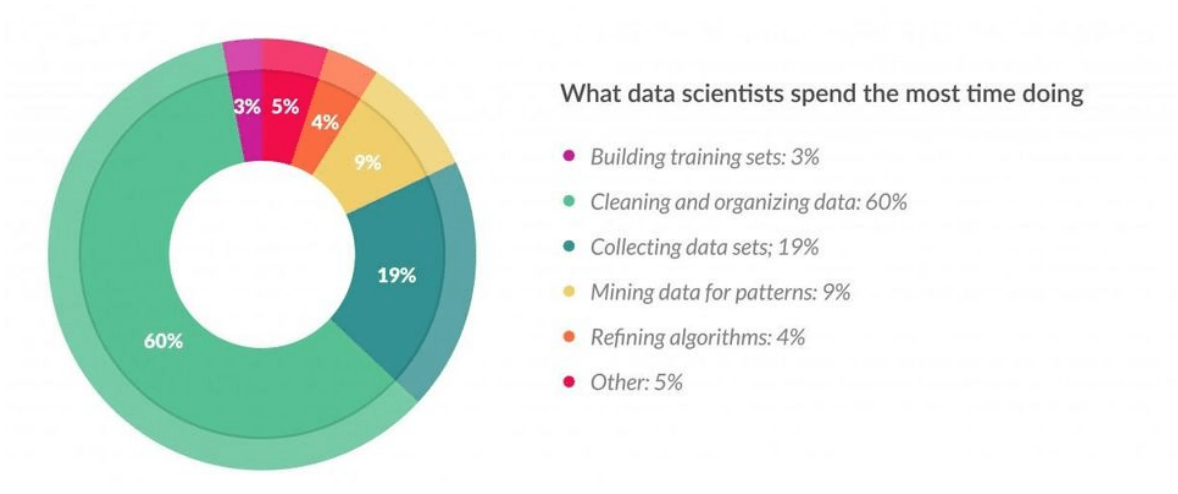
According to a survey by Forbes, data scientists and machine learning engineers spend around **60%** of their time preparing data for analysis and machine learning. A large chunk of that time is spent on **feature engineering**.




*Time data scientists dedicate to different tasks*

Feature engineering is the process of taking a data set and constructing explanatory variables, or predictor features, that are then passed onto the prediction model to train a machine learning algorithm. It is a crucial step in all machine learning models, but can often be challenging and very time consuming.

Feature engineering involves aspects such as imputing missing values, encoding categorical variables, transforming and discretizing numerical variables, removing or censoring outliers and scaling, among others.

In the last few years, a growing number of open source Python libraries that support feature engineering techniques have started to emerge. Among these, the Featuretools library supports an exhaustive array of functions to work with transaction data and time series; the Category encoders library supports a comprehensive selection of methods to encode categorical variables; and the Scikit-learn and Feature-engine libraries support a wide range of transformations including imputation, categorical encoding, discretization, mathematical transformations and more (see Table 1 below for a snapshot summary).

| Feature engineering aspect | Scikit-learn | Feature-engine | Category encoders | Featuretools |
|---|---|---|---|---|
| Missing data imputation | yes | yes | no | no |
| Categorical encoding | yes | yes | yes | no |
| Discretization | yes | yes | no | no |
| Mathematical transformations | yes | yes | no | no |
| Outlier handling | no | yes | no | no |
| Scaling | yes | no | no | no |
| Text | yes | no | no | no |
| Transaction data | no | no | no | yes |
| Time Series | no | no | no | yes |

*Table 1: Feature engineering methods supported by the different Python libraries*

In the following sections of this blog post, we will be covering brief practical code implementations and comparisons of some of the commonly available open-source feature engineering Python packages, including:

- Scikit-learn
- Feature-engine
- Category Encoders

If you are looking for a refresher, our earlier article "Feature Engineering for Machine Learning: A Comprehensive Overview" provides further insights on the basics of each of the individual feature engineering techniques discussed below. For more in depth knowledge or code recipes check out the course "Feature Engineering for Machine Learning" or the book "Python Feature Engineering Cookbook"

