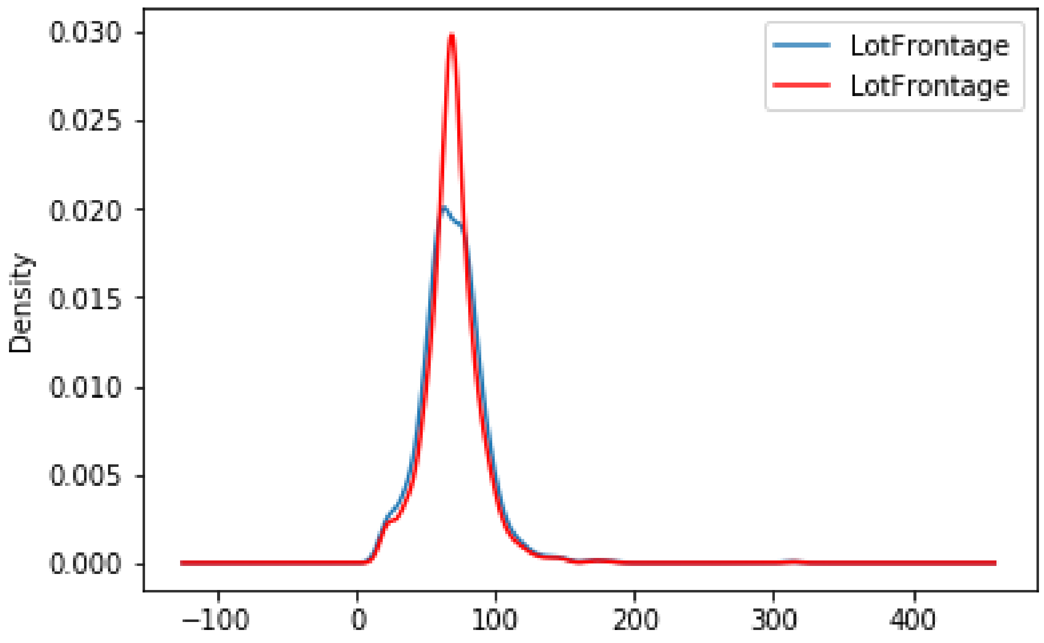
```
$ conda install -c conda-forge feature_engine
```

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

### 7.1.2 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

# 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')
```



### 7.1.3 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 feature engineering pipeline in one object and save it in one pickle (.pkl). Here is an example on 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
    )
```

(continues on next page)

(continued from previous page)

```
# 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']
    ]

# 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
```

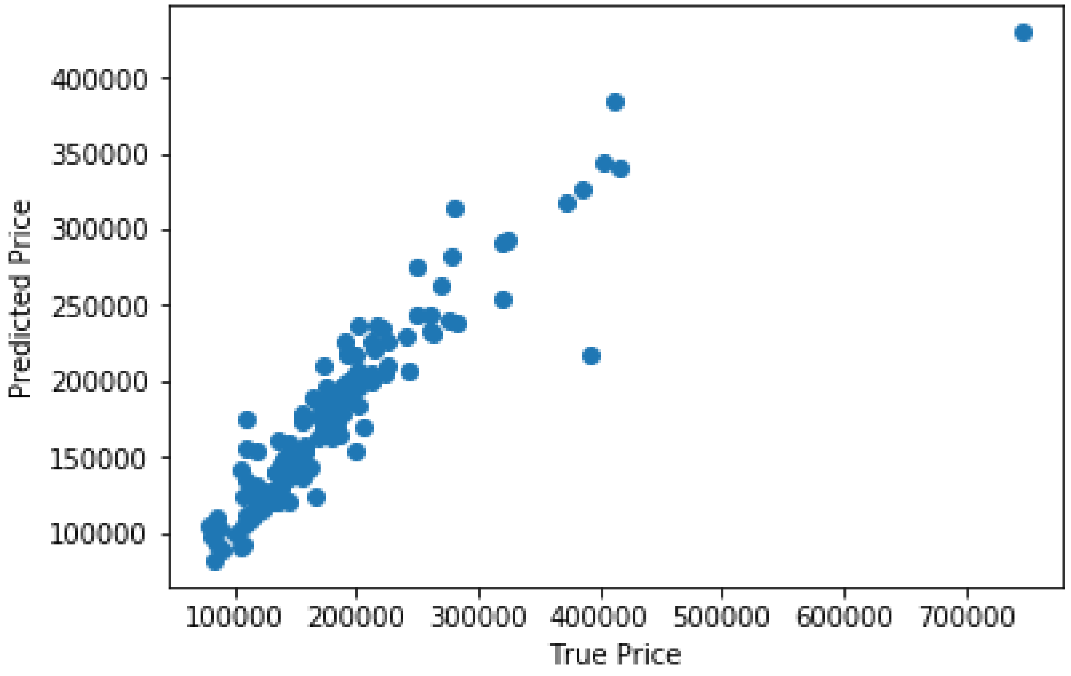
(continues on next page)

(continued from previous page)

```
    ('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
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 can be found in:

- The API documentation
- Tutorials
- How To

Check the navigation panel on the left.

## 7.2 Installation

Feature-engine is a Python 3 package and works well with 3.6 or later. Earlier versions have not been tested. The simplest way to install Feature-engine is from PyPI with pip, Python's preferred package installer:

```
$ 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 with new or updated transformers. To upgrade Feature-engine to the latest version, use pip like this:

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

If you're using Anaconda, you can take advantage of the conda utility to install the Anaconda Feature-engine package:

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

## 7.3 Getting Help

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

1. The docs (you're here!).
2. Stack Overflow. If you ask a question, please tag it with "feature-engine".
3. If you are enrolled in the Feature Engineering for Machine Learning course in Udemy , post a question in a relevant section.
4. Join our mailing list.
5. Ask a question in the repo by filing an issue.

## 7.4 About

Coming Soon!

## 7.5 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
```

(continues on next page)

(continued from previous page)

```python
varnames = ['A'+str(s) for s in range(1,17)]
data.columns = varnames

# replace ? by np.nan
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)
```

## 7.6 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.

### 7.6.1 MeanMedianImputer

#### API Reference

**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.

We can pass a list of variables to be imputed. Alternatively, the MeanMedianImputer() will automatically select all variables of type numeric in the training set.

The imputer:

- first calculates the mean / median values of the variables (fit).
- Then replaces the missing data with the estimated mean / median (transform).

**Parameters**

**imputation_method** [str, default=median] Desired method of imputation. Can take 'mean' or 'median'.

**variables** [list, default=None] The list of variables to be imputed. If None, the imputer will select all variables of type numeric.

### Attributes

| | |
|---|---|
| **imputer_dict_ :** | Dictionary with the mean or median values per variable. |

### Methods

| | |
|---|---|
| **fit:** | Learn the mean or median values. |
| **transform:** | Impute missing data. |
| **fit_transform:** | Fit to the data, then transform it. |

`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.

**Returns**

**self**

**Raises**

**TypeError**

- If the input is not a Pandas DataFrame
- If any of the user provided variables are not numerical

**ValueError** If there are no numerical variables in the df or the df is empty

`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** [pandas dataframe of shape = [n_samples, n_features]] The dataframe without missing values in the selected variables.

**rtype** `DataFrame` ..

**Raises**

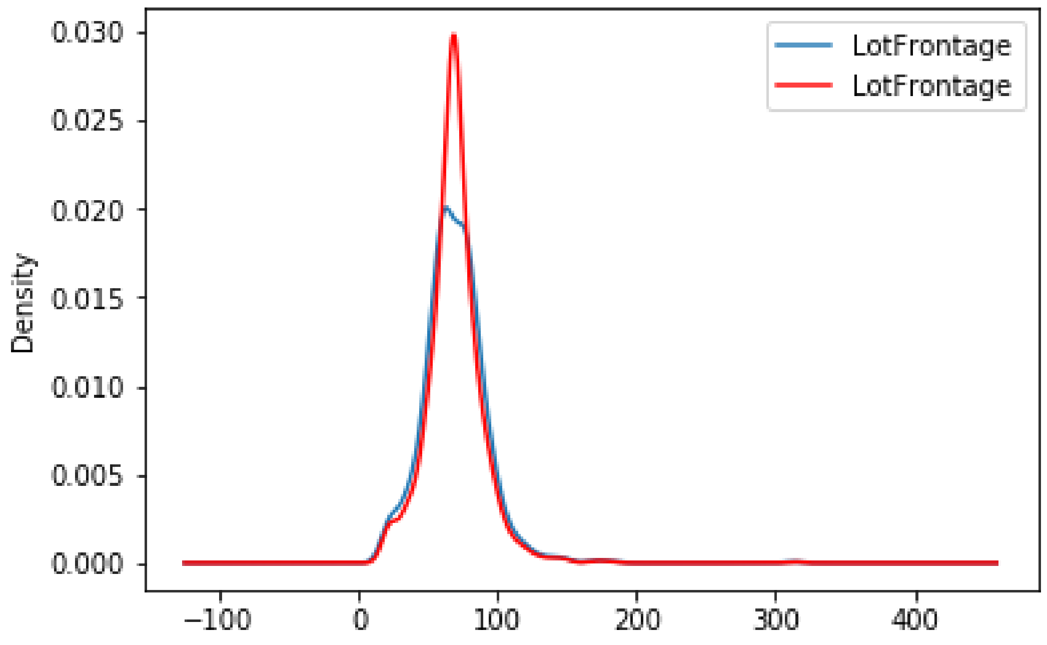
**TypeError** If the input is not a Pandas DataFrame

**ValueError** If the dataframe is not of same size as that used in fit()

### Example

The MeanMedianImputer() replaces missing data with the mean or median of the variable. It works only with numerical variables. A list of variables to impute can be indicated, or the imputer will automatically select all numerical variables in the train set. For more details, check the API Reference below.

```
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)

# 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')
```



### 7.6.2 ArbitraryNumberImputer

#### API Reference

**class** feature_engine.imputation.**ArbitraryNumberImputer**(*arbitrary_number=999, variables=None, imputer_dict=None*)

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

We can impute all variables with the same number, in which case we need to define the variables to impute in `variables` and the imputation number in `arbitrary_number`. Alternatively, we can pass a dictionary of variable and numbers to use for their imputation.

For example, we can impute varA and varB with 99 like this:

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

Xt = transformer.fit_transform(X)
```

Alternatively, we can impute varA with 1 and varB with 99 like this:

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

Xt = transformer.fit_transform(X)
```

**Parameters**

**arbitrary_number** [int or float, default=999] The number to be used to replace missing data.

**variables** [list, default=None] The list of variables to be imputed. If None, the imputer will find and select all numerical type 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.

### Attributes

| | |
|---|---|
| **imputer_dict_ :** | Dictionary with the values to replace NAs in each variable. |

**See also:**

*`feature_engine.imputation.EndTailImputer`*

### Methods

| | |
|---|---|
| **fit:** | This transformer does not learn parameters. |
| **transform:** | Impute missing data. |
| **fit_transform:** | Fit to the data, then transform it. |

**`fit`** (*X*, *y=None*)

This method does not learn any parameter. Checks dataframe and finds numerical variables, or checks that the variables entered by user are numerical.

**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.

**Returns**

**self**

**Raises**

**TypeError**

- If the input is not a Pandas DataFrame
- If any of the user provided variables are not numerical

**ValueError** If there are no numerical variables in the df or the df is empty

**`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** [pandas dataframe of shape = [n_samples, n_features]] The dataframe without missing values in the selected variables.

**rtype** DataFrame ..

**Raises**

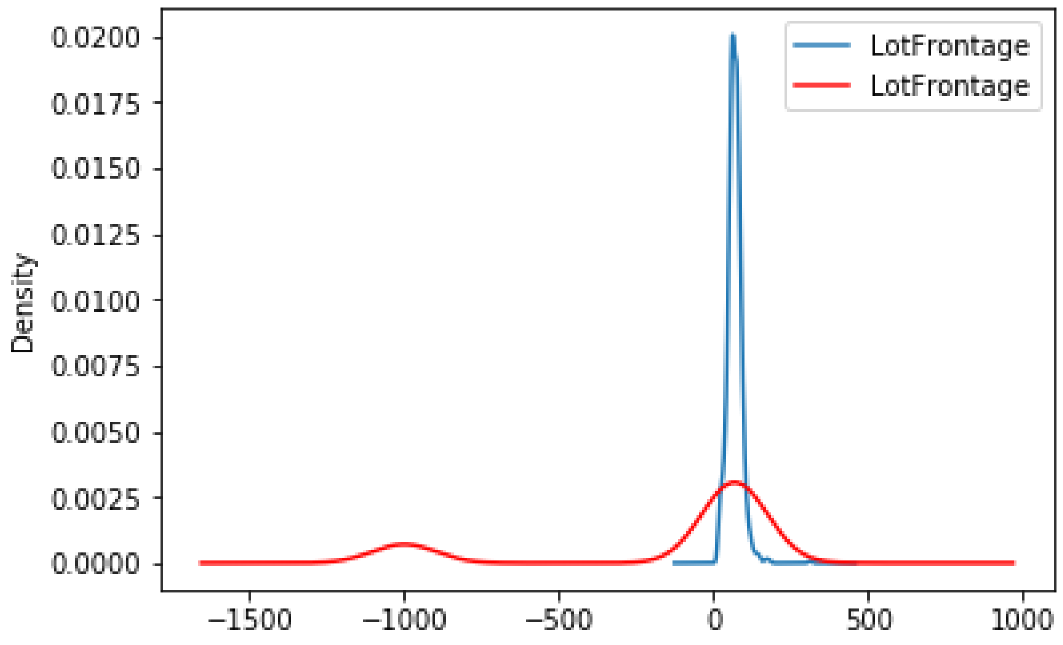
**TypeError** If the input is not a Pandas DataFrame

**ValueError** If the dataframe is not of same size as that used in fit()

## Example

The ArbitraryNumberImputer() replaces missing data with an arbitrary value determined by the user. It works only with numerical variables. A list of variables can be indicated, or the imputer will automatically select all numerical variables in the train set. A dictionary with variables and their arbitrary values can be indicated to use different arbitrary values for variables.

```
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, random_
↪state=0)

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

# fit the imputer
arbitrary_imputer.fit(X_train)

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



## 7.6.3 EndTailImputer

### API Reference

**class** feature_engine.imputation.**EndTailImputer**(*imputation_method='gaussian', tail='right', fold=3, variables=None*)

The EndTailImputer() transforms features by replacing missing data by a value at either tail of the distribution. Ti works only with numerical variables.

The user can indicate the variables to be imputed in a list. Alternatively, the EndTailImputer() will automatically find and select all variables of type numeric.

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’.

The imputer then replaces the missing data with the estimated values (transform).

**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 variables to be imputed. If None, the imputer will find and select all variables of type numeric.

## Attributes

| | |
|---|---|
| **imputer_dict_:** | Dictionary with the values at the end of the distribution per variable. |

## Methods

| | |
|---|---|
| **fit:** | Learn values to replace missing data. |
| **transform:** | Impute missing data. |
| **fit_transform:** | Fit to the data, then transform it. |

`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.

**Returns**

**self**

**Raises**

**TypeError**

- If the input is not a Pandas DataFrame
- If any of the user provided variables are not numerical

**ValueError** If there are no numerical variables in the df or the df is empty

`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** [pandas dataframe of shape = [n_samples, n_features]] The dataframe without missing values in the selected variables.

**rtype** `DataFrame` ..

**Raises**

**TypeError** If the input is not a Pandas DataFrame

**ValueError** If the dataframe is not of same size as that used in fit()

## Example

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. See the API Reference below for more details.

The user decides whether the missing data should be placed at the right or left tail of the variable distribution.

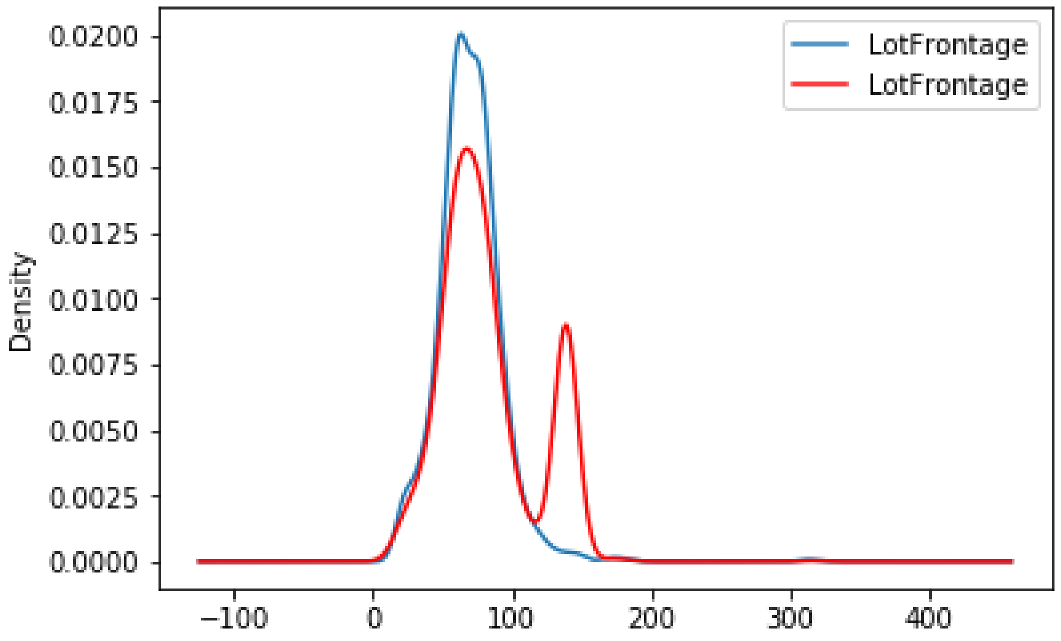
It works only with numerical variables. A list of variables can be indicated, or the imputer will automatically select all numerical variables in the train set.

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

# 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)

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

fig = plt.figure()
ax = fig.add_subplot(111)
X_train['LotFrontage'].plot(kind='kde', ax=ax)
```

(continues on next page)

(continued from previous page)

```
train_t['LotFrontage'].plot(kind='kde', ax=ax, color='red')
lines, labels = ax.get_legend_handles_labels()
ax.legend(lines, labels, loc='best')
```



### 7.6.4 CategoricalImputer

#### API Reference

**class** feature_engine.imputation.**CategoricalImputer**(*imputation_method='missing', fill_value='Missing', variables=None, return_object=False*)

The CategoricalImputer() replaces missing data in categorical variables by a string like 'Missing' or any other entered by the user. Alternatively, it replaces missing data by the most frequent category.

The CategoricalVariableImputer() works only with categorical variables.

The user can pass a list with the variables to be imputed. Alternatively, the CategoricalImputer() will automatically find and select all variables of type object.

**Note**

If you want to impute numerical variables with this transformer, you first need to cast them as object. It may well be that after the imputation, they are re-casted by pandas as numeric. Thus, if planning to do categorical encoding with feature-engine to this variables after the imputation, make sure to return the variables as object by setting `return_object=True`.

**Parameters**

**imputation_method** [str, default=missing] Desired method of imputation. Can be 'frequent' or 'missing'.

**fill_value** [str, default='Missing'] Only used when `imputation_method='missing'`. Can be used to set a user-defined value to replace the missing data.

**variables** [list, default=None] The list of variables to be imputed. If None, the imputer will find and select all object type variables.

**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.

### Attributes

| | |
|---|---|
| **imputer_dict_:** | Dictionary with most frequent category or string per variable. |

### Methods

| | |
|---|---|
| **fit:** | Learn more frequent category, or assign string to variable. |
| **transform:** | Impute missing data. |
| **fit_transform:** | Fit to the data, than transform it. |

**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.

**Returns**

**self**

**Raises**

**TypeError**

- If the input is not a Pandas DataFrame.
- If any user provided variable is not categorical

**ValueError** If there are no categorical variables in the df or the df is empty

**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** [pandas dataframe of shape = [n_samples, n_features]] The dataframe without missing values in the selected variables.

**rtype** `DataFrame` ..

**Raises**

**TypeError** If the input is not a Pandas DataFrame

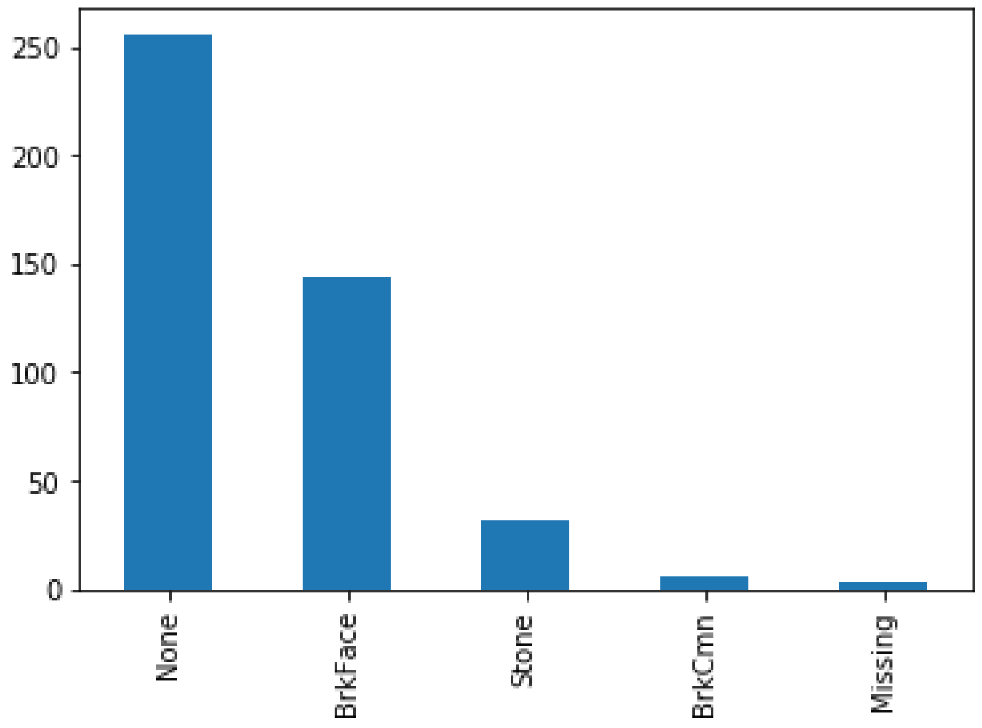
**ValueError** If the dataframe is not of same size as that used in fit()

### Example

The CategoricalImputer() replaces missing data in categorical variables with the string ‘Missing’ or by the most frequent category.

It works only with categorical variables. A list of variables can be indicated, or the imputer will automatically select all categorical variables in the train set.

