# feature_engine Documentation

***Release 0.5.1***

**Soledad Galli**

**Jul 10, 2020**

# 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 fit() and transform() methods to learn parameters from and then transform data.

Feature-engine includes transformers for:

- Missing value imputation
- Categorical variable encoding
- Outlier capping
- Discretisation
- Numerical variable transformation

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

Feature-engine's 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.

More details into what is unique about Feature-engine can be found in this article: **`Feature-engine: A new open source Python package for feature engineering <<https://www.trainindata.com/post/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, Python's preferred package installer.

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

CHAPTER

TWO

# 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 onboard. 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.

For more details on how to contribute check the contributing page. Click on the "Contributing" link in the "Table of Contents" on the left of this page.

Thank you for your contributions!

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
- *CategoricalVariableImputer*: 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

## 3.2 Categorical Variable Encoders: Encoders

- *OneHotCategoricalEncoder*: performs one hot encoding, optional: of popular categories
- *CountFrequencyCategoricalEncoder*: replaces categories by observation count or percentage
- *OrdinalCategoricalEncoder*: replaces categories by numbers arbitrarily or ordered by target
- *MeanCategoricalEncoder*: replaces categories by the target mean
- *WoERatioCategoricalEncoder*: replaces categories by the weight of evidence
- *RareLabelCategoricalEncoder*: 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

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

## 3.5 Outlier Capping: Cappers

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

## 3.6 Scikit-learn Wrapper:

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

CHAPTER

FOUR

# GETTING HELP

Can't get something to work? Here are places 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.

CHAPTER

# FIVE

# FIND A BUG?

Check if there's already an open issue on the topic. If needed, file an issue.

CHAPTER

SIX

# 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.

## 6.1 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.

### 6.1.1 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, Python's preferred package installer.

```
$ 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
```

You can also use the `-U` flag to update Scikit-learn, pandas, NumPy, or any other libraries that work well with Feature-engine to their latest versions.

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

### 6.1.2 Example Use

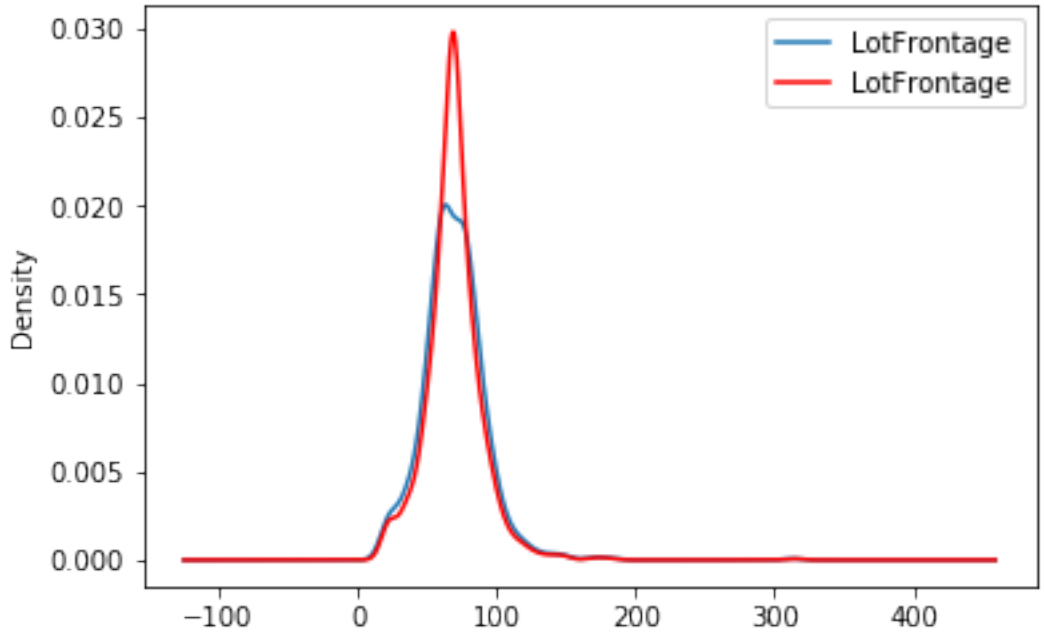
This is an example of how to use Feature-engine's transformers to perform missing data imputation.

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

import feature_engine.missing_data_imputers as mdi
```

(continues on next page)

(continued from previous page)

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

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

# set up the imputer
median_imputer = mdi.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')
```




More examples can be found in the documentation for each transformer and in a dedicated section in the repository with Jupyter notebooks.

### 6.1.3 Feature-engine with 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 import categorical_encoders as ce
from feature_engine import discretisers as dsc
from feature_engine import missing_data_imputers as mdi

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

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

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

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

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

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

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

 # 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', mdi.AddMissingIndicator(variables = ['LotFrontage'])),
```

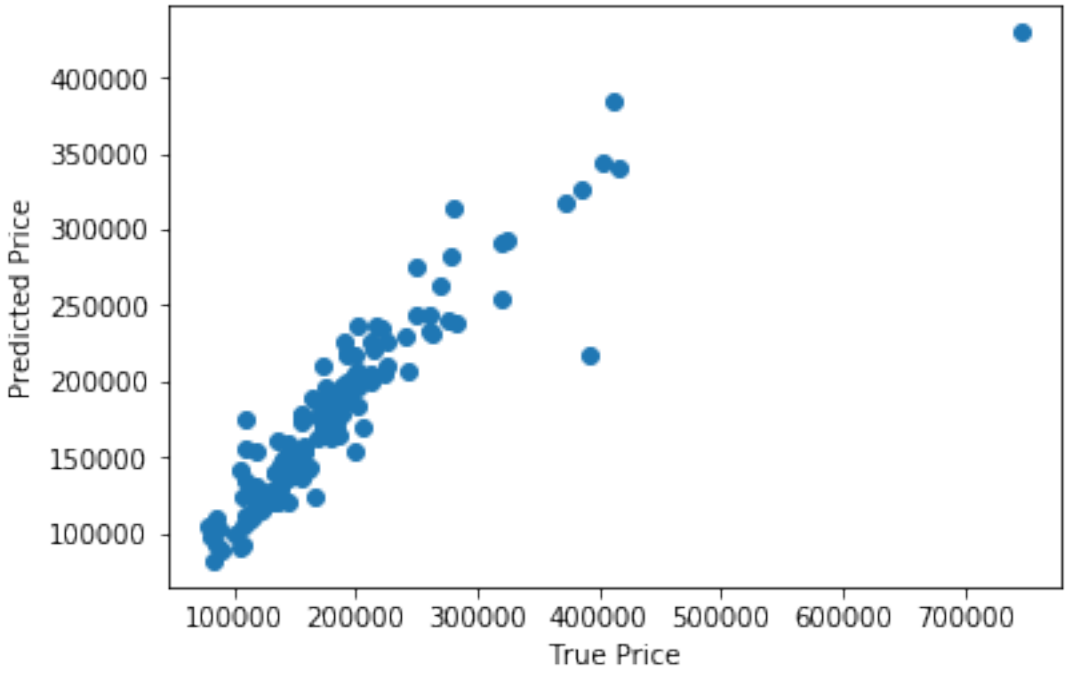
(continues on next page)

(continued from previous page)

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

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

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

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

    # encode categorical and discrete variables using the target mean
    ('categorical_encoder', ce.MeanCategoricalEncoder(variables =␣
↪categorical+discrete)),

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

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

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

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

# Evaluate
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 documentation for each transformer and in a dedicated section of Jupyter notebooks.

### 6.1.4 Dataset attribution

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

**Titanic dataset**

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

**Ames House Prices dataset**

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

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

The original data and documentation can be found here:

- Documentation
- Data

**Credit Approval dataset**

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

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

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

To prepare the data for the examples:

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

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

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

# replace ? by np.nan
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)
```

## 6.2 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
```

(continues on next page)

(continued from previous page)

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

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

# replace ? by np.nan
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)
```

## 6.3 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.

### 6.3.1 MeanMedianImputer

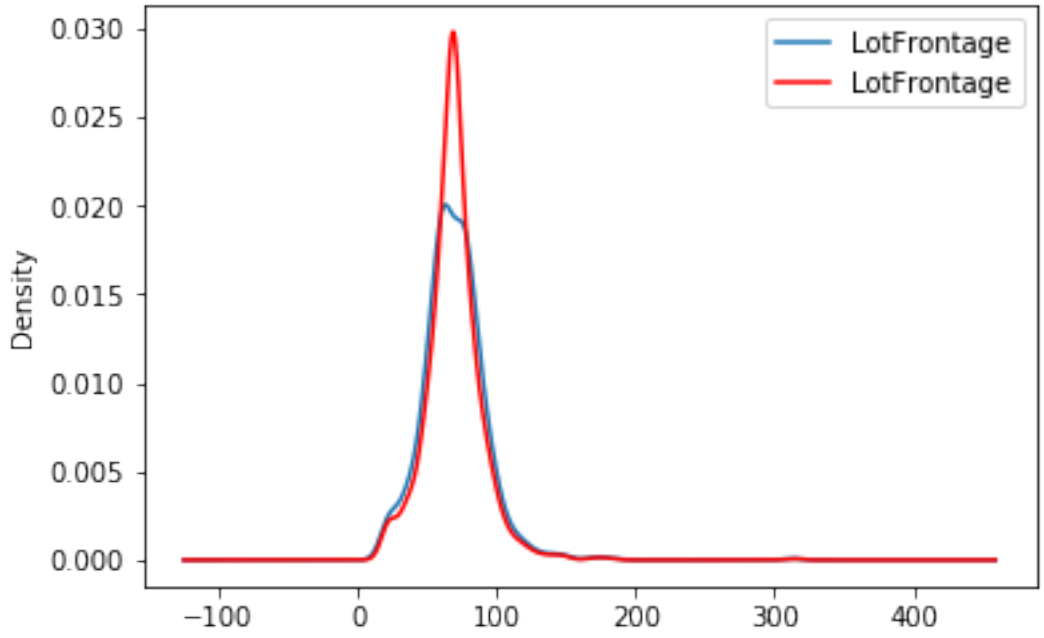
The MeanMedianImputer() replaces missing data with the mean or median of the variable. It works only with numerical variables. A list of variables to impute can be indicated, or the imputer will automatically select all numerical variables in the train set. For more details, check the API Reference below.