Many, if not most, feature engineering techniques learn parameters from the data. For example, to impute data with the mean, we derive the value from the training set. To encode categorical variables, we define mappings of strings to numbers from the training

data as well. Among the mathematical transformations for example, the transformation of BoxCox also needs to learn the optimal exponent to transform data from the train set. As a result, open source Python packages need to have the functionality to first learn and store these parameters, and then retrieve them to transform incoming data. For this exact reason, in this blog, we will be focusing on Scikit-learn, Feature-engine and Category encoders, all of which have this functionality. We will not be discussing libraries such as pandas, which, while containing intrinsic methods to, say, impute missing data ( fillna() ), or map ( map() ) a variable value to some other value, does not contain the functionality to learn and perpetuate the necessary parameters.

# Open Source Libraries for Feature Engineering: General Overview

Scikit-learn, Feature-engine and Category encoders share the fit() and transform() functionality to learn parameters from data, and then transform the variables. The transformers from these libraries, that is, the 'classes' in technical terms, can all store the learned parameters within their attributes. Yet, there are some nuances and subtleties that are different across these packages in terms of i) their output, ii) whether they operate on the entire dataframe or on a slice, and iii) if they allow grid search of engineering methods.

## Outputs: NumPy array vs Pandas dataframe

Feature engineering is performed ahead of training machine learning models. Often, we want to understand how these transformations affect the variable characteristics and their relationships with each other. Pandas is a great tool for data analysis and visualization, and thus, libraries that return pandas dataframes instead of NumPy arrays are inherently more data analysis "friendly".

Feature-engine and Category encoders return pandas dataframes, while Scikit-learn returns NumPy arrays instead. NumPy arrays are optimized for machine learning, as the library is generally more computationally efficient but are less suited for data visualization. The difference in data output type between the packages means that we may need to add

another line of code or two to convert between the NumPy arrays and Pandas dataframes before inserting back into your workflow.

Table 2 below summarizes these key differences between the three packages.

| Transformer characteristics | Scikit-learn | Feature-engine | Category encoders |
|---|---|---|---|
| Output | NumPy array | Pandas dataframe | Pandas dataframe |
| Select variables | no | yes | yes |
| Allows Grid Search | yes | not really | no |

*Table 2: Main differences of the different Python packages for Feature Engineering*

# Data slice vs full data set

Feature engineering techniques are usually applied to different variable subsets. For example, we would only impute variables that contain missing data, and not necessarily the entire data set. Also, there are imputation techniques more suited to numerical variables and those more suited to categorical variables. Similarly, we may also want to discretize a group of variables while transform mathematically another. Thus, the ability to select variables within the feature engineering transformer, or class, allows for an easier flow of the feature engineering pipeline.

Feature-engine and Category encoders allow us to select which variables to transform within the transformer. On the other hand, Scikit-learn transformers will operate over the entire data set (Table 2); meaning we need to slice the dataframe into categorical and numerical or into the variable subgroups to which we will apply each technique before using Scikit-learn transformers, which we can do manually using pandas, or with help of Scikit-learn's ColumnTransformer or Feature-engine's SklearnWrapper.

# Grid Search

Sometimes, we may wonder which transformation technique returns the most predictive variable. For example, should we do equal-width or equal-frequency discretization? Should we impute with the mean, median or an arbitrary number? Should we transform with the logarithm or maybe some other mathematical function?

Most Scikit-learn transformers are centralized, meaning that one transformer, or class, can carry out different transformations. For example, we can apply 3 discretization techniques by simply changing the parameters of the KBinsDiscretizer() class from Scikit-learn, whereas, Feature-engine presents 3 different transformers for discretization. The same

holds true for imputation; by changing the parameters of SimpleImputer(), we can perform different imputation techniques with Scikit-learn, whereas Feature-engine has several transformers, each of which can perform at most 2 different imputation variations.

This adds additional versatility to Scikit-learn transformers through allowing for GridSearch, giving itself an edge on the jack-of-all-trades time sensitive kind of applications such as Data Science competitions and hackathons. On the other hand, Feature-engine transformers offer a more general use-case, oriented more towards mainstream industry usage where their applications are much more streamlined.