```
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 CategoricalImputer

# 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
imputer = CategoricalImputer(variables=['Alley', 'MasVnrType'])

# fit the imputer
imputer.fit(X_train)

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

test_t['MasVnrType'].value_counts().plot.bar()
```



### 7.6.5 RandomSampleImputer

#### API Reference

**class** `feature_engine.imputation.`**`RandomSampleImputer`**(*variables=None, random_state=None, seed='general', seeding_method='add'*)

The RandomSampleImputer() replaces missing data in each feature with a random sample extracted from the variables in the training set. The RandomSampleImputer() works with both numerical and categorical variables.

**Note**

Random samples will vary from execution to execution. This may affect the results of your work. Remember to set a seed before running the RandomSampleImputer().

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

If seed = 'general' then the random_state can be either None or an integer. The seed will be used as the random_state and all observations will be imputed in one go. This is equivalent to `pandas.sample(n, random_state=seed)` where n is the number of observations with missing data.

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 seeding variable values, and passed to the random_state. Then, a value will be extracted from the train set using that seed and used to replace the NAN in 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 more details on why this functionality is important refer to the course Feature Engineering for Machine Learning in Udemy: https://www.udemy.com/feature-engineering-for-machine-learning/

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

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.

**Parameters**

**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 variables 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'.

**variables** [list, default=None] The list of variables to be imputed. If None, the imputer will select all variables in the train set.

## Attributes

| | |
|---|---|
| **X_ :** | Copy of the training dataframe from which to extract the random samples. |

## Methods

| | |
|---|---|
| **fit:** | Make a copy of the dataframe |
| **transform:** | Impute missing data. |
| **fit_transform:** | Fit to the data, then transform it. |

`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. Only a copy of the indicated variables will be stored in the transformer.

**y** [None] y is not needed in this imputation. You can pass None or y.

**Returns**

**self**

**Raises**

**TypeError** If the input is not a Pandas DataFrame

**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** [pandas dataframe of shape = [n_samples, n_features]] The dataframe without missing values in the transformed variables.

**rtype** `DataFrame` ..

**Raises**

**TypeError** If the input is not a Pandas DataFrame

## Example

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.

A seed can be set to a pre-defined number and all observations will be replaced in batch. Alternatively, a seed can be set using the values of 1 or more numerical variables. In this case, the observations will be imputed individually, one at a time, using the values of the variables as a seed.

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 observation will be replaced using pandas sample as follows:

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

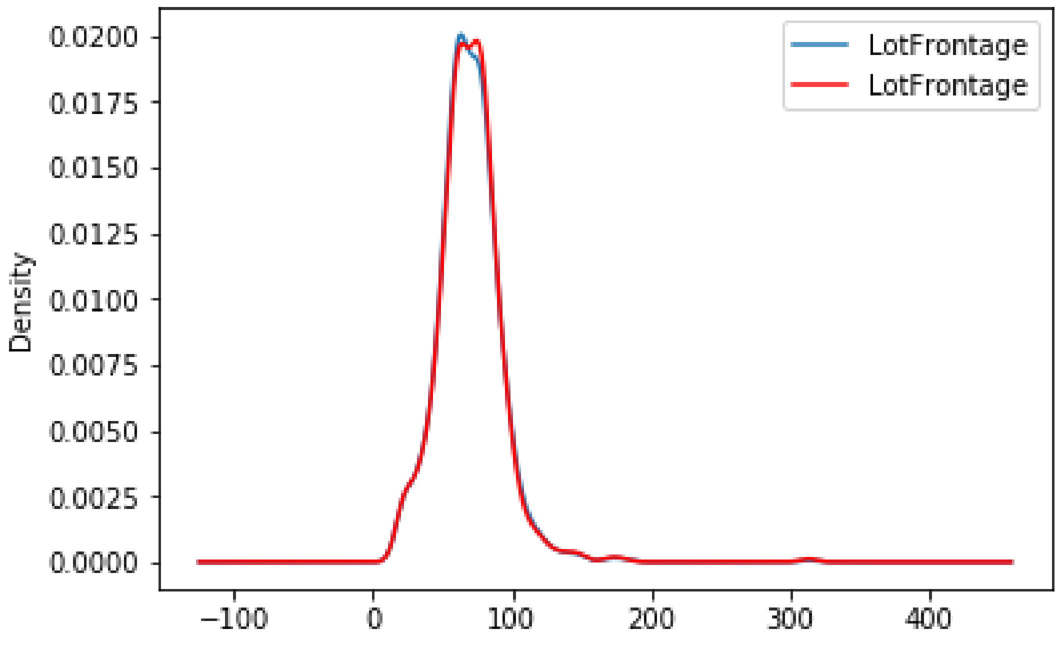
More details on how to use the RandomSampleImputer():

```
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
)
```

(continues on next page)

(continued from previous page)

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

# fit the imputer
imputer.fit(X_train)

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



## 7.6.6 AddMissingIndicator

### API Reference

**class** `feature_engine.imputation.`**`AddMissingIndicator`**(*missing_only=True, variables=None*)

The AddMissingIndicator() adds an additional column or binary variable that indicates if data is missing.

AddMissingIndicator() will add as many missing indicators as variables indicated by the user, or variables with missing data in the train set.

The AddMissingIndicator() works for both numerical and categorical variables. The user can pass a list with the variables for which the missing indicators should be added as a list. Alternatively, the imputer will select and add missing indicators to all variables in the training set that show missing data.

**Parameters**

**missing_only** [bool, defatult=True] Indicates 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 be imputed. If None, the imputer will find and select all variables with missing data.

****Note****

**The transformer will first select all variables or all user entered**

**variables and if how=missing_only, it will re-select from the original group**

**only those that show missing data in during fit.**

## Attributes

| | |
|---|---|
| **variables_:** | List of variables for which the missing indicators will be created. |

## Methods

| | |
|---|---|
| **fit:** | Learn the variables for which the missing indicators will be created |
| **transform:** | Add the missing indicators. |
| **fit_transform:** | Fit to the data, then trasnform it. |

`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.

**Returns**

**self.variables_** [list] The list of variables for which missing indicators will be added.

**Raises**

**TypeError** If the input is not a Pandas DataFrame

`transform`(*X*)

Add the binary missing indicators.

**Parameters**

**X** [pandas dataframe of shape = [n_samples, n_features]] The dataframe to be transformed.

**Returns**

**X_transformed** [pandas dataframe of shape = [n_samples, n_features]] The dataframe containing the additional binary variables. Binary variables are named with the original variable name plus '_na'.

**rtype** `DataFrame` ..

## Example

The AddMissingIndicator() adds a binary variable indicating if observations are missing (missing indicator). It adds a missing indicator for both categorical and numerical variables. A list of variables for which to add a missing indicator can be passed, or the imputer will automatically select all variables.

The imputer has the option to select if binary variables should be added to all variables, or only to those that show missing data in the train set, by setting the option how='missing_only'.

```
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)

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

# fit the imputer
addBinary_imputer.fit(X_train)

# 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 |

### 7.6.7 DropMissingData

#### API Reference

**class** `feature_engine.imputation.`**`DropMissingData`**(*missing_only=True*, *variables=None*)

works for both numerical and categorical variables. DropMissingData can automatically select all the variables, or alternatively, all the variables with missing data in the train set. Then the observations with NA will be dropped for these variable groups.

The user has the option to indicate for which variables the observations with NA should be removed.

**Parameters**

**missing_only** [bool, default=True] If true, missing observations will be dropped only for the variables that were seen to have NA in the train set, during fit. If False, observations with NA will be dropped from all variables.

**variables** [list, default=None] The list of variables to be imputed. If None, the imputer will find and select all variables with missing data.

****Note****

**The transformer will first select all variables or all user entered**

**variables and if `missing_only=True`, it will re-select from the original group**

**only those that show missing data in during fit, that is in the train set.**

#### Attributes

| | |
|---|---|
| **variables_:** | List of variables for which the rows with NA will be deleted. |

#### Methods

| | |
|---|---|
| **fit:** | Learn the variables for which the rows with NA will be deleted |
| **transform:** | Remove observations with NA |
| **fit_transform:** | Fit to the data, then transform it. |
| **return_na_data:** | Returns the dataframe with the rows that contain NA . |

**`fit`**(*X*, *y=None*)

Learn the variables for which the rows with NA will be deleted.

**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.

**Returns**

**self**

**Raises**

**TypeError** If the input is not a Pandas DataFrame

**return_na_data**(*X*)

Returns the subset of the dataframe which contains the rows with missing values. This method could be useful in production, in case we want to store the observations that will not be fed into the model.

**Parameters**

**X** [pandas dataframe of shape = [n_samples, n_features]] The dataset to from which rows containing NA should be retained.

**Returns**

**X** [pandas dataframe of shape = [obs_with_na, features]] The cdataframe portion that contains only the rows with missing values.

**rtype** `DataFrame` ..

**Raises**

**TypeError** If the input is not a Pandas DataFrame

**transform**(*X*)

Remove rows with missing values.

**Parameters**

**X** [pandas dataframe of shape = [n_samples, n_features]] The dataframe to be transformed.

**Returns**

**X_transformed** [pandas dataframe] The complete case dataframe for the selected variables, of shape [n_samples - rows_with_na, n_features]

**rtype** `DataFrame` ..

## Example

DropMissingData() deletes rows with NA values. It works with numerical and categorical variables. The user can pass a list of variables for which to delete rows with NA. Alternatively, DropMissingData() will default to all variables. The trasformer has the option to learn the variables with NA in the train set, and then remove observations with NA in only those variables.

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

from feature_engine.imputation import DropMissingData

# 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
missingdata_imputer = DropMissingData(variables=['LotFrontage', 'MasVnrArea'])
```

(continues on next page)

(continued from previous page)

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

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

# Number of NA before the transformation:
X_train['LotFrontage'].isna().sum()
```

```
189
```

```
# Number of NA after the transformation:
    train_t['LotFrontage'].isna().sum()
```

```
0
```

```
# Number of rows before and after transformation
print(X_train.shape)
    print(train_t.shape)
```

```
(1022, 79)
(829, 79)
```

## 7.7 Categorical Variable Encoding

Feature-engine's categorical encoders replace variable strings by estimated or arbitrary numbers.

Feature-engine's categorical encoders work only with categorical variables. A list of variables can be indicated, or the encoders will automatically select and encode all categorical variables in the train set.

### 7.7.1 OneHotEncoder

#### API Reference

**class** `feature_engine.encoding.`**`OneHotEncoder`**(*top_categories=None*, *variables=None*, *drop_last=False*)

One hot encoding consists in replacing the categorical variable by a combination of binary variables which take value 0 or 1, to indicate if a certain category is present in an observation. The binary variables are also known as dummy variables.

For example, from the categorical variable "Gender" with categories "female" and "male", we can generate the boolean variable "female", which takes 1 if the observation is female or 0 otherwise. We can also generate the variable "male", which takes 1 if the observation is "male" and 0 otherwise.

The encoder can create k binary variables per categorical variable, k being the number of unique categories, or alternatively k-1 to avoid redundant information. This behaviour can be specified using the parameter `drop_last`.

The encoder has the additional option to generate binary variables only for the top n 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`.

**Note**

Only when creating binary variables for all categories of the variable, we can specify if we want to encode into k or k-1 binary variables, where k is the number if unique categories. If we encode only the top n most popular categories, the encoder will create only n binary variables per categorical variable. Observations that do not show any of these popular categories, will have 0 in all the binary variables.

The encoder will encode only categorical variables (type 'object'). A list of variables can be passed as an argument. If no variables are passed as argument, the encoder will find and encode categorical variables (object type).

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.

**Parameters**

**top_categories** [int, default=None] If None, a dummy variable will be created for each category of the variable. Alternatively, we can indicate in `top_categories` 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.

**variables** [list] The list of categorical variables to encode. If None, the encoder will find and select all object type variables in the train set.

**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 of the list (k-1 dummies).

## Attributes

| | |
|---|---|
| **encoder_dict_ :** | Dictionary with the categories for which dummy variables will be created. |

### 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]

### Methods

| | |
|---|---|
| **fit:** | Learn the unique categories per variable |
| **transform:** | Replace the categorical variables by the binary variables. |
| **fit_transform:** | Fit to the data, then transform it. |

**`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.

**Returns**

**self**

**Raises**

**TypeError**

- If the input is not a Pandas DataFrame.
- If any user provided variable is not categorical

**ValueError**

- If there are no categorical variables in the df or the df is empty
- If the variable(s) contain null values

**`inverse_transform`**(*X*)

inverse_transform is not implemented for this transformer.

**`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** [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` ..

**Raises**

**TypeError** If the input is not a Pandas DataFrame

**ValueError**

- If the variable(s) contain null values.
- If the dataframe is not of same size as that used in fit()

## Example

The OneHotEncoder() replaces categorical variables by a set of binary variables, one per unique category. The encoder has the option to create k or k-1 binary variables, where k is the number of unique categories.

The encoder can also create binary variables for the n most popular categories, n being determined by the user. This means, if we encode the 6 more popular categories, we will only create binary variables for those categories, and the rest will be dropped.

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

from feature_engine.encoding import OneHotEncoder

# 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)
        return data

data = load_titanic()

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

# set up the encoder
encoder = OneHotEncoder( top_categories=2, variables=['pclass', 'cabin', 'embarked'],␣
↪drop_last=False)

# fit the encoder
encoder.fit(X_train)

# transform the data
```

(continues on next page)

(continued from previous page)

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

encoder.encoder_dict_
```

```
{'pclass': [3, 1], 'cabin': ['n', 'C'], 'embarked': ['S', 'C']}
```

### 7.7.2 CountFrequencyEncoder

#### API Reference

**class** feature_engine.encoding.**CountFrequencyEncoder**(*encoding_method='count'*, *variables=None*)

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 (type 'object'). A list of variables to encode can be passed as argument. Alternatively, the encoder will find and encode all categorical variables (object type).

The encoder first maps the categories to the counts or frequencies for each variable (fit). The encoder then replaces the categories by those mapped numbers (transform).

**Parameters**

**encoding_method** [str, default='count'] Desired method of encoding.

'count': number of observations per category

'frequency': percentage of observations per category

**variables** [list] The list of categorical variables that will be encoded. If None, the encoder will find and transform all object type variables.

#### Attributes

| | |
|---|---|
| **encoder_dict_:** | Dictionary with the count or frequency} per category, per variable. |

**See also:**

*feature_engine.encoding.RareLabelEncoder*

### 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().

### Methods

| | |
|---|---|
| **fit:** | Learn the count or frequency per category, per variable. |
| **transform:** | Encode the categories to numbers. |
| **fit_transform:** | Fit to the data, then transform it. |
| **inverse_transform:** | Encode the numbers into the original categories. |

**`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.

**Returns**

**self**

**Raises**

**TypeError**

- If the input is not a Pandas DataFrame.
- If any user provided variable is not categorical

**ValueError**

- If there are no categorical variables in the df or the df is empty
- If the variable(s) contain null 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** [pandas dataframe of shape = [n_samples, n_features].] The un-transformed dataframe, with the categorical variables containing the original values.

**rtype** `DataFrame` ..

**Raises**

**TypeError**

- If the input is not a Pandas DataFrame

**ValueError**

- If the variable(s) contain null values

- If the dataframe is not of same size as that used in fit()

**transform**(*X*)

Replace categories with the learned parameters.

**Parameters**

**X** [pandas dataframe of shape = [n_samples, n_features].] The dataset to transform.

**Returns**

**X** [pandas dataframe of shape = [n_samples, n_features].] The dataframe containing the categories replaced by numbers.

**rtype** `DataFrame` ..

**Raises**

**TypeError** If the input is not a Pandas DataFrame

**ValueError**

- If the variable(s) contain null values
- If dataframe is not of same size as that used in fit()

**Warning** If after encoding, NAN were introduced.

## Example

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

from feature_engine.encoding import CountFrequencyEncoder

# 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)
        return data

data = load_titanic()

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

# set up the encoder
encoder = CountFrequencyEncoder(encoding_method='frequency',
                         variables=['cabin', 'pclass', 'embarked'])

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

(continues on next page)

(continued from previous page)

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

encoder.encoder_dict_
```

```
{'cabin': {'n': 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': 0.19759825327510916,
  'Q': 0.0906113537117904}}
```

### 7.7.3 OrdinalEncoder

#### API Reference

**class** `feature_engine.encoding.`**`OrdinalEncoder`**(*encoding_method='ordered', variables=None*)

The OrdinalCategoricalEncoder() 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.

**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.

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

The encoder will encode only categorical variables (type 'object'). A list of variables can be passed as an argument. If no variables are passed, the encoder will find and encode all categorical variables (type 'object').

The encoder first maps the categories to the numbers for each variable (fit). The encoder then transforms the categories to the mapped numbers (transform).

**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 select all object type variables.

### Attributes

| | |
|---|---|
| **encoder_dict_ :** | Dictionary with the ordinal number per category, per variable. |

**See also:**

*feature_engine.encoding.RareLabelEncoder*

### 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().

### References

Encoding into integers ordered following target mean was discussed in the following talk at PyData London 2017:

[1]

### Methods

| | |
|---|---|
| **fit:** | Find the integer to replace each category in each variable. |
| **transform:** | Encode the categories to numbers. |
| **fit_transform:** | Fit to the data, then transform it. |
| **inverse_transform:** | Encode the numbers into the original categories. |

**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.

**Returns**

**self**

**Raises**

**TypeError**

- If the input is not a Pandas DataFrame.
- If any user provided variable is not categorical

**ValueError**

- If there are no categorical variables in the df or the df is empty
- If the variable(s) contain null 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** [pandas dataframe of shape = [n_samples, n_features].] The un-transformed dataframe, with the categorical variables containing the original values.

**rtype** `DataFrame` ..

**Raises**

**TypeError**

- If the input is not a Pandas DataFrame

**ValueError**

- If the variable(s) contain null values
- If the dataframe is not of same size as that used in fit()

**transform**(*X*)

Replace categories with the learned parameters.

**Parameters**

**X** [pandas dataframe of shape = [n_samples, n_features].] The dataset to transform.

**Returns**

**X** [pandas dataframe of shape = [n_samples, n_features].] The dataframe containing the categories replaced by numbers.

**rtype** `DataFrame` ..

**Raises**

**TypeError** If the input is not a Pandas DataFrame

**ValueError**

- If the variable(s) contain null values
- If dataframe is not of same size as that used in fit()

**Warning** If after encoding, NAN were introduced.

## Example

The OrdinalEncoder() replaces the categories by digits, starting from 0 to k-1, where k is the number of different categories. If we select “arbitrary”, then the encoder will assign numbers as the labels appear in the variable (first come first served). If we 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.

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

from feature_engine.encoding import OrdinalEncoder

# 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)
        return data

data = load_titanic()

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

# set up the encoder
encoder = OrdinalEncoder(encoding_method='ordered', variables=['pclass', 'cabin',
↪'embarked'])

# fit the encoder
encoder.fit(X_train, y_train)

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

encoder.encoder_dict_
```

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

### 7.7.4 MeanEncoder

#### API Reference

**class** `feature_engine.encoding.`**`MeanEncoder`**(*variables=None*)

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.

The encoder will encode only categorical variables (type 'object'). A list of variables can be passed as an argument. If no variables are passed as argument, the encoder will find and encode all categorical variables (object type).

The encoder first maps the categories to the numbers for each variable (fit). The encoder then replaces the categories with the mapped numbers (transform).

**Parameters**

**variables** [list, default=None] The list of categorical variables to encode. If None, the encoder will find and select all object type variables.

#### Attributes

| | |
|---|---|
| **encoder_dict_ :** | Dictionary with the target mean value per category per variable. |

**See also:**

*`feature_engine.encoding.RareLabelEncoder`*

#### 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().

#### References

[1]

#### Methods

| | |
|---|---|
| **fit:** | Learn the target mean value per category, per variable. |
| **transform:** | Encode the categories to numbers. |
| **fit_transform:** | Fit to the data, then transform it. |
| **inverse_transform:** | Encode the numbers into the original categories. |

**`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.

**Returns**

**self**

**Raises**

**TypeError**

- If the input is not a Pandas DataFrame.
- If any user provided variable is not categorical

**ValueError**

- If there are no categorical variables in the df or the df is empty
- If the variable(s) contain null 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** [pandas dataframe of shape = [n_samples, n_features].] The un-transformed dataframe, with the categorical variables containing the original values.

**rtype** `DataFrame` ..

**Raises**

**TypeError**

- If the input is not a Pandas DataFrame

**ValueError**

- If the variable(s) contain null values
- If the dataframe is not of same size as that used in fit()

**transform**(*X*)

Replace categories with the learned parameters.

**Parameters**

**X** [pandas dataframe of shape = [n_samples, n_features].] The dataset to transform.

**Returns**

**X** [pandas dataframe of shape = [n_samples, n_features].] The dataframe containing the categories replaced by numbers.

**rtype** `DataFrame` ..

**Raises**

**TypeError** If the input is not a Pandas DataFrame

**ValueError**

- If the variable(s) contain null values
- If dataframe is not of same size as that used in fit()

**Warning** If after encoding, NAN were introduced.

### Example

The MeanEncoder() replaces categories with the mean of the target per category. For example, if we are trying to predict default rate, and our data has the variable city, with categories, London, Manchester and Bristol, and the default rate per city is 0.1, 0.5, and 0.3, respectively, the encoder will replace London by 0.1, Manchester by 0.5 and Bristol by 0.3.

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

from feature_engine.encoding import MeanEncoder

# 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)
        return data

data = load_titanic()

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

# set up the encoder
encoder = MeanEncoder(variables=['cabin', 'pclass', 'embarked'])

# fit the encoder
encoder.fit(X_train, y_train)

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

encoder.encoder_dict_
```

```
{'cabin': {'A': 0.5294117647058824,
  'B': 0.7619047619047619,
  'C': 0.5633802816901409,
  'D': 0.71875,
  'E': 0.71875,
  'F': 0.6666666666666666,
  'G': 0.5,
```

(continues on next page)

(continued from previous page)

```
  'T': 0.0,
  'n': 0.30484330484330485},
 'pclass': {1: 0.6173913043478261,
  2: 0.43617021276595747,
  3: 0.25903614457831325},
 'embarked': {'C': 0.5580110497237569,
  'Q': 0.37349397590361444,
  'S': 0.3389570552147239}}
```

### 7.7.5 WoEEncoder

#### API Reference

**class** feature_engine.encoding.**WoEEncoder**(*variables=None*)

The WoERatioCategoricalEncoder() 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 (type 'object'). A list of variables can be passed as an argument. If no variables are passed the encoder will find and encode all categorical variables (object type).

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).

**Note**

This categorical encoding is exclusive for binary classification.

**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**

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.

**Parameters**

**variables** [list, default=None] The list of categorical variables that will be encoded. If None, the encoder will find and select all object type variables.

### Attributes

| | |
|---|---|
| **encoder_dict_ :** | Dictionary with the WoE per variable. |

**See also:**

*feature_engine.encoding.RareLabelEncoder*

`feature_engine.discretisation`

### 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

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.

NAN are introduced when encoding categories that were not present in the training dataset. If this happens, try grouping infrequent categories using the RareLabelEncoder().

### Methods

| | |
|---|---|
| **fit:** | Learn the WoE per category, per variable. |
| **transform:** | Encode the categories to numbers. |
| **fit_transform:** | Fit to the data, then transform it. |
| **inverse_transform:** | Encode the numbers into the original categories. |

**fit** (*X*, *y*)

Learn the 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 [0,1].

**Returns**

**self**

**Raises**

**TypeError**

- If the input is not the Pandas DataFrame.
- If any user provided variables are not categorical.

**ValueError**

- If there are no categorical variables in df or df is empty

- If variable(s) contain null values.
- If y is not binary with values 0 and 1.
- If p(0) = 0 or p(1) = 0.

**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** [pandas dataframe of shape = [n_samples, n_features].] The un-transformed dataframe, with the categorical variables containing the original values.

**rtype** `DataFrame` ..

**Raises**

**TypeError**

- If the input is not a Pandas DataFrame

**ValueError**

- If the variable(s) contain null values
- If the dataframe is not of same size as that used in fit()

**transform**(*X*)

Replace categories with the learned parameters.

**Parameters**

**X** [pandas dataframe of shape = [n_samples, n_features].] The dataset to transform.

**Returns**

**X** [pandas dataframe of shape = [n_samples, n_features].] The dataframe containing the categories replaced by numbers.

**rtype** `DataFrame` ..

**Raises**

**TypeError** If the input is not a Pandas DataFrame

**ValueError**

- If the variable(s) contain null values
- If dataframe is not of same size as that used in fit()

**Warning** If after encoding, NAN were introduced.

### Example

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

from feature_engine.encoding import WoEEncoder, RareLabelEncoder

# 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)
        return data

data = load_titanic()

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

# set up a rare label encoder
rare_encoder = RareLabelEncoder(tol=0.03, n_categories=2, variables=['cabin', 'pclass
↪', 'embarked'])

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

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

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

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

woe_encoder.encoder_dict_
```

```
{'cabin': {'B': 1.6299623810120747,
'C': 0.7217038208351837,
'D': 1.405081209799324,
'E': 1.405081209799324,
'Rare': 0.7387452866900354,
'n': -0.35752781962490193},
'pclass': {1: 0.9453018143294478,
2: 0.21009172435857942,
3: -0.5841726684724614},
'embarked': {'C': 0.6999054533737715,
'Q': -0.05044494288988759,
'S': -0.20113381737960143}}
```

### 7.7.6 PRatioEncoder

#### API Reference

**class** `feature_engine.encoding.`**`PRatioEncoder`**(*encoding_method='ratio'*, *variables=None*)

The PRatioEncoder() replaces categories by the ratio of the probability of the target = 1 and the probability of the target = 0.

The target probability ratio is given by:

$$p(1)/p(0)$$

The log of the target probability ratio is:

$$log(p(1)/p(0))$$

**Note**

This categorical encoding is exclusive for binary classification.

For example in the variable colour, if the mean of the target = 1 for blue is 0.8 and the mean of the target = 0 is 0.2, blue will be replaced by: 0.8 / 0.2 = 4 if ratio is selected, or log(0.8/0.2) = 1.386 if log_ratio is selected.

Note: the division by 0 is not defined and the log(0) is not defined. Thus, if p(0) = 0 for the ratio encoder, or either p(0) = 0 or p(1) = 0 for log_ratio, in any of the variables, the encoder will return an error.

The encoder will encode only categorical variables (type 'object'). A list of variables can be passed as an argument. If no variables are passed the encoder will find and encode all categorical variables (object type).

The encoder first maps the categories to the numbers for each variable (fit). The encoder then transforms the categories into the mapped numbers (transform).

**Parameters**

**encoding_method** [str, default=woe] Desired method of encoding.

'ratio' : probability ratio

'log_ratio' : log probability ratio

**variables** [list, default=None] The list of categorical variables to encode. If None, the encoder will find and select all object type variables.

#### Attributes

| | |
|---|---|
| **encoder_dict_ :** | Dictionary with the probability ratio per category per variable. |

**See also:**

*`feature_engine.encoding.RareLabelEncoder`*

### 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().

### Methods

|  |  |
|---|---|
| **fit:** | Learn probability ratio per category, per variable. |
| **transform:** | Encode categories into numbers. |
| **fit_transform:** | Fit to the data, then transform it. |
| **inverse_transform:** | Encode the numbers into the original categories. |

**`fit`**(*X*, *y*)

Learn the numbers that should be used to replace the categories in each variable. That is the ratio of probability.

**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 [0,1].

**Returns**

**self**

**Raises**

**TypeError**

- If the input is not the Pandas DataFrame.
- If any user provided variables are not categorical.

**ValueError**

- If there are no categorical variables in df or df is empty
- If variable(s) contain null values.
- If y is not binary with values 0 and 1.
- If p(0) = 0 or any of p(0) or p(1) are 0.

**`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** [pandas dataframe of shape = [n_samples, n_features].] The un-transformed dataframe, with the categorical variables containing the original values.

**rtype** `DataFrame` ..

**Raises**

**TypeError**

- If the input is not a Pandas DataFrame

**ValueError**

- If the variable(s) contain null values
- If the dataframe is not of same size as that used in fit()

**transform**(*X*)

Replace categories with the learned parameters.

**Parameters**

**X** [pandas dataframe of shape = [n_samples, n_features].] The dataset to transform.

**Returns**

**X** [pandas dataframe of shape = [n_samples, n_features].] The dataframe containing the categories replaced by numbers.

**rtype** `DataFrame` ..

**Raises**

**TypeError** If the input is not a Pandas DataFrame

**ValueError**

- If the variable(s) contain null values
- If dataframe is not of same size as that used in fit()

**Warning** If after encoding, NAN were introduced.

## Example

The PRatioEncoder() replaces the labels by the ratio of probabilities. It only works for binary classification.

The target probability ratio is given by: p(1) / p(0)

The log of the target probability ratio is: np.log( p(1) / p(0) )

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

from feature_engine.encoding import PRatioEncoder, RareLabelEncoder

# 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)
        return data

data = load_titanic()

# 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(['survived', 'name', 'ticket'], axis=1),
                data['survived'], test_size=0.3, random_state=0)

# set up a rare label encoder
rare_encoder = RareLabelEncoder(tol=0.03, n_categories=2, variables=['cabin', 'pclass
↪', 'embarked'])

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

# set up a weight of evidence encoder
pratio_encoder = PRatioEncoder(encoding_method='ratio', variables=['cabin', 'pclass',
↪'embarked'])

# fit the encoder
pratio_encoder.fit(train_t, y_train)

# transform
train_t = pratio_encoder.transform(train_t)
test_t = pratio_encoder.transform(test_t)

pratio_encoder.encoder_dict_
```

```
{'cabin': {'B': 3.1999999999999993,
 'C': 1.2903225806451615
 'D': 2.5555555555555554,
 'E': 2.5555555555555554,
 'Rare': 1.3124999999999998,
 'n': 0.4385245901639344},
 'pclass': {1: 1.6136363636363635,
  2: 0.7735849056603774,
  3: 0.34959349593495936},
  'embarked': {'C': 1.2625000000000002,
  'Q': 0.5961538461538461,
  'S': 0.5127610208816704}}
```

## 7.7.7 DecisionTreeEncoder

### API Reference

**class** feature_engine.encoding.**DecisionTreeEncoder**(*encoding_method='arbitrary', cv=3, scoring='neg_mean_squared_error', param_grid=None, regression=True, random_state=None, variables=None*)

The DecisionTreeEncoder() encodes categorical variables with predictions of a decision tree model.

Each categorical feature is recoded by training a decision tree, typically of limited depth (2, 3 or 4) using that feature alone, and let the tree directly predict the target. The probabilistic predictions of this decision tree are used as the new values of the original categorical feature, that now is linearly (or at least monotonically) correlated with the target.

In practice, the categorical variable will be first encoded into integers with the OrdinalCategoricalEncoder(). The integers can be assigned arbitrarily to the categories or following the mean value of the target in each category. Then a decision tree will fit the resulting numerical variable to predict the target variable. Finally, the original categorical variable values will be replaced by the predictions of the decision tree.

Note that a decision tree is fit per every single categorical variable to encode.

**Parameters**

**encoding_method** [str, default='arbitrary'] The categorical encoding method that will be used to encode the original categories to numerical values.

'ordered': the categories are numbered in ascending order according to the target mean value per category.

'arbitrary' : categories are numbered arbitrarily.

**cv** [int, default=3] Desired number of cross-validation fold to be used to fit the decision tree.

**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

**regression** [boolean, default=True] Indicates whether the encoder should train a regression or a classification decision tree.

**param_grid** [dictionary, default=None] The list of parameters over which the decision tree should be optimised during the grid search. The param_grid can contain any of the permitted parameters for Scikit-learn's DecisionTreeRegressor() or DecisionTreeClassifier().

If None, then param_grid = {'max_depth': [1, 2, 3, 4]}.

**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 select all object type variables.

## Attributes

| | |
|---|---|
| **encoder_ :** | sklearn Pipeline containing the ordinal encoder and the decision tree. |

**See also:**

**`sklearn.ensemble.DecisionTreeRegressor`**

**`sklearn.ensemble.DecisionTreeClassifier`**

*`feature_engine.discretisation.DecisionTreeDiscretiser`*

*`feature_engine.encoding.RareLabelEncoder`*

### Notes

The authors designed this method originally, to work with numerical variables. We can replace numerical variables by the preditions of a decision tree utilising the DecisionTreeDiscretiser().

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]

### Methods

| | |
|---|---|
| **fit:** | Fit a decision tree per variable. |
| **transform:** | Replace categorical variable by the predictions of the decision tree. |
| **fit_transform:** | Fit to the data, then transform it. |

`fit` (*X*, *y=None*)

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.

**Returns**

**self**

**Raises**

**TypeError**

- If the input is not a Pandas DataFrame.
- If any user provided variable is not categorical

**ValueError**

- If there are no categorical variables in the df or the df is empty
- If the variable(s) contain null values

`inverse_transform` (*X*)

inverse_transform is not implemented for this transformer yet.

`transform` (*X*)

Replace categorical variable by the predictions of the decision tree.

**Parameters**

**X** [pandas dataframe of shape = [n_samples, n_features]] The input samples.

**Returns**

**X** [pandas dataframe of shape = [n_samples, n_features].] Dataframe with variables encoded with decision tree predictions.

**rtype** `DataFrame` ..

**Raises**

**TypeError** If the input is not a Pandas DataFrame

**ValueError**

- If the variable(s) contain null values
- If dataframe is not of same size as that used in fit()

**Warning** If after encoding, NAN were introduced.

## Example

The DecisionTreelEncoder() replaces categories in the variable with the predictions of a decision tree. The transformer first encodes categorical variables into numerical variables using ordinal encoding. 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. 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.

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

from feature_engine.encoding import DecisionTreeEncoder

# 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)
        return data

data = load_titanic()

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

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

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

```
# set up the encoder
encoder = DecisionTreeEncoder(variables=['cabin', 'pclass', 'embarked'], random_
↪state=0)

# fit the encoder
encoder.fit(X_train, y_train)

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

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

```
     cabin    pclass  embarked
501   0.304843  0.307580  0.338957
588   0.304843  0.307580  0.338957
402   0.304843  0.307580  0.558011
1193  0.304843  0.307580  0.373494
686   0.304843  0.307580  0.373494
971   0.304843  0.307580  0.373494
117   0.649533  0.617391  0.558011
540   0.304843  0.307580  0.338957
294   0.649533  0.617391  0.558011
261   0.649533  0.617391  0.338957
```

## 7.7.8 RareLabelEncoder

### API Reference

**class** feature_engine.encoding.**RareLabelEncoder**(*tol=0.05*, *n_categories=10*, *max_n_categories=None*, *variables=None*, *replace_with='Rare'*)

The RareLabelCategoricalEncoder() groups rare / 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 (type 'object'). A list of variables can be passed as an argument. If no variables are passed as argument, the encoder will find and encode all categorical variables (object type).

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).

**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.

**variables** [list, default=None] The list of categorical variables to encode. If None, the encoder will find and select all object type variables.

**replace_with** [string, default='Rare'] The category name that will be used to replace infrequent categories.

## Attributes

| | |
|---|---|
| **encoder_dict_:** | Dictionary with the frequent categories, i.e.., those that will be kept, per variable. |

## Methods

| | |
|---|---|
| **fit:** | Find frequent categories. |
| **transform:** | Group rare categories |
| **fit_transform:** | Fit to data, then transform it. |

**`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.

**Returns**

**self**

**Raises**

**TypeError**

- If the input is not a Pandas DataFrame.
- If any user provided variable is not categorical

**ValueError**

- If there are no categorical variables in the df or the df is empty
- If the variable(s) contain null values

**Warning** If the number of categories in any one variable is less than the indicated in `n_categories`.

**`inverse_transform`**(*X*)

inverse_transform is not implemented for this transformer yet.

**`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` ..

**Raises**

**TypeError** If the input is not a Pandas DataFrame

**ValueError**

- If the variable(s) contain null values
- If dataframe is not of same size as that used in fit()

### Example

The RareLabelEncoder() groups infrequent categories altogether 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 show to be preserved and the minimum number of unique categories a variable should have to be re-grouped.

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

from feature_engine.encoding import RareLabelEncoder

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)
    return data

data = load_titanic()

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

# set up the encoder
encoder = RareLabelEncoder(tol=0.03, n_categories=2, variables=['cabin', 'pclass',
↪'embarked'],
                           replace_with='Rare')

# fit the encoder
encoder.fit(X_train)

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

(continues on next page)

(continued from previous page)

```
test_t = encoder.transform(X_test)

encoder.encoder_dict_
```

```
{'cabin': Index(['n', 'C', 'B', 'E', 'D'], dtype='object'),
 'pclass': array([2, 3, 1], dtype='int64'),
 'embarked': array(['S', 'C', 'Q'], dtype=object)}
```

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

```
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
```

```
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
```

```
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
```

## 7.8 Variable Transformation

Feature-engine's variable transformers transform numerical variables with various mathematical transformations.

### 7.8.1 LogTransformer

#### API Reference

**class** `feature_engine.transformation.`**`LogTransformer`**(*base='e'*, *variables=None*)

The LogTransformer() applies the natural logarithm or the base 10 logarithm to numerical variables. The natural logarithm is logarithm in base e.

The LogTransformer() only works with numerical non-negative 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.

**Parameters**

**base: string, default='e'** Indicates if the natural or base 10 logarithm should be applied. Can take values 'e' or '10'.

**variables** [list, default=None] The list of numerical variables to be transformed. If None, the transformer will find and select all numerical variables.

#### Methods

| | |
|---|---|
| **fit:** | This transformer does not learn parameters. |
| **transform:** | Transforms the variables using log transformation. |
| **fit_transform:** | Fit to data, then transform it. |

**`fit`**(*X*, *y=None*)

This transformer does not learn parameters.

Select the numerical variables and determines whether the logarithm can be applied on the selected variables (it checks if the variables are all 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.

**Returns**

**self**

**Raises**

**TypeError**

- If the input is not a Pandas DataFrame
- If any of the user provided variables are not numerical

**ValueError**

- If there are no numerical variables in the df or the df is empty
- If the variable(s) contain null values
- If some variables contain zero or negative values

**transform**(*X*)
Transforms the variables using log 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` ..

**Raises**

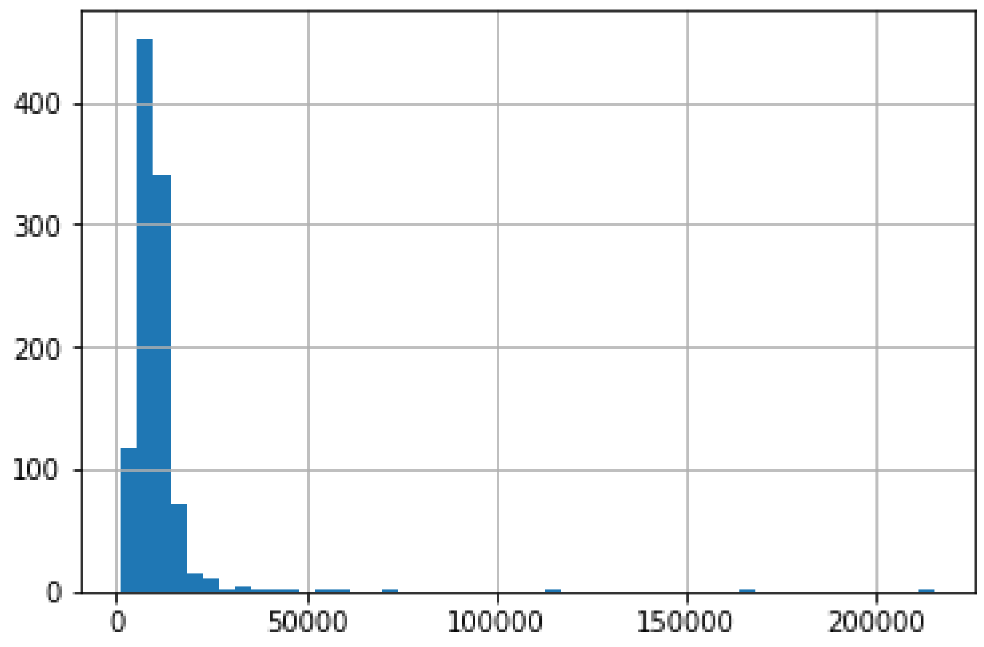
**TypeError** If the input is not a Pandas DataFrame

**ValueError**

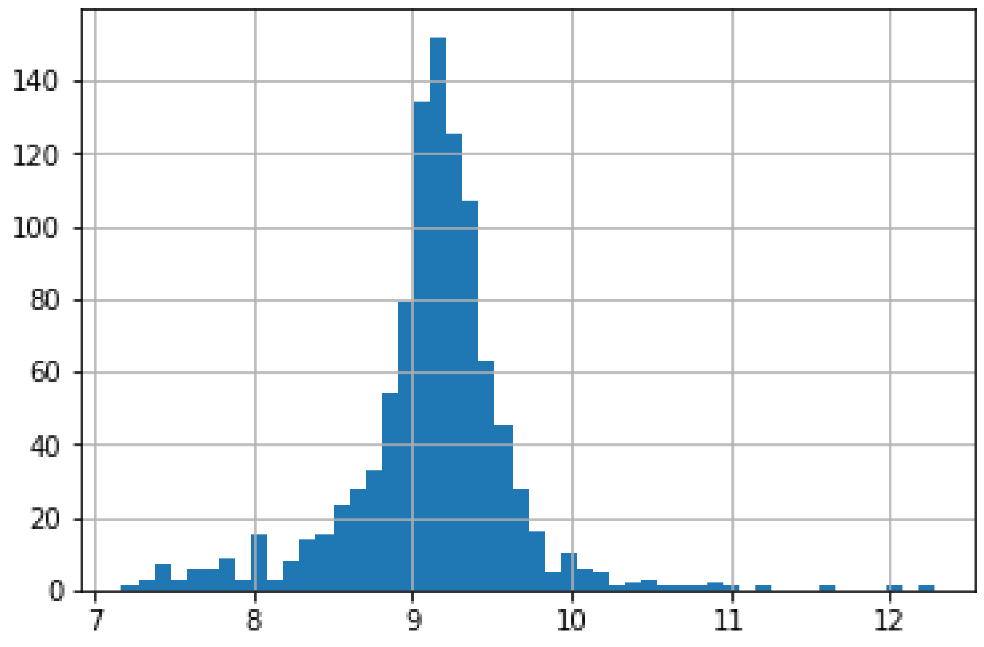
- If the variable(s) contain null values.
- If the dataframe not of the same size as that used in fit().
- If some variables contains zero or negative values.

## Example

```
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)

# set up the variable transformer
tf = vt.LogTransformer(variables = ['LotArea', 'GrLivArea'])

# fit the transformer
tf.fit(X_train)

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

# un-transformed variable
X_train['LotArea'].hist(bins=50)
```



```
# transformed variable
train_t['LotArea'].hist(bins=50)
```



### 7.8.2 ReciprocalTransformer

#### API Reference

**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.

**Parameters**

**variables** [list, default=None] The list of numerical variables that will be transformed. If None, the transformer will automatically find and select all numerical variables.

#### Methods

| | |
|---|---|
| **fit:** | This transformer does not learn parameters. |
| **transform:** | Apply the reciprocal 1 / x transformation. |
| **fit_transform:** | Fit to data, then transform it. |

**`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.

**Returns**

**self**

**Raises**

**TypeError**

- If the input is not a Pandas DataFrame
- If any of the user provided variables are not numerical

**ValueError**

- If there are no numerical variables in the df or the df is empty
- If the variable(s) contain null values
- If some variables contain zero as values

**`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** [pandas dataframe] The dataframe with the transformed variables.

**rtype** `DataFrame` ..

**Raises**

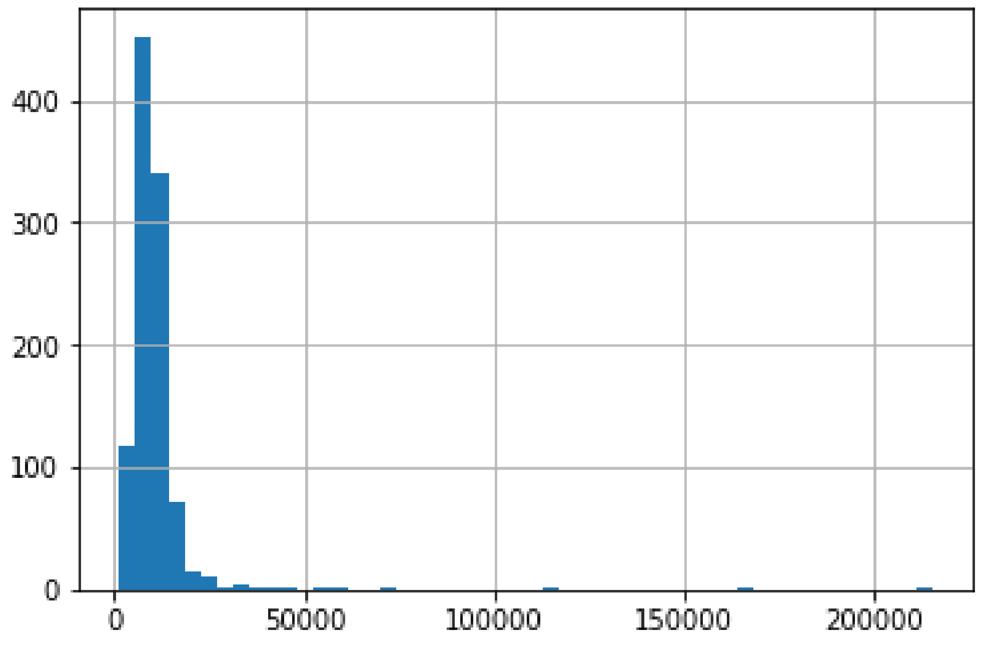
**TypeError** If the input is not a Pandas DataFrame

**ValueError**

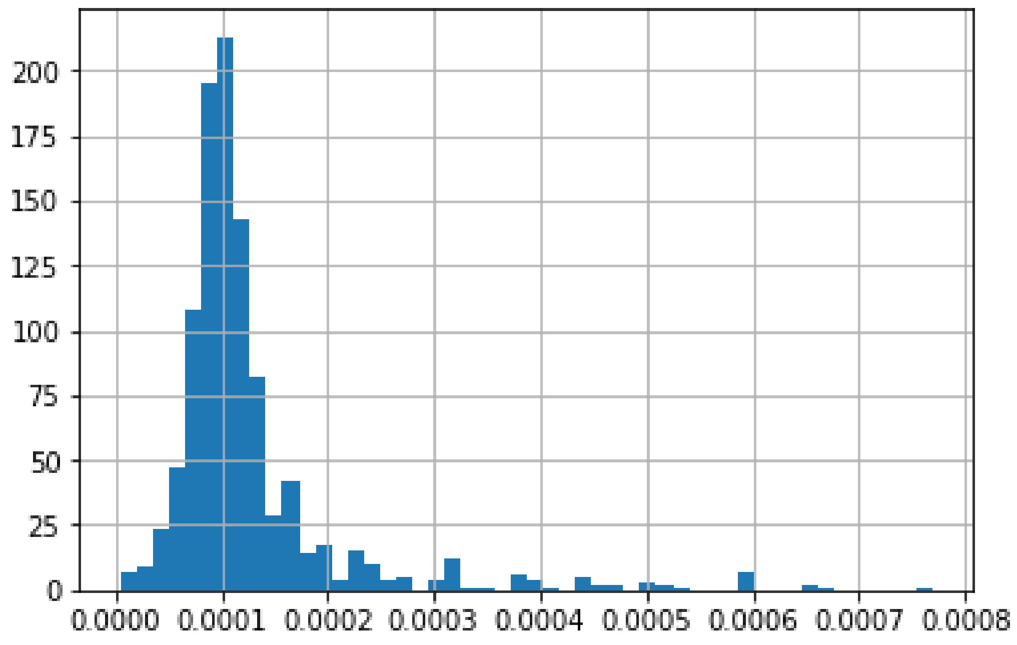
- If the variable(s) contain null values.
- If the dataframe not of the same size as that used in fit().
- If some variables contain zero values.

## Example

```
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)

# set up the variable transformer
tf = vt.ReciprocalTransformer(variables = ['LotArea', 'GrLivArea'])

# fit the transformer
tf.fit(X_train)

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

# un-transformed variable
X_train['LotArea'].hist(bins=50)
```



```
# transformed variable
train_t['LotArea'].hist(bins=50)
```



### 7.8.3 PowerTransformer

#### API Reference

**class** `feature_engine.transformation.`**`PowerTransformer`**(*exp=0.5*, *variables=None*)

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.

**Parameters**

**variables** [list, default=None] The list of numerical variables that will be transformed. If None, the transformer will automatically find and select all numerical variables.

**exp** [float or int, default=0.5] The power (or exponent).

#### Methods

| | |
|---|---|
| **fit:** | This transformer does not learn parameters. |
| **transform:** | Apply the power transformation to the variables. |
| **fit_transform:** | Fit to data, then transform it. |

**`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.

**Returns**

**self**

**Raises**

**TypeError**

- If the input is not a Pandas DataFrame
- If any of the user provided variables are not numerical

**ValueError**

- If there are no numerical variables in the df or the df is empty
- If the variable(s) contain null values

**`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** [pandas Dataframe] The dataframe with the power transformed variables.

**rtype** `DataFrame` ..

**Raises**

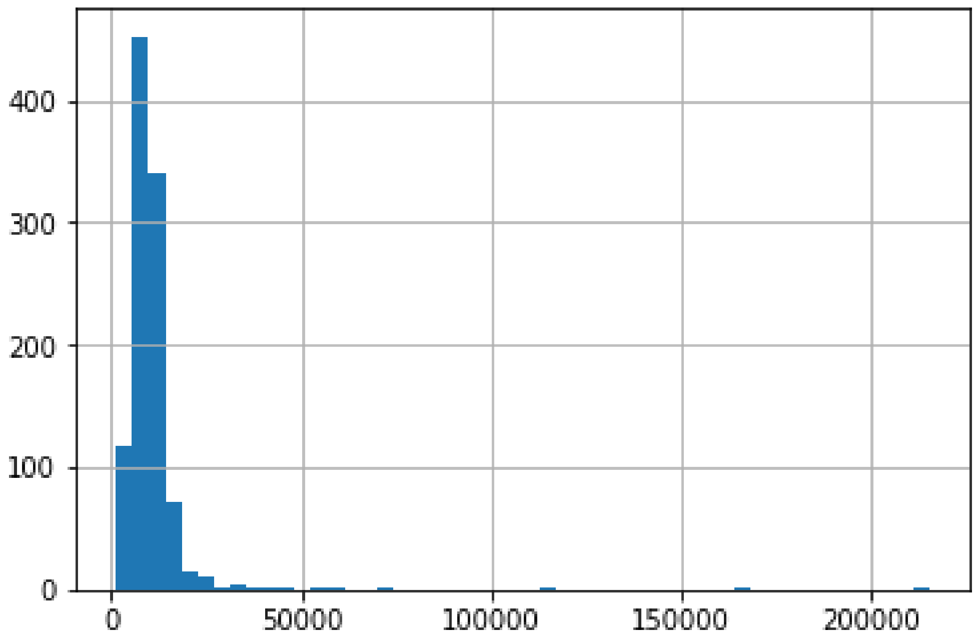
**TypeError** If the input is not a Pandas DataFrame

**ValueError**

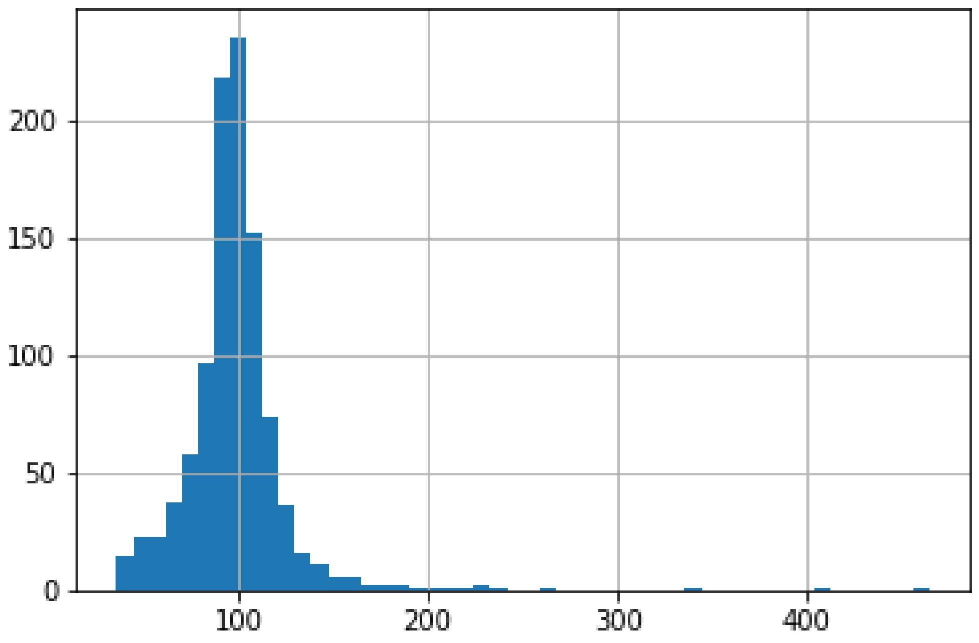
- If the variable(s) contain null values.
- If the dataframe not of the same size as that used in fit().

## Example

```
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)

# set up the variable transformer
tf = vt.PowerTransformer(variables = ['LotArea', 'GrLivArea'], exp=0.5)

# fit the transformer
tf.fit(X_train)

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

# un-transformed variable
X_train['LotArea'].hist(bins=50)
```



```
# transformed variable
train_t['LotArea'].hist(bins=50)
```



### 7.8.4 BoxCoxTransformer

#### API Reference

**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, the transformer also works for zero values).

A list of variables can be passed as an argument. Alternatively, the transformer will automatically select and transform all numerical variables.

**Parameters**

**variables** [list, default=None] The list of numerical variables that will be transformed. If None, the transformer will automatically find and select all numerical variables.

#### Attributes

| | |
|---|---|
| **lambda_dict_ :** | Dictionary with the best BoxCox exponent per variable. |

#### References

[1]

#### Methods

| | |
|---|---|
| **fit:** | Learn the optimal lambda for the BoxCox transformation. |
| **transform:** | Apply the BoxCox transformation. |
| **fit_transform:** | Fit to data, then transform it. |

**`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.

**Returns**

**self**

**Raises**

**TypeError**

- If the input is not a Pandas DataFrame
- If any of the user provided variables are not numerical

**ValueError**

- If there are no numerical variables in the df or the df is empty
- If the variable(s) contain null values
- If some variables contain zero values

**transform**(*X*)

Apply the BoxCox 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` ..

**Raises**

**TypeError** If the input is not a Pandas DataFrame

**ValueError**

- If the variable(s) contain null values.
- If the dataframe not of the same size as that used in fit().
- If some variables contain negative values.

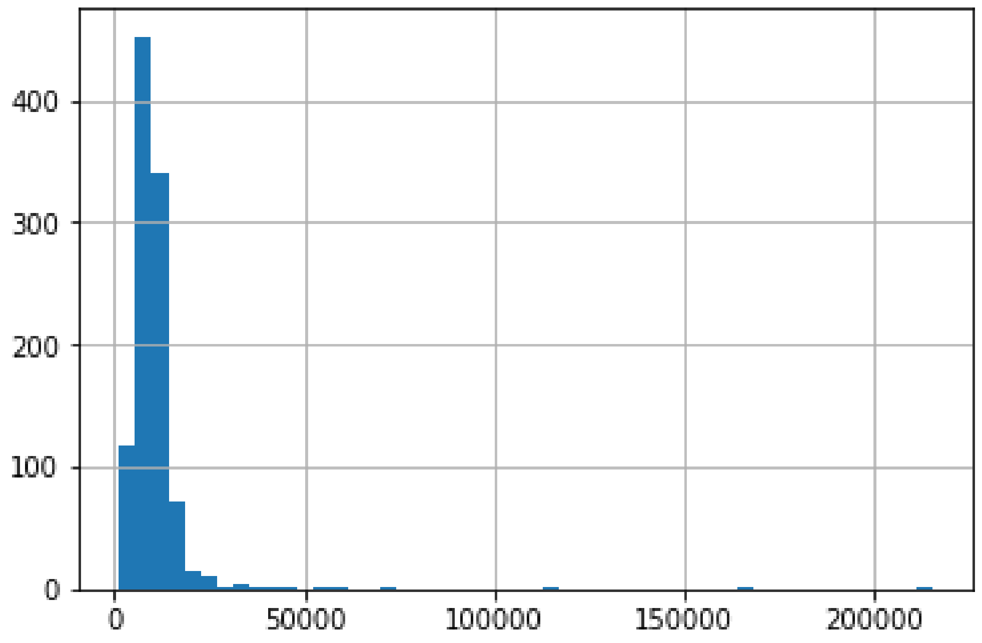
## Example

```
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)

# set up the variable transformer
tf = vt.BoxCoxTransformer(variables = ['LotArea', 'GrLivArea'])

# fit the transformer
```

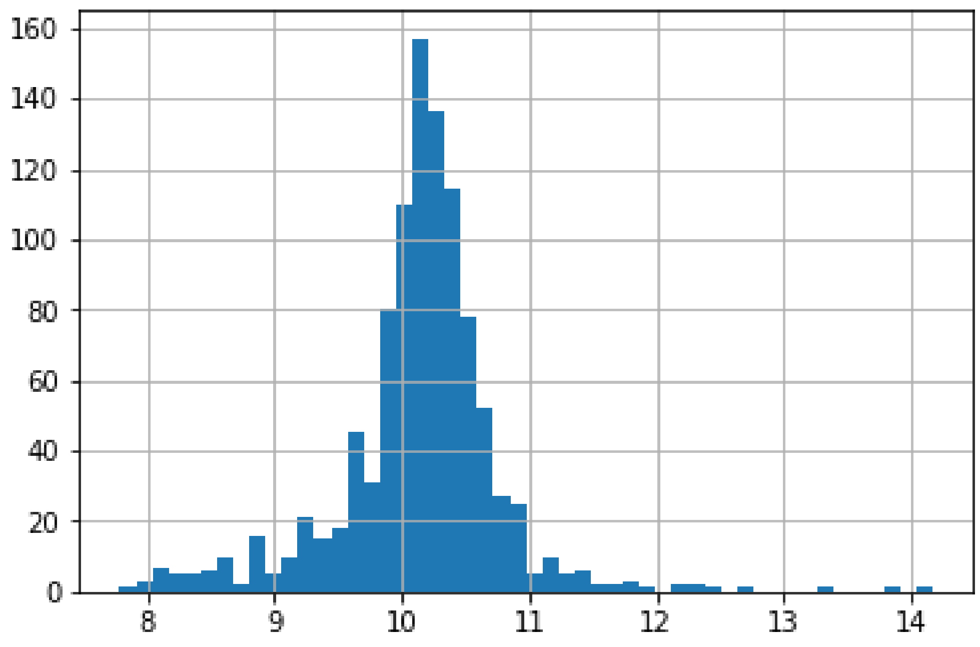
(continues on next page)

(continued from previous page)

```
tf.fit(X_train)

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

# un-transformed variable
X_train['LotArea'].hist(bins=50)
```



```
# transformed variable
train_t['GrLivArea'].hist(bins=50)
```



### 7.8.5 YeoJohnsonTransformer

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.

#### API Reference

**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.

**Parameters**

**variables** [list, default=None] The list of numerical variables that will be transformed. 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. |

### References

[1]

### Methods

|  |  |
|---|---|
| **fit:** | Learn the optimal lambda for the Yeo-Johnson transformation. |
| **transform:** | Apply the Yeo-Johnson transformation. |
| **fit_transform:** | Fit to data, then transform it. |

`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.

**Returns**

**self**

**Raises**

**TypeError**

- If the input is not a Pandas DataFrame
- If any of the user provided variables are not numerical

**ValueError**

- If there are no numerical variables in the df or the df is empty
- If the variable(s) contain null values

`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` ..

**Raises**

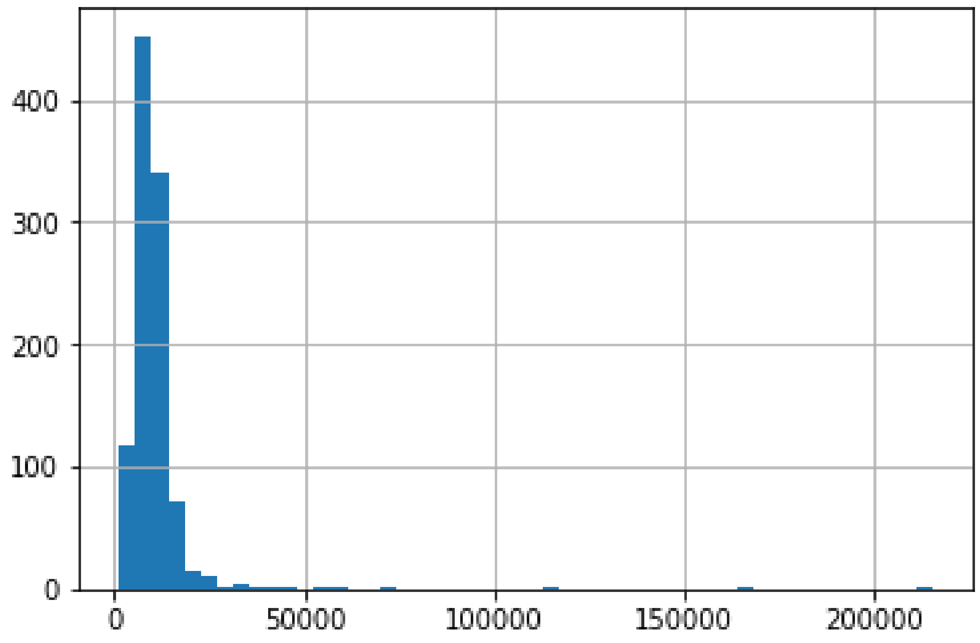
**TypeError** If the input is not a Pandas DataFrame

**ValueError**

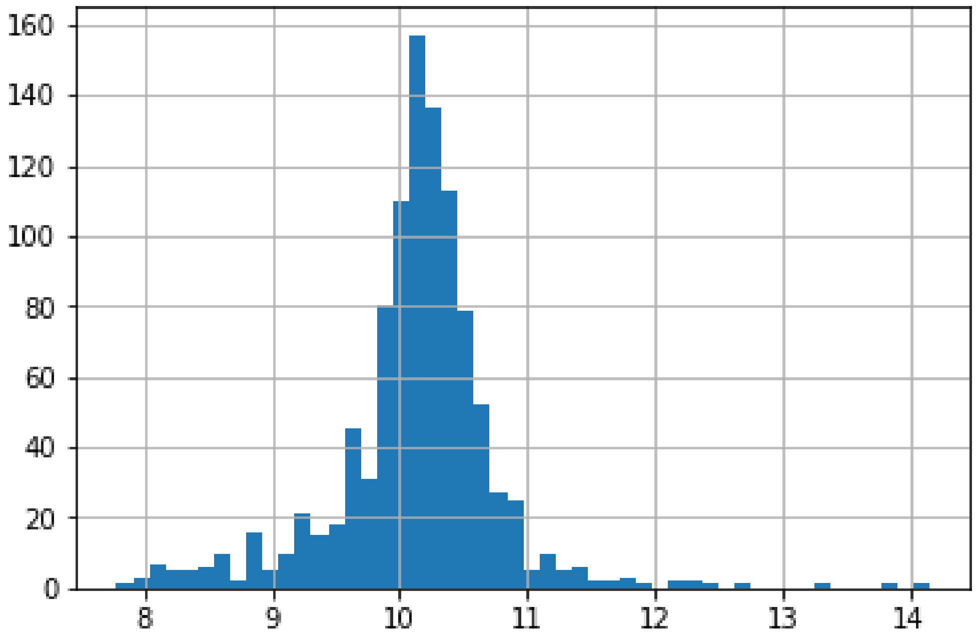
- If the variable(s) contain null values.
- If the dataframe not of the same size as that used in fit().

## Example

```
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)

# set up the variable transformer
tf = vt.YeoJohnsonTransformer(variables = ['LotArea', 'GrLivArea'])

# fit the transformer
tf.fit(X_train)

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

# un-transformed variable
X_train['LotArea'].hist(bins=50)
```



```
# transformed variable
train_t['LotArea'].hist(bins=50)
```



## 7.9 Variable 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.

### 7.9.1 EqualFrequencyDiscretiser

#### API Reference

**class** `feature_engine.discretisation.`**`EqualFrequencyDiscretiser`**(*q=10*, *variables=None*, *return_object=False*, *return_boundaries=False*)

The EqualFrequencyDiscretiser() divides continuous numerical variables into contiguous equal frequency intervals, that is, intervals that contain approximately the same proportion of observations.

The interval limits are determined using `pandas.qcut()`, in other words, the interval limits are determined by the quantiles. The number of intervals, i.e., the number of quantiles in which the variable should be divided is determined by the user.

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.

**Parameters**

**q** [int, default=10] Desired number of equal frequency intervals / bins. In other words the number of quantiles in which the variables should be divided.

**variables** [list] The list of numerical variables that will be discretised. If None, the EqualFrequencyDiscretiser() will select all numerical variables.

**return_object** [bool, default=False] Whether the numbers in the discrete variable should be returned as numeric or as object. The decision is made by the user based on whether they would like to proceed the engineering of the variable as if it was numerical or categorical.

**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.

## Attributes

| | |
|---|---|
| **binner_dict_:** | Dictionary with the interval limits per variable. |

**See also:**

`pandas.qcut` https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.qcut.html

## References

[1], [2]

## Methods

| | |
|---|---|
| **fit:** | Find the interval limits. |
| **transform:** | Sort continuous variable values into the intervals. |
| **fit_transform:** | Fit to the data, then transform it. |

`fit` (*X*, *y=None*)

Learn the limits of the equal frequency intervals, that is the percentiles 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.

**Returns**

**self**

**Raises**

**TypeError**

- If the input is not a Pandas DataFrame
- If any of the user provided variables are not numerical

**ValueError**

- If there are no numerical variables in the df or the df is empty
- If the variable(s) contain null values

**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** [pandas dataframe of shape = [n_samples, n_features]] The transformed data with the discrete variables.

**rtype** `DataFrame` ..

**Raises**

**TypeError** If the input is not a Pandas DataFrame

**ValueError**

- If the variable(s) contain null values
- If the dataframe is not of the same size as the one used in fit()

## Example

The EqualFrequencyDiscretiser() sorts the variable values into contiguous intervals of equal proportion of observations. The limits of the intervals are calculated according to the quantiles. The number of intervals or quantiles should be determined by the user. The transformer can return the variable as numeric or object (default = numeric).

The EqualFrequencyDiscretiser() works only with numerical variables. A list of variables can be indicated, or the discretiser will automatically select all numerical variables in the train set.

```python
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 EqualFrequencyDiscretiser

# 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)

# set up the discretisation transformer
disc = EqualFrequencyDiscretiser(q=10, variables=['LotArea', 'GrLivArea'])

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

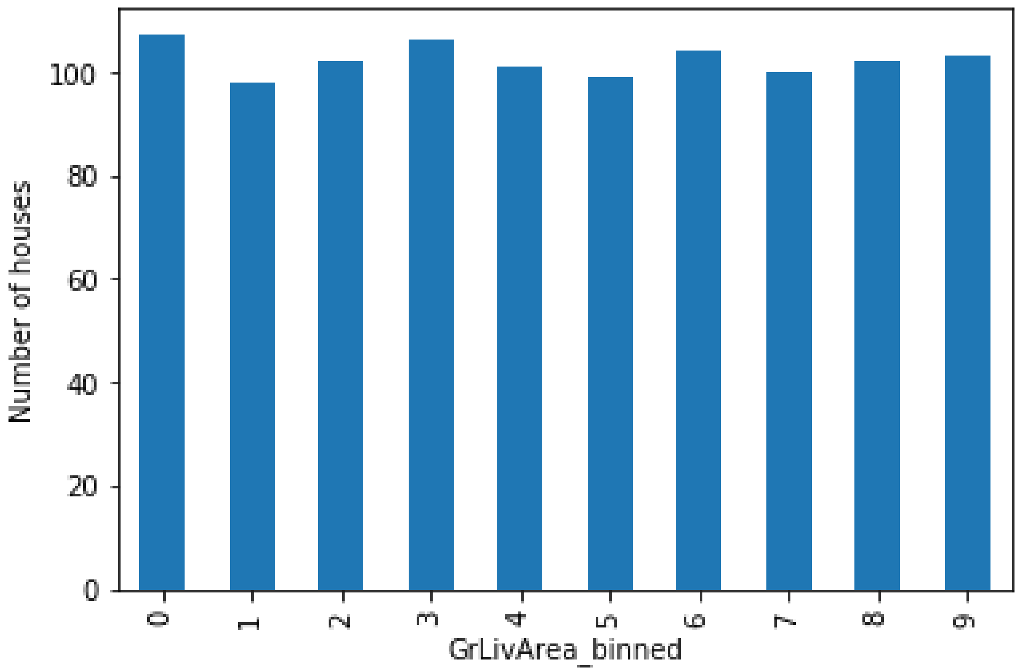
(continues on next page)

(continued from previous page)

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

disc.binner_dict_
```

```
{'LotArea': [-inf,
  5007.1,
  7164.6,
  8165.700000000001,
  8882.0,
  9536.0,
  10200.0,
  11046.300000000001,
  12166.400000000001,
  14373.9,
  inf],
 'GrLivArea': [-inf,
  912.0,
  1069.6000000000001,
  1211.3000000000002,
  1344.0,
  1479.0,
  1603.2000000000003,
  1716.0,
  1893.0000000000005,
  2166.3999999999996,
  inf]}
```

```
# with equal frequency discretisation, each bin contains approximately
# the same number of observations.
train_t.groupby('GrLivArea')['GrLivArea'].count().plot.bar()
plt.ylabel('Number of houses')
```



### 7.9.2 EqualWidthDiscretiser

#### API Reference

**class** feature_engine.discretisation.**EqualWidthDiscretiser**(*bins=10*, *variables=None*, *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, should be determined by the user.

The interval limits are determined using `pandas.cut()`. The number of intervals in which the variable should be divided must be indicated 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.

**Parameters**

**bins** [int, default=10] Desired number of equal width intervals / bins.

**variables** [list] The list of numerical variables to transform. If None, the discretiser will automatically select all numerical type variables.

**return_object** [bool, default=False] Whether the numbers in the discrete variable should be returned as numeric or as object. The decision should be made by the user based on whether they would like to proceed the engineering of the variable as if it was numerical or categorical.

**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.

### Attributes

| | |
|---|---|
| **binner_dict_:** | Dictionary with the interval limits per variable. |

**See also:**

`pandas.cut` https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.cut.html

### References

[1], [2]

### Methods

| | |
|---|---|
| **fit:** | Find the interval limits. |
| **transform:** | Sort continuous variable values into the intervals. |
| **fit_transform:** | Fit to the data, then transform it. |

`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.

**Returns**

**self**

**Raises**

**TypeError**

- If the input is not a Pandas DataFrame
- If any of the user provided variables are not numerical

**ValueError**

- If there are no numerical variables in the df or the df is empty
- If the variable(s) contain null values

`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** [pandas dataframe of shape = [n_samples, n_features]] The transformed data with the discrete variables.

**rtype** `DataFrame` ..

**Raises**

**TypeError** If the input is not a Pandas DataFrame

**ValueError**

- If the variable(s) contain null values
- If the dataframe is not of the same size as the one used in fit()

## Example

The EqualWidthDiscretiser() sorts the variable values into contiguous intervals of equal size. The size of the interval is calculated as:

( max(X) - min(X) ) / bins

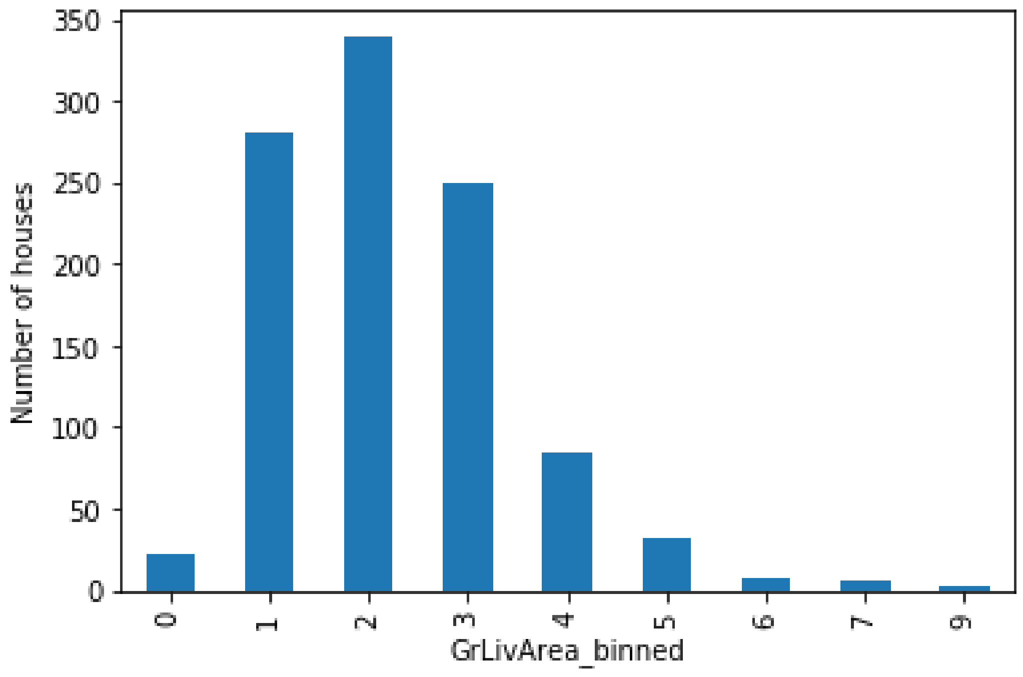
where bins, which is the number of intervals, should be determined by the user. The transformer can return the variable as numeric or object (default = numeric).

The EqualWidthDiscretiser() works only with numerical variables. A list of variables can be indicated, or the discretiser will automatically select all numerical variables in the train set.

```python
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 EqualWidthDiscretiser

# 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)

# set up the discretisation transformer
disc = EqualWidthDiscretiser(bins=10, variables=['LotArea', 'GrLivArea'])

# fit the transformer
disc.fit(X_train)

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

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,
  768.2,
  1202.4,
  1636.6,
  2070.8,
  2505.0,
  2939.2,
  3373.4,
  3807.6,
  4241.799999999999,
  inf]}
```

```
# with equal width 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')
```



### 7.9.3 ArbitraryDiscretiser

#### API Reference

**class** feature_engine.discretisation.**ArbitraryDiscretiser**(*binning_dict*, *return_object=False*, *return_boundaries=False*)

The ArbitraryDiscretiser() divides continuous numerical variables into contiguous intervals which limits are determined arbitrarily by the user.

The user needs to enter a dictionary with variable names as keys, and a list of the limits of the intervals as values. For example {'var1':[0, 10, 100, 1000], 'var2':[5, 10, 15, 20]}.

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`.

The ArbitraryDiscretiser() works only with numerical variables. The discretiser will check if the dictionary entered by the user contains variables present in the training set, and if these variables are numerical, before doing any transformation.

Then it transforms the variables, that is, it sorts the values into the intervals.

**Parameters**

**binning_dict** [dict] The dictionary with the variable : interval limits pairs, provided by the user. A valid dictionary looks like this:

binning_dict = {'var1':[0, 10, 100, 1000], 'var2':[5, 10, 15, 20]}.

**return_object** [bool, default=False] Whether the numbers in the discrete variable should be returned as numeric or as object. The decision is made by the user based on whether they would like to proceed the engineering of the variable as if it was numerical or categorical.

Categorical encoders in Feature-engine work only with variables of type object, thus, if you wish to encode the returned bins, set return_object to True.

**return_boundaries** [bool, default=False] whether the output, that is the bin names / values, should be the interval boundaries. If True, it returns the interval boundaries. If False, it returns integers.

#### Attributes

| | |
|---|---|
| **binner_dict_ :** | Dictionary with the interval limits per variable. |

**See also:**

**`pandas.cut`** https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.cut.html

### Methods

| | |
|---|---|
| **fit:** | This transformer does not learn any parameter. |
| **transform:** | Sort continuous variable values into the intervals. |
| **fit_transform:** | Fit to the data, then transform it. |

**fit**(*X*, *y=None*)

This transformer does not learn any parameter.

Check dataframe and variables. Checks that the user entered variables are in the train set and cast as numerical.

**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.

**Returns**

**self**

**Raises**

**TypeError**

- If the input is not a Pandas DataFrame
- If any of the user provided variables are not numerical

**ValueError**

- If there are no numerical variables in the df or the df is empty
- If the variable(s) contain null values

**transform**(*X*)

Sort the variable values into the intervals.

**Parameters**

**X** [pandas dataframe of shape = [n_samples, n_features]] The dataframe to be transformed.

**Returns**

**X** [pandas dataframe of shape = [n_samples, n_features]] The transformed data with the discrete variables.

**rtype** `DataFrame` ..

**Raises**

**TypeError** If the input is not a Pandas DataFrame

**ValueError**

- If the variable(s) contain null values
- If the dataframe is not of the same size as the one used in fit()

### Example

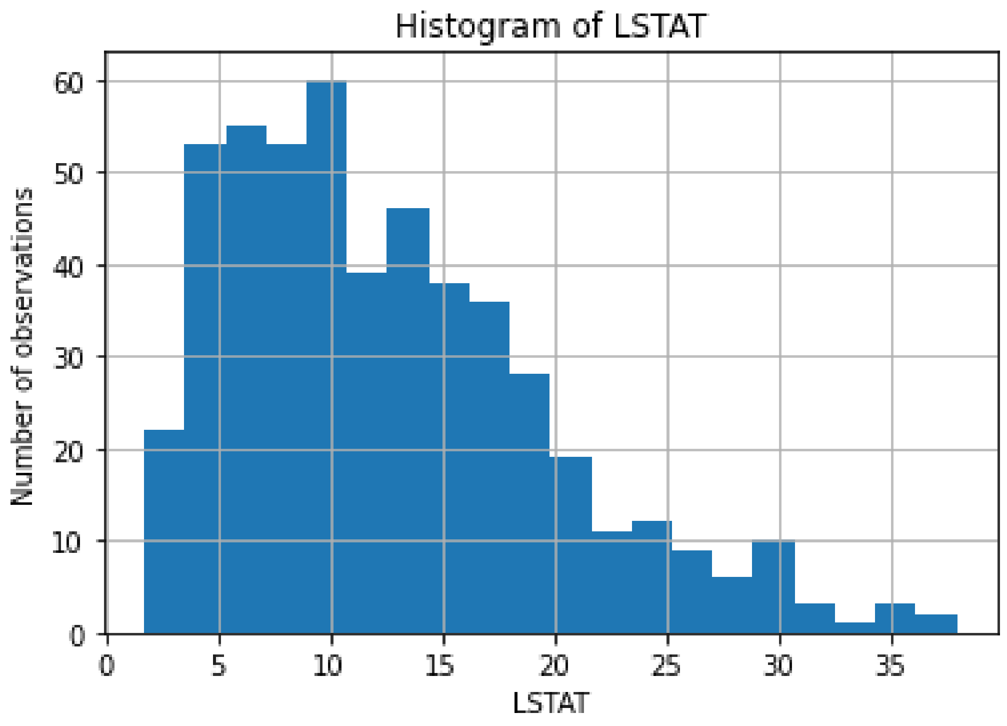
The ArbitraryDiscretiser() sorts the variable values into contiguous intervals which limits are arbitrarily defined by the user.

The user must provide a dictionary of variable:list of limits pair 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.

First, let's load a dataset and plot a histogram of a continuous variable.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import load_boston
from feature_engine.discretisation import ArbitraryDiscretiser

boston_dataset = load_boston()
data = pd.DataFrame(boston_dataset.data, columns=boston_dataset.feature_names)

data['LSTAT'].hist(bins=20)
plt.xlabel('LSTAT')
plt.ylabel('Number of observations')
plt.title('Histogram of LSTAT')
plt.show()
```



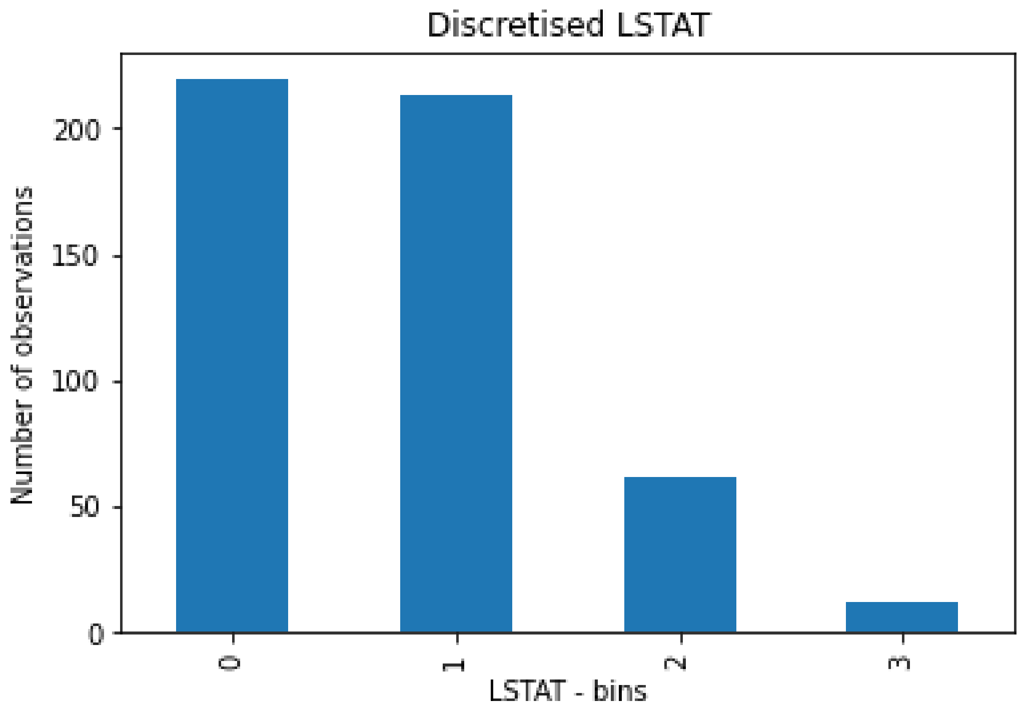
Now, let's discretise the variable into arbitrarily determined intervals. We want the interval names as integers, so we set return_boundaries to False.

```
user_dict = {'LSTAT': [0, 10, 20, 30, np.Inf]}

transformer = ArbitraryDiscretiser(
```

(continues on next page)

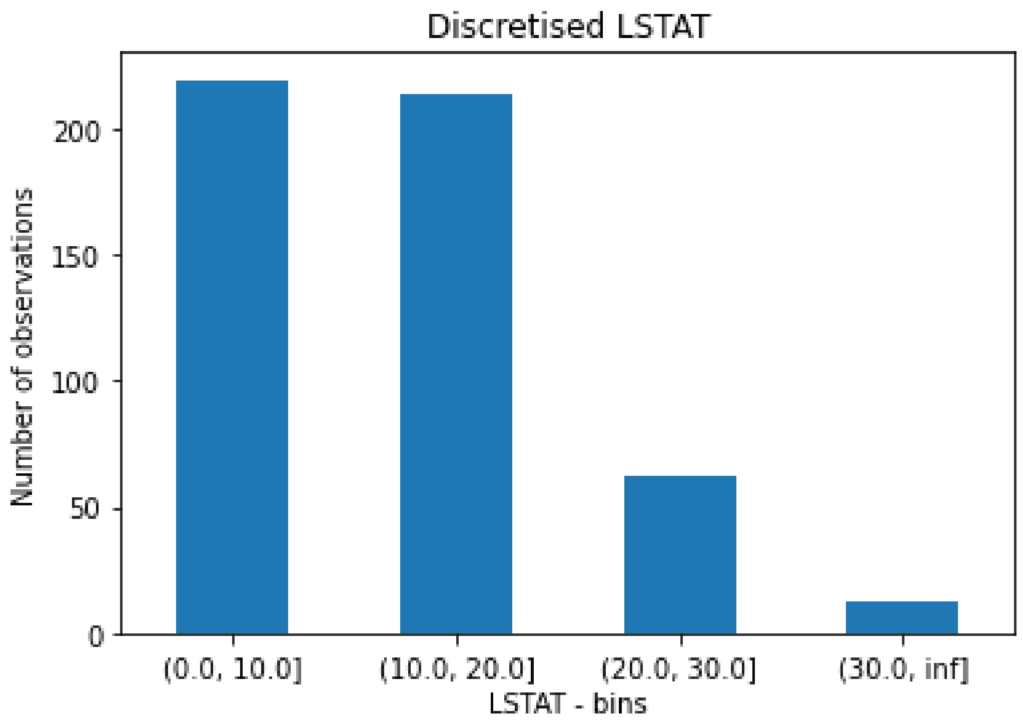
(continued from previous page)

```
    binning_dict=user_dict, return_object=False, return_boundaries=False)
X = transformer.fit_transform(data)

X['LSTAT'].value_counts().plot.bar()
plt.xlabel('LSTAT - bins')
plt.ylabel('Number of observations')
plt.title('Discretised LSTAT')
plt.show()
```



Alternatively, we can return the interval limits in the discretised variable by setting return_boundaries to True.

```
transformer = ArbitraryDiscretiser(
    binning_dict=user_dict, return_object=False, return_boundaries=True)
X = transformer.fit_transform(data)

X['LSTAT'].value_counts().plot.bar(rot=0)
plt.xlabel('LSTAT - bins')
plt.ylabel('Number of observations')
plt.title('Discretised LSTAT')
plt.show()
```



### 7.9.4 DecisionTreeDiscretiser

#### API Reference

**class** feature_engine.discretisation.**DecisionTreeDiscretiser**(*cv=3*, *scoring='neg_mean_squared_error'*, *variables=None*, *param_grid=None*, *regression=True*, *random_state=None*)

The DecisionTreeDiscretiser() replaces continuous numerical variables by discrete, finite, values estimated by a decision tree.

The methods 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() works only with numerical variables. A list of variables can be passed as an argument. Alternatively, the discretiser will automatically select all numerical variables.

The DecisionTreeDiscretiser() first trains a decision tree for each variable.

The DecisionTreeDiscretiser() then transforms the variables, that is, makes predictions based on the variable values, using the trained decision tree.

**Parameters**

**cv** [int, default=3] Desired number of cross-validation fold to be used to fit the decision tree.

**scoring: str, default='neg_mean_squared_error'** Desired metric to optimise the performance for the tree. Comes from sklearn.metrics. See DecisionTreeRegressor or DecisionTreeClassifier model evaluation documentation for more options: https://scikit-learn.org/stable/modules/model_evaluation.html

**variables** [list] The list of numerical variables that will be transformed. If None, the discretiser will automatically select all numerical variables.

**regression** [boolean, default=True] Indicates whether the discretiser should train a regression or a classification decision tree.

**param_grid** [dictionary, default=None] The list of parameters over which the decision tree should be optimised during the grid search. The param_grid can contain any of the permitted parameters for Scikit-learn's DecisionTreeRegressor() or DecisionTreeClassifier().

If None, then param_grid = {'max_depth': [1, 2, 3, 4]}

**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, over the train set. |

**See also:**

**`sklearn.tree.DecisionTreeClassifier`**

**`sklearn.tree.DecisionTreeRegressor`**

## References

[1]

## Methods

| | |
|---|---|
| **fit:** | Fit a decision tree per variable. |
| **transform:** | Replace continuous values by the predictions of the decision tree. |
| **fit_transform:** | Fit to the data, then transform it. |

**`fit`** (*X*, *y*)

Fit the decision trees. One tree per variable to be transformed.

**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.

**Returns**

**self**

**Raises**

**TypeError**

- If the input is not a Pandas DataFrame
- If any of the user provided variables are not numerical

**ValueError**

- If there are no numerical variables in the df or the df is empty
- If the variable(s) contain null values

**transform**(*X*)
Replaces original variable with the predictions of the tree. The tree outcome is finite, aka, discrete.

**Parameters**

**X** [pandas dataframe of shape = [n_samples, n_features]] The input samples.

**Returns**

**X_transformed** [pandas dataframe of shape = [n_samples, n_features]] The dataframe with transformed variables.

**rtype** `DataFrame` ..

**Raises**

**TypeError** If the input is not a Pandas DataFrame

**ValueError**

- If the variable(s) contain null values
- If the dataframe is not of the same size as the one used in fit()

## Example

In the original article, each feature of the challenge dataset was recoded 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.

In the following example, we recode 2 numerical variables using decision trees.

```python
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)
```

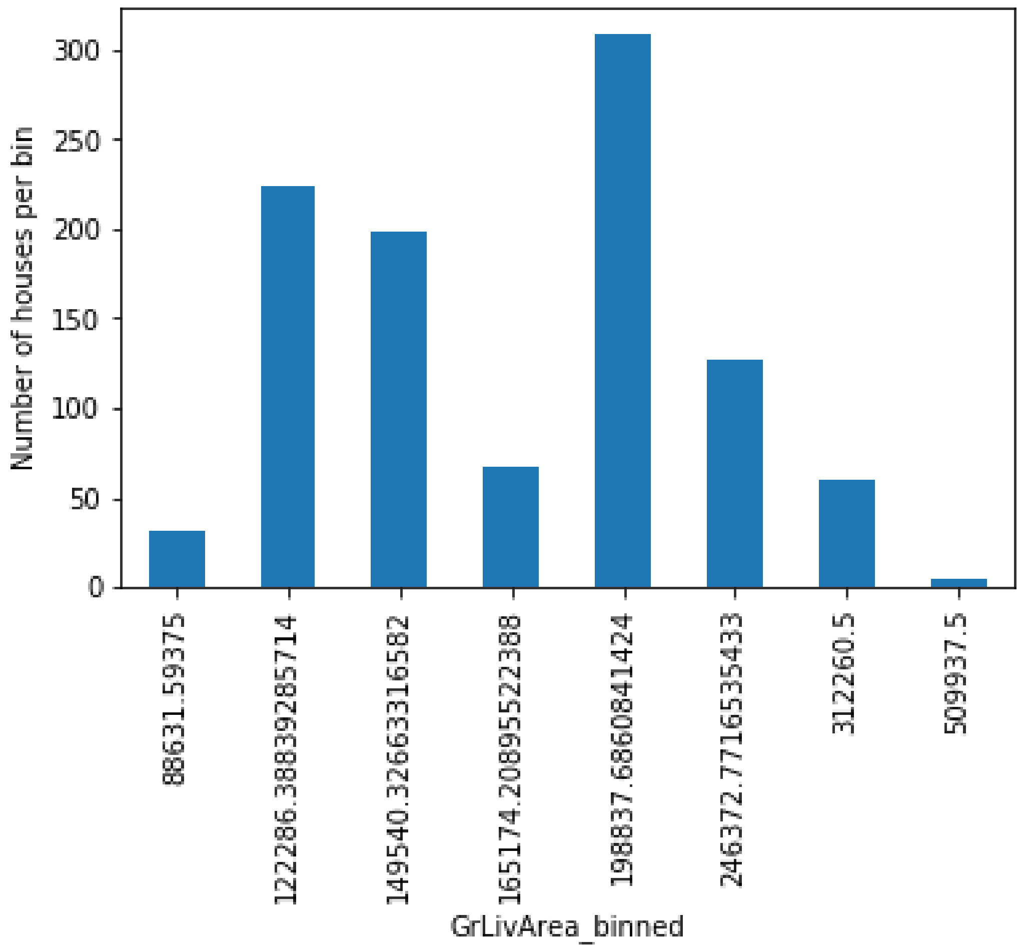
(continues on next page)

(continued from previous page)

```
# 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)

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

disc.binner_dict_
```

```
{'LotArea': 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),
 '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 does not necessarily contain
# the same number of observations.
train_t.groupby('GrLivArea')['GrLivArea'].count().plot.bar()
plt.ylabel('Number of houses')
```



## 7.10 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.

### 7.10.1 Winsorizer

#### API Reference

**class** `feature_engine.outliers.`**`Winsorizer`**(*capping_method='gaussian'*, *tail='right'*, *fold=3*, *variables=None*, *missing_values='raise'*)

The Winsorizer() caps maximum and / or minimum values of a variable.

The Winsorizer() works only with numerical variables. A list of variables can be indicated. Alternatively, the Winsorizer() will select all numerical variables in the train set.

The Winsorizer() first calculates the capping values at the end of the distribution. The values are determined using:

- a Gaussian approximation,

- the inter-quantile range proximity rule (IQR)
- 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.

**percentiles or quantiles:**

- 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='quantile'`, 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 transformer first finds the values at one or both tails of the distributions (fit). The transformer then caps the variables (transform).

**Parameters**

**capping_method** [str, default=gaussian] Desired capping method. Can take 'gaussian', 'iqr' or 'quantiles'.

'gaussian': the transformer will find the maximum and / or minimum values to cap the variables using the Gaussian approximation.

'iqr': the transformer will find the boundaries using the IQR proximity rule.

'quantiles': the limits are given by the percentiles.

**tail** [str, default=right] Whether to cap outliers on the right, left or both tails of the distribution. Can take 'left', 'right' or 'both'.

**fold: int or float, default=3** How far out to to place the capping values. The number that will multiply the std or IQR to calculate the capping values. Recommended values, 2 or 3 for the gaussian approximation, or 1.5 or 3 for the IQR proximity 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 variables for which the outliers will be capped. If None, the transformer will find and select all numerical variables.

**missing_values: string, default='raise'** Indicates if missing values should be ignored or raised. Sometimes we want to remove outliers in the raw, original data, sometimes, we may want to remove outliers in the already pre-transformed data. If missing_values='ignore', the transformer will ignore missing data when learning the capping parameters or transforming the data. If missing_values='raise' the transformer will return an error if the training or the datasets to transform contain missing values.

### Attributes

| | |
|---|---|
| **right_tail_caps_:** | Dictionary with the maximum values at which variables will be capped. |
| **left_tail_caps_ :** | Dictionary with the minimum values at which variables will be capped. |

### Methods

| | |
|---|---|
| **fit:** | Learn the values that should be used to replace outliers. |
| **transform:** | Cap the variables. |
| **fit_transform:** | Fit to the data. Then transform it. |

**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.

**Returns**

**self**

**Raises**

**TypeError** If the input is not a Pandas DataFrame

**transform**(*X*)

Cap the variable values, that is, censors outliers.

**Parameters**

**X** [pandas dataframe of shape = [n_samples, n_features]] The data to be transformed.

**Returns**

**X** [pandas dataframe of shape = [n_samples, n_features]] The dataframe with the capped variables.

**rtype** `DataFrame` ..

**Raises**

**TypeError** If the input is not a Pandas DataFrame

**ValueError** If the dataframe is not of same size as that used in fit()

### Example

Censors variables at predefined minimum and maximum 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.

**percentiles or quantiles:** right tail: 95th percentile

left tail: 5th percentile

See the API Reference for more details.

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

from feature_engine.outliers import Winsorizer

# 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['fare'] = data['fare'].astype('float')
    data['fare'].fillna(data['fare'].median(), inplace=True)
    data['age'] = data['age'].astype('float')
    data['age'].fillna(data['age'].median(), inplace=True)
    return data

data = load_titanic()

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

# set up the capper
capper = Winsorizer(capping_method='gaussian', tail='right', fold=3, variables=['age',
↪ 'fare'])

# fit the capper
capper.fit(X_train)

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

capper.right_tail_caps_
```

```
{'age': 72.03416424092518, 'fare': 174.78162171790427}
```

```
train_t[['fare', 'age']].max()
```

```
fare    174.781622
age      67.490484
dtype: float64
```

### 7.10.2 ArbitraryOutlierCapper

#### API Reference

**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.

The user must provide the maximum or minimum values that will be used to cap each variable in a dictionary {feature:capping value}

**Parameters**

**max_capping_dict** [dictionary, default=None] Dictionary containing the user specified capping values for the right tail of the distribution of each variable (maximum values).

**min_capping_dict** [dictionary, default=None] Dictionary containing user specified capping values for the eft tail of the distribution of each variable (minimum values).

**missing_values** [string, default='raise'] Indicates if missing values should be ignored or raised. If `missing_values='raise'` the transformer will return an error if the training or the datasets to transform contain missing values.

#### Attributes

| | |
|---|---|
| **right_tail_caps_:** | Dictionary with the maximum values at which variables will be capped. |
| **left_tail_caps_ :** | Dictionary with the minimum values at which variables will be capped. |

#### Methods

| | |
|---|---|
| **fit:** | This transformer does not learn any parameter. |
| **transform:** | Cap the variables. |
| **fit_transform:** | Fit to the data. Then transform it. |

**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.

**Returns**

**self**

**Raises**

**TypeError** If the input is not a Pandas DataFrame

**transform**(*X*)

Cap the variable values, that is, censors outliers.

**Parameters**

**X** [pandas dataframe of shape = [n_samples, n_features]] The data to be transformed.

**Returns**

**X** [pandas dataframe of shape = [n_samples, n_features]] The dataframe with the capped variables.

**rtype** `DataFrame` ..

**Raises**

**TypeError** If the input is not a Pandas DataFrame

**ValueError** If the dataframe is not of same size as that used in fit()

## Example

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

from feature_engine.outliers import ArbitraryOutlierCapper

# 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['fare'] = data['fare'].astype('float')
        data['fare'].fillna(data['fare'].median(), inplace=True)
        data['age'] = data['age'].astype('float')
        data['age'].fillna(data['age'].median(), inplace=True)
        return data

data = load_titanic()

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

# set up the capper
capper = ArbitraryOutlierCapper(max_capping_dict={'age': 50, 'fare': 200}, min_
↪capping_dict=None)
```

(continues on next page)

(continued from previous page)

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

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

capper.right_tail_caps_
```

```
{'age': 50, 'fare': 200}
```

```
train_t[['fare', 'age']].max()
```

```
fare    200
age      50
dtype: float64
```

### 7.10.3 OutlierTrimmer

#### API Reference

**class** `feature_engine.outliers.`**`OutlierTrimmer`**(*capping_method='gaussian'*, *tail='right'*, *fold=3*, *variables=None*, *missing_values='raise'*)

The OutlierTrimmer() removes observations with outliers from the dataset.

It works only with numerical variables. A list of variables can be indicated. Alternatively, the OutlierTrimmer() will select all numerical variables.

The OutlierTrimmer() first calculates the maximum and /or minimum values beyond which a value will be considered an outlier, and thus removed.

Limits are determined using:

- a Gaussian approximation
- the inter-quantile range proximity rule
- 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.

**percentiles or quantiles:**

- 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='quantile'`, 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 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).

**Parameters**

**capping_method** [str, default=gaussian] Desired capping method. Can take ‘gaussian’, ‘iqr’ or ‘quantiles’.

‘gaussian’: the transformer will find the maximum and / or minimum values to cap the variables using the Gaussian approximation.

‘iqr’: the transformer will find the boundaries using the IQR proximity rule.

‘quantiles’: the limits are given by the percentiles.

**tail** [str, default=right] Whether to cap outliers on the right, left or both tails of the distribution. Can take ‘left’, ‘right’ or ‘both’.

**fold: int or float, default=3** How far out to to place the capping values. The number that will multiply the std or IQR to calculate the capping values. Recommended values, 2 or 3 for the gaussian approximation, or 1.5 or 3 for the IQR proximity 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 variables for which the outliers will be removed If None, the transformer will find and select all numerical variables.

**missing_values: string, default=’raise’** Indicates if missing values should be ignored or raised. Sometimes we want to remove outliers in the raw, original data, sometimes, we may want to remove outliers in the already pre-transformed data. If missing_values=’ignore’, the transformer will ignore missing data when learning the capping parameters or transforming the data. If missing_values=’raise’ the transformer will return an error if the training or the datasets to transform contain missing values.

### Attributes

| | |
|---|---|
| **right_tail_caps_:** | Dictionary with the maximum values above which values will be removed |
| **left_tail_caps_ :** | Dictionary with the minimum values below which values will be removed |

### Methods

| | |
|---|---|
| **fit:** | Find maximum and minimum values. |
| **transform:** | Remove outliers. |
| **fit_transform:** | Fit to the data. Then transform it. |

**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** [pandas dataframe of shape = [n_samples, n_features]] The dataframe without outlier observations.

**rtype** `DataFrame` ..

**Raises**

**TypeError** If the input is not a Pandas DataFrame

**ValueError** If the dataframe is not of same size as that used in fit()

## Example

Removes values beyond predefined minimum and maximum values from the data set. 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.

**percentiles or quantiles:** right tail: 95th percentile

left tail: 5th percentile

See the API Reference for more details.

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

from feature_engine.outliers import OutlierTrimmer

# 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')
```

(continues on next page)

(continued from previous page)

```
    data['embarked'].fillna('C', inplace=True)
    data['fare'] = data['fare'].astype('float')
    data['fare'].fillna(data['fare'].median(), inplace=True)
    data['age'] = data['age'].astype('float')
    data['age'].fillna(data['age'].median(), inplace=True)
    return data

data = load_titanic()

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

# set up the capper
capper = OutlierTrimmer(capping_method='iqr', tail='right', fold=1.5, variables=['age
↪', 'fare'])

# fit the capper
capper.fit(X_train)

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

capper.right_tail_caps_
```

```
{'age': 53.0, 'fare': 66.34379999999999}
```

```
train_t[['fare', 'age']].max()
```

```
fare    65.0
age     53.0
dtype: float64
```

## 7.11 Feature Creation

Feature-engine's creation transformers combine features into new variables through various mathematical and other methods.

### 7.11.1 MathematicalCombination

#### API Reference

**class** feature_engine.creation.**MathematicalCombination**(*variables_to_combine*, *math_operations=None*, *new_variables_names=None*)

MathematicalCombination() applies basic mathematical operations to multiple features, returning one or more additional features as a result. That is, it sums, multiplies, takes the average, maximum, minimum or standard deviation of a group of variables and returns the result into new variables.

For example, if we have the variables **number_payments_first_quarter**, **number_payments_second_quarter**, **number_payments_third_quarter** and **number_payments_fourth_quarter**, we can use MathematicalCombination() to calculate the total number of payments and mean number of payments as follows:

```
transformer = MathematicalCombination(
    variables_to_combine=[
        'number_payments_first_quarter',
        'number_payments_second_quarter',
        'number_payments_third_quarter',
        'number_payments_fourth_quarter'
    ],
    math_operations=[
        'sum',
        'mean'
    ],
    new_variables_name=[
        'total_number_payments',
        'mean_number_payments'
    ]
)

Xt = transformer.fit_transform(X)
```

The transformed X, Xt, will contain the additional features **total_number_payments** and **mean_number_payments**, plus the original set of variables.

**Parameters**

**variables_to_combine** [list] The list of numerical variables to be combined.

**math_operations** [list, default=None] The list of basic math operations to be used to create the new features.

If None, all of [‘sum’, ‘prod’, ‘mean’, ‘std’, ‘max’, ‘min’] will be performed over the `variables_to_combine`. Alternatively, the user can enter the list of operations to carry out.

Each operation should be a string and must be one of the elements from the list: [‘sum’, ‘prod’, ‘mean’, ‘std’, ‘max’, ‘min’]

Each operation will result in a new variable that will be added to the transformed dataset.

**new_variables_names** [list, default=None] Names of the newly created variables. The user can enter a name or a list of names for the newly created features (recommended). The user must enter one name for each mathematical transformation indicated in the `math_operations` parameter. That is, if you want to perform mean and sum of features, you should enter 2 new variable names. If you perform only mean of features, enter 1 variable name. Alternatively, if you chose to perform all mathematical transformations, enter 6 new variable names.

The name of the variables indicated by the user should coincide with the order in which the mathematical operations are initialised in the transformer. That is, if you set math_operations = [‘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.

If `new_variable_names = None`, the transformer will assign an arbitrary name to the newly created features starting by the name of the mathematical operation, followed by the variables combined separated by -.

### Attributes

| | |
|---|---|
| **combination_dict_ :** | Dictionary containing the mathematical operation to column name pairs |
| **math_operations_ :** | List with the mathematical operations to be applied to the `variables_to_combine`. |

### Notes

Although the transformer in essence allows us to combine any feature with any of the allowed mathematical operations, its used is intended mostly for the creation of new features based on some domain knowledge. Typical examples within the financial sector are:

- Sum debt across financial products, i.e., credit cards, to obtain the total debt.
- Take the average payments to various financial products per month.
- 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.

### Methods

| | |
|---|---|
| **fit:** | This transformer does not learn parameters. |
| **transform:** | Combine the variables with the mathematical operations. |
| **fit_transform:** | Fit to the data, then transform it. |

**`fit`**(*X*, *y=None*)

This transformer does not learn parameters.

Perform dataframe checks. Creates dictionary of operation to new feature name pairs.

**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. Defaults to None.] It is not needed in this transformer. You can pass y or None.

**Returns**

**self**

**Raises**

**TypeError**

- If the input is not a Pandas DataFrame
- If any user provided variables in variables_to_combine are not numerical

**ValueError** If the variable(s) contain null values

**`transform`**(*X*)

Combine the variables with the mathematical operations.

**Parameters**

**X** [pandas dataframe of shape = [n_samples, n_features]] The data to transform.

**Returns**

**X** [Pandas dataframe, shape = [n_samples, n_features + n_operations]] The dataframe with the original variables plus the new variables.

**rtype** `DataFrame` ..

**Raises**

**TypeError** If the input is not a Pandas DataFrame

**ValueError**

- If the variable(s) contain null values
- If the dataframe is not of the same size as that used in fit()

## Example

MathematicalCombination() applies basic mathematical operations to multiple features, returning one or more additional features as a result. That is, it sums, multiplies, takes the average, maximum, minimum or standard deviation of a group of variables and returns the result into new variables.

In this example, we sum 2 variables from the house prices dataset.

```
import pandas as pd
from sklearn.model_selection import train_test_split

from feature_engine.creation import MathematicalCombination

data = pd.read_csv('houseprice.csv').fillna(0)

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
)

math_combinator = MathematicalCombination(
    variables_to_combine=['LotFrontage', 'LotArea'],
    math_operations = ['sum'],
    new_variables_names = ['LotTotal']
)

math_combinator.fit(X_train, y_train)
X_train_ = math_combinator.transform(X_train)
```

```
print(math_combinator.combination_dict_)
```

```
{'LotTotal': 'sum'}
```

```
print(X_train_.loc[:,['LotFrontage', 'LotArea', 'LotTotal']].head())
```

```
      LotFrontage  LotArea  LotTotal
64            0.0     9375    9375.0
682           0.0     2887    2887.0
960          50.0     7207    7257.0
1384         60.0     9060    9120.0
1100         60.0     8400    8460.0
```

### 7.11.2 CombineWithReferenceFeature

#### API Reference

**class** `feature_engine.creation.`**`CombineWithReferenceFeature`**(*variables_to_combine*, *reference_variables*, *operations=['sub']*, *new_variables_names=None*, *missing_values='ignore'*)

CombineWithReferenceFeature() applies basic mathematical operations between one or more reference variables and a group of variables, returning one or more additional features as a result. That is, it sums, multiplies, substracts or divides a group of features to or by a group of reference variables and returns the result into new variables.

For example, if we have the variables **number_payments_first_quarter**, **number_payments_second_quarter**, **number_payments_third_quarter**, **number_payments_fourth_quarter**, and **total_payments** we can use CombineWithReferenceFeature() to determine the percentage of total payments per month as follows:

```
transformer = CombineWithReferenceFeature(
    variables_to_combine=[
        'number_payments_first_quarter',
        'number_payments_second_quarter',
        'number_payments_third_quarter',
        'number_payments_fourth_quarter',
    ],

    reference_variables=['total_payments'],

    operations=['div'],

    new_variables_name=[
        'perc_payments_first_quarter',
        'perc_payments_second_quarter',
        'perc_payments_third_quarter',
        'perc_payments_fourth_quarter',
    ]
)

Xt = transformer.fit_transform(X)
```

The transformed X, Xt, will contain the additional features indicated in the new_variables_name list plus the original set of variables.

**Parameters**

**variables_to_combine** [list] The list of numerical variables to be combined with the reference variables.

**reference_variables** [list] The list of numerical reference variables that will be added, multiplied, or substracted from the variables_to_combine, or used as denominator for division.

**operations** [list, default=['sub']] The list of basic mathematical operations to be used in transformation.

If none, all of ['sub', 'div','add','mul'] will be performed over the variables. Alternatively, the user can enter the list of operations to carry out.

Each operation should be a string and must be one of the elements from the list: ['sub', 'div','add','mul']

Each operation will result in a new variable that will be added to the transformed dataset.

**new_variables_names** [list, default=None] Names of the newly created variables. The user can enter a list with the names for the newly created features (recommended). The user must enter as many names as new features created by the transformer. The number of new features is the number of operations times the number of reference variables times the number of variables to combine.

Thus, if you want to perform 2 operations, sub and div, combining 4 variables with 2 reference variables, you should enter 2 X 4 X 2 new variable names.

The name of the variables indicated by the user should coincide with the order in which the operations are performed by the transformer. The transformer will first carry out 'sub', then 'div', then 'add' and finally 'mul'.

If new_variable_names=None, the transformer will assign an arbitrary name to the newly created features.

**missing_values** [string, default='ignore'] Indicates if missing values should be ignored or raised. If missing_values='ignore', the transformer will ignore missing data when transforming the data. If missing_values='raise' the transformer will return an error if the training or the datasets to transform contain missing values.

## Notes

Although the transformer in essence allows us to combine any feature with any of the allowed mathematical operations, its used is intended mostly for the creation of new features based on some domain knowledge. 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.

## Methods

| | |
|---|---|
| **fit :** | This transformer does not learn parameters. |
| **transform :** | Combine the variables with the mathematical operations. |
| **fit_transform :** | Fit to the data, then transform it. |

`fit` (*X*, *y=None*)

This transformer does not learn any parameter. Performs dataframe checks.

**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. Defaults to None.] It is not needed in this transformer. You can pass y or None.

**Returns**

**self**

**Raises**

**TypeError**

- If the input is not a Pandas DataFrame
- If any user provided variables are not numerical

**ValueError** If any of the reference variables contain null values and the mathematical operation is 'div'.

**transform**(*X*)

Combine the variables with the mathematical operations.

**Parameters**

**X** [pandas dataframe of shape = [n_samples, n_features]]

**The data to transform.**

**Returns**

**X** [Pandas dataframe, shape = [n_samples, n_features + n_operations]]

**The dataframe with the operations results added as columns.**

**rtype** `DataFrame` ..

## Example

CombineWithReferenceFeature() combines a group of variables with a group of reference variables utilizing basic mathematical operations (subtraction, division, addition and multiplication), returning one or more additional features in the dataframe as a result.

In this example, we subtract 2 variables from the house prices dataset.

```
import pandas as pd
from sklearn.model_selection import train_test_split

from feature_engine.creation import CombineWithReferenceFeature

data = pd.read_csv('houseprice.csv').fillna(0)

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
)

combinator = CombineWithReferenceFeature(
```

(continues on next page)

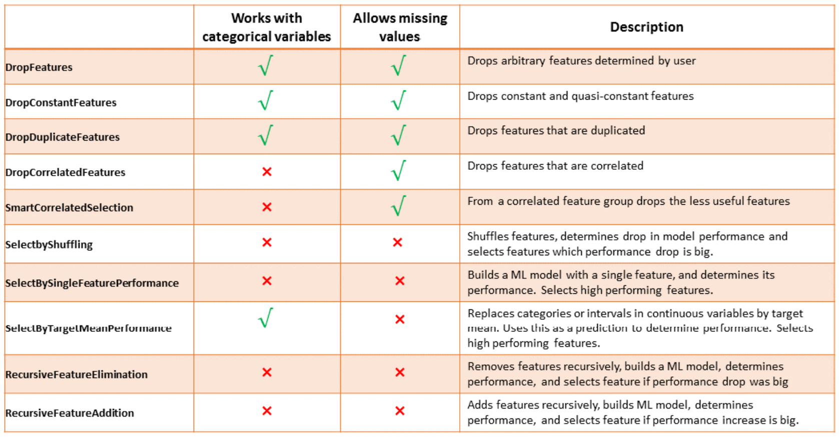
(continued from previous page)

```
    variables_to_combine=['LotArea'],
    reference_variables=['LotFrontage'],
    operations = ['sub'],
    new_variables_names = ['LotPartial']
    )

combinator.fit(X_train, y_train)
X_train_ = combinator.transform(X_train)

print(X_train_[["LotPartial","LotFrontage","LotArea"]].head())
```

```
    LotTotal  LotFrontage  LotArea
64       9375.0          0.0     9375
682      2887.0          0.0     2887
960      7157.0         50.0     7207
1384     9000.0         60.0     9060
1100     8340.0         60.0     8400
```

## 7.12 Feature Selection

Feature-engine's feature selection transformers are used to drop subsets of variables. Or in other words to select subsets of variables.



| | Works with categorical variables | Allows missing values | 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 |
| SelectbyShuffling | × | × | Shuffles features, determines drop in model performance and selects features which performance drop is big. |
| SelectBySingleFeaturePerformance | × | × | Builds a ML model with a single feature, and determines its performance. Selects high performing features. |
| SelectByTargetMeanPerformance | √ | × | Replaces categories or intervals in continuous variables by target mean. Uses this as a prediction to determine performance. Selects high performing features. |
| RecursiveFeatureElimination | × | × | Removes features recursively, builds a ML model, determines performance, and selects feature if performance drop was big |
| RecursiveFeatureAddition | × | × | Adds features recursively, builds ML model, determines performance, and selects feature if performance increase is big. |

Fig. 1: Summary of Feature-engine's selectors main characteristics

### 7.12.1 DropFeatures

#### API Reference

**class** `feature_engine.selection.`**`DropFeatures`**(*features_to_drop*)

DropFeatures() drops a list of variable(s) indicated by the user from the dataframe.

**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.

**Parameters**

**features_to_drop** [str or list, default=None] Variable(s) to be dropped from the dataframe

### Methods

| | |
|---|---|
| **fit:** | This transformer does not learn any parameter. |
| **transform:** | Drops indicated features. |
| **fit_transform:** | Fit to data, then transform it. |

**fit**(*X*, *y=None*)

This transformer does not learn any parameter.

Verifies that the input X is a pandas dataframe, and that the variables to drop exist in the training dataframe.

**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.

**Returns**

**self**

**transform**(*X*)

Return dataframe with selected features.

**Parameters**

**X** [pandas dataframe of shape = [n_samples, n_features].] The input dataframe.

**Returns**

**X_transformed: pandas dataframe of shape = [n_samples, n_selected_features]** Pandas dataframe with the selected features.

**rtype** `DataFrame` ..

### Example

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.

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

from feature_engine.selection import DropFeatures

# Load dataset
def load_titanic():
        data = pd.read_csv('https://www.openml.org/data/get_csv/16826755/phpMYEkMl')
```

(continues on next page)

(continued from previous page)

```
        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)
        return data

# load data as pandas dataframe
data = load_titanic()

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

# set up the transformer
transformer = DropFeatures(
    features_to_drop=['sibsp', 'parch', 'ticket', 'fare', 'body', 'home.dest']
)

# fit the transformer
transformer.fit(X_train)

# transform the data
train_t = transformer.transform(X_train)

train_t.columns
```

```
Index(['pclass', 'sex', 'age', 'cabin', 'embarked' 'boat'],
      dtype='object')
```

### 7.12.2 DropConstantFeatures

#### API Reference

**class** feature_engine.selection.**DropConstantFeatures**(*tol=1*, *variables=None*, *missing_values='raise'*)

Drop constant and quasi-constant variables from a dataframe. Constant variables show the same value across all the observations in the dataset. Quasi-constant variables show the same value in almost all the observations in the dataset.

By default, DropConstantFeatures() drops only constant variables. This transformer works with both 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.

**Parameters**

**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.

**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=raises] Whether the missing values should be raised as error, ignored or included as an additional value of the variable, when considering if the feature is constant or quasi-constant. Takes values 'raise', 'ignore', 'include'.

### Attributes

| | |
|---|---|
| **features_to_drop_:** | List with constant and quasi-constant features. |

**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

### Methods

| | |
|---|---|
| **fit:** | Find constant and quasi-constant features. |
| **transform:** | Remove constant and quasi-constant features. |
| **fit_transform:** | Fit to the data. Then transform it. |

**`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.

**Returns**

**self**

**`transform`**(*X*)

Return dataframe with selected features.

**Parameters**

**X** [pandas dataframe of shape = [n_samples, n_features].] The input dataframe.

**Returns**

**X_transformed: pandas dataframe of shape = [n_samples, n_selected_features]**
Pandas dataframe with the selected features.

**rtype** `DataFrame` ..

### Example

The DropConstantFeatures() drops constant and quasi-constant variables from a dataframe. By default, DropConstant-Features drops only constant variables. This transformer works with both numerical and categorical variables.

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

from feature_engine.selection import DropConstantFeatures

# 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)
        return data

# load data as pandas dataframe
data = load_titanic()

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

# set up the transformer
transformer = DropConstantFeatures(tol=0.7)

# fit the transformer
transformer.fit(X_train)

# transform the data
train_t = transformer.transform(X_train)

transformer.constant_features_
```

```
['parch', 'cabin', 'embarked']
```

```
X_train['embarked'].value_counts() / len(X_train)
```

```
S    0.711790
C    0.197598
Q    0.090611
Name: embarked, dtype: float64
```

```
X_train['parch'].value_counts() / len(X_train)
```

```
0    0.771834
1    0.125546
2    0.086245
3    0.005459
4    0.004367
5    0.003275
```

(continues on next page)

(continued from previous page)

```
6    0.002183
9    0.001092
Name: parch, dtype: float64
```

## 7.12.3 DropDuplicateFeatures

### API Reference

**class** feature_engine.selection.**DropDuplicateFeatures**(*variables=None*, *missing_values='ignore'*)

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.

The transformer will first identify and store the duplicated variables. Next, the transformer will drop these variables from a dataframe.

**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] Takes values 'raise' and 'ignore'. Whether the missing values should be raised as error or ignored when finding duplicated features.

#### 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. |

#### Methods

| | |
|---|---|
| **fit:** | Find duplicated features. |
| **transform:** | Remove duplicated features |
| **fit_transform:** | Fit to data. Then transform it. |

**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.

**Returns**

**self**

**transform**(*X*)

Return dataframe with selected features.

**Parameters**

**X** [pandas dataframe of shape = [n_samples, n_features].] The input dataframe.

**Returns**

**X_transformed: pandas dataframe of shape = [n_samples, n_selected_features]** Pandas dataframe with the selected features.

**rtype** `DataFrame` ..

## Example

The DropDuplicateFeatures() finds and removes duplicated variables from a dataframe. The user can pass a list of variables to examine, or alternatively the selector will examine all variables in the data set.

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

from feature_engine.selection import DropDuplicateFeatures

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 = data[['pclass', 'survived', 'sex', 'age', 'sibsp', 'parch', 'cabin',
↪'embarked']]
        data = pd.concat([data, data[['sex', 'age', 'sibsp']]], axis=1)
        data.columns = ['pclass', 'survived', 'sex', 'age', 'sibsp', 'parch', 'cabin',
↪ 'embarked',
                        'sex_dup', 'age_dup', 'sibsp_dup']
        return data

# load data as pandas dataframe
data = load_titanic()
data.head()

# 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)

# set up the transformer
transformer = DropDuplicateFeatures()

# fit the transformer
transformer.fit(X_train)

# transform the data
train_t = transformer.transform(X_train)

train_t.columns
```

```
Index(['pclass', 'sex', 'age', 'sibsp', 'parch', 'cabin', 'embarked'], dtype='object')
```

```
transformer.duplicated_features_
```

```
{'age_dup', 'sex_dup', 'sibsp_dup'}
```

```
transformer.duplicated_feature_sets_
```

```
[{'sex', 'sex_dup'}, {'age', 'age_dup'}, {'sibsp', 'sibsp_dup'}]
```

### 7.12.4 DropCorrelatedFeatures

#### API Reference

**class** `feature_engine.selection.`**`DropCorrelatedFeatures`**(*variables=None*, *method='pearson'*, *threshold=0.8*, *missing_values='ignore'*)

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.

**Parameters**

**variables** [list, default=None] The list of variables to evaluate. If None, the transformer will evaluate all numerical variables in the dataset.

**method** [string, default='pearson'] Can take 'pearson', 'spearman' or'kendall'. 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

**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] Takes values 'raise' and 'ignore'. Whether the missing values should be raised as error or ignored when determining correlation.

#### 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. |

**See also:**

**`pandas.corr`**

**`feature_engine.selection.SmartCorrelationSelection`**

### Methods

| | |
|---|---|
| **fit:** | Find correlated features. |
| **transform:** | Remove correlated features. |
| **fit_transform:** | Fit to the data. Then transform it. |

**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.

**Returns**

**self**

**transform** (*X*)
Return dataframe with selected features.

**Parameters**

**X** [pandas dataframe of shape = [n_samples, n_features].] The input dataframe.

**Returns**

**X_transformed: pandas dataframe of shape = [n_samples, n_selected_features]**
Pandas dataframe with the selected features.

**rtype** `DataFrame` ..

## Example

The DropCorrelatedFeatures() finds and removes correlated variables from a dataframe. The user can pass a list of variables to examine, or alternatively the selector will examine all numerical variables in the data set.

```python
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,
                               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
```

(continues on next page)

(continued from previous page)

```
X = make_data()

tr = DropCorrelatedFeatures(variables=None, method='pearson', threshold=0.8)

Xt = tr.fit_transform(X)

tr.correlated_feature_sets_
```

```
[{'var_0', 'var_8'}, {'var_4', 'var_6', 'var_7', 'var_9'}]
```

```
tr.correlated_features_
```

```
{'var_6', 'var_7', 'var_8', 'var_9'}
```

```
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

     var_11
0 -1.989335
1 -1.309524
2 -0.852703
3  0.484649
4 -0.186530
```

## 7.12.5 SmartCorrelatedSelection

### API Reference

**class** feature_engine.selection.**SmartCorrelatedSelection**(*variables=None, method='pearson', threshold=0.8, missing_values='ignore', selection_method='missing_values', estimator=None, scoring='roc_auc', cv=3*)

SmartCorrelatedSelection() finds groups of correlated features and then selects, from each group, a feature following certain criteria:

- Feature with least missing values
- Feature with most unique values
- Feature with highest variance
- Best performing feature according to estimator entered by user

SmartCorrelatedSelection() returns a dataframe containing from each group of correlated features, the selected variable, plus all original 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.

**Parameters**

**variables** [list, default=None] The list of variables to evaluate. If None, the transformer will evaluate all numerical variables in the dataset.

**method** [string, default='pearson'] Can take 'pearson', 'spearman' or'kendall'. 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

**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] Takes values 'raise' and 'ignore'. Whether the missing values should be raised as error or ignored when determining correlation.

**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 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 the correlated feature group and retains the feature with the highest importance.

**estimator** [object, default = None] A Scikit-learn estimator for regression or classification.

**scoring** [str, default='roc_auc'] Desired metric to optimise 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, default=3] Cross-validation fold to be used to fit the estimator.

## Attributes

| | |
|---|---|
| **correlated_feature_sets_:** | Groups of correlated features. Each list is a group of correlated features. |
| **features_to_drop_:** | The correlated features to remove from the dataset. |

**See also:**

**`pandas.corr`**

*`feature_engine.selection.DropCorrelatedFeatures`*

### Methods

| | |
|---|---|
| **fit:** | Find best feature from each correlated groups. |
| **transform:** | Return selected features. |
| **fit_transform:** | Fit to the data. Then transform it. |

**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'.

**Returns**

**self**

**transform**(*X*)

Return dataframe with selected features.

**Parameters**

**X** [pandas dataframe of shape = [n_samples, n_features].] The input dataframe.

**Returns**

**X_transformed: pandas dataframe of shape = [n_samples, n_selected_features]**
Pandas dataframe with the selected features.

**rtype** `DataFrame` ..

### Example

```
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)

    # 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()
```

(continues on next page)

(continued from previous page)

```
# set up the selector
tr = SmartCorrelatedSelection(
    variables=None,
    method="pearson",
    threshold=0.8,
    missing_values="raise",
    selection_method="variance",
    estimator=None,
)

Xt = tr.fit_transform(X)

tr.correlated_feature_sets_
```

```
[{'var_0', 'var_8'}, {'var_4', 'var_6', 'var_7', 'var_9'}]
```

```
tr.selected_features_
```

```
['var_1', 'var_2', 'var_3', 'var_5', 'var_10', 'var_11', 'var_8', 'var_7']
```

```
print(print(Xt.head()))
```

```
      var_1     var_2     var_3     var_5    var_10    var_11     var_8  \
0 -2.376400 -0.247208  1.210290  0.091527  2.070526 -1.989335  2.070483
1  1.969326 -0.126894  0.034598 -0.186802  1.184820 -1.309524  2.421477
2  1.499174  0.334123 -2.233844 -0.313881 -0.066448 -0.852703  2.263546
3  0.075341  1.627132  0.943132 -0.468041  0.713558  0.484649  2.792500
4  0.372213  0.338141  0.951526  0.729005  0.398790 -0.186530  2.186741

      var_7
0 -2.230170
1 -1.447490
2 -2.240741
3 -3.534861
4 -2.053965
```

## 7.12.6 SelectByShuffling

### API Reference

**class** feature_engine.selection.**SelectByShuffling**(*estimator=RandomForestClassifier(), scoring='roc_auc', cv=3, threshold=None, variables=None, random_state=None*)

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.

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 the drop in performance evaluated.

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.

**Parameters**

**variables** [str or list, default=None] The list of variable(s) to be shuffled from the dataframe. If None, the transformer will shuffle all numerical variables in the dataset.

**estimator** [object, default = RandomForestClassifier()] A Scikit-learn estimator for regression or classification.

**scoring** [str, default='roc_auc'] Desired metric to optimise the performance for 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 = None] The value that defines if a feature will be kept or removed. Note that for metrics like roc-auc, r2_score and accuracy, the thresholds will be floats between 0 and 1. For metrics like the mean_square_error and the root_mean_square_error the threshold will be a big number. The threshold can be defined by the user. If None, the selector will select features which performance drift is smaller than the mean performance drift across all features.

**cv** [int, default=3] Desired number of cross-validation fold to be used to fit the estimator.

**random_state: int, default=None** Controls the randomness when shuffling features.

## Attributes

| | |
|---|---|
| **initial_model_performance_:** | Performance of the model trained using the original dataset. |
| **performance_drifts_:** | Dictionary with the performance drift per shuffled feature. |
| **features_to_drop_:** | List with the features to remove from the dataset. |

## Methods

| | |
|---|---|
| **fit:** | Find the important features. |
| **transform:** | Reduce X to the selected features. |
| **fit_transform:** | Fit to data, then transform it. |

**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.

**Returns**

**self**

**transform**(*X*)

Return dataframe with selected features.

**Parameters**

**X** [pandas dataframe of shape = [n_samples, n_features].] The input dataframe.

**Returns**

**X_transformed: pandas dataframe of shape = [n_samples, n_selected_features]**
Pandas dataframe with the selected features.

**rtype** `DataFrame` ..

## Example

The SelectByShuffling() selects important features if permutation their values at random produces a decrease in the initial model performance.

```
import pandas as pd
from sklearn.datasets import load_diabetes
from sklearn.linear_model import LinearRegression
from feature_engine.selection import ShuffleFeaturesSelector

# load dataset
diabetes_X, diabetes_y = load_diabetes(return_X_y=True)
X = pd.DataFrame(diabetes_X)
y = pd.DataFrame(diabetes_y)

# initialize linear regresion estimator
linear_model = LinearRegression()

# initialize feature selector
tr = SelectByShuffling(estimator=linear_model, scoring="r2", cv=3)

# fit transformer
Xt = tr.fit_transform(X, y)

tr.initial_model_performance_
```

```
0.488702767247119
```

```
tr.performance_drifts_
```

```
{0: -0.02368121940502793,
 1: 0.017909161264480666,
 2: 0.18565460365508413,
 3: 0.07655405817715671,
 4: 0.4327180164470878,
 5: 0.16394693824418372,
 6: -0.012876023845921625,
 7: 0.01048781540981647,
 8: 0.3921465005640224,
 9: -0.01427065640301245}
```

```
tr.selected_features_
```

```
[1, 2, 3, 4, 5, 7, 8]
```

```
print(Xt.head())
```

```
          1         2         3         4         5         7         8
0  0.050680  0.061696  0.021872 -0.044223 -0.034821 -0.002592  0.019908
1 -0.044642 -0.051474 -0.026328 -0.008449 -0.019163 -0.039493 -0.068330
2  0.050680  0.044451 -0.005671 -0.045599 -0.034194 -0.002592  0.002864
3 -0.044642 -0.011595 -0.036656  0.012191  0.024991  0.034309  0.022692
4 -0.044642 -0.036385  0.021872  0.003935  0.015596 -0.002592 -0.031991
None
```

### 7.12.7 SelectBySingleFeaturePerformance

#### API Reference

**class** feature_engine.selection.**SelectBySingleFeaturePerformance**(*estimator=RandomForestClassifier(), scoring='roc_auc', cv=3, threshold=None, variables=None*)

SelectBySingleFeaturePerformance() selects features based on the performance obtained from a machine learning model trained utilising a single feature. In other words, it trains a machine learning model for every single feature, utilising that individual feature, then determines each model performance. If the performance of the model based on the single feature is greater than a user specified threshold, then the feature is retained, otherwise removed.

The models are trained on the individual features using cross-validation. The performance metric to evaluate and the machine learning model to train are specified by the user.

**Parameters**

**variables** [str or list, default=None] The list of variable(s) to be evaluated. If None, the transformer will evaluate all numerical variables in the dataset.

**estimator** [object, default = RandomForestClassifier()] A Scikit-learn estimator for regression or classification.

**scoring** [str, default='roc_auc'] Desired metric to optimise the performance for 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 = None] The value that defines if a feature will be kept or removed.

The r2varies between 0 and 1. So a threshold needs to be set-up within these boundaries.

The roc-auc varies between 0.5 and 1. So a threshold needs to be set-up within these boundaries.

For metrics like the mean_square_error and the root_mean_square_error the threshold will be a big number.

The threshold can be specified by the user. If None, it will be automatically set to the mean performance value of all features.

**cv** [int, default=3] Desired number of cross-validation fold to be used to fit the estimator.

### Attributes

| | |
|---|---|
| **features_to_drop_:** | List with the features to remove from the dataset. |
| **feature_performance_:** | Dictionary with the single feature model performance per feature. |

### Methods

| | |
|---|---|
| **fit:** | Find the important features. |
| **transform:** | Reduce X to the selected features. |
| **fit_transform:** | Fit to data, then transform it. |

**fit** (*X*, *y*)

Select 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.

**Returns**

**self**

**transform** (*X*)

Return dataframe with selected features.

**Parameters**

**X** [pandas dataframe of shape = [n_samples, n_features].] The input dataframe.

**Returns**

**X_transformed: pandas dataframe of shape = [n_samples, n_selected_features]**
Pandas dataframe with the selected features.

**rtype** `DataFrame` ..

## Example

The SelectBySingleFeaturePerformance()selects features based on the performance of machine learning models trained using individual features. In other words, selects features based on their individual performance, returned by estimators trained on only that particular feature.

```
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.DataFrame(diabetes_y)

# initialize feature selector
sel = SelectBySingleFeaturePerformance(
```

(continues on next page)

(continued from previous page)

```
        estimator=LinearRegression(), scoring="r2", cv=3, threshold=0.01)

# fit transformer
sel.fit(X, y)

sel.selected_features_
```

```
[0, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
sel.feature_importance_
```

```
{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}
```

### 7.12.8 SelectByTargetMeanPerformance

#### API Reference

**class** feature_engine.selection.**SelectByTargetMeanPerformance**(*variables=None, scoring='roc_auc_score', threshold=0.5, bins=5, strategy='equal_width', cv=3, random_state=None*)

SelectByTargetMeanPerformance() selects features by using the mean value of the target per category or bin, if the variable is numerical, as proxy of target estimation, by determining its performance.

Works with both numerical and categorical variables.

The transformer works as follows:

1. Separates the training set into train and test sets.

Then, for each categorical variable:

2. Determine the mean value of the target for each category of the variable using the train set (equivalent of Target mean encoding)

3. Replaces the categories in the test set, by the target mean values determined from the train set

4. Using the encoded variable calculates the roc-auc or r2

5. Selects the features which roc-auc or r2 is bigger than the indicated threshold

For each numerical variable:

2. Discretize the variable into intervals of equal width or equal frequency (uses the discretizers of Feature-engine)

3. Determine the mean value of the target for each interval of the variable using the train set (equivalent of Target mean encoding)

4. Replaces the intervals in the test set, by the target mean values determined from the train set

5. Using the encoded variable calculates the roc-auc or r2

6. Selects the features which roc-auc or r2 is bigger than the indicated threshold

**Parameters**

**variables** [list, default=None] The list of variables to evaluate. If None, the transformer will evaluate all variables in the dataset.

**scoring** [string, default='roc_auc_score'] This indicates the metrics score to perform the feature selection. The current implementation supports 'roc_auc_score' and 'r2_score'.

**threshold** [float, default = None] The performance threshold above which a feature will be selected.

**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 to create the bins for discretization of numerical variables of equal width or equal frequency.

**cv** [int, default=3] Desired number of cross-validation fold to be used to fit the estimator.

**random_state** [int, default=0] The random state setting in the train_test_split method.

## Attributes

| | |
|---|---|
| **features_to_drop_:** | List with the features to remove from the dataset. |
| **feature_performance_:** | Dictionary with the performance proxy per feature. |

## Methods

| | |
|---|---|
| **fit:** | Find the important features. |
| **transform:** | Reduce X to the selected features. |
| **fit_transform:** | Fit to data, then transform it. |

**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.

**Returns**

**self**

**transform** (*X*)

Return dataframe with selected features.

**Parameters**

**X** [pandas dataframe of shape = [n_samples, n_features].] The input dataframe.

**Returns**

**X_transformed: pandas dataframe of shape = [n_samples, n_selected_features]**
Pandas dataframe with the selected features.

**rtype** `DataFrame` ..

## Example

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_auc_score
from feature_engine.selection import SelectByTargetMeanPerformance

# load data
data = pd.read_csv('../titanic.csv')

# extract cabin letter
data['cabin'] = data['cabin'].str[0]

# 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')

# 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=0.3,
    random_state=0)


# feature engine automates the selection for both categorical and numerical
# variables
sel = SelectByTargetMeanPerformance(
    variables=None,
    scoring="roc_auc_score",
    threshold=0.6,
    bins=3,
    strategy="equal_frequency",
    cv=2,# cross validation
    random_state=1, #seed for reproducibility
)

# find important features
sel.fit(X_train, y_train)
```

(continues on next page)

(continued from previous page)

```
sel.variables_categorical_
```

```
['pclass', 'sex', 'sibsp', 'parch', 'cabin', 'embarked']
```

```
sel.variables_numerical_
```

```
['age', 'fare']
```

```
sel.feature_performance_
```

```
{'pclass': 0.6802934787230475,
 'sex': 0.7491365252482871,
 'age': 0.5345141148737766,
 'sibsp': 0.5720480307315783,
 'parch': 0.5243557188989476,
 'fare': 0.6600883312700917,
 'cabin': 0.6379782658154696,
 'embarked': 0.5672382248783936}
```

```
sel.features_to_drop_
```

```
['age', 'sibsp', 'parch', 'embarked']
```

```
# remove features
X_train = sel.transform(X_train)
X_test = sel.transform(X_test)

X_train.shape, X_test.shape
```

```
((914, 4), (392, 4))
```

### 7.12.9 RecursiveFeatureElimination

#### API Reference

**class** feature_engine.selection.**RecursiveFeatureElimination**(*estimator=RandomForestClassifier()*, *scoring='roc_auc'*, *cv=3*, *threshold=0.01*, *variables=None*)

RecursiveFeatureElimination selects features following a recursive 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 one feature -the least important- and fit a new estimator with the remaining features.
4. Calculate the performance of the new estimator.
5. Calculate the difference in performance between the new and the original estimator.

6. If the performance drops beyond the threshold, then that feature is important and will be kept. Otherwise, that feature is removed.

7. Repeat steps 3-6 until all features have been evaluated.

Model training and performance calculation are done with cross-validation.

**Parameters**

**variables** [str or list, default=None] The list of variable to be evaluated. If None, the transformer will evaluate all numerical features in the dataset.

**estimator** [object, default = RandomForestClassifier()] A Scikit-learn estimator for regression or classification. The estimator must have either a `feature_importances` or `coef_` attribute after fitting.

**scoring** [str, default='roc_auc'] Desired metric to optimise 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 if a feature will be kept or removed. Note that for metrics like roc-auc, r2_score and accuracy, the thresholds will be floats between 0 and 1. For metrics like the mean_square_error and the root_mean_square_error the threshold will be a big number. The threshold must be defined by the user. Bigger thresholds will select less features.

**cv** [int, default=3] Cross-validation fold to be used to fit the estimator.

## Attributes

| | |
|---|---|
| **initial_model_performance_ :** | Performance of the model trained using the original dataset. |
| **feature_importances_ :** | Pandas Series with the feature importance |
| **performance_drifts_:** | Dictionary with the performance drift per examined feature. |
| **features_to_drop_:** | List with the features to remove from the dataset. |

## Methods

| | |
|---|---|
| **fit:** | Find the important features. |
| **transform:** | Reduce X to the selected features. |
| **fit_transform:** | Fit to data, then transform it. |

**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.

**Returns**

**self**

**transform**(*X*)

Return dataframe with selected features.

**Parameters**

**X** [pandas dataframe of shape = [n_samples, n_features].] The input dataframe.

**Returns**

**X_transformed: pandas dataframe of shape = [n_samples, n_selected_features]**
Pandas dataframe with the selected features.

**rtype** `DataFrame` ..

## Example

```
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.DataFrame(diabetes_y)

# initialize linear regresion estimator
linear_model = LinearRegression()

# initialize feature selector
tr = RecursiveFeatureElimination(estimator=linear_model, scoring="r2", cv=3)

# fit transformer
Xt = tr.fit_transform(X, y)

# get the initial linear model performance, using all features
tr.initial_model_performance_
```

```
0.488702767247119
```

```
# 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}
```

```
# get the selected features
tr.selected_features_
```

```
[1, 3, 5, 2, 8, 4]
```

```
print(Xt.head())
```

```
          1         3         5         2         8         4
0  0.050680  0.021872 -0.034821  0.061696  0.019908 -0.044223
1 -0.044642 -0.026328 -0.019163 -0.051474 -0.068330 -0.008449
2  0.050680 -0.005671 -0.034194  0.044451  0.002864 -0.045599
3 -0.044642 -0.036656  0.024991 -0.011595  0.022692  0.012191
4 -0.044642  0.021872  0.015596 -0.036385 -0.031991  0.003935
```

### 7.12.10 RecursiveFeatureAddition

#### API Reference

**class** `feature_engine.selection.`**`RecursiveFeatureAddition`**(*estimator=RandomForestClassifier()*, *scoring='roc_auc'*, *cv=3*, *threshold=0.01*, *variables=None*)

RecursiveFeatureAddition selects features following a recursive 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 its performance.
4. Add the second most important feature and train a new estimator.

5. Calculate the difference in performance between the last estimator and the previous one.

6. If the performance increases beyond the threshold, then that feature is important and will be kept. Otherwise, that feature is removed . 7. Repeat steps 4-6 until all features have been evaluated.

Model training and performance calculation are done with cross-validation.

**Parameters**

**variables** [str or list, default=None] The list of variable to be evaluated. If None, the transformer will evaluate all numerical features in the dataset.

**estimator** [object, default = RandomForestClassifier()] A Scikit-learn estimator for regression or classification. The estimator must have either a `feature_importances` or `coef_` attribute after fitting.

**scoring** [str, default='roc_auc'] Desired metric to optimise 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 if a feature will be kept or removed. Note that for metrics like roc-auc, r2_score and accuracy, the thresholds will be floats between 0 and 1. For metrics like the mean_square_error and the root_mean_square_error the threshold will be a big number. The threshold must be defined by the user. Bigger thresholds will select less features.

**cv** [int, default=3] Cross-validation fold to be used to fit the estimator.

### Attributes

| | |
|---|---|
| **initial_model_performance_ :** | Performance of the model trained using the original dataset. |
| **feature_importances_ :** | Pandas Series with the feature importance. |
| **performance_drifts_:** | Dictionary with the performance drift per examined feature. |
| **features_to_drop_:** | List with the features to remove from the dataset. |

### Methods

| | |
|---|---|
| **fit:** | Find the important features. |
| **transform:** | Reduce X to the selected features. |
| **fit_transform:** | Fit to data, then transform it. |

**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.

**Returns**

**self**

**transform**(*X*)
Return dataframe with selected features.

**Parameters**

**X** [pandas dataframe of shape = [n_samples, n_features].] The input dataframe.

**Returns**

**X_transformed: pandas dataframe of shape = [n_samples, n_selected_features]**
Pandas dataframe with the selected features.

**rtype** `DataFrame` ..

### Example

```
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.DataFrame(diabetes_y)

# initialize linear regresion estimator
linear_model = LinearRegression()
```

(continues on next page)

(continued from previous page)

```

# initialize feature selector
tr = RecursiveFeatureElimination(estimator=linear_model, scoring="r2", cv=3)

# fit transformer
Xt = tr.fit_transform(X, y)

# get the initial linear model performance, using all features
tr.initial_model_performance_
```

```
0.488702767247119
```

```
# Get the performance drift of each feature
tr.performance_drifts_
```

```
{4: 0,
 8: 0.2837159006046677,
 2: 0.1377700238871593,
 5: 0.0023329006089969906,
 3: 0.0187608758643259,
 1: 0.0027994385024313617,
 7: 0.0026951300105543807,
 6: 0.002683967832484757,
 9: 0.0003040126429713075,
 0: -0.007386876030245182}
```

```
# get the selected features
tr.selected_features_
```

```
[4, 8, 2, 3]
```

```
print(Xt.head())
```

```
          4         8         2         3
0 -0.044223  0.019908  0.061696  0.021872
1 -0.008449 -0.068330 -0.051474 -0.026328
2 -0.045599  0.002864  0.044451 -0.005671
3  0.012191  0.022692 -0.011595 -0.036656
4  0.003935 -0.031991 -0.036385  0.021872
```

## 7.13 Scikit-learn Wrapper

Feature-engine's Scikit-learn wrappers wrap Scikit-learn transformers allowing their implementation only on a selected subset of features.

### 7.13.1 SklearnTransformerWrapper

#### API Reference

**class** `feature_engine.wrappers.`**`SklearnTransformerWrapper`**(*variables=None*, *transformer=None*)

Wrapper for Scikit-learn pre-processing transformers like the SimpleImputer() or OrdinalEncoder(), to allow the use of the transformer on a selected group of variables.

**Parameters**

**variables** [list, default=None] The list of variables to be imputed.

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.

**transformer** [sklearn transformer, default=None] The desired Scikit-learn transformer.

#### Methods

| | |
|---|---|
| **fit:** | Fit Scikit-learn transformers |
| **transform:** | Transforms with Scikit-learn transformers |
| **fit_transform:** | Fit to data, then transform it. |

**fit**(*X*, *y=None*)

The `fit` method allows Scikit-learn transformers to learn the required parameters from the training data set.

If transformer is OneHotEncoder, OrdinalEncoder or SimpleImputer, all variables indicated in the `` `variables` `` parameter will be transformed. When the variables parameter is None, the SklearnWrapper will automatically select and transform all features in the dataset, numerical or otherwise.

For all other Scikit-learn transformers only numerical variables will be transformed. The SklearnWrapper will check that the variables indicated in the variables parameter are numerical, or alternatively, if variables is None, it will automatically select the numerical variables in the data set.

**Parameters**

**X** [Pandas DataFrame] The dataset to fit the transformer

**y** [pandas Series, default=None] This parameter exists only for compatibility with sklearn.pipeline.Pipeline.

**Returns**

**self**

**Raises**

**TypeError** If the input is not a Pandas DataFrame

**transform**(*X*)

Apply the transformation to the dataframe. Only the selected features will be modified.

If transformer is OneHotEncoder, dummy features are concatenated to the source dataset. Note that the original categorical variables will not be removed from the dataset after encoding. If this is the desired effect, please use Feature-engine's OneHotEncoder instead.

**Parameters**

**X** [Pandas DataFrame] The data to transform

**Returns**

**X** [Pandas DataFrame] The transformed dataset.

**rtype** `DataFrame` ..

**Raises**

**TypeError** If the input is not a Pandas DataFrame

## Example

Implements Scikit-learn transformers like the SimpleImputer, the OrdinalEncoder or most scalers only to the selected subset of features.

```
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(
    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)
```

## 7.14 Tutorials

### 7.14.1 Code tutorials

Coming Soon!

### 7.14.2 Kaggle Kernels

- Feature selection for bank customer satisfaction prediction
- Feature engineering and selection for house price prediction
- Feature creation for wine quality prediction

### 7.14.3 Video tutorials

You can find some videos on how to use Feature-engine in our you tube Feature-engine playlist. The list is a bit short at the moment, apologies.

## 7.15 How To

Find jupyter notebooks with examples of each transformer functionality. Within each folder, you will find a jupyter notebook showcasing the functionality of each transformer.

## 7.16 Books

You can learn more about how to use Feature-engine and feature engineering in general in the following books:

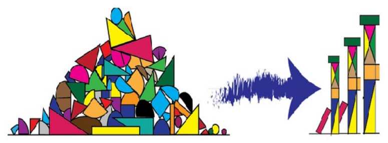


Fig. 3: Feature Engineering for Machine Learning

## 7.17 Courses

You can learn more about how to use Feature-engine and, feature engineering and feature selection in general in the following online courses:

## 7.18 Blogs and Videos

Articles and videos about Feature-engine and feature engineering and selection in general.

### 7.18.1 Blogs

- 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.
- Feature Selection for Machine Learning: A comprehensive Overview.

### 7.18.2 Videos

- Optimising Feature Engineering Pipelines with Feature-engine, Pydata Cambridge 2020, from minute 51:43.

### 7.18.3 En Español

- Ingeniería de variables para machine learning, Curso Online.
- Ingeniería de variables, MachinLenin, charla con video online.
- Ingeniería de atributos para aprendizaje automático: una revisión completa, Artículo.
- Paquetes Python de código abierto para la ingeniería de atributos: Comparaciones y conclusiones, Artículo.
- Feature-engine: Un paquete Python de código abierto para la ingeniería de atributos, Artículo.

More resources will be added as they appear online!

## 7.19 Contributing

Feature-engine is an open source project, originally designed to support the online course Feature Engineering for Machine Learning in Udemy, but has now gained popularity and supports transformations beyond those taught in the course.

Feature-engine is currently supported by a small 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.

There are many ways to contribute to Feature-engine:

- Create a new variable transformer
- Add additional functionality to current transformers
- Fix a bug
- Submit a bug report or feature request on GitHub issues.

- Add a Jupyter notebook to our Jupyter notebooks example gallery.
- Improve our documentation.
- Write unit tests.
- Write a blog, tweet, or share our project with others.

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!.

### 7.19.1 Getting in touch

We prefer to handle most contributions through the github repository. You can also join our mailing list.

1. Open issues.
2. Mailing list.

### 7.19.2 Contributing Guide

#### Getting Started with Feature-engine on GitHub

Feature-engine is hosted on GitHub.

A typical contributing workflow goes like this:

1. **Find** a bug while using Feature-engine, **suggest** new functionality, or **pick up** an issue from our repo.
2. **Discuss** with us your approach to resolve the issue.
3. Then, **fork** the repository into your GitHub account.
4. **Clone** your fork into your local computer.
5. **Code** the feature, the tests and update or add the documentation.
6. Make a **Pull Request (PR)** with your changes.
7. **Review** the code with one of us, who will guide you to a final submission.
8. **Merge** your contribution into the Feature-engine source code base.

It is important that we communicate right from the beginning, so we have a clear understanding of how you would like to get involved and what is needed to complete the task.

### Forking the Repository

When you fork the repository, you create a copy of Feature-engine's source code into your account, which you can edit. To fork Feature-engine's repository, click the **fork** button in the upper right corner of Feature-engine's GitHub page.

### Setting up the Development Environment

Once you forked the repository, follow these steps to set up your development environment:

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/solegalli/feature_engine.git
$ git remote -v
origin    https:/
↪/github.com/YOUR_USERNAME/feature_engine.git (fetch)
origin    https:/
↪/github.com/YOUR_USERNAMEfeature_engine.git (push)
upstream ␣
↪https://github.com/solegalli/feature_engine.git (fetch)
upstream␣
↪ https://github.com/solegalli/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 for tips on **Keeping your fork up to date**.

3. Optional but highly advisable: Create a virtual environment. Use any virtual environment tool of your choice. Some examples include:

1. venv
2. conda environments

4. Change directory into the cloned repository:

```
$ cd feature_engine
```

5. Install Feature_engine in developer mode:

```
$ pip install -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.

6. Install the additional dependencies for tests and documentation:

```
$ pip install -r test_requirements.txt
$ pip install -r docs/requirements.txt
```

7. Make sure that your local master is up to date with the remote master:

```
$ git pull --rebase upstream master
```

   If you just cloned your fork, your local master 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 master to the main repository master, read below the section **Keeping your fork up to date**.

8. Create a new branch where you will develop your feature:

```
$ git checkout -b myfeaturebranch
```

   There are 3 things to keep in mind when creating a feature branch. First, give the branch a name that identifies the feature you are going to build. Second, make sure you checked out your branch from master branch. Third, make sure your local master was updated with the upstream master.

9. Once your code is ready, commit your changes and push your branch to your fork:

```
$ git add .
$ git commit -m "my commit message"
$ git push origin myfeaturebranch
```

   This will add a new branch to your fork. In the commit message, be succint, describe what is being added and if it resolves an issue, make sure to **reference the issue in the commit message** (you can also do this from Github).

   10. Go to your fork in Github, you will see the branch you just pushed and next to it a button to create a PR. Go ahead and create a PR from your feature branch to Feature_engine's master branch.

### Developing a New Feature

First thing, make a pull request (PR). Once you have written a bit of code for your new feature, or bug fix, or example, or whatever task you are working on, make a PR. The PR should be made from your feature_branch (in your fork), to Feature-engine's master branch in the main repository.

When you develop a new feature, or bug, or any 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, should a major catastrophe arise
4. If your feature takes some time to develop, make sure you rebase upstream/master onto your feature branch

Once your contribution contains the new code, the tests, and ideally the documentation, the review process will start. Likely, there will be some back and forth until the final submission.

Once the submission is reviewed and provided the continuous integration tests have passed and the code is up to date with Feature-engine's master branch, we will be ready to "Squash and Merge" your contribution into the `master` 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 master, you will be listed as a Feature-engine contributor :)

### Testing the Code in the PR

You can test the code functionality either in your development environment or using tox. If you want to use tox:

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
```

If the tests pass, the local setup is complete.

If you want to know more about tox follow this link. If you want to know why we prefer tox, this article will tell you everything ;)

If you prefer not to use tox, there are a few options. If you are using Pycharm:

1. In your project directory (where you have all the files and scripts), click with the mouse right button on the folder "tests".

2. Select "Run pytest in tests".
3. Done!!

Sweet, isn't it?

You can also run the tests from your command line:

1. Open a command line and change into the repo directory.
2. Run:

```
$ pytest
```

These command will run all the test scripts within the test folder. Alternatively, you can run specific scripts as follows:

1. Change into the tests folder:

```
$ cd tests
```

2. Run a specific script, for example:

```
$ pytest test_categorical_encoder.py
```

If running pytest without tox, that is in your development environment, make sure you have the test dependencies installed. If not, from the root directory of the repo and in your development environment run:

```
$ pip install -r test_requirements.txt
```

If tests pass, your code is functional. If not, try and fix the issue following the error messages. If stuck, get in touch.

### Keeping 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 days or weeks to develop, it may happen that new code changes are made to Feature_engine's master 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 master into your feature branch, fairly often. To keep your branches up to date:

1. Check out your local master:

```
$ git checkout master
```

If your feature branch has uncommited changes, it will ask you to commit or stage those first.

2. Pull and rebase the upstream master on your local master:

```
$ git pull --rebase upstream master
```

Your master should be a copy of the upstream master. 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 master
```

The previous command will update your fork so that your fork's master is in sync with Feature-engine's master. Now, you need to rebase master onto your feature branch.

4. Check out your feature branch:

```
$ git checkout myfeaturebranch
```

5. Rebase master onto it:

```
$ git rebase master
```

Again, if conflicts arise, try and resolve them and continue the rebase. Now you are good to go to continue developing your feature.

### Merging Pull Requests

Only Core contributors have write access to the repository, can review and can 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 includes the PR # and the issue #.

### After your PR is merged

Update your local fork (see section **Keeping your fork updated**) and delete the feature branch.

Well done and thank you very much for your support!

### Releases

After a few features have been added to the master branch by yourself and other contributors, we will merge master into a release branch, e.g. 0.6.X, to release a new version of Feature-engine to PyPI.

### Getting started with Feature-engine documentation

Feature-engine documentation is built using Sphinx and is hosted on Read the Docs.

To learn more about Sphinx follow the Sphinx Quickstart documentation.

### Building the documentation

First, make sure you have properly installed Sphinx and the required dependencies.

1. If you haven't done so, in your virtual environment, from the root folder of the repository, install the requirements for the documentation:

```
$ pip install -r docs/requirements.txt
```

2. To build the documentation (and test if it is working properly):

```
$ 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 navigate the html files located in build. Alternatively, you need to troubleshoot through the error messages returned by sphinx.

Good luck and get in touch if stuck!

## 7.20 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 :).

## 7.21 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.

### 7.21.1 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 aprove 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 together with the Founder (see below) and input from the community can decide on the fate of the Feature-engine project.

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.

### 7.21.2 Join the community

Feature-engine welcomes contributors who would like to take on the role of additional Core Contributors and Contributors.

Get in touch using our Github issues page or through our mailing list:

1. Github issues.
2. Mailing list.

## 7.22 What's new

Feature-engine's creation transformers combine features into new variables through various mathematical and other methods.

### 7.22.1 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

### 7.22.2 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.

### 7.22.3 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)

### 7.22.4 Former Releases

#### 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`.**

  - **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_`.

# 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] 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] Weisberg S. "Yeo-Johnson Power Transformations". https://www.stat.umn.edu/arc/yjpower.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] 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

# INDEX