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

import feature_engine.missing_data_imputers as mdi

# 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 = mdi.MeanMedianImputer(imputation_method='median',
                                       variables=['LotFrontage', 'MasVnrArea'])
# fit the imputer
median_imputer.fit(X_train)
```

(continues on next page)

(continued from previous page)

```
# 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')
```




## API Reference

**class** feature_engine.missing_data_imputers.**MeanMedianImputer**(*imputation_method='median'*, *variables=None*)

The MeanMedianImputer() transforms features by replacing missing data by the mean or median value of the variable.

The MeanMedianImputer() works only with numerical variables.

Users can pass a list of variables to be imputed as argument. Alternatively, the MeanMedianImputer() will automatically find and select all variables of type numeric.

The imputer first calculates the mean / median values of the variables (fit).

The imputer 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.

**fit** (*X*, *y=None*)
Learns the mean or median values.

**Parameters**

- **X** (*pandas dataframe of shape = [n_samples, n_features]*) – The training input samples. User can pass the entire dataframe, not just the variables that need imputation.
- **y** (*pandas series or None, default=None*) – y is not needed in this imputation. You can pass None or y.

**imputer_dict_**
The dictionary containing the mean / median values per variable. These values will be used by the imputer to replace missing data. The **imputer_dict_** is created when fitting the imputer.
**Type** dictionary

**transform** (*X*)
Replaces missing data with the learned parameters.

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

**Returns** **X_transformed** – The dataframe without missing values in the selected variables.

**Return type** pandas dataframe of shape = [n_samples, n_features]

### 6.3.2 ArbitraryNumberImputer

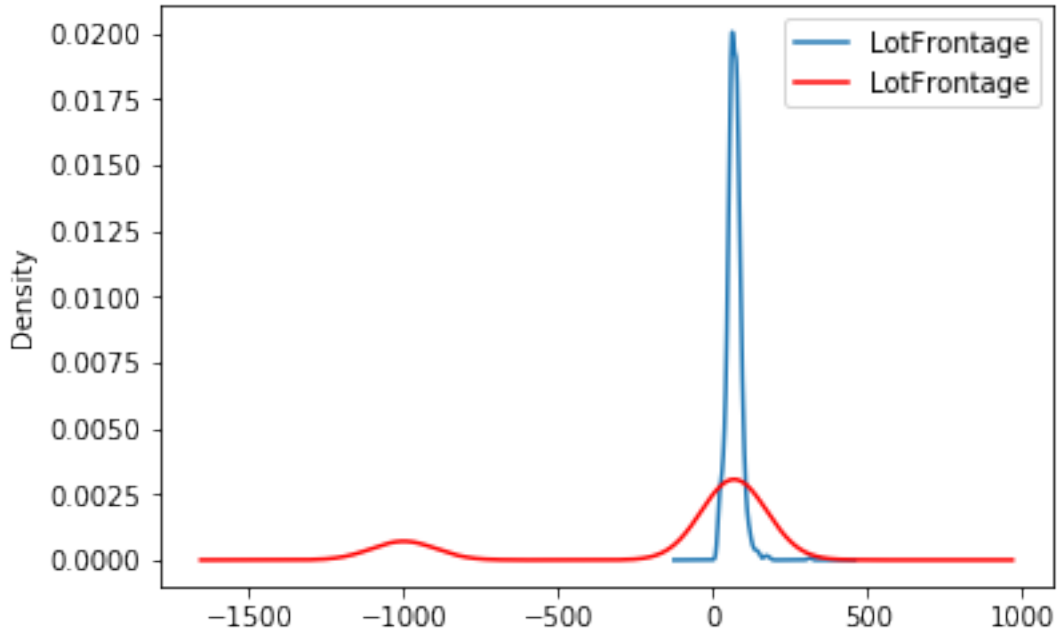
The ArbitraryNumberImputer() replaces missing data with an arbitrary value determined by the user. It works only with numerical variables. A list of variables can be indicated, or the imputer will automatically select all numerical variables in the train set.

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

import feature_engine.missing_data_imputers as mdi

# 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 = mdi.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)
```

(continues on next page)

(continued from previous page)

```
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')
```




## API Reference

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

The ArbitraryNumberImputer() replaces missing data in each variable by an arbitrary value determined by the user.

**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.

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

Checks that the variables are numerical.

**Parameters**

- **X** (*pandas dataframe of shape = [n_samples, n_features]*) – The training input samples. User can pass the entire dataframe, not just the variables to impute.
- **y** (*None*) – y is not needed in this imputation. You can pass None or y.

**transform**(*X*)

Replaces missing data with the learned parameters.

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

**Returns** **X_transformed** – The dataframe without missing values in the selected variables.

**Return type** pandas dataframe of shape = [n_samples, n_features]

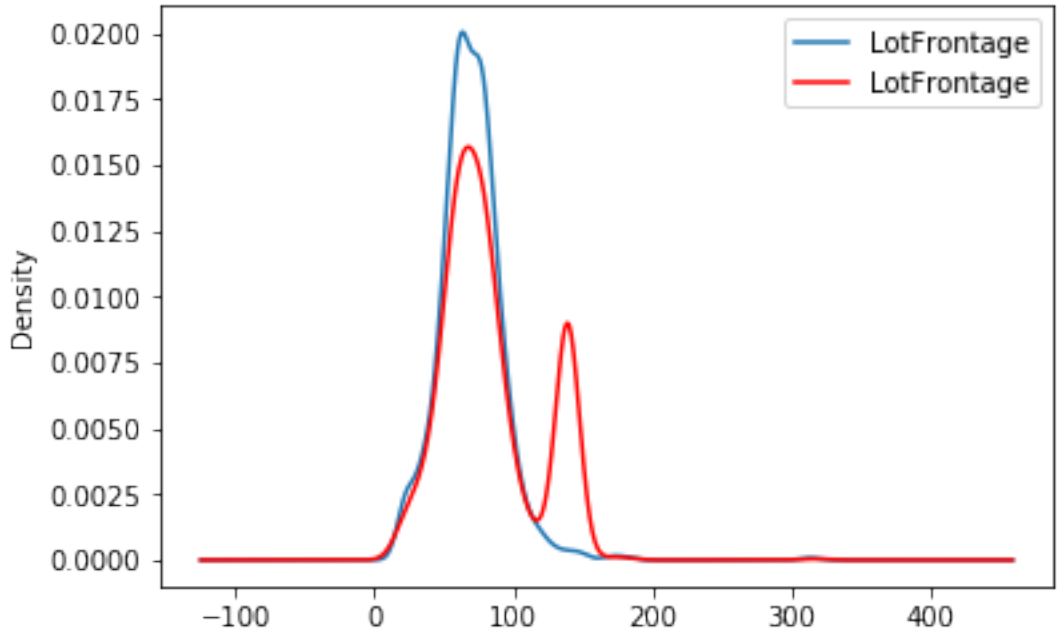
### 6.3.3 EndTailImputer

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

import feature_engine.missing_data_imputers as mdi

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




### API Reference

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

The EndTailImputer() transforms features by replacing missing data by a value at either tail of the distribution.

The EndTailImputer() 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**

- **distribution** (*str, default=gaussian*) – Method to be used to find the replacement values. Can take 'gaussian', 'skewed' or 'max'.

gaussian: the imputer will use the Gaussian limits to find the values to replace missing data.

skewed: 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 skewed.
- **variables** (*list, default=None*) – The list of variables to be imputed. If None, the imputer will find and select all variables of type numeric.

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

Learns the values at the end of the variable distribution.

**Parameters**

- **X** (*pandas dataframe of shape = [n_samples, n_features]*) – The training input samples. The user can pass the entire dataframe, not just the variables that need imputation.
- **y** (*None*) – y is not needed in this imputation. You can pass None or y.

**imputer_dict_**

The dictionary containing the values at the end of the distribution per variable. These values will be used by the imputer to replace missing data.

**Type** dictionary

**transform** (*X*)

Replaces missing data with the learned parameters.

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

**Returns** **X_transformed** – The dataframe without missing values in the selected variables.

**Return type** pandas dataframe of shape = [n_samples, n_features]

### 6.3.4 CategoricalVariableImputer

The CategoricalVariableImputer() replaces missing data in categorical variables with the string 'Missing' or by the most frequent category.

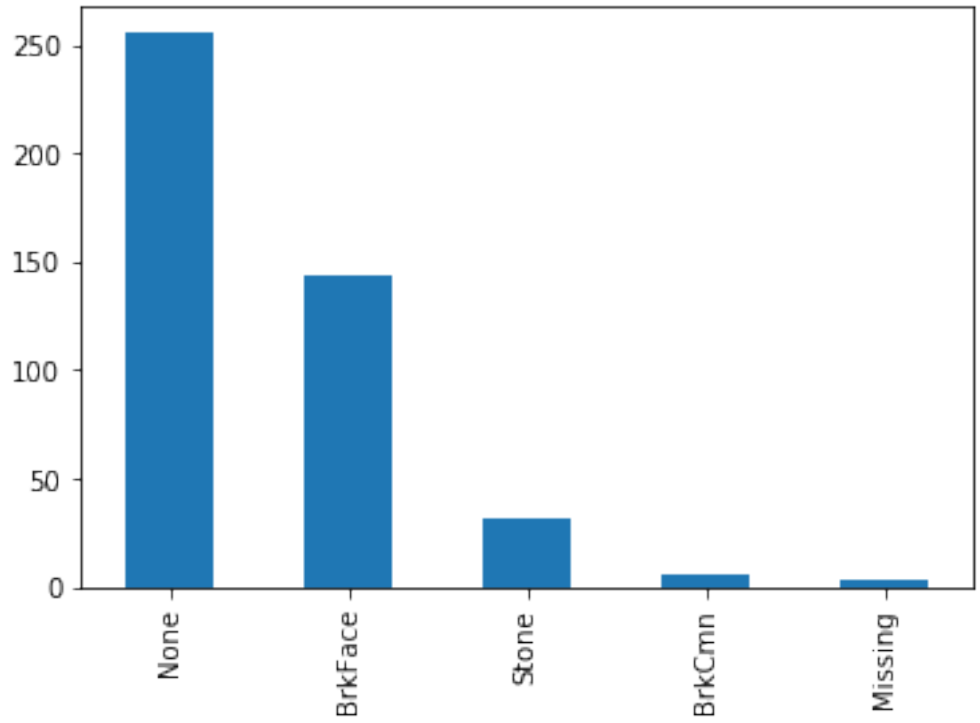
It works only with categorical variables. A list of variables can be indicated, or the imputer will automatically select all categorical variables in the train set.

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

import feature_engine.missing_data_imputers as mdi

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

(continues on next page)

(continued from previous page)

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

# set up the imputer
imputer = mdi.CategoricalVariableImputer(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()
```




## API Reference

**class** feature_engine.missing_data_imputers.**CategoricalVariableImputer**(*imputation_method='missing'*, *variables=None*, *return_object=False*)

The CategoricalVariableImputer() replaces missing data in categorical variables by the string 'Missing' or 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 CategoricalVariableImputer() will

automatically find and select all variables of type object.

**Parameters**

- **imputation_method** (*str, default=missing*) – Desired method of imputation. Can take 'missing' or 'frequent'.
- **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.

  Tip: return the variables as object if planning to do categorical encoding with feature-engine.

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

Learns the most frequent category if the imputation method is set to frequent.

**Parameters**

- **X** (*pandas dataframe of shape = [n_samples, n_features]*) – The training input samples. Can be the entire dataframe, not just the selected variables.
- **y** (*None*) – y is not needed in this imputation. You can pass None or y.

**imputer_dict_**

The dictionary mapping each variable to the most frequent category, or to the value 'Missing' depending on the imputation_method. The most frequent category is calculated when fitting the transformer.
**Type** dictionary

**transform**(*X*)

Replaces missing data with the learned parameters.

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

**Returns** **X_transformed** – The dataframe without missing values in the selected variables.

**Return type** pandas dataframe of shape = [n_samples, n_features]

### 6.3.5 RandomSampleImputer

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

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:

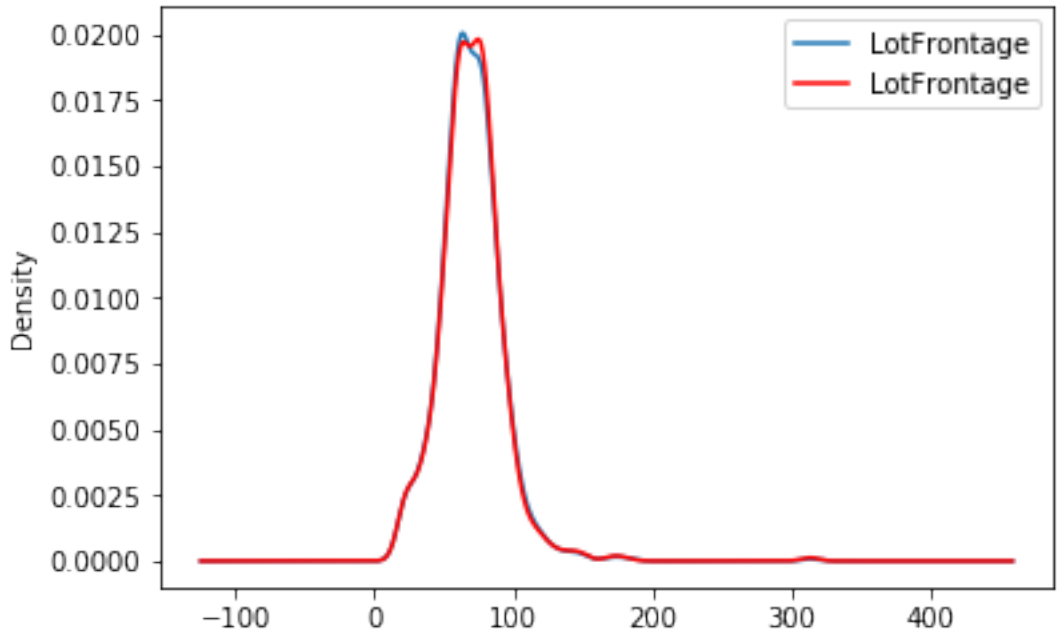
```
RandomSampleImputer(random_state=['height', 'weight'],
                            seed='observation',
                            seeding_method='add'))
```

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

import feature_engine.missing_data_imputers as mdi

# 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 = mdi.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')
```



### API Reference

**class** feature_engine.missing_data_imputers.**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).

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 for that observation, and passed to the random_state. Thus, a value will be extracted using that seed, and used to replace that particular observation. This is the equivalent of pandas.sample(1, random_state=var1+var2) if the 'seeding_method' is set to 'add' or pandas.sample(1, random_state=var1*var2) if the 'seeding_method' is set to 'multiply'.

For 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. The imputer replaces missing data with a random sample from the training set.

**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.

**fit** (*X*, *y=None*)
Makes a copy of the variables to impute in the training dataframe from which it will randomly extract the values to fill the missing data during transform.

**Parameters**

- **X** (*pandas dataframe of shape = [n_samples, n_features]*) – The training input samples. Can be the entire dataframe, not just he variables to impute.
- **y** (*None*) – y is not needed in this imputation. You can pass None or y.

**X_**
Copy of the training dataframe from which to extract the random samples.
**Type** dataframe.

**transform** (*X*)
Replaces 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_transformed** – The dataframe without missing values in the transformed variables.

**Return type** pandas dataframe of shape = [n_samples, n_features]

### 6.3.6 AddMissingIndicator

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

import feature_engine.missing_data_imputers as mdi

# 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 = mdi.AddMissingIndicator(
    variables=['Alley', 'MasVnrType', 'LotFrontage', 'MasVnrArea'])

# fit the imputer
addBinary_imputer.fit(X_train)

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

(continues on next page)

(continued from previous page)

```
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 |

### API Reference

**class** feature_engine.missing_data_imputers.**AddMissingIndicator**(*how='missing_only'*, *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**

- **how** (*string, defatul='missing_only'*) – Indicates if missing indicators should be added to variables with missing data or to all variables.

  missing_only: indicators will be created only for those variables that showed missing data during fit.

  all: 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.

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

Learns the variables for which the missing indicators will be created.

**Parameters**

- **X** (*pandas dataframe of shape = [n_samples, n_features]*) – The training input samples.
- **y** (*None*) – y is not needed in this imputation. You can pass None or y.

**variables_**

the lit of variables for which the missing indicator will be created.

**Type** list

**transform**(*X*)
Adds the binary missing indicators.

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

**Returns** **X_transformed** – The dataframe containing the additional binary variables. Binary variables are named with the original variable name plus '_na'.

**Return type** pandas dataframe of shape = [n_samples, n_features]

## 6.4 Categorical variable encoding

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

### 6.4.1 OneHotCategoricalEncoder

The OneHotCategoricalEncoder() 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.

The OneHotCategoricalEncoder() works only with categorical variables. A list of variables can be indicated, or the encoder 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 import categorical_encoders as ce

# 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 = ce.OneHotCategoricalEncoder(
    top_categories=2,
    variables=['pclass', 'cabin', 'embarked'],
    drop_last=False)
```

(continues on next page)

(continued from previous page)

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

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

encoder.encoder_dict_
```

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

### API Reference

**class** feature_engine.categorical_encoders.**OneHotCategoricalEncoder**(*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.

Each one of 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 person is female or 0 otherwise. We can also generate the variable male, which takes 1 if the person is "male" and 0 otherwise.

The encoder has the option to generate one dummy variable per category, or to create dummy variables only for the top n most popular categories, that is, the categories that are shown by the majority of the observations.

If dummy variables are created for all the categories of a variable, you have the option to drop one category not to create information redundancy. That is, encoding into k-1 variables, where k is the number if unique categories.

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).

**Parameters**

- **top_categories** (*int, default=None*) – If None, a dummy variable will be created for each category of the variable. Alternatively, top_categories indicates the number of most frequent categories to encode. Dummy variables will be created only for those popular categories and the rest will be ignored. Note that this is equivalent to grouping all the remaining categories in one group.
- **variables** (*list*) – The list of categorical variables that will be encoded. If None, the encoder will find and select all object type variables.
- **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 variable of the list (k-1 dummies).

**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.

**encoder_dict_**
The dictionary containing the categories for which dummy variables will be created.
**Type** dictionary

**transform** (*X*)
Creates the dummy / binary variables.

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

**Returns** **X_transformed** – The shape of the dataframe will be different from the original as it includes the dummy variables.

**Return type** pandas dataframe.

### 6.4.2 CountFrequencyCategoricalEncoder

The CountFrequencyCategoricalEncoder() replaces categories with the number of observations or percentage of observations per category. For example, if 10 observations show the category blue for the variable color, blue will be replaced by 10. If, using frequency, if 20% of observations show the category red, red will be replaced by 0.20.

The CountFrequencyCategoricalEncoder() works only with categorical variables. A list of variables can be indicated, or the encoder 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 import categorical_encoders as ce

# 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)
```

(continues on next page)

(continued from previous page)

```
# set up the encoder
encoder = ce.CountFrequencyCategoricalEncoder(encoding_method='frequency',
                          variables=['cabin', 'pclass', 'embarked'])

# fit the encoder
encoder.fit(X_train)

# 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}}
```

### API Reference

**class** feature_engine.categorical_encoders.**CountFrequencyCategoricalEncoder**(*encoding_method='cou*
*vari-*
*ables=None*)

The CountFrequencyCategoricalEncoder() replaces categories by the count of observations per category or by 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 CountFrequencyCategoricalEncoder() will encode only categorical variables (type 'object'). A list of variables to be encoded 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 numbers (counts or frequencies) for each variable (fit).

The encoder then transforms the categories to 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.

**fit** (*X*, *y=None*)
Learns the counts or frequencies which will be used to replace the categories.

**Parameters**

- **X** (*pandas dataframe of shape = [n_samples, n_features]*) – The training input samples. The user can pass the entire dataframe.
- **y** (*None*) – y is not needed in this encoder. You can pass y or None.

**encoder_dict_**
Dictionary containing the {category: count / frequency} pairs for each variable.
**Type** dictionary

**inverse_transform** (*X*)
Convert the data back to the original representation.

**Parameters** **X_transformed** (*pandas dataframe of shape = [n_samples, n_features]*) – The transformed dataframe.

**Returns** **X** – The un-transformed dataframe, that is, containing the original values of the categorical variables.

**Return type** pandas dataframe of shape = [n_samples, n_features]

**transform** (*X*)
Replaces categories with the learned parameters.

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

**Returns** **X_transformed** – The dataframe containing categories replaced by numbers.

**Return type** pandas dataframe of shape = [n_samples, n_features]

### 6.4.3 OrdinalCategoricalEncoder

The OrdinalCategoricalEncoder() 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.

The OrdinalCategoricalEncoder() works only with categorical variables. A list of variables can be indicated, or the encoder 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 import categorical_encoders as ce

# 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)
```

(continues on next page)

(continued from previous page)

```
        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 = ce.OrdinalCategoricalEncoder(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}}
```

### API Reference

**class** feature_engine.categorical_encoders.**OrdinalCategoricalEncoder**(*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.

**encoder_dict_**
The dictionary containing the {category: ordinal number} pairs for every variable.

**Type** dictionary

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

**inverse_transform** (*X*)
Convert the data back to the original representation.

**Parameters** **X_transformed** (*pandas dataframe of shape = [n_samples, n_features]*) – The transformed dataframe.

**Returns** **X** – The un-transformed dataframe, that is, containing the original values of the categorical variables.

**Return type** pandas dataframe of shape = [n_samples, n_features]

**transform** (*X*)
Replaces categories with the learned parameters.

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

**Returns** **X_transformed** – The dataframe containing categories replaced by numbers.

**Return type** pandas dataframe of shape = [n_samples, n_features]

### 6.4.4 MeanCategoricalEncoder

The MeanCategoricalEncoder() 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.

The MeanCategoricalEncoder() works only with categorical variables. A list of variables can be indicated, or the encoder 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
```

(continues on next page)

(continued from previous page)

```
from feature_engine import categorical_encoders as ce

# 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 = ce.MeanCategoricalEncoder(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,
  '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}}
```

### API Reference

**class** feature_engine.categorical_encoders.**MeanCategoricalEncoder**(*variables=None*)

The MeanCategoricalEncoder() 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 transforms the categories to the mapped numbers (transform).

**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.

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

Learns 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.

**encoder_dict_**

The dictionary containing the {category: target mean} pairs used to replace categories in every variable.

**Type** dictionary

**inverse_transform**(*X*)

Convert the data back to the original representation.

**Parameters** **X_transformed** (*pandas dataframe of shape = [n_samples, n_features]*) – The transformed dataframe.

**Returns** **X** – The un-transformed dataframe, that is, containing the original values of the categorical variables.

**Return type** pandas dataframe of shape = [n_samples, n_features]

**transform**(*X*)

Replaces categories with the learned parameters.

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

**Returns** **X_transformed** – The dataframe containing categories replaced by numbers.

**Return type** pandas dataframe of shape = [n_samples, n_features]

### 6.4.5 WoERatioCategoricalEncoder

The WoERatioCategoricalEncoder() replaces the labels by the weight of evidence or the ratio of probabilities. It only works for binary classification.

The weight of evidence is given by: np.log( p(1) / p(0) )

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

The CountFrequencyCategoricalEncoder() works only with categorical variables. A list of variables can be indicated, or the encoder 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 import categorical_encoders as ce

# 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 = ce.RareLabelCategoricalEncoder(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 = ce.WoERatioCategoricalEncoder(
encoding_method='woe', 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,
```

(continues on next page)

(continued from previous page)

```
'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}}
```

### API Reference

**class** feature_engine.categorical_encoders.**WoERatioCategoricalEncoder**(*encoding_method='woe'*, *variables=None*)

The WoERatioCategoricalEncoder() replaces categories by the weight of evidence or by the ratio between the probability of the target = 1 and the probability of the target = 0.

The weight of evidence is given by: np.log(P(X=xj|Y = 1)/P(X=xj|Y=0))

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

And the log of the target probability ratio is: np.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: np.log(0.8/0.2) = 1.386 if log_ratio is selected. Alternatively, blue will be replaced by 0.8 / 0.2 = 4 if ratio is selected.

For details on the calculation of the weight of evidence visit: https://www.listendata.com/2015/03/weight-of-evidence-woe-and-information.html

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 woe or 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 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 transforms the categories into the mapped numbers (transform).

**Parameters**

- **encoding_method** (*str, default=woe*) – Desired method of encoding.

  'woe': weight of evidence

  'ratio' : probability ratio

- **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.

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

Learns the numbers that should be used to replace the categories in each variable. That is the WoE or 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].

**encoder_dict_**
The dictionary containing the {category: WoE / ratio} pairs per variable.
**Type** dictionary

**inverse_transform**(*X*)
Convert the data back to the original representation.

**Parameters** **X_transformed** (*pandas dataframe of shape = [n_samples, n_features]*) – The transformed dataframe.

**Returns** **X** – The un-transformed dataframe, that is, containing the original values of the categorical variables.

**Return type** pandas dataframe of shape = [n_samples, n_features]

**transform**(*X*)
Replaces categories with the learned parameters.

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

**Returns** **X_transformed** – The dataframe containing categories replaced by numbers.

**Return type** pandas dataframe of shape = [n_samples, n_features]

### 6.4.6 RareLabelCategoricalEncoder

The RareLabelCategoricalEncoder() 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.

The RareLabelCategoricalEncoder() works only with categorical variables. A list of variables can be indicated, or the encoder 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 import categorical_encoders as ce

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),
```

(continues on next page)

(continued from previous page)

```
    data['survived'], test_size=0.3, random_state=0)

# set up the encoder
encoder = ce.RareLabelCategoricalEncoder(tol=0.03, n_categories=2,
                                         variables=['cabin', 'pclass', 'embarked'],
                                         replace_with='Rare', return_object=True)

# fit the encoder
encoder.fit(X_train)

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

### API Reference

**class** feature_engine.categorical_encoders.**RareLabelCategoricalEncoder**(*tol=0.05*, *n_categories=10*, *variables=None*, *replace_with='Rare'*, *return_object=False*)

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.

- **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.
- **replace_with** (*string, default='Rare'*) – The category name that will be used to replace infrequent categories.
- **return_object** (*bool, default=False*) – Whether the variables should be re-cast as object, in case they have numerical values.

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

Learns 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.

**encoder_dict_**

The dictionary containing the frequent categories (that will be kept) for each variable. Categories not present in this list will be replaced by 'Rare' or by the user defined value.

**Type** dictionary

**transform** (*X*)

Groups rare labels under separate group 'Rare' or any other name provided by the user.

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

**Returns** **X_transformed** – The dataframe where rare categories have been grouped.

**Return type** pandas dataframe of shape = [n_samples, n_features]

## 6.5 Variable transformation

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

### 6.5.1 LogTransformer

#### API Reference

**class** feature_engine.variable_transformers.**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.

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

Selects 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** (*None*) – y is not needed in this transformer. You can pass y or None.

**transform**(*X*)

Transforms the variables using logarithm.

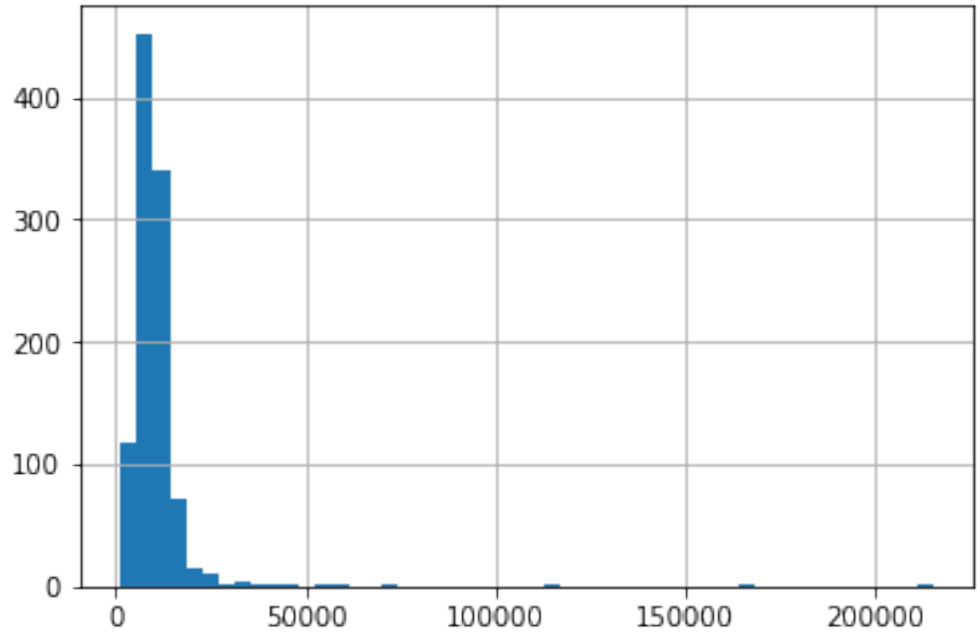
**Parameters** **X** (*pandas dataframe of shape = [n_samples, n_features]*) – The data to transform.

**Returns** **X_transformed** – The log transformed dataframe.

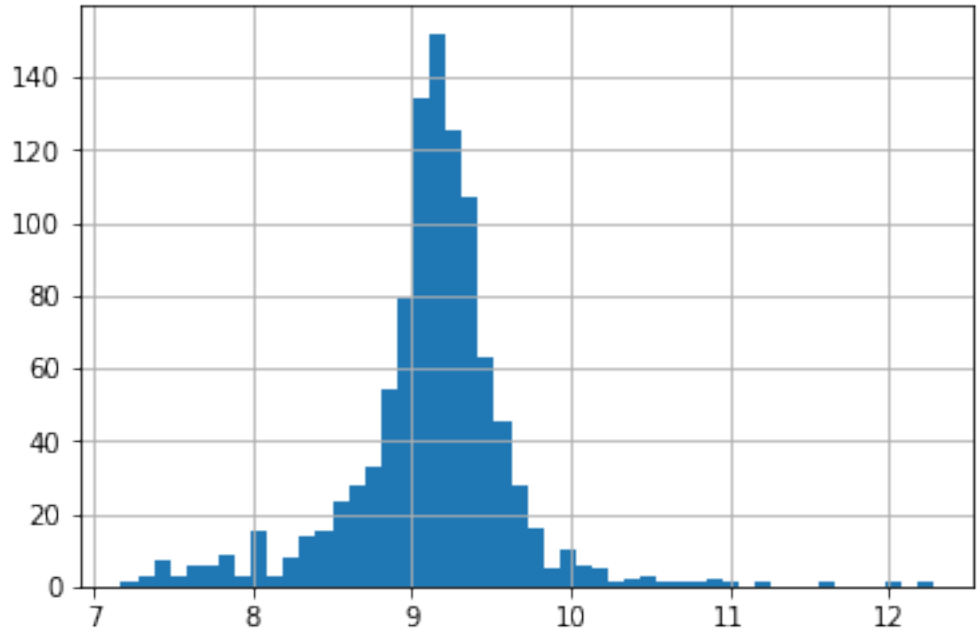
**Return type** pandas dataframe of shape = [n_samples, n_features]

### Example Use

```
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 variable_transformers 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)
```



### 6.5.2 ReciprocalTransformer

#### API Reference

**class** feature_engine.variable_transformers.**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.

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

**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** (*None*) – y is not needed in this encoder. You can pass y or None.

**transform**(*X*)

Applies the reciprocal 1 / x transformation.

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

**Returns** **X_transformed** – The dataframe with reciprocally transformed variables.

**Return type** pandas dataframe of shape = [n_samples, n_features]

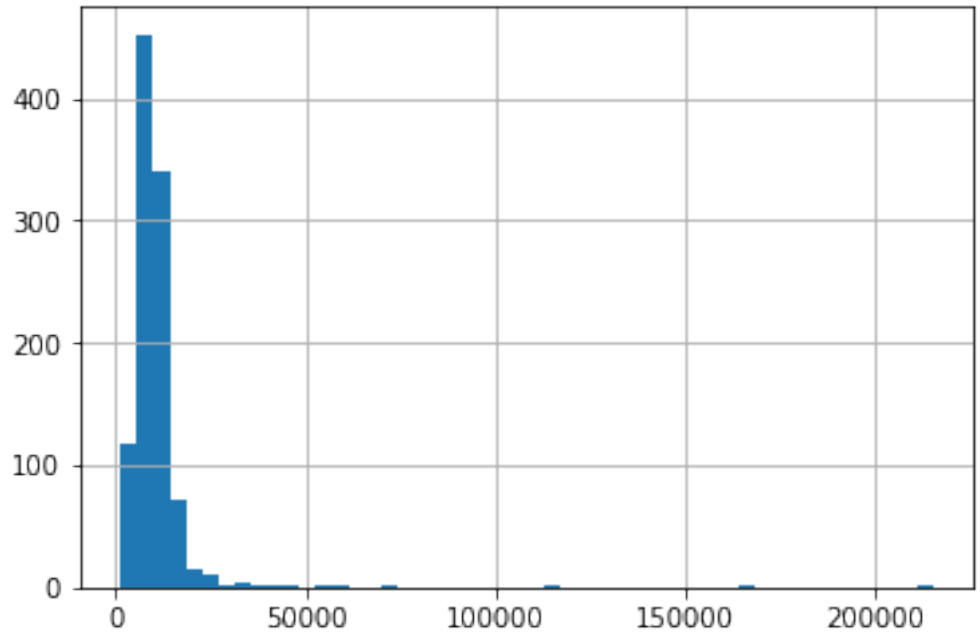
#### Example Use

```
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 variable_transformers 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
```

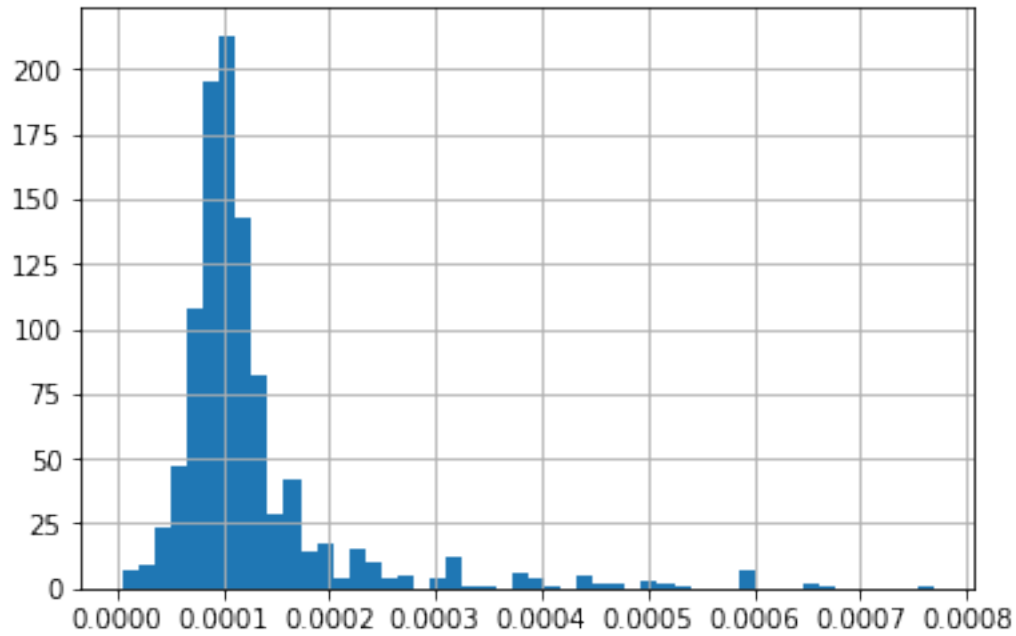
(continues on next page)

(continued from previous page)

```
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)
```



### 6.5.3 PowerTransformer

#### API Reference

**class** feature_engine.variable_transformers.**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).

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

**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** (*None*) – y is not needed in this transformer. You can pass y or None.

**transform**(*X*)

Applies the power transformation to the variables.

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

**Returns** **X_transformed** – The dataframe with the power transformed variables.

**Return type** pandas dataframe of shape = [n_samples, n_features]

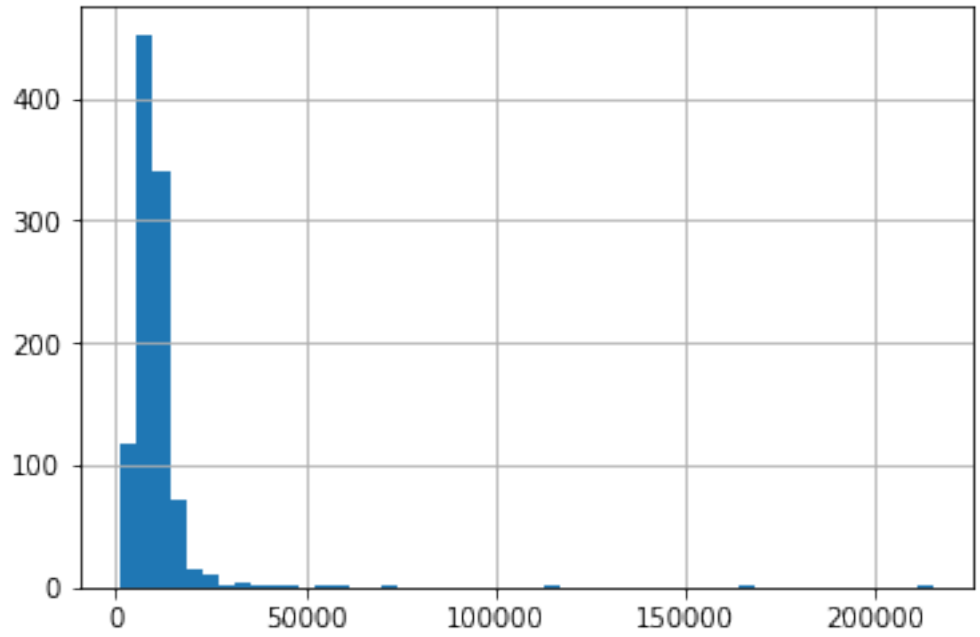
#### Example Use

```
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 variable_transformers 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)
```

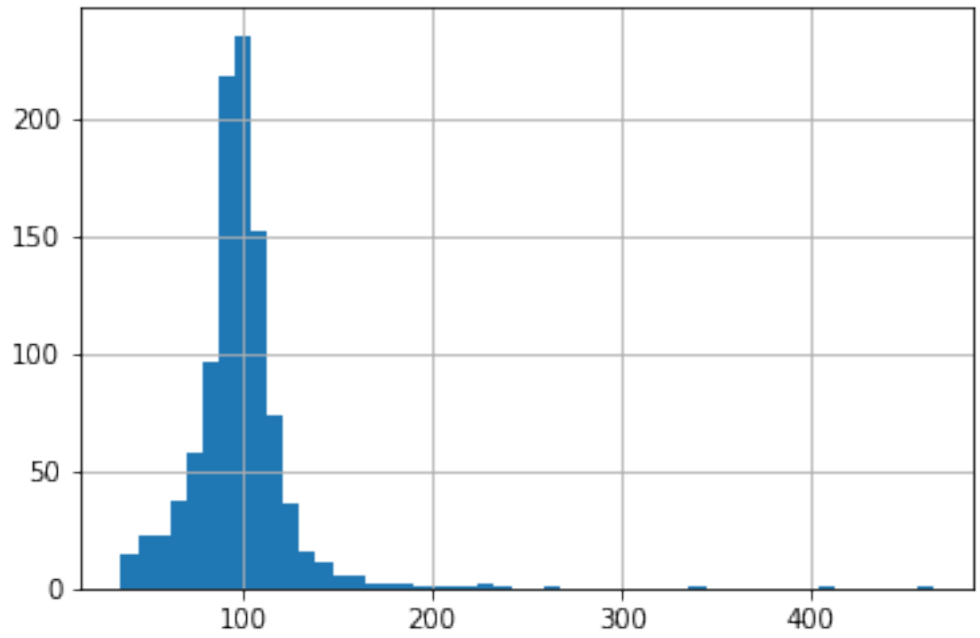
(continues on next page)

(continued from previous page)

```
# 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)
```




### 6.5.4 BoxCoxTransformer

The Box-Cox transformation is defined as:

T(Y)=(Y exp()1)/ if !=0, or 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.

#### API Reference

**class** feature_engine.variable_transformers.**BoxCoxTransformer**(*variables=None*)

The BoxCoxTransformer() applies the BoxCox transformation to numerical variables.

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.

**lamda_dict_**

The dictionary containing the {variable: best exponent for the BoxCox transfomration} pairs. These are determined automatically.

**Type** dictionary

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

Learns 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** (*None*) – y is not needed in this transformer. You can pass y or None.

**transform**(*X*)

Applies the BoxCox transformation.

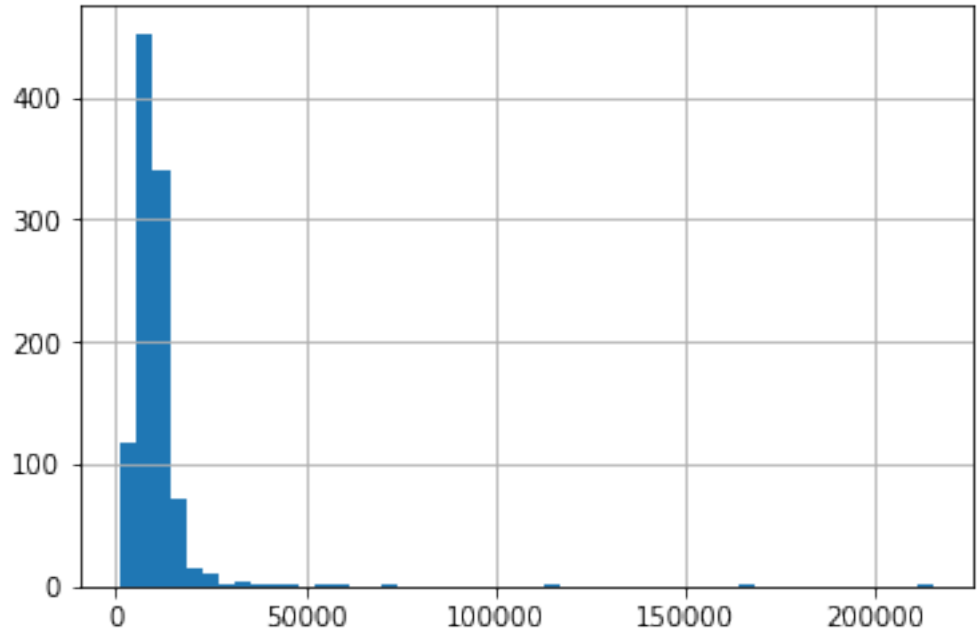
**Parameters** **X** (*pandas dataframe of shape = [n_samples, n_features]*) – The data to be transformed.

**Returns** **X_transformed** – The dataframe with the transformed variables.

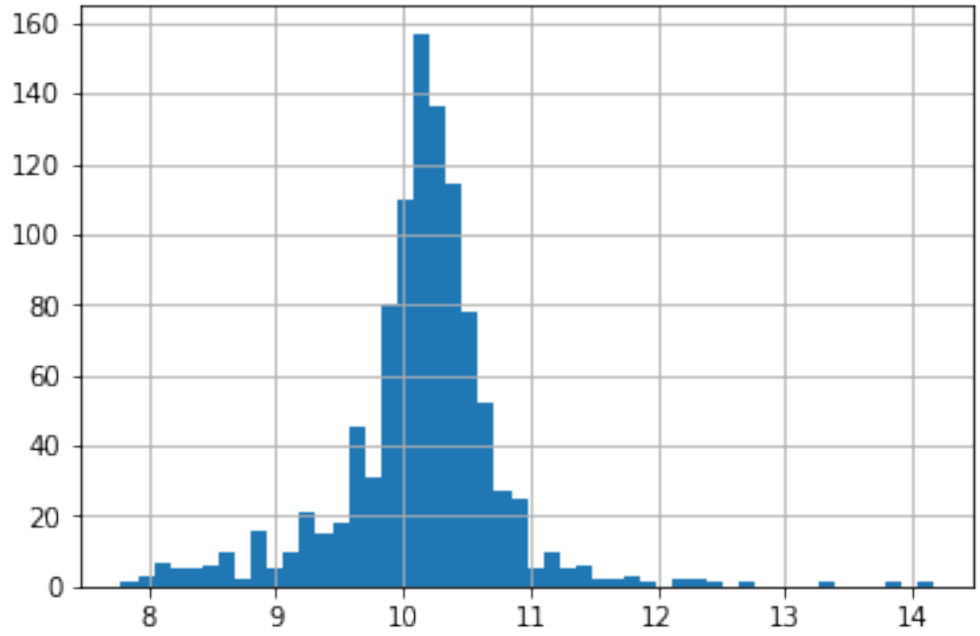
**Return type** pandas dataframe of shape = [n_samples, n_features]

### Example Use

```
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 variable_transformers 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
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)
```



### 6.5.5 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.variable_transformers.**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.

**lamda_dict_**

The dictionary containing the {variable: best lambda for the Yeo-Johnson transformation} pairs.

**Type** dictionary

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

Learns 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** (*None*) – y is not needed in this transformer. You can pass y or None.

**transform**(*X*)

Applies the Yeo-Johnson transformation.

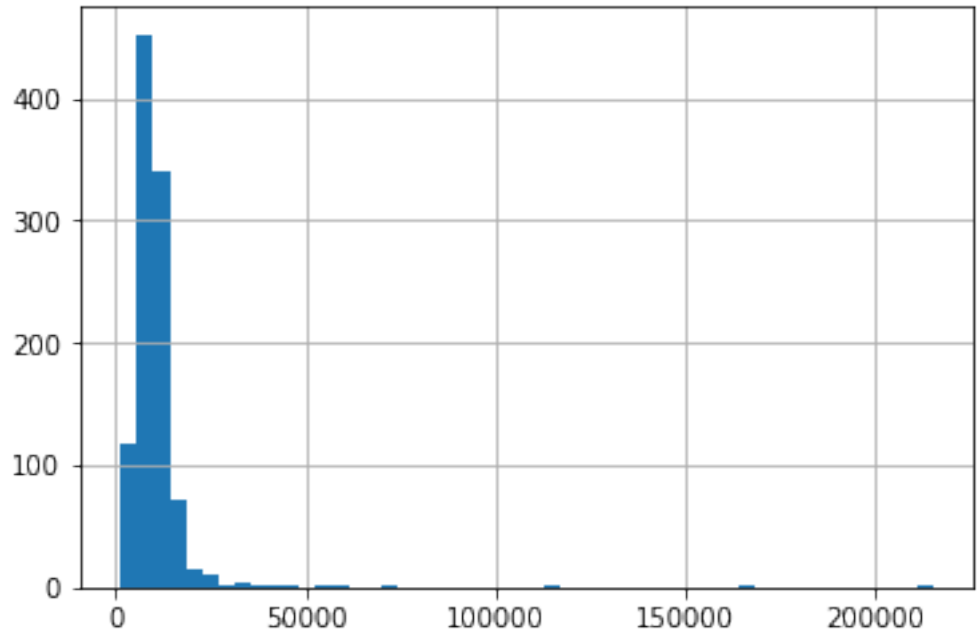
**Parameters** **X** (*pandas dataframe of shape = [n_samples, n_features]*) – The data to be transformed.

**Returns** **X_transformed** – The dataframe with the transformed variables.

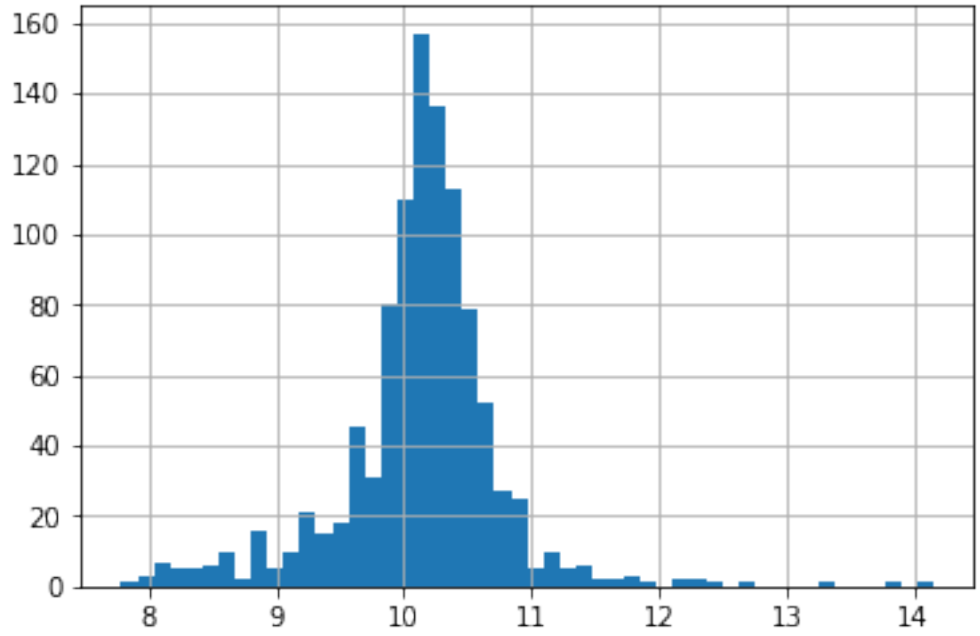
**Return type** pandas dataframe of shape = [n_samples, n_features]

### Example Use

```
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 variable_transformers 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)
```




## 6.6 Variable discretisation

Feature-engine's variable discretisation transformers transform continuous numerical variables into discrete variables of contiguous intervals.

### 6.6.1 EqualFrequencyDiscretiser

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 indiacated, 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 import discretisers as dsc

# 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
```

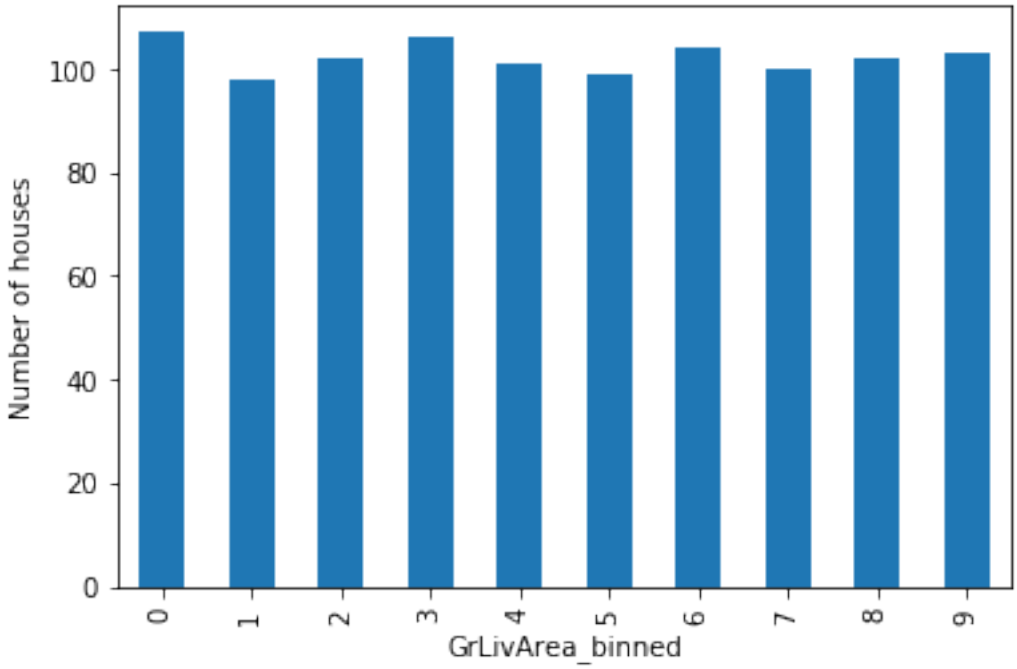
(continues on next page)

(continued from previous page)

```
disc = dsc.EqualFrequencyDiscretiser(q=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,
  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')
```




### API Reference

**class** feature_engine.discretisers.**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, fit.

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

**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.

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

Learns the limits of the equal frequency intervals, that is the quantiles for 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 transformed.
- **y** (*None*) – y is not needed in this encoder. You can pass y or None.

**binner_dict_**

The dictionary containing the {variable: interval limits} pairs used to sort the values into discrete intervals.

**Type** dictionary

**transform** (*X*)

Sorts the variable values into the intervals.

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

**Returns** **X_transformed** – The transformed data with the discrete variables.

**Return type** pandas dataframe of shape = [n_samples, n_features]

### 6.6.2 EqualWidthDiscretiser

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 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 import discretisers as dsc

# 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 = dsc.EqualWidthDiscretiser(bins=10, variables=['LotArea', 'GrLivArea'])
```

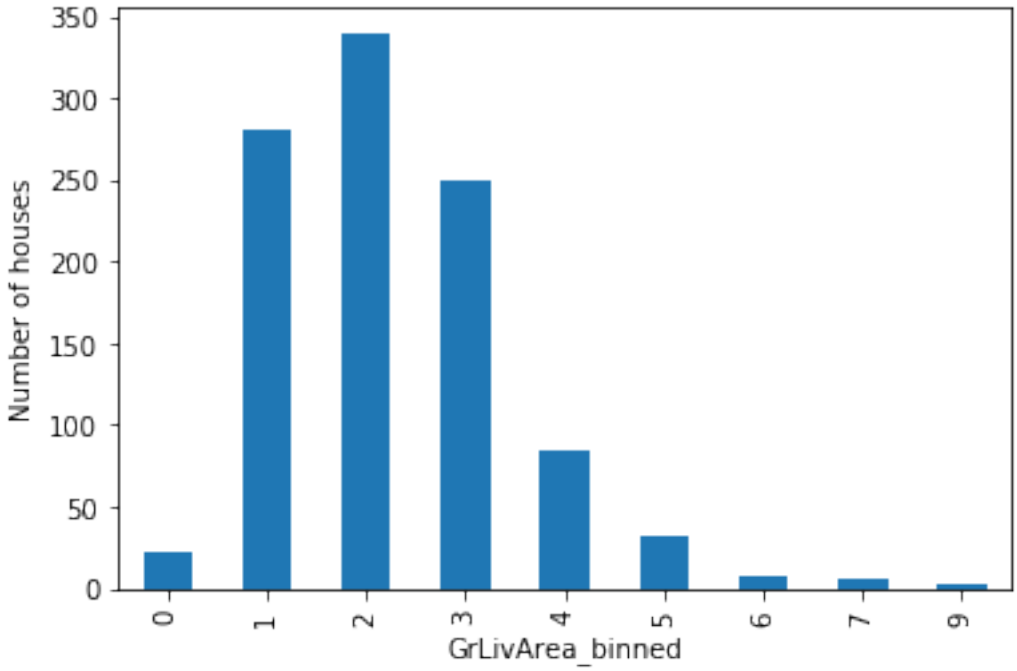
(continues on next page)

(continued from previous page)

```
# 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')
```




### API Reference

**class** feature_engine.discretisers.**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 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, fit.

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

**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.

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

Learns the boundaries of the equal width intervals / bins for each variable.

**Parameters**

- **X** (*pandas dataframe of shape = [n_samples, n_features]*) – The training input samples. Can be the entire dataframe, not just the variables to transform.
- **y** (*None*) – y is not needed in this encoder. You can pass y or None.

**binner_dict_**
The dictionary containing the {variable: interval boundaries} pairs used to transform each variable.
**Type** dictionary

**transform**(*X*)
Sorts the variable values into the intervals.

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

**Returns** **X_transformed** – The transformed data with the discrete variables.

**Return type** pandas dataframe of shape = [n_samples, n_features]

### 6.6.3 DecisionTreeDiscretiser

The DecisionTreeDiscretiser() divides the numerical variable into groups estimated by a decision tree. In other words, the bins are the predictions made by a decision tree. More details in the API Reference section at the end of this page. A grid with parameters can be passed to find the best performing tree, determining the scoring metric and cross-validation fold.

The DecisionTreeDiscretiser() 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 import discretisers as dsc

# 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 = dsc.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)
```

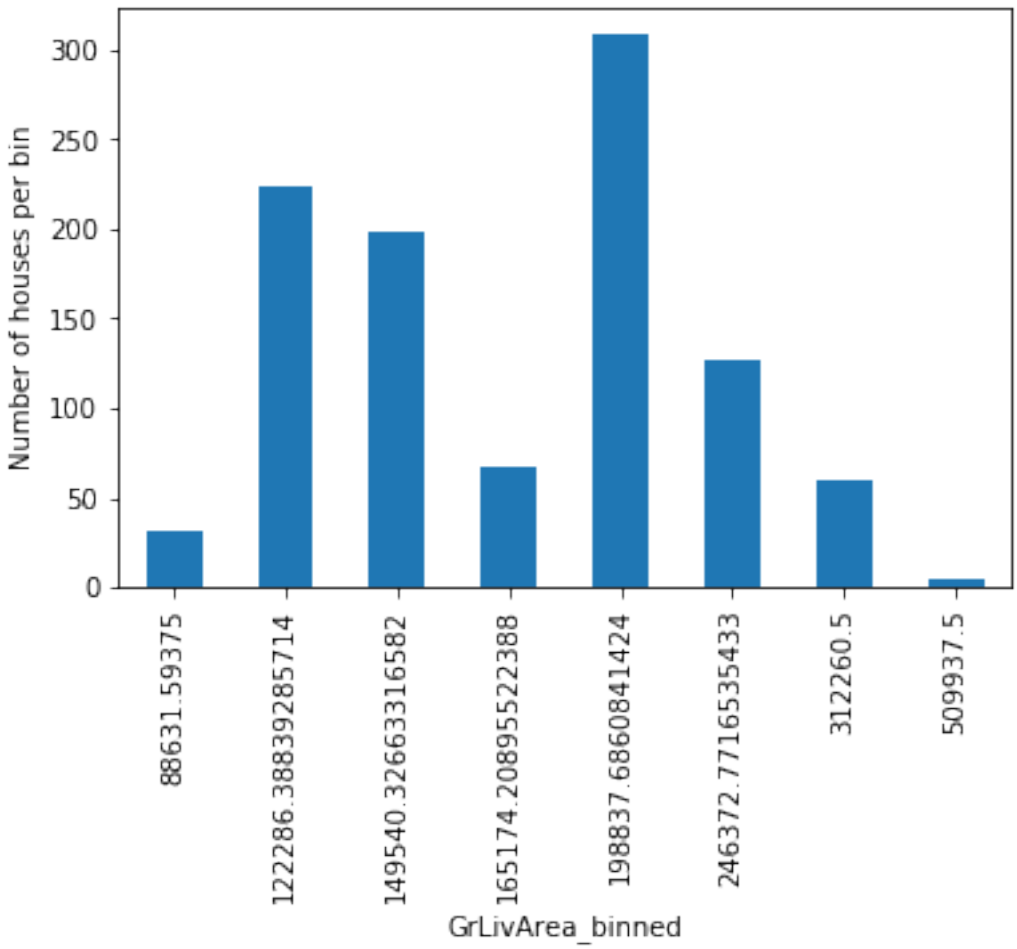
(continues on next page)

(continued from previous page)

```
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')
```




### API Reference

**class** feature_engine.discretisers.**DecisionTreeDiscretiser**(*cv=3*, *scoring='neg_mean_squared_error'*, *variables=None*, *param_grid={'max_depth': [1, 2, 3, 4]}*, *regression=True*, *random_state=None*)

The DecisionTreeDiscretiser() divides continuous numerical variables into 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

At the moment, this transformer only works for binary classification or regression. Multi-class classification is not supported.

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, fit.

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

**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 type variables.
- **regression** (*boolean, default=True*) – Indicates whether the discretiser should train a regression or a classification decision tree.
- **param_grid** (*dictionary, default={'max_depth': [1,2,3,4]}*) – 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().
- **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.

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

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

**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.*) – Target variable. Required to train the decision tree.

**binner_dict_**

The dictionary containing the {variable: fitted tree} pairs.

**Type** dictionary

**scores_dict_**

The score of the best decision tree, over the train set. Provided in case the user wishes to understand the performance of the decision tree.

**Type** dictionary

**transform** (*X*)

Returns the predictions of the tree, based of the variable original values. The tree outcome is finite, aka, discrete.

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

**Returns** **X_transformed** – The dataframe with transformed variables.

**Return type** pandas dataframe of shape = [n_samples, n_features]

### 6.6.4 UserInputDiscretiser

The UserInputDiscretiser() 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 UserInputDiscretiser() 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.

```
import numpy as np
import pandas as pd
from sklearn.datasets import load_boston
from feature_engine.discretisers import UserInputDiscretiser

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

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

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

X['LSTAT'].head()
```

```
'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]}
```

```
0    0
1    0
2    0
3    0
4    0
Name: LSTAT, dtype: int64
```

### API Reference

**class** feature_engine.discretisers.**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 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, fit.

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

**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.

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

Learns the boundaries of the equal width intervals / bins for each variable.

**Parameters**

- **X** (*pandas dataframe of shape = [n_samples, n_features]*) – The training input samples. Can be the entire dataframe, not just the variables to transform.
- **y** (*None*) – y is not needed in this encoder. You can pass y or None.

**binner_dict_**

The dictionary containing the {variable: interval boundaries} pairs used to transform each variable.
**Type** dictionary

**transform**(*X*)

Sorts the variable values into the intervals.

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

**Returns** **X_transformed** – The transformed data with the discrete variables.

**Return type** pandas dataframe of shape = [n_samples, n_features]

## 6.7 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.

### 6.7.1 Winsorizer

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 import outlier_removers as outr

# 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 = outr.Winsorizer(
    distribution='gaussian', tail='right', fold=3, variables=['age', 'fare'])

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

(continues on next page)

(continued from previous page)

```
# 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
```

### API Reference

**class** feature_engine.outlier_removers.**Winsorizer**(*distribution='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 1) a Gaussian approximation, 2) the inter-quantile range proximity rule or 3) 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 distribution='gaussian' fold gives the value to multiply the std.

If distribution='skewed' fold is the value to multiply the IQR.

If distribution='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**

- **distribution** (*str, default=gaussian*) – Desired distribution. Can take 'gaussian', 'skewed' or 'quantiles'.

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

  skewed: 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 distribution='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 distribution='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.

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

Learns the values that should be used to replace outliers.

**Parameters**

- **X** (*pandas dataframe of shape = [n_samples, n_features]*) – The training input samples.
- **y** (*None*) – y is not needed in this transformer. You can pass y or None.

**right_tail_caps_**

The dictionary containing the maximum values at which variables will be capped.

**Type** dictionary

**left_tail_caps_**

The dictionary containing the minimum values at which variables will be capped.

**Type** dictionary

**transform** (*X*)

Caps the variable values, that is, censors outliers.

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

**Returns** **X_transformed** – The dataframe with the capped variables.

**Return type** pandas dataframe of shape = [n_samples, n_features]

### 6.7.2 ArbitraryOutlierCapper

The ArbitraryOutlierCapper censors variable values at user pre-defined maximum and minimum values. For more details, read 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 import outlier_removers as outr

# 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['age'] = data['age'].astype('float')
        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 = outr.ArbitraryOutlierCapper(
                max_capping_dict={'age': 50, 'fare': 200}, min_capping_dict=None)

# 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
```

### API Reference

**class** feature_engine.outlier_removers.**ArbitraryOutlierCapper**(*max_capping_dict=None*, *min_capping_dict=None*, *missing_values='raise'*)

The ArbitraryOutlierCapper() caps the maximum or minimum values of a variable by 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**

- **capping_max** (*dictionary, default=None*) – user specified capping values on right tail of the distribution (maximum values).
- **capping_min** (*dictionary, default=None*) – user specified capping values on left tail of the distribution (minimum values).

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

**Parameters**

- **X** (*pandas dataframe of shape = [n_samples, n_features]*) – The training input samples.
- **y** (*None*) – y is not needed in this transformer. You can pass y or None.

**right_tail_caps_**
The dictionary containing the maximum values at which variables will be capped.
**Type** dictionary

**left_tail_caps_**
The dictionary containing the minimum values at which variables will be capped.
**Type** dictionary

**transform**(*X*)
Caps the variable values, that is, censors outliers.

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

**Returns** **X_transformed** – The dataframe with the capped variables.

**Return type** pandas dataframe of shape = [n_samples, n_features]

## 6.7.3 OutlierTrimmer

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 import outlier_removers as outr

# 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 = outr.OutlierTrimmer(
    distribution='skewed', 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
```

### API Reference

**class** feature_engine.outlier_removers.**OutlierTrimmer**(*distribution='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 1) a Gaussian approximation, 2) the inter-quantile range proximity rule or 3) 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 allow the maximum or minimum values with the parameter 'fold'.

If distribution='gaussian' fold gives the value to multiply the std.

If distribution='skewed' fold is the value to multiply the IQR.

If distribution='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**

- **distribution** (*str, default=gaussian*) – Desired distribution. Can take 'gaussian', 'skewed' or 'quantiles'.

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

  skewed: 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 distribution='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 distribution='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.

**transform**(*X*)

Removes observations with outliers from the dataframe.

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

**Returns** **X_transformed** – The dataframe without outlier observations.

**Return type** pandas dataframe of shape = [n_samples, n_features]

## 6.8 Scikit-learn Wrapper

Feature-engine's Scikit-learn wrappers wrap Scikit-learn various transformers and allows their implementation only on a selected subset of features.

### 6.8.1 SklearnTransformerWrapper

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)
```

For more details, check more examples in the Jupyter notebooks in our repository,

### 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 imputer will select all variables of type numeric.
- **transformer** (*sklearn transformer, default=None*) – The desired Scikit-learn transformer.

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

The *fit* method allows scikit-learn transformers to learn the required parameters from the training data set.

**transform**(*X*)

Apply the transformation to the dataframe.

## 6.9 Contributing

Feature-engine is an open source project, originally designed to support the online course Feature Engineering for Machine Learning course 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 or integration tests for our project.
- Write a blog post, 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!.

### 6.9.1 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 how you would like to get involved and 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 ideally as well, the documentation.
6. **Review** the code with one of us, who will guide you to a final submission.
7. **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.

Once forked, 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. Create a virtual environment with the virtual environment tool of your choice.
3. Change directory into the cloned repository.:

```
$ cd feature_engine
```

4. 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.

5. Install the additional dependencies for tests and documentation.:

```
$ pip install -r test_requirements.txt
$ pip install -r docs/requirements.txt
```

6. Checkout and switch to the develop branch.

Feature-engine's repository has a `develop` branch where we develop the new functionality before the next version is released. Switch to the develop branch as follows.:

```
$ git fetch
$ git checkout develop
```

7. Create a new branch where you will develop your feature.:

```
$ git checkout -b myfeaturebranch develop
```

Please give the branch a name that identifies which feature you are going to build.

8. If you haven't done so, set up up an `upstream` remote from where you can pull the latest code changes occurying 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)
```

Now you are ready to start developing your feature.

#### Developing a new feature

9. First thing, make a pull request (PR). The PR should be made from your feature_branch (in your fork), to Feature-engine's develop branch in the main repository.

At this point, we should have discussed the contribution. But if we haven't, we will get in touch to do so.

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.

One the submission is reviewd and provided the continuous integration tests have passed and the code is up to date with Feature-engine's develop branch, we will be ready to "Squash and Merge" your contribution, into the `develop` 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.

After a few features have been developed in the develop branch, by yourself and others, it will be merged into master and released in a new Feature-engine version to PyPi. Once your contribution has been merged into master, you will be listed as a Feature-engine contributor :)

#### After your PR is merged

Update your local fork:

```
$ git checkout develop
$ git pull upstream develop
$ git push origin develop
```

Finally, delete the old feature branch, both locally and on GitHub. Well done and thank you very much!

## 6.10 Changelog

### 6.10.1 Version 0.5.0

- Deployed: Friday, July 10, 2020
- Contributors: Soledad Galli

**Major Changes:**

- **Bug fix: fixed error in weight of evidence formula in the `WoERatioCategoricalEncoder`. The old formula, that**

  - **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.

### 6.10.2 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.

### 6.10.3 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.

### 6.10.4 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_`.

# INDEX