feature_engine Documentation

Release 0.3.0

Soledad Galli

Aug 05, 2019

Table of Contents

1 Installation **3**

2 Contributing **5**

3 Feature-engine’s Transformers **7**
- 3.1 Missing Data Imputation: Imputers 7
- 3.2 Categorical Variable Encoders: Encoders 7
- 3.3 Numerical Variable Transformation: Transformers 7
- 3.4 Variable Discretisation: Discretisers 8
- 3.5 Outlier Capping: Cappers 8

4 Getting Help **9**

5 Find a Bug? **11**

6 Open Source **13**
- 6.1 Quick Start 13
- 6.2 Missing data imputation: imputers 17
- 6.3 Categorical variable encoding: encoders 30
- 6.4 Variable transformation: variable transformers 42
- 6.5 Variable discretisation: discretisers 53
- 6.6 Outlier capping: cappers 62
- 6.7 Changelog 65

Index **67**

Fig. 1: Feature-engine rocks!

Feature-engine is a Python library that contains several 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 to select which variables to engineer within each transformer.

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.

CHAPTER 1

Installation

Feature-engine is a Python 3 package and works well with 3.5 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 2

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.

More details on how to contribute will come soon! Meanwhile, feel free to fork the Github repo and make pull requests, create an issue, or send feedback. More details on how to reach us in the **Getting help** section below.

Thank you for your contributions!

CHAPTER 3

Feature-engine's Transformers

3.1 Missing Data Imputation: Imputers

- *MeanMedianImputer*: replaces missing data in numerical variables by 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'
- *FrequentCategoryImputer*: replaces missing data in categorical variables by the mode
- *RandomSampleImputer*: replaces missing data with random samples of the variable
- *AddNaNBinaryImputer*: 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 number 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 in one group

3.3 Numerical Variable Transformation: Transformers

- *LogTransformer*: perform logarithmic transformation of numerical variables

- *ReciprocalTransformer*: perform reciprocal transformation of numerical variables
- *PowerTransformer*: perform 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 percentage of obs intervals
- *EqualWidthDiscretiser*: sorts variable into equal size contiguous intervals
- *DecisionTreeDiscretiser*: uses decision trees to create finite variables

3.5 Outlier Capping: Cappers

- *Winsorizer*: caps maximum or minimum values using Gaussian approx or IQR rule
- *ArbitraryOutlierCapper*: caps maximum and minimum values arbitrarily

CHAPTER 4

Getting Help

Can't get someting 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 5

Find a Bug?

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

CHAPTER 6

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 3.5 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
```

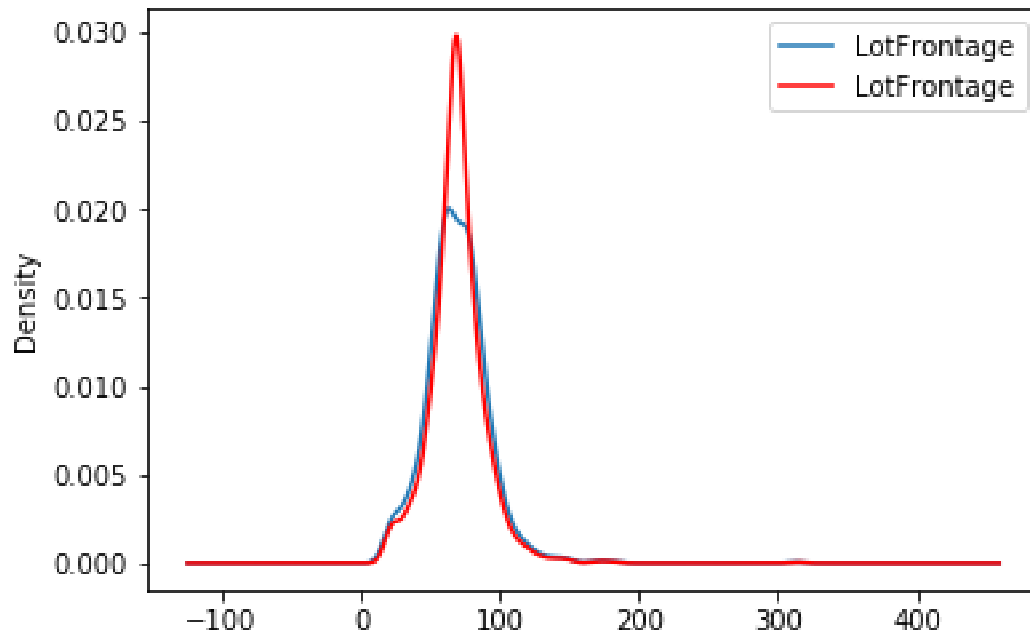
You can also use the `-U` flag to update Scikit-learn, pandas, NumPy, or any other libraries that work well with Feature-engine to their latest versions.

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

# 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 of 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)

# categorical encoders work only with object type variables
data[discrete]= data[discrete].astype('O')

# 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.AddNaNBinaryImputer(variables = ['LotFrontage'])),

    # 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 numerical 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
```

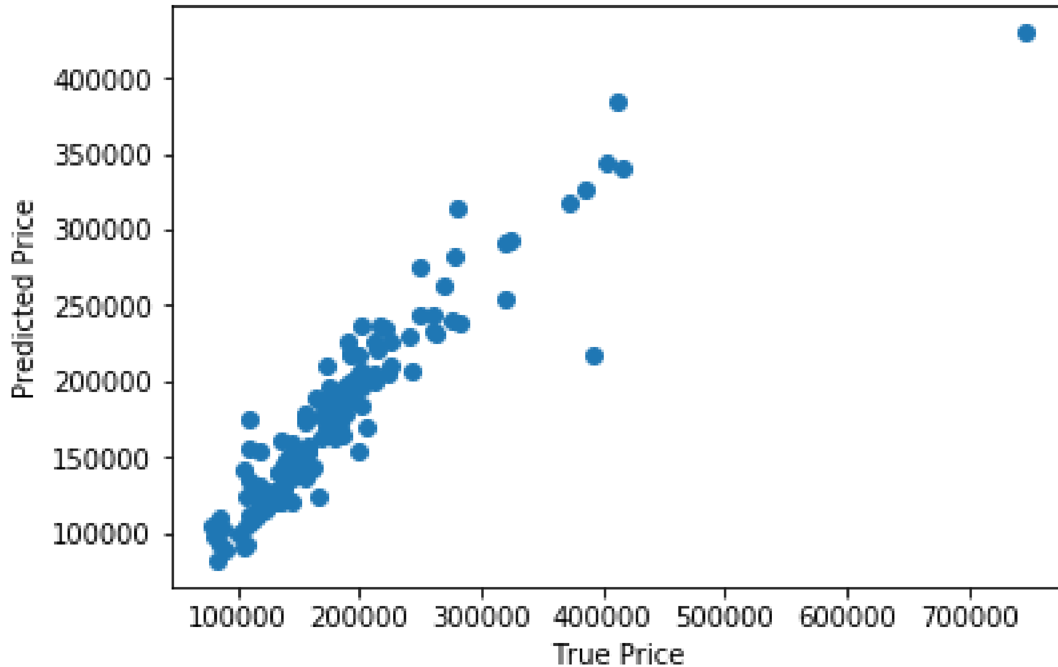
(continues on next page)

(continued from previous page)

```
    ('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 train 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 documetation are based on these 2 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.

However, original data and documentation can be found here:

- Documentation
- Data

6.2 Missing data imputation: imputers

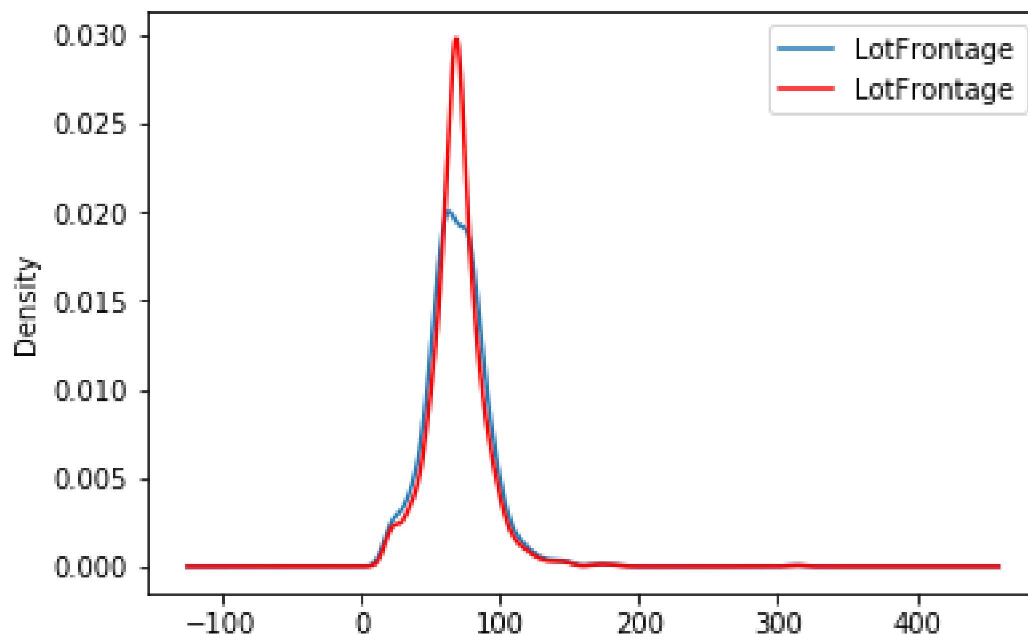
Feature-engine's missing data imputers replace missing data by estimated parameters or arbitrary values.

6.2.1 MeanMedianImputer

The MeanMedianImputer() replaces missing data with the mean or median of the variable. It works only with numerical variables. A list of variables 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

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)

# 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 in each variable by the mean or median value of the variable.

The MeanMedianImputer() works only with numerical variables.

A list of variables to impute can be indicated in a list. If no variable list is passed the MeanMedianImputer() will automatically find and select all variables of type numeric.

The estimator first calculates the mean / median values for the indicated variables (fit).

The estimator then fills 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

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

Type list, default=None

imputer_dict_

The dictionary containing the mean / median values to use for each variable to replace missing data. It is calculated when fitting the imputer.

Type dictionary

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

Learns the mean or median values that should be used to replace mising data in each 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** (*None*) – y is not needed in this transformer, yet the sklearn pipeline API requires this parameter for checking. You can pass None or y.

6.2.2 ArbitraryNumberImputer

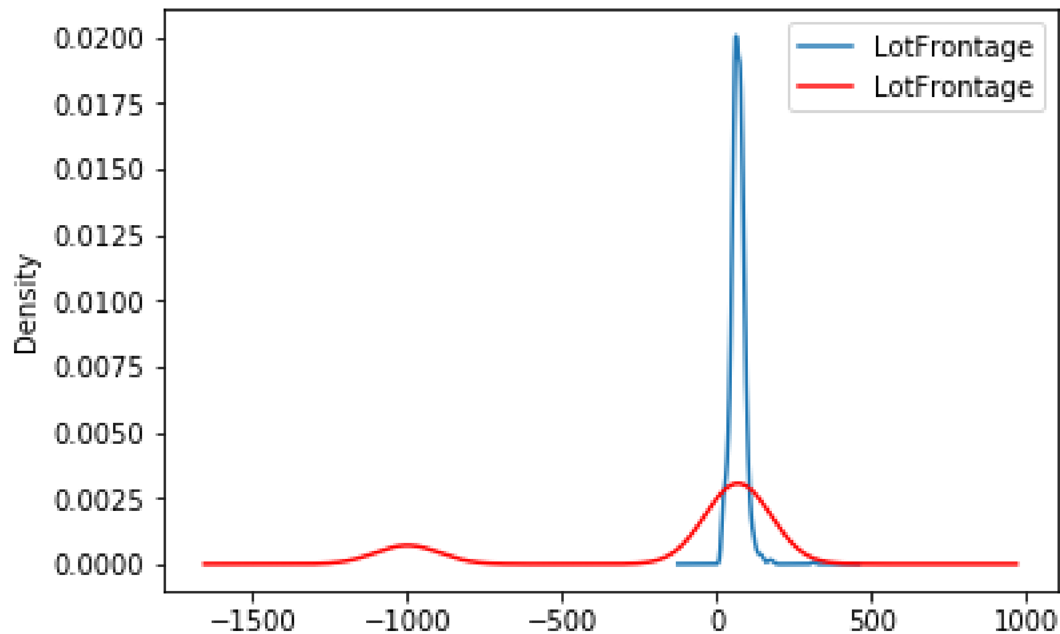
The ArbitraryNumberImputer() replaces missing data with an arbitrary value determined by the user. It works only with numerical variables. A list of variables can be indiacated, or the imputer will automatically select all numerical variables in the train set.

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

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)
```

(continues on next page)

(continued from previous page)

```
# 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)

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() transforms features by replacing missing data in each feature for a given 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 (*self*, *X*, *y=None*)
Checks that the variables are numerical.

Parameters

- **X** (*pandas dataframe of shape = [n_samples, n_features]*) – The training input samples. Can be the entire dataframe, not just the variables to impute.
- **y** (*None*) – y is not needed in this transformer, yet the sklearn pipeline API requires this parameter for checking.

transform (*self*, *X*)
Replaces missing data with the arbitrary values.

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

Returns X_transformed – The dataframe containing no missing values for the selected variables

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

6.2.3 EndTailImputer

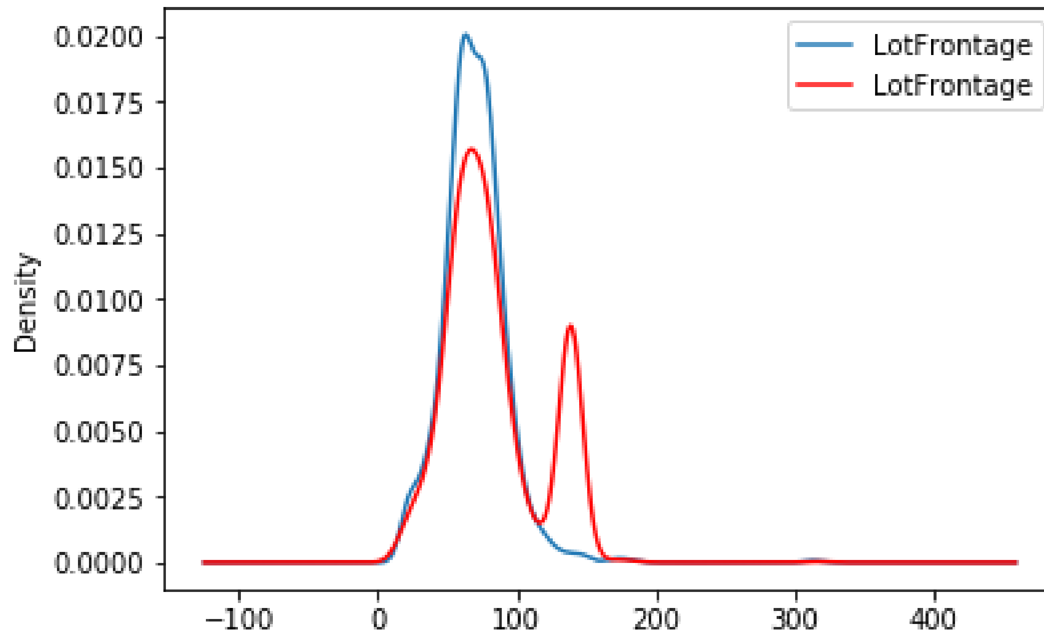
The EndTailImputer() replaces missing data with a value at the end of the distribution. The value can be determined using the mean plus or minus a number of times the standard deviation, or using the inter-quantile range proximity rule. The user decides whether the missing data should be placed at the right or left tail of the variable distribution. It works only with numerical variables. A list of variables can be 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

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)
```

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

The EndTailImputer() transforms features by replacing missing data in each feature for a given value at the tail of the distribution.

The EndTailImputer() works only with numerical variables. A list of variables to impute can be indicated in a list. If no variable list is passed the EndTailImputer() will automatically find and select all variables of type numeric.

The estimator first calculates the values at the end of the distribution for the indicated features. The values at the end of the distribution can be given by the Gaussian limits or the IQR proximity rule limits.

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-quantal range = 75th Quantile - 25th quantile

You can select how far out you want to place your missing values by tuning the number by which you multiply the std or the IQR, using the parameter 'fold'.

The encoder then fills the missing data with the estimated values.

Parameters

- **distribution** (*str, default=gaussian*) – Desired distribution. Can take 'gaussian' or 'skewed'. gaussian: the encoder will use the Qaussian limits to find the values to replace missing data. skewed: the encoder will use the IQR limits to find the values to replace missing data.
- **tail** (*str, default=right*) – Whether the missing values will be placed at the right or left tail of the variable distribution. Can take 'left' or 'right'.
- **fold** (*int, default=3*) – How far out to place the missing data. Fold will multiply the std or the IQR. Recommended values, 2 or 3 for Gaussian, 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 numerical type variables.

imputer_dict_

The dictionary containing the values at end of the distribution to use to replace missing data for each variable. The values are calculated when fitting the imputer.

Type dictionary

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

Learns the values that should be used to replace mising data in 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 needed in this transformer, yet the sklearn pipeline API requires this parameter for checking.

6.2.4 CategoricalVariableImputer

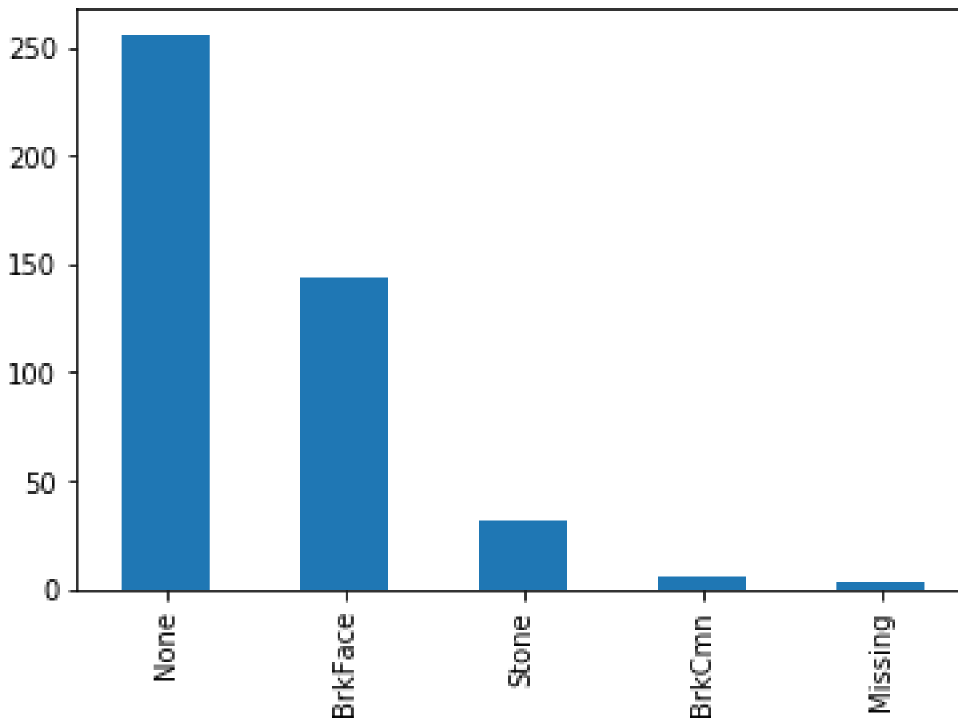
The CategoricalVariableImputer() replaces missing data in categorical variables with the string 'Missing'. It works only with categorial variables. A list of variables can be indiacated, or the imputer will automatically select all categorical variables in the train set.

```
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')

# 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
```

(continues on next page)

(continued from previous page)

```
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**(*variables=None*)

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

The CategoricalVariableImputer() works only with categorical variables.

A list of variables to impute can be indicated. If no variable list is passed the CategoricalVariableImputer() will automatically find and select all variables of type object.

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

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

Checks that the selected variables are categorical.

Parameters

- **X** (*pandas dataframe of shape = [n_samples, n_features]*) – The training input samples. Can be the entire dataframe, not just seleted variables
- **y** (*None*) – y is not needed in this transformer, yet the sklearn pipeline API requires this parameter for checking.

transform(*self, X*)

Replaces missing values with the new Label 'Missing'.

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

Returns X_transformed – The dataframe containing no missing values for the selected variables

Return type dataframe of shape = [n_samples, n_features]

6.2.5 FrequentCategoryImputer

The FrequentCategoryImputer() replaces missing data in categorical variables with the most frequent category. It works only with categorial variables. A list of variables can be indiacated, 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')

# 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.FrequentCategoryImputer(variables='MasVnrType')

# fit the imputer
imputer.fit(X_train)

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

API Reference

class feature_engine.missing_data_imputers.**FrequentCategoryImputer**(*variables=None*)

The FrequentCategoryImputer() replaces missing data in categorical variables by the most frequent category.

The FrequentCategoryImputer() works only with categorical variables.

A list of variables to impute can be indicated. If no variable list is passed the FrequentCategoryImputer() will automatically find and select all variables of type object.

The encoder first finds the most frequent category per variable (fit).

The encoder then fills the missing data with the most frequent category (transform).

Parameters variables (*list, default=None*) – The list of variables to be imputed. If None, the transformer will find and select all object type variables.

imputer_dict_
The dictionary mapping each variable to the most frequent category which will be used to fill missing data. These are calculated when fitting the transformer.

Type dictionary

fit (*self*, *X*, *y=None*)
Learns the most frequent category for each 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** (*None*) – y is not needed in this transformer, yet the sklearn pipeline API requires this parameter for checking.

transform (*self*, *X*)
Replaces missing data with the most frequent category of the variable.

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

Returns X_transformed – The dataframe containing no missing values for the selected variables

Return type dataframe of shape = [n_samples, n_features]

6.2.6 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 indiacated, or the imputer will automatically select all variables in the train set.

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

For example, if the observvation 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))
```

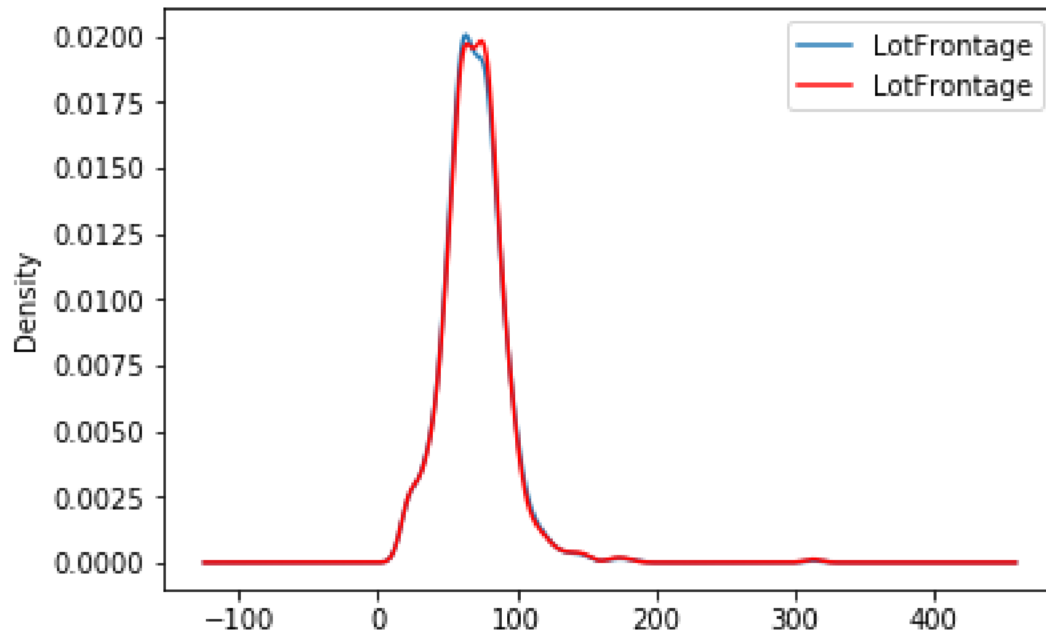
More details on how to use the RandomSampleImputer():

```
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')
```

(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.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 the seed = 'general' then the random_state can be either None or an integer. The seed will be initialised to the random_state and all observations will be imputed in one go. If the 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 adding or multiplying the variable values indicated in the list passed to the random_state.

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.

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 dataset contains Personal Information.

The transformer fills the 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 set the seed, observation per observation.
- **seed** (*str, default='general'*) – Indicates wheter the seed should be set for each observation to impute or one seed should be used for a batch of imputations. general: one seed will be used to impute the entire dataframe. This is the equivalent of setting the seed in pandas.sample(random_state) observation: the seed will be set per each observation using the values of the variables indicated in the random_state.
- **seeding_method** (*str, default='add'*) – If more than one variable are indicated to seed the randoms 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 present in the train set.

X

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

Type dataframe.

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

Makes a copy of the indicated variables in the training dataframe, from which it will randomly extract the values during transform.

Parameters

- **X** (*pandas dataframe of shape = [n_samples, n_features]*) – The training input samples. Can be the entire dataframe, not just seleted variables.
- **y** (*None*) – y is not needed in this transformer, yet the sklearn pipeline API requires this parameter for checking.

transform(*self*, *X*)

Replaces missing data with random values taken from the train set.

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

Returns X_transformed – The dataframe containing NO missing values for all indicated variables

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

6.2.7 AddNaNBinaryImputer

The AddNaNBinaryImputer() 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.

```
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.AddNaNBinaryImputer(
    variables=['Alley', 'MasVnrType', 'LotFrontage', 'MasVnrArea'])

# fit the imputer
addBinary_imputer.fit(X_train)

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

train_t[['Alley_na', 'MasVnrType_na', 'LotFrontage_na', 'MasVnrArea_na']].head()
```

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

API Reference

class feature_engine.missing_data_imputers.**AddNaNBinaryImputer**(*variables=None*)

The AddNaNBinaryImputer() adds an additional column or binary variable to indicating if data is missing for the selected variables. AddNaNBinaryImputer() will add as many missing indicators as variables are selected.

The AddNaNBinaryImputer() works with both numerical and categorical variables. A list of variables can be indicated. If None, the imputer will select and add missing indicators to all variables present in the training set.

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

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

Learns the variables for which the additionalmissing indicator 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 transformer, yet the sklearn pipeline API requires this parameter for checking.

transform(*self, X*)

Adds the additional binary missing indicator variables indicating the presence of missing data per observation.

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

Returns X_transformed – The dataframe containing the additional binary variables

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

6.3 Categorical variable encoding: encoders

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

6.3.1 OneHotCategoricalEncoder

The OneHotCategoricalEncoder() replaces original categorical variable, 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 by the n most popular categories, n being determined by the user.

The OneHotCategoricalEncoder() works only with categorical variables. A list of variables can be indiacated, or the imputer will automatically select all categorical variables in the train set.

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

from feature_engine 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)

# 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 boolean variables which take value 0 or 1, to indicate if a certain category is present for an observation.

Each one of the boolean variables are also known as dummy variables or binary 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 present in a variable, or to create dummy variables only for the top n most popular categories, that is, the categories that are present in 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 (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 only encode categorical variables (object type) and ignore the rest.

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).

Parameters

- **top_categories** (*int, default=None*) – If None is selected, a dummy variable will be created for each category per variable.If set to True, the encoder will find the most frequent categories. 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 dropped. 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 available categories, or if set to True, it will ignore the last variable of the list.

encoder_dict_

The dictionary containg the frequent categories (that will be kept) for each variable.

Type dictionary

fit(*self, 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** (*Target*) –

transform(*self, X*)

Creates the dummy / boolean variables.

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

Returns

- **X_transformed** (*pandas dataframe. The shape of the dataframe will*)
- *be different from the original as it includes the dummy variables.*

6.3.2 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 indiacated, or the imputer will automatically select all categorical variables in the train set.

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

from feature_engine 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.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='co 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 can be passed as an argument. If no variables are passed as argument, the encoder will only encode categorical variables (object type) and ignore the rest.

The encoder first maps the categories to the numbers 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.

encoder_dict_

The dictionary containing the {count / frequency: category} pairs used to replace categories for every variable.

Type dictionary

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

Learns the numbers that should 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 seleted variables.
- **y** (*None*) – y is not needed in this encoder, yet the sklearn pipeline API requires this parameter for checking. You can either leave it as None or pass y.

6.3.3 OrdinalCategoricalEncoder

The OrdinalCategoricalEncoder() replaces the categories by digits, from 1 to the number of different labels. 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 1, and those where the mean of the target is smallest will get the number n.

The OrdinalCategoricalEncoder() works only with categorical variables. A list of variables can be indiacated, 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
```

(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.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.

For the ordered ordinal encoding 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 1, red by 2 and grey by 0.

For the 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 as argument, the encoder will only encode categorical variables (object type) and ignore the rest.

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 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 {ordinal number: category} pairs used to replace categories for every variable.

Type dictionary

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

Learns the numbers that should be used to replace the labels in each 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** (*Target. Can be None if selecting encoding_method = 'arbitrary'.*) –
- **needs to be passed when fitting the transformer.** (*Otherwise,*) –

6.3.4 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 default rate 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 indiacated, or the imputer will automatically select all categorical variables in the train set.

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

from feature_engine 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]
```

(continues on next page)

(continued from previous page)

```
        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 of the target.

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 only encode categorical variables (object type) and ignore the rest.

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.

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

Type dictionary

fit (*self*, *X*, *y*)
Learns the numbers that should be used to replace the labels in each 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** (*Target*) –

6.3.5 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 indiacated, or the imputer will automatically select all categorical variables in the train set.

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

from feature_engine 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=5,
                                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
```

(continues on next page)

(continued from previous page)

```
encoder = ce.WoERatioCategoricalEncoder(
encoding_method='woe', variables=['cabin', 'pclass', 'embarked'])

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

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

encoder.encoder_dict_
```

```
{'cabin': {'B': 1.1631508098056806,
  'C': 0.2548922496287902,
  'D': 0.9382696385929302,
  'E': 0.9382696385929302,
  'Rare': 0.2719337154836416,
  'n': -0.8243393908312957},
 'pclass': {1: 0.4784902431230542,
  2: -0.25671984684781396,
  3: -1.0509842396788551},
 'embarked': {'C': 0.23309388216737797,
  'Q': -0.5172565140962812,
  'S': -0.6679453885859952}}
```

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(1) / p(0))

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

Note: This categorical encoder 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 woe is selected. Alternatively, blue will be replaced by 0.8 / 0.2 = 4.

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, 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 only encode categorical variables (object type) and ignore the rest.

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.

encoder_dict_

The dictionary containing the {woe: category} pairs or the {prob ratio: category} pairs used to replace the categories in each variable.

Type dictionary

fit (*self, X, y*)

Learns the numbers that should 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 seleted variables.
- **y** (*Target, must be binary [0,1]*) –

6.3.6 RareLabelCategoricalEncoder

The RareLabelCategoricalEncoder() groups infrequent categories altogether into one new category called 'Rare'. 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 indiacated, or the imputer will automatically select all categorical variables in the train set.

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

from feature_engine 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.RareLabelCategoricalEncoder(tol=0.03, n_categories=5,
                                         variables=['cabin', 'pclass', 'embarked'])

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

(continues on next page)

(continued from previous page)

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

encoder.encoder_dict_
```

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

API Reference

class feature_engine.categorical_encoders.**RareLabelCategoricalEncoder**(*tol=0.05*, *n_categories=10*, *variables=None*)

The RareLabelCategoricalEncoder() groups rare / infrequent categories in a new category called "Rare".

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".

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 only encode categorical variables (object type) and ignore the rest.

The encoder first finds the frequent labels for each variable (fit). The encoder then groups the infrequent labels under the new label 'Rare' (transform).

Parameters

- **tol** (*float, default=0.05*) – the minimum frequency a label should have to be considered frequent and not be removed.
- **n_categories** (*int, default=10*) – the minimum number of categories a variable should have in order 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.

encoder_dict_

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

Type dictionary

fit(*self, 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 seleted variables
- **y** (*None*) – There is no need of a target in a transformer, yet the pipeline API requires this parameter. You can leave y as None, or pass it as an argument.

transform(*self, X*)

Groups rare labels under separate group 'Rare'.

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]

6.4 Variable transformation: variable transformers

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

6.4.1 LogTransformer

API Reference

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

The LogTransformer() applies the logarithmic transformation to numerical variables.

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

The LogTransformer() will transform only numerical variables. A list of variables can be passed as an argument. But, if no variables are indicated, the transformer will automatically select and transform numerical variables and ignore the rest.

Parameters **variables** (*list, default=None*) – The list of numerical variables to be transformed. If None, the transformer will find and select all numerical type variables.

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

Selects the numerical variables and determines whether the logarithm can be applied on 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 seleted variables.
- **y** (*None*) – y is not needed in this transformer, yet the sklearn pipeline API requires this parameter for checking. You can either leave it as None or pass y.

transform(*self, X*)

Transforms the variables using logarithm.

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

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

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

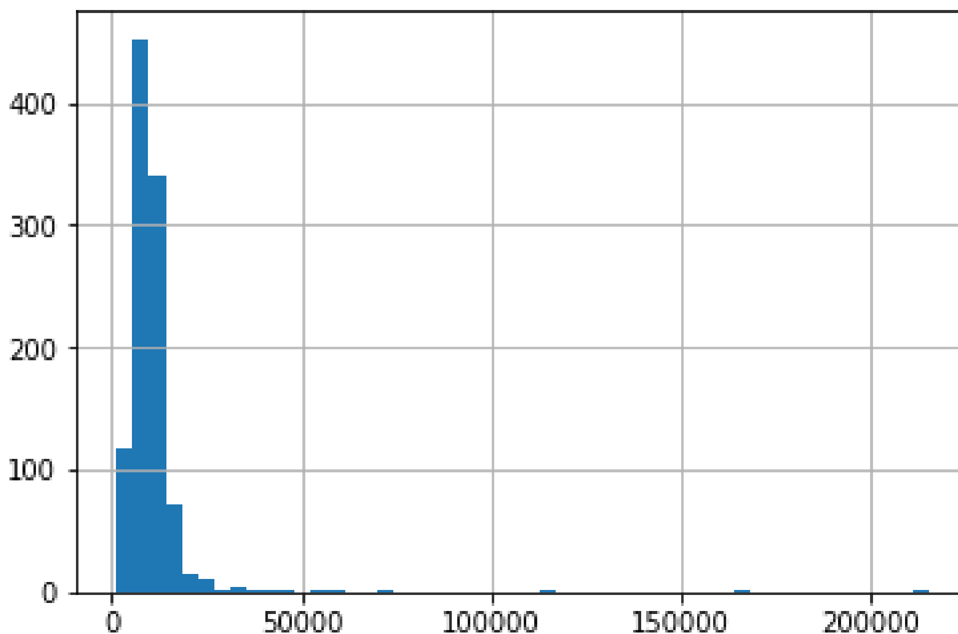
Example Use

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

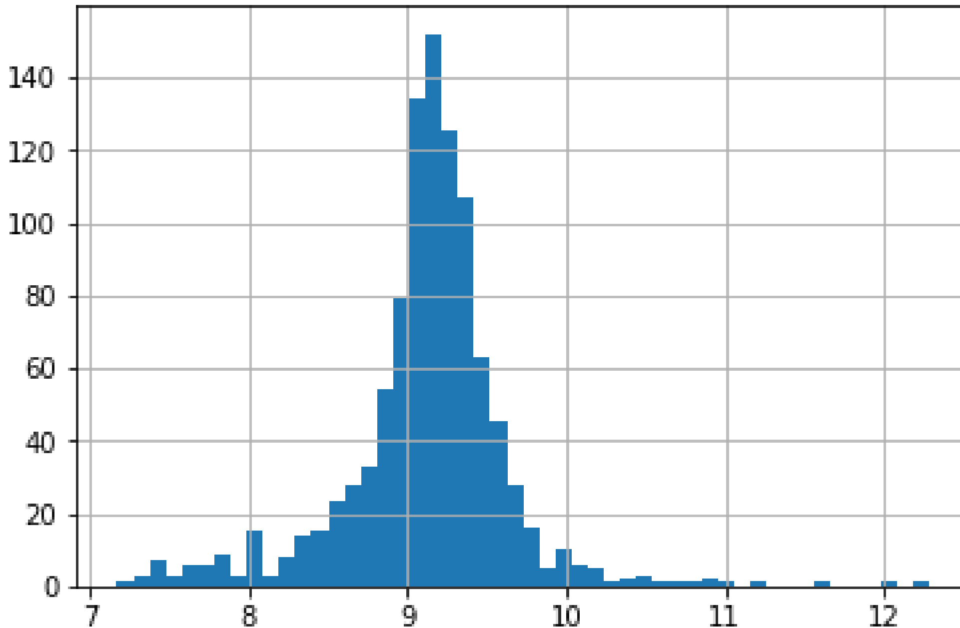
(continues on next page)

(continued from previous page)

```
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['GrLivArea'].hist(bins=50)
```



6.4.2 ReciprocalTransformer

API Reference

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

The ReciprocalTransformer() applies the reciprocal transformation 1 / x to the numerical variables.

The ReciprocalTransformer() only works with numerical variables with non-zero values.

The ReciprocalTransformer() will transform only numerical variables. A list of variables can be passed as an argument. But, if no variables are indicated, the transformer will automatically select and transform numerical variables and ignore the rest.

Parameters **variables** (*list, default=None*) – The list of numerical variables that will be transformed. If None, the transformer will automaticalle find and select all numerical type variables.

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

Parameters

- **X** (*pandas dataframe of shape = [n_samples, n_features]*) – The training input samples. Can be the entire dataframe, not just seleted variables.
- **y** (*None*) – y is not needed in this encoder, yet the sklearn pipeline API requires this parameter for checking. You can either leave it as None or pass y.

transform(*self, X*)

Applies the reciprocal 1 / x transformation.

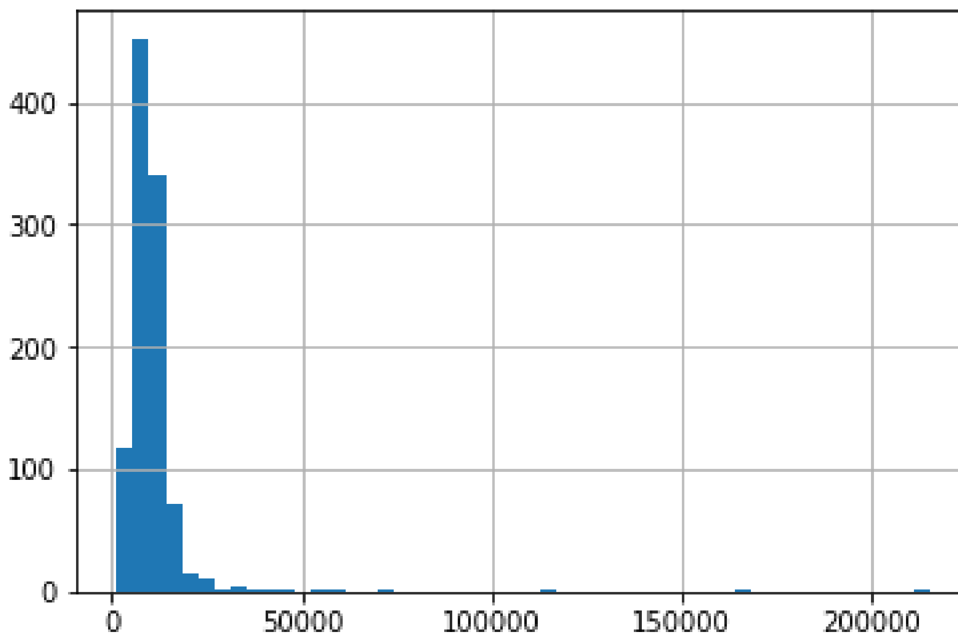
Parameters **X** (*pandas dataframe of shape = [n_samples, n_features]*) – The input samples.

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

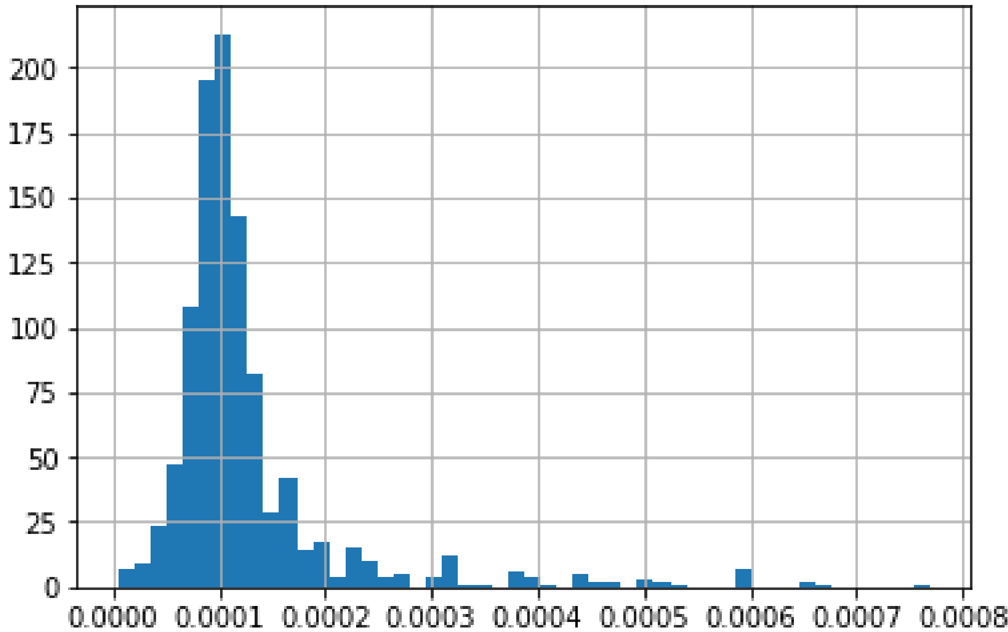
Return type pandas dataframe of shape = [n_samples, n_features]

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
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)
```



6.4.3 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.

The PowerTransformer() will transform only numerical variables. A list of variables can be passed as an argument. But, if no variables are indicated, the transformer will automatically select and transform numerical variables and ignore the rest.

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 type variables.
- **exp** (*float, default=0.5*) – The power (or exponent).

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

Learns the numerical variables that should be transformed

Parameters

- **X** (*pandas dataframe of shape = [n_samples, n_features]*) – The training input samples. Can be the entire dataframe, not just seleted variables.
- **y** (*None*) – y is not needed in this transformer, yet the sklearn pipeline API requires this parameter for checking. You can either leave it as None or pass y.

transform(*self*, *X*)

Applies the power transformation to the variables.

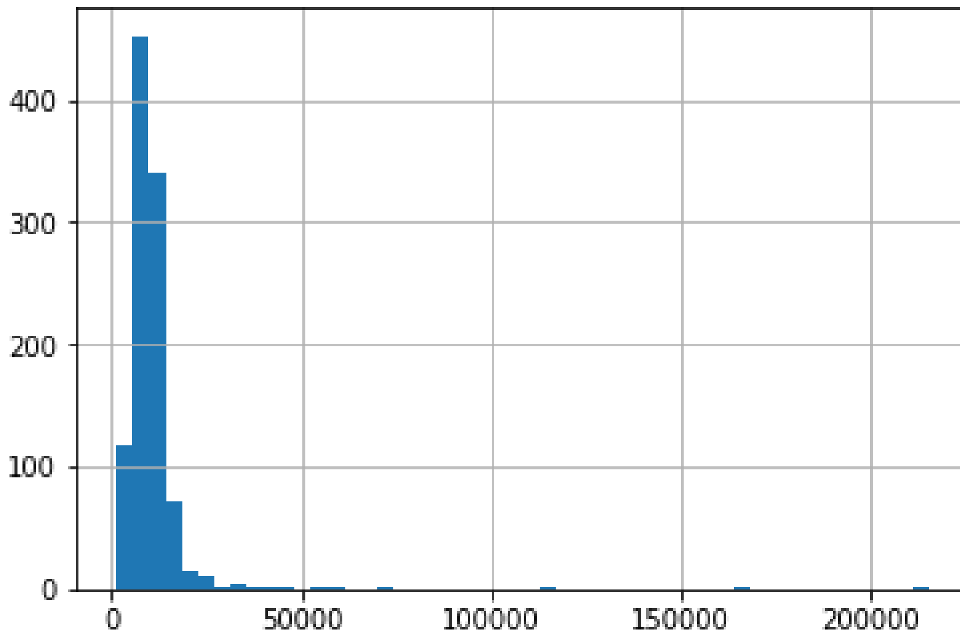
Parameters X (*pandas dataframe of shape = [n_samples, n_features]*) – The input samples.

Returns X_transformed – The dataframe with power transformed variables.

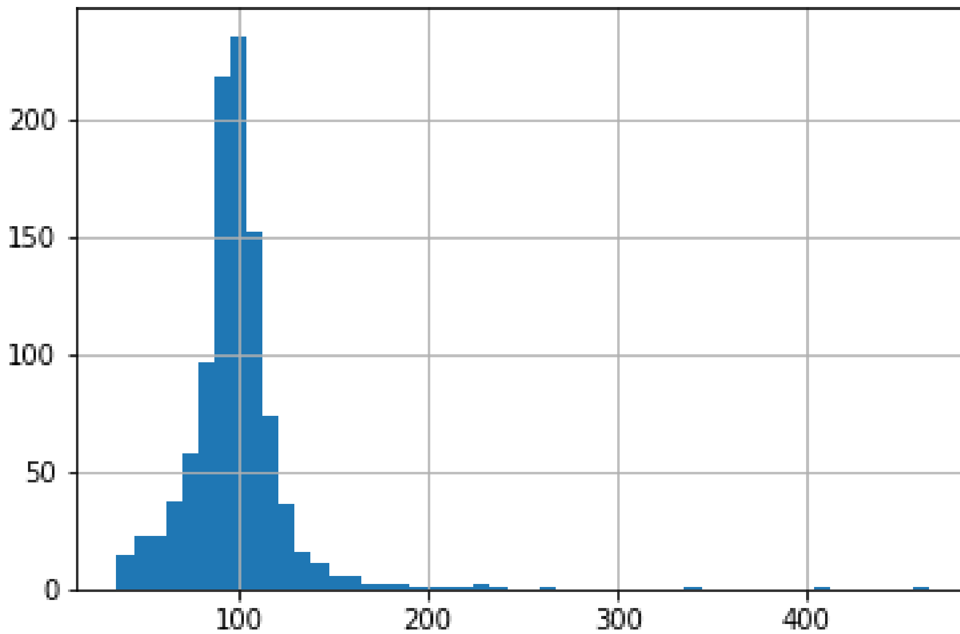
Return type pandas dataframe of shape = [n_samples, n_features]

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)

# 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)
```



6.4.4 BoxCoxTransformer

The Box-Cox transformation is defined as:

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

where Y is the response variable and λ is the transformation parameter. λ varies, typically from -5 to 5. In the

transformation, all values of λ 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 the 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).

The BoxCoxTransformer() will transform only numerical variables. A list of variables can be passed as an argument. But, if no variables are indicated, the transformer will automatically select and transform numerical variables and ignore the rest.

Parameters **variables** (*list, default=None*) – The list of numerical variables that will be transformed. If None, the transformer will automaticalle find and select all numerical type variables.

lamda_dict_

The dictionary containing the {variable: best exponent for the BoxCox transfomration} pairs.

Type dictionary

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

Learns the numerical variables. Captures 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 seleted variables.
- **y** (*None*) – y is not needed in this transformer, yet the sklearn pipeline API requires this parameter for checking. You can either leave it as None or pass y.

transform(*self, X*)

Applies the BoxCox transformation.

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

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

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

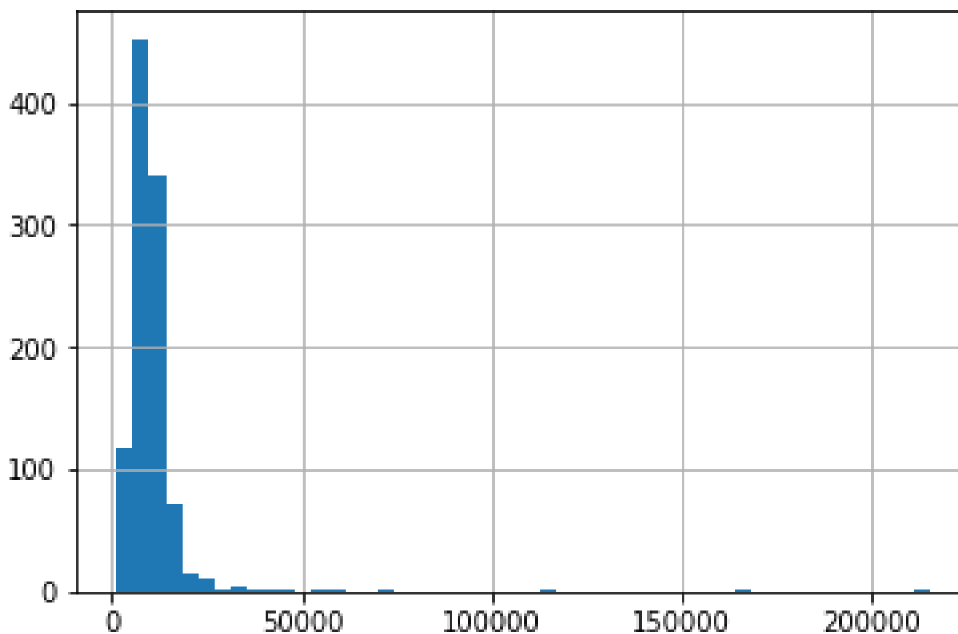
Example Use

```
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')
```

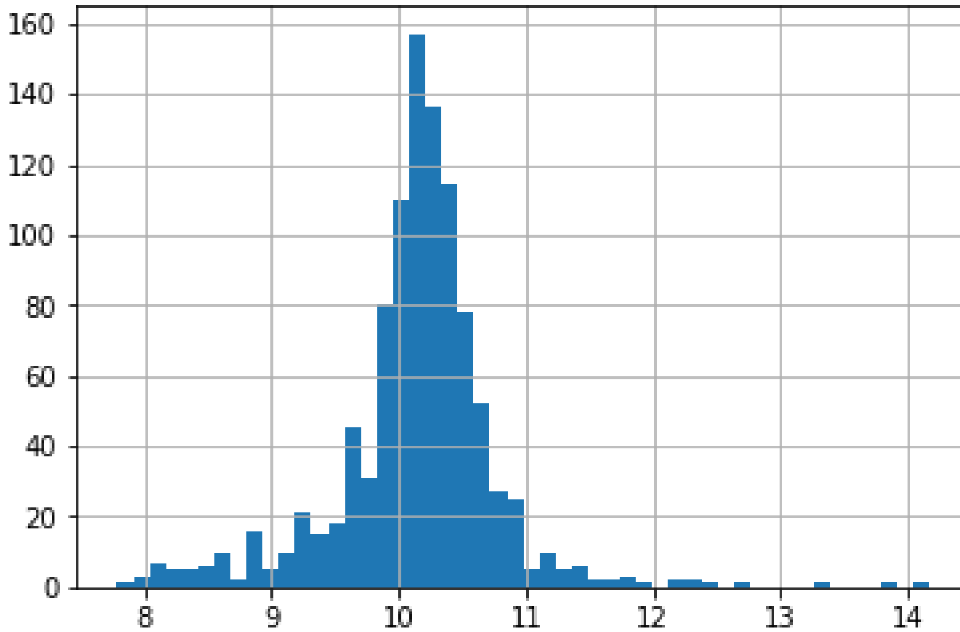
(continues on next page)

(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 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)
```



6.4.5 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 λ 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.

The YeoJohnsonTransformer() will transform only numerical variables. A list of variables can be passed as an argument. But, if no variables are indicated, the transformer will automatically select and transform numerical variables and ignore the rest.

Parameters **variables** (*list, default=None*) – The list of numerical variables that will be trasnformed. If None, the transformer will automatically find and select all numerical type variables.

lamda_dict_

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

Type dictionary

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

Learns the numerical variables. Captures the optimal lambda for the transformation.

Parameters

- **X** (*pandas dataframe of shape = [n_samples, n_features]*) – The training input samples. Can be the entire dataframe, not just seleted variables.
- **y** (*None*) – y is not needed in this transformer, yet the sklearn pipeline API requires this parameter for checking. You can either leave it as None or pass y.

transform(*self*, *X*)

Applies the BoxCox trasformation.

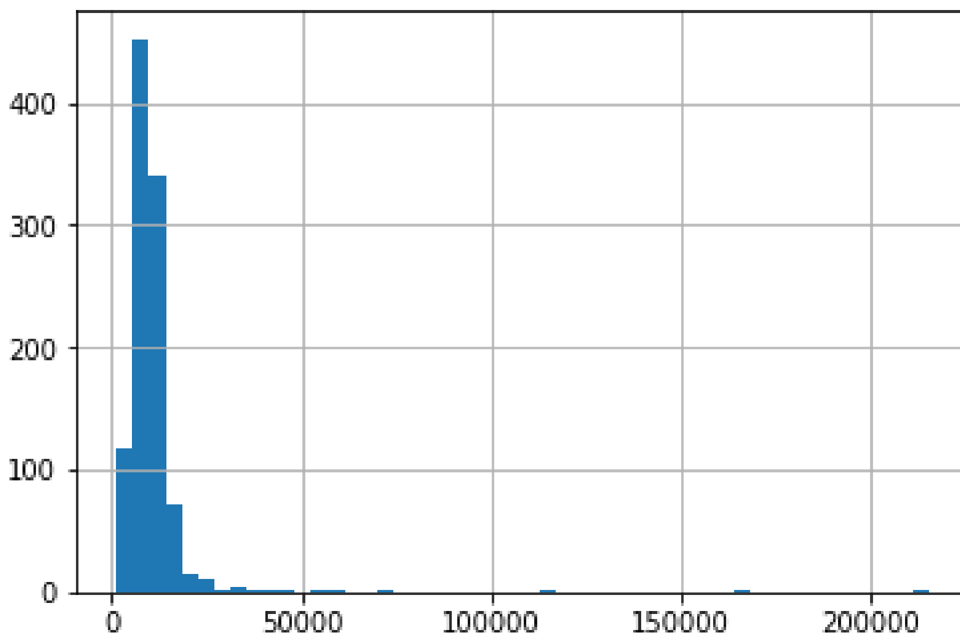
Parameters X (*pandas dataframe of shape = [n_samples, n_features]*) – The input samples.

Returns X_transformed – The dataframe with the transformed variables.

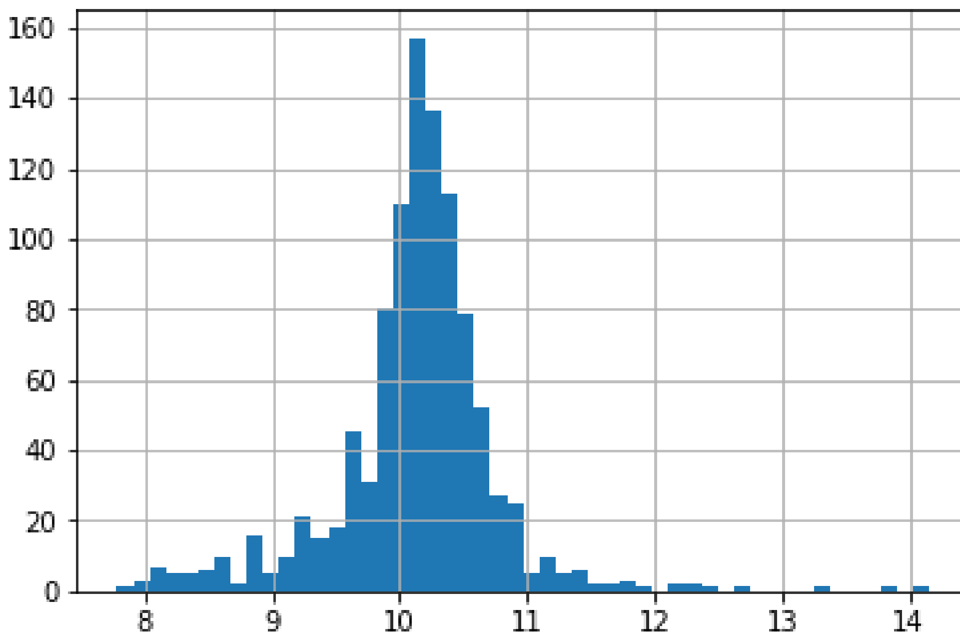
Return type pandas dataframe of shape = [n_samples, n_features]

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['GrLivArea'].hist(bins=50)
```



6.5 Variable discretisation: discretisers

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

6.5.1 EqualFrequencyDiscretiser

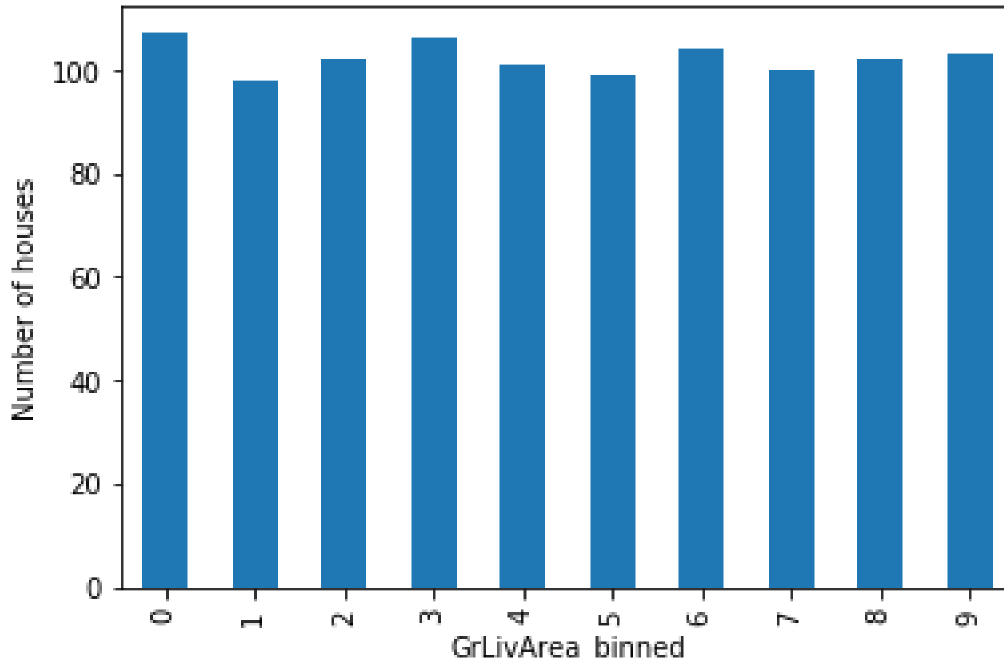
The EqualFrequencyDiscretiser() sorts the variable values into contiguous intervals of equal proportion of observations. The limits of the intervald are calculated according to the quantiles. The number of intervals or quantiles should be determined by the user. The transformer can return the variable 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
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_binned')['GrLivArea'].count().plot.bar()
plt.ylabel('Number of houses')
```



API Reference

class feature_engine.discretisers.**EqualFrequencyDiscretiser**(*q=10*, *variables=None*, *return_object=False*)

The EqualFrequencyDiscretiser() divides the numerical variable values 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() will binnarise only numerical variables. A list of variables can be passed as an argument. But, if no variables are indicated, the discretiser will automatically select and binnarise numerical variables and ignore the rest.

The EqualFrequencyDiscretiser() must be fit to a training set, so that it can first find the boundaries for the intervals / quantiles for each variable (fit()).

The EqualFrequencyDiscretiser() can then transform the variables, that is, sort 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 variable 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 discretised 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.

binner_dict_
The dictionary containing the {interval limits: variable} pairs used to binnarise / discretise variable.

Type dictionary

fit (*self*, *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 seleted variables.
- **y** (*None*) – y is not needed in this encoder, yet the sklearn pipeline API requires this parameter for checking. You can either leave it as None or pass y.

6.5.2 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 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
disc = dsc.EqualWidthDiscretiser(bins=10, variables=['LotArea', 'GrLivArea'])

# fit the transformer
disc.fit(X_train)

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

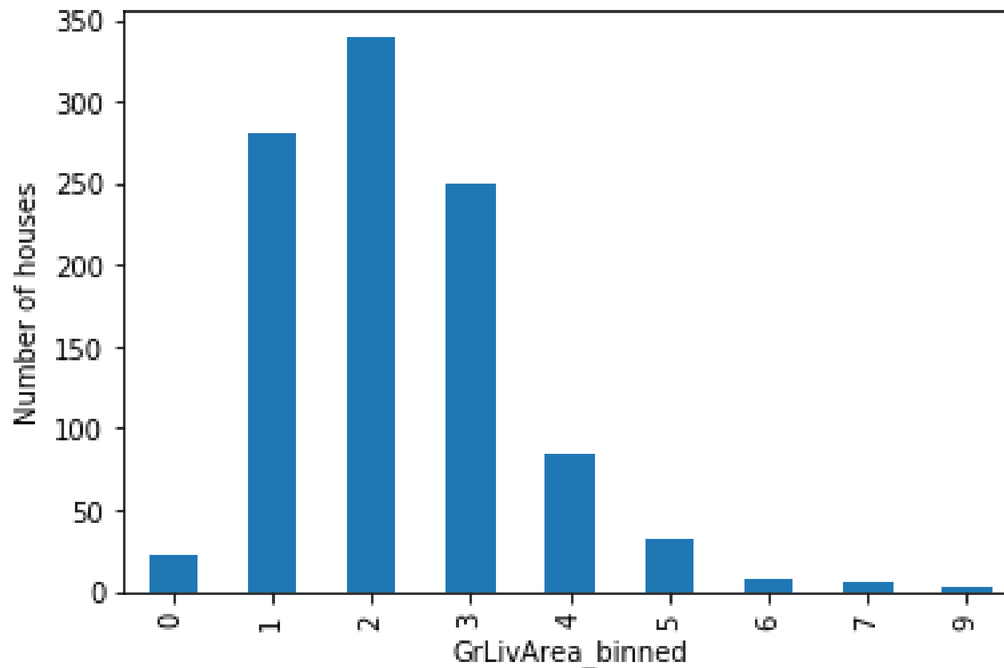
(continues on next page)

(continued from previous page)

```
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_binned')['GrLivArea'].count().plot.bar()
plt.ylabel('Number of houses')
```



API Reference

class feature_engine.discretisers.**EqualWidthDiscretiser**(*bins=10*, *variables=None*, *return_object=False*)

The EqualWidthDiscretiser() divides the numerical variable values 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() will binnarise only numerical variables. A list of variables can be passed as an argument. But, if no variables are indicated, the discretiser will automatically select and binnarise numerical variables and ignore the rest.

The EqualWidthDiscretiser() must be fit to a training set, so that it can first find the boundaries for the intervals for each variable (fit()).

The EqualWidthDiscretiser() can then transform the variables, that is, sort the values into the intervals (transform()).

Parameters

- **bins** (*int, default=10*) – Desired number of equal widht buckets / bins.
- **variables** (*list*) – The list of numerical variables that will be discretised. If None, the discretiser will automatically select all numerical type variables.
- **return_object** (*bool, default=False*) – Whether the numbers in the discretised 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.

binner_dict_

The dictionary containing the {interval boundaries: variable} pairs used to binnarise / discretise each variable.

Type dictionary

fit(*self*, *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 seleted variables.
- **y** (*None*) – y is not needed in this encoder, yet the sklearn pipeline API requires this parameter for checking. You can either leave it as None or pass y.

6.5.3 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 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
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)

disc.binner_dict_
```

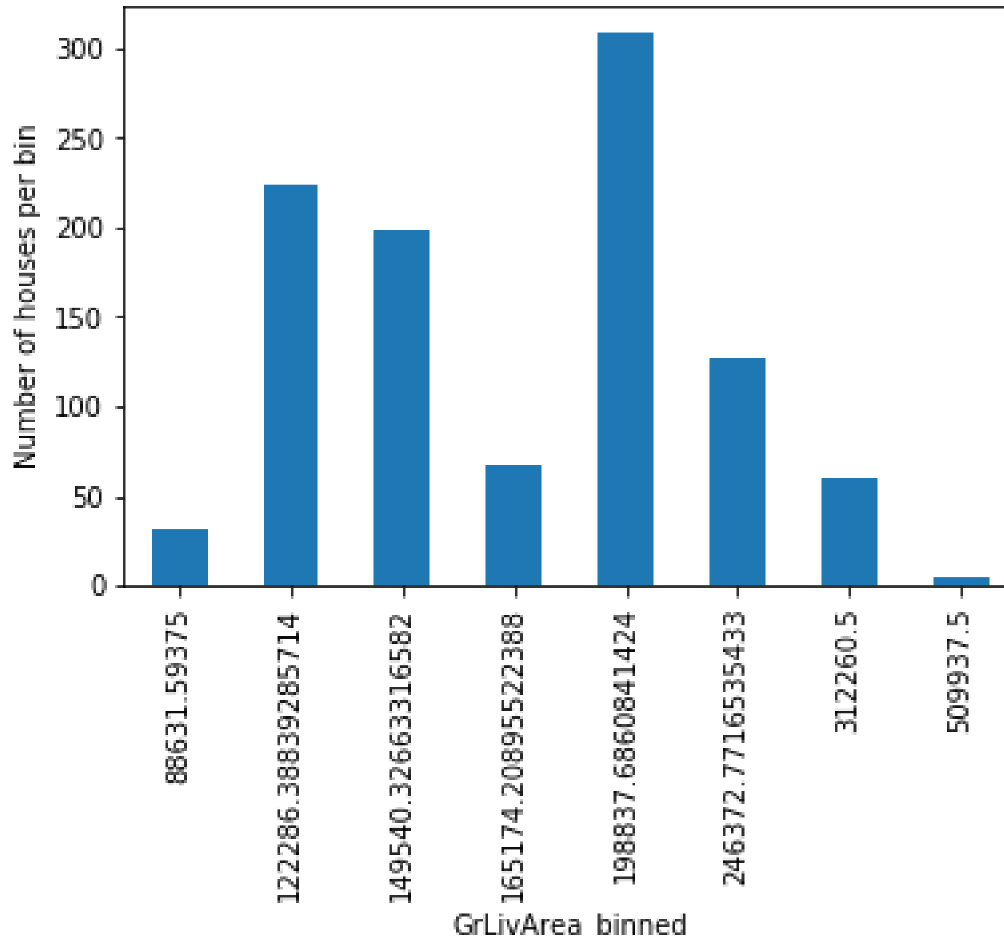
```
{'LotArea': GridSearchCV(cv=3, error_score='raise-deprecating',
              estimator=DecisionTreeRegressor(criterion='mse', max_depth=None,
                                              max_features=None,
                                              max_leaf_nodes=None,
                                              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,
```

(continues on next page)

(continued from previous page)

```
                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_binned')['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 the numerical variable into groups estimated by a decision tree. In other words, the intervals are the predictions made 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, the discretiser only works for binary classification or regression. Multiclass classification is not supported.

The DecisionTreeDiscretiser() will binnarise only numerical variables. A list of variables can be passed as an argument. But, if no variables are indicated, the discretiser will automatically select and binnarise numerical variables and ignore the rest.

The DecisionTreeDiscretiser() must be fit to a training set, so that it can first train a decision tree for each variable (fit()).

The DecisionTreeDiscretiser() can then transform the variables, that is, make 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 for each variable.
- **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 discretised. 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 for the best tree. 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 initilise the training of the decision tree. It is one of the normal parameters of the Scikit-learn's DecisionTreeRegressor() or DecisionTreeClassifier(). For reproducibility it is recommended to set the random_state to an integer.

binner_dict_

The dictionary containing the {fitted tree: variable} pairs, used to transform each variable.

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

fit (*self, X, y*)

Fits the decision tree.

Parameters

- **X** (*pandas dataframe of shape = [n_samples, n_features]*) – The training input samples. Can be the entire dataframe, not just seleted variables.
- **y** (*target variable. Required for this transformer to train the decision*) – tree.

transform(*self*, *X*)
Discretises the variables using the trained tree. That is, returns the predictions of the tree, based of the variable values.

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 Outlier capping: cappers

Feature-engine's outlier cappers cap maximum or minimum values of a variable at an arbitrary or derived value.

6.6.1 Winsorizer

API Reference

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

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. If no list of variable names is passed, the Winsorizer() will find and select all numerical variables seen in the train set.

The Winsorizer() first calculates the capping values at the end of the distribution for the indicated features. The values at the end of the distribution are calculated wither using a Gaussian approximation or the inter-quantile range proximity rule.

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-quantal range: 75th Quantile - 25th Quantile.

You can select to tune how far out to cap your maximum or minimum values by tuning the number by which you multiply the std or the IQR, using the parameter 'fold'.

The transformer first finds the values at one or both tails of the distributions at which it will cap the variables (fit).

The transformer then caps the variables (transform).

Parameters

- **distribution** (*str, default=gaussian*) – Desired distribution. Can take 'gaussian' or 'skewed'. If 'gaussian' the transformer will find the maximum and / or minimum values to cap the variables using the Gaussian approximation. If 'skewed' the transformer will find the boundaries using the IQR proximity rule.
- **end** (*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, default=3*) – How far out to to place the capping value. 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.

- **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.

outlier_capper_dict_
The dictionary containg the values at the end of the distributions to use to cap each variable.

Type dictionary

fit (*self, X, y=None*)
Learns the values that should be used to replace outliers in each variable.

Parameters

- **X** (*pandas dataframe of shape = [n_samples, n_features]*) – The training input samples. Can contain all the variables
- **y** (*None*) – y is not needed in this transformer, yet the sklearn pipeline API requires this parameter for checking.

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

# fit the capper
capper.fit(X_train)

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

capper.right_tail_caps_
```

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

6.6.2 ArbitraryOutlierCapper

API Reference

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

The ArbitraryOutlierCapper() caps the maximum or minimum values of a variable by an arbitrary value indicated by the user.

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

The transformer caps the variables.

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(*self, X, y=None*)

Parameters

- **X** (*pandas dataframe of shape = [n_samples, n_features]*) – The training input samples. Can contain all the variables, not necessarily only those to remove outliers
- **y** (*None*) – y is not needed in this transformer, yet the sklearn pipeline API requires this parameter for checking.

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 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()
```

(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(['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}
```

6.7 Changelog

6.7.1 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

A

AddNaNBinaryImputer (*class in feature_engine.missing_data_imputers*), 30
ArbitraryNumberImputer (*class in feature_engine.missing_data_imputers*), 20
ArbitraryOutlierCapper (*class in feature_engine.outlier_removers*), 64

B

binner_dict_ (*feature_engine.discretisers.DecisionTreeDiscretiser attribute*), 61
binner_dict_ (*feature_engine.discretisers.EqualFrequencyDiscretiser attribute*), 56
binner_dict_ (*feature_engine.discretisers.EqualWidthDiscretiser attribute*), 58
BoxCoxTransformer (*class in feature_engine.variable_transformers*), 49

C

CategoricalVariableImputer (*class in feature_engine.missing_data_imputers*), 24
CountFrequencyCategoricalEncoder (*class in feature_engine.categorical_encoders*), 34

D

DecisionTreeDiscretiser (*class in feature_engine.discretisers*), 60

E

encoder_dict_ (*feature_engine.categorical_encoders.CountFrequencyCategoricalEncoder attribute*), 34
encoder_dict_ (*feature_engine.categorical_encoders.MeanCategoricalEncoder attribute*), 37
encoder_dict_ (*feature_engine.categorical_encoders.OneHotCategoricalEncoder attribute*), 32
encoder_dict_ (*feature_engine.categorical_encoders.OrdinalCategoricalEncoder attribute*), 36
encoder_dict_ (*feature_engine.categorical_encoders.RareLabelCategoricalEncoder attribute*), 41
encoder_dict_ (*feature_engine.categorical_encoders.WoERatioCategoricalEncoder attribute*), 40
EndTailImputer (*class in feature_engine.missing_data_imputers*), 22
EqualFrequencyDiscretiser (*class in feature_engine.discretisers*), 55
EqualWidthDiscretiser (*class in feature_engine.discretisers*), 58

F

fit() (*feature_engine.categorical_encoders.CountFrequencyCategoricalE method*), 34
fit() (*feature_engine.categorical_encoders.MeanCategoricalEncoder method*), 38
fit() (*feature_engine.categorical_encoders.OneHotCategoricalEncoder method*), 32
fit() (*feature_engine.categorical_encoders.OrdinalCategoricalEncoder method*), 36
fit() (*feature_engine.categorical_encoders.RareLabelCategoricalEncode method*), 41
fit() (*feature_engine.categorical_encoders.WoERatioCategoricalEncode method*), 40
fit() (*feature_engine.discretisers.DecisionTreeDiscretiser method*), 61
fit() (*feature_engine.discretisers.EqualFrequencyDiscretiser method*), 56
fit() (*feature_engine.discretisers.EqualWidthDiscretiser method*), 58
fit() (*feature_engine.missing_data_imputers.AddNaNBinaryImputer method*), 30

fit() (*feature_engine.missing_data_imputers.ArbitraryNumberImputer method*), 21
fit() (*feature_engine.missing_data_imputers.CategoricalVariableImputer method*), 24
fit() (*feature_engine.missing_data_imputers.EndTailImputer method*), 23
fit() (*feature_engine.missing_data_imputers.FrequentCategoryImputer method*), 26
fit() (*feature_engine.missing_data_imputers.MeanMedianImputer method*), 19
fit() (*feature_engine.missing_data_imputers.RandomSampleImputer method*), 28
fit() (*feature_engine.outlier_removers.ArbitraryOutlierCapper method*), 64
fit() (*feature_engine.outlier_removers.Winsorizer method*), 63
fit() (*feature_engine.variable_transformers.BoxCoxTransformer method*), 49
fit() (*feature_engine.variable_transformers.LogTransformer method*), 42
fit() (*feature_engine.variable_transformers.PowerTransformer method*), 46
fit() (*feature_engine.variable_transformers.ReciprocalTransformer method*), 44
fit() (*feature_engine.variable_transformers.YeoJohnsonTransformer method*), 51
FrequentCategoryImputer (*class in feature_engine.missing_data_imputers*), 25

I

imputer_dict_ (*feature_engine.missing_data_imputers.EndTailImputer attribute*), 23
imputer_dict_ (*feature_engine.missing_data_imputers.FrequentCategoryImputer attribute*), 25
imputer_dict_ (*feature_engine.missing_data_imputers.MeanMedianImputer attribute*), 19

L

lamda_dict_ (*feature_engine.variable_transformers.BoxCoxTransformer attribute*), 49
lamda_dict_ (*feature_engine.variable_transformers.YeoJohnsonTransformer attribute*), 51
LogTransformer (*class in feature_engine.variable_transformers*), 42

M

MeanCategoricalEncoder (*class in feature_engine.categorical_encoders*), 37
MeanMedianImputer (*class in feature_engine.missing_data_imputers*), 19

O

OneHotCategoricalEncoder (*class in feature_engine.categorical_encoders*), 31
OrdinalCategoricalEncoder (*class in feature_engine.categorical_encoders*), 35
outlier_capper_dict_ (*feature_engine.outlier_removers.Winsorizer attribute*), 63

P

PowerTransformer (*class in feature_engine.variable_transformers*), 46

R

RandomSampleImputer (*class in feature_engine.missing_data_imputers*), 28
RareLabelCategoricalEncoder (*class in feature_engine.categorical_encoders*), 41
ReciprocalTransformer (*class in feature_engine.variable_transformers*), 44

S

scores_dict_ (*feature_engine.discretisers.DecisionTreeDiscretiser attribute*), 61

T

transform() (*feature_engine.categorical_encoders.OneHotCategorical method*), 32
transform() (*feature_engine.categorical_encoders.RareLabelCategoric method*), 41
transform() (*feature_engine.discretisers.DecisionTreeDiscretiser method*), 61
transform() (*feature_engine.missing_data_imputers.AddNaNBinaryIm method*), 30
transform() (*feature_engine.missing_data_imputers.ArbitraryNumberI method*), 21
transform() (*feature_engine.missing_data_imputers.CategoricalVariab method*), 24
transform() (*feature_engine.missing_data_imputers.FrequentCategory method*), 26
transform() (*feature_engine.missing_data_imputers.RandomSampleIm method*), 29
transform() (*feature_engine.variable_transformers.BoxCoxTransforme method*), 49
transform() (*feature_engine.variable_transformers.LogTransformer method*), 42
transform() (*feature_engine.variable_transformers.PowerTransformer method*), 46
transform() (*feature_engine.variable_transformers.ReciprocalTransfor method*), 44
transform() (*feature_engine.variable_transformers.YeoJohnsonTransfo method*), 52

V

variables (*feature_engine.missing_data_imputers.MeanMedianImputer attribute*), 19

W

Winsorizer (*class in feature_engine.outlier_removers*), 62
WoERatioCategoricalEncoder (*class in feature_engine.categorical_encoders*), 39

X

X (*feature_engine.missing_data_imputers.RandomSampleImputer attribute*), 28

Y

YeoJohnsonTransformer (*class in feature_engine.variable_transformers*), 51