Through the rest of the blog, we will compare the implementation of missing data imputation, categorical encoding, mathematical transformation and discretization among Scikit-learn, Feature-engine and Category encoders whenever possible.

# Missing data Imputation

Imputation is the process of replacing missing data in a column, or variable, with a probable value estimated by other available information in the data set, typically within the same variable. There are multiple missing data imputation techniques available, each of which serve different purposes. If you want to learn more about these techniques, their advantages and limitations and when we should use them, check out the course "Feature engineering for Machine Learning". Here, we will compare their implementation with current open source libraries.

Scikit-learn and Feature-engine offer a variety of transformers for data imputation for numerical and categorical variables. Each of these libraries come with their own subtle differences in implementation and output. We will be doing walkthroughs of a few of the imputation methods in the next few paragraphs and discussing a few of those differences below.

As we discussed in the previous section, in terms of the output, Feature-engine returns the imputed data sets as pandas dataframes while Scikit-learn returns NumPy arrays. Now, depending on the stage of your data pre-processing workflow and your personal preferences, you could be dealing with either NumPy arrays or pandas dataframes, requiring you to add a line of code or two to convert between the two.

Table 4 below summarizes the techniques supported by each package and the main takeaways of their advantages and shortcomings.

| | Feature-engine | | | Scikit-learn | | |
|---|---|---|---|---|---|---|
| | Supported | How to do it | Anything else I need to know? | Supported | How to do it | Anything else I need to know? |
| Mean imputation | Yes | MeanMedianImputer(imputation_method='mean') | • Feature-engine will automatically select all numerical variables. | Yes | SimpleImputer(streategy='mean') | • SimpleImputer() will raise an error if there are categorical variables in the dataset. |
| Median imputation | Yes | MeanMedianImputer( imputation_method='median') | • Feature-engine will automatically select all numerical variables. | Yes | SimpleImputer(streategy='median') | • SimpleImputer() will raise an error if there are categorical variables in the dataset. |
| Arbitrary number imputation | Yes | ArbitraryNumberImputer( arbitrary_number=99) | • Feature-engine will automatically select all numerical variables. | Yes | SimpleImputer(streategy='constant', fill_value=99) | • SimpleImputer() will replace missing values with an arbitrary number even in categorical variables. |
| End of tail imputation | Yes | EndTailImputer( distribution='gaussian', tail='right', fold=3) | • Feature-engine will automatically select all numerical variables. | No | - | - |
| Frequent category imputation | Yes | FrequentCategoryImputer() | • Feature-engine will automatically select all categorical variables. | Yes | SimpleImputer( streategy='most_frequent') | • SimpleImputer() will replace missing data by the mode in both numerical and categorical variables. |
| Missing label imputation | Yes | CategoricalVariableImputer() | • Feature-engine will automatically select all categorical variables. | Yes | SimpleImputer(streategy='constant', fill_value='Missing') | • SimpleImputer() will replace missing data with a string even numerical variables, therefore re-casting the variables as object. |
| Random sample imputation | Yes | RandomSampleImputer( random_state=10) | • Can be applied to all variables | No | - | - |
| Adding a missing indicator | Yes | AddNaNBinaryImputer() | • Can be applied to all variables. • Missing indicators are added to the original dataframe with clear variable names | Yes | MissingIndicator() | • MissingIndicator() returns a Numpy array with missing indicators only, without names. • Missing indicators can be also added when using SimpleImputer() if setting parameter add_indicator=True. |
| MICE - Multivariate Imputation of Chained Equations | No | - | - | Yes | IterativeImputer() | |

*Table 3: Comparison of Imputation techniques supported by each Python library*

Both libraries contain functionality for most common imputation techniques. Feature-engine can additionally do **End of tail imputation** and **Random sample imputation**, while Scikit-learn offers **Multivariate imputation of chained equations** in its functionality.

As we mentioned previously, Feature-engine allows us to select the variables that we want to impute within each transformer, whereas Scikit-learn transformers will impute over the entire dataframe. Feature-engine transformers can also automatically identify between numerical and categorical, depending on the imputation technique we would like to apply. That way, we will not end up inadvertently adding a string when we impute numerical variables, or a number to categorical ones. The method for selecting the variables will become clearer when we demonstrate a walkthrough in the next few paragraphs.

Finally, in Table 4, we are reminded that the same Scikit-learn transformer, SimpleImputer(), can perform all imputation techniques just by adjusting the strategy and the fill_value parameters. Thus, giving us the freedom of performing a GridSearch of imputation techniques, as shown for example in the code implementation in Scikit-learn's documentation.

To compare the implementation of both libraries, we will first carry out median imputation followed by imputation by the most frequent category. For median imputation, which is only applicable to numerical variables, the Feature-engine library offers the MeanMedianImputer(), and Scikit-learn offers the SimpleImputer(). During implementation, Feature-engine's MeanMedianImputer() automatically selects all numerical variables in the training data set, leaving out the categorical variables, whereas Scikit-learn's SimpleImputer() transforms all variables in the data set and raises an error if there are categorical variables during the execution. Therefore, the SimpleImputer() may

require an additional step for defining the specific set of numerical variables within the training data as compared to MeanMedianImputer().

The SimpleImputer() class also supports imputing categorical data represented as string values or pandas categorical, using the 'most_frequent' or 'constant' strategy. However, when using either of these imputation strategies, the transformation is automatically applied to both numerical and categorical variables, although they are almost exclusively intended to be used on categorical variables. In those cases, the numerical variables are re-cast as objects, which may not be ideal for our workflow. Feature-engine has a separate CategoricalVariableImputer() transformer, which automatically selects the categorical variables for imputation if they are not specifically declared.

In the next few sections, we will be covering a few of the categorical encoding techniques and walkthroughs.

# Mean/Median Imputation

The 2 walkthroughs below will demonstrate median imputation of numerical variables.

## Feature-engine

In the walkthrough below, we can see the implementation of the MeanMedianImputer() using the median as the imputation method on predicting variables on both the test and train datasets. Mean imputation can be implemented similarly by simply replacing "median" with "mean" for imputation_method.

```
import pandas as pd
from sklearn.model_selection import train_test_split
from feature_engine.missing_data_imputers import MeanMedia

# 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['SalePric
    test_size=0.3, random_state=0)
```

```
# set up the imputer
median_imputer = MeanMedianImputer(imputation_method='medi
  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)
```

Feature-engine returns the original dataframe, where only the applicable variables were modified. For more details visit the MeanMedianImputer() documentation.

## Scikit-learn

Similar to Feature-engine's MeanMedianImputer(), the mean imputation method can also be used by specifying it in the strategy parameter within the SimpleImputer() class.

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

# 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']

# Set up the imputer
median_imputer = SimpleImputer(strategy='median')

# fit the imputer
median_imputer.fit(X_train[['LotFrontage', 'MasVnrArea']])

# transform the data
X_train_t = median_imputer.transform(X_train[['LotFrontage
```

```
X_test_t = median_imputer.transform(X_test[['LotFrontage',
```

As we can see above, Scikit-learn requires that we slice the dataframe before or as we pass it onto the imputation function, whereas this step was not required for its Feature-engine counterpart. The return is a NumPy array with only the sliced data, which in this case is only the 2 numerical variables.

# Frequent Category Imputation

This method applies to categorical variables and replaces missing data with the most frequent category (i.e. the mode), identified in the variable in the training set. The walkthroughs below will demonstrate the most frequent category imputation.

## Feature-engine

The CategoricalVariableImputer() replaces missing data in categorical variables by its mode if we set the imputation_method parameter to 'frequent'. A list of variables can be declared, as is done below; otherwise, the imputer will automatically select all categorical variables in the training data set.

```
import pandas as pd
from sklearn.model_selection import train_test_split
from feature_engine.missing_data_imputers import Categorica

# 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']

# set up the imputer
imputer = CategoricalVariableImputer(imputation_method='fr
  variables=['Alley', 'MasVnrType'])

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

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

## Scikit-learn

The SimpleImputer() class is also used for frequent category imputation by using "most_frequent" as the imputation strategy. The categorical variables must, however, be explicitly declared in this case since the SimpleImputer()'s "most_frequent" imputation strategy will apply to both numerical and categorical variables if left unspecified.

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


# 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']

# set up the imputer
mode_imputer = SimpleImputer(strategy='most_frequent')

# fit the imputer
mode_imputer.fit(X_train[['Alley', 'MasVnrType']])

# transform the data
X_train= mode_imputer.transform(X_train[['Alley', 'MasVnrT
X_test= mode_imputer.transform(X_test[['Alley', 'MasVnrTyp
```

# Categorical Variable Encoding

Machine learning models require input data in a numerical format, which necessitates categorically labelled variables to be converted to numerical values. The method of encoding we choose is completely data context and business problem driven; how we represent and engineer these features could have a major impact on the performance of the model.

Scikit-learn, Feature-engine and Category encoders offer a wide range of categorical label encoders. All three offer the commonly used encoders such as One Hot Encoding and Ordinal Encoding, one that we will be demonstrating below. Feature-engine and Category encoders also offer target-based encoding methods such as target mean encoding and weight of evidence encoding.

Feature-engine maintains its advantage of automatically detecting categorical variables that neither Scikit-learn nor Category encoders are capable of; albeit we could manually define the variables in the transformers for Category encoders, as well as in Feature-engine.

| | Feature-engine | Scikit-learn | Category encoders |
|---|---|---|---|
| Select variable within transformer | yes | no | yes |
| Automatically select categorical variables | yes | no | no |
| Unseen categories | returns a warning | can raise error or ignore | can raise error or ignore |
| inverse transform | no | OrdinalEncoder() yes, OneHotEncoder() no | some transformers yes |

*Table 4: Main advantages and characteristics of the Python open source libraries in terms of categorical encoding techniques*

Overall, Category encoders appear to be the front runners in this field of categorical encoding, offering the widest arsenal of encoding techniques. They were originally derived from a host of scientific publications, developed almost exclusively for categorical data encoding. Supporting both NumPy arrays and pandas dataframes input formats, the Category encoders transformers are fully compatible Scikit-learn functionality and can be used in pipelines in your existing scripts. In addition to the more commonly implemented encoders mentioned above, Category encoders also offer some special use-case encoders including Backward Difference, Helmert, Polynomial and Sum Coding, as well as a handful selection of experimental encoders such as LeaveOneOut, Binary and BaseN.

| | Feature-engine | | Scikit-learn | | Category encoders | |
|---|---|---|---|---|---|---|
| | Supported | Transformer | Supported | Transformer | Supported | Transformer |
| One hot encoding | Yes | OneHotCategoricalEncoder()<br>Drops original variable when replacing by the dummy variables | Yes | OneHotEncoder()<br>Returns dummy variables in a NumPy array | Yes | OneHotEncoder()<br>Drops original variable when replacing by the dummy variables |
| One hot encoding of top categor | Yes | OneHotCategoricalEncoder(top_categories=10) | Yes | OneHotEncoder()<br>Need to pre-select the top categories and include as argument | No | |
| Count or Frequency encoding | Yes | CountFrequencyCategoricalEncoder() | No | | No | |
| Ordinal encoding | Yes | OrdinalCategoricalEncoder(encoding_method='arbitrary') | Yes | OrdinalEncoder() | Yes | OrdinalEncoder) |
| Ordinal Monotonic encoding | Yes | OrdinalCategoricalEncoder(encoding_method='ordered') | No | | Yes | OrdinalEncoder()<br>The mappings need to be pre-defined and passed as argument |
| Target mean encoding | Yes | MeanCategoricalEncoder()<br>Replaces by mean value of target in each category as found in training data | No | | Yes | TargetEncoder()<br>Uses blend of posterior probability of the target given particular categorical value and the prior probability of the target over all the training data. |
| Weight of Evidence | Yes | WoERatioCategoricalEncoder() | No | | Yes | WOEEncoder() |
| Rare Label encoding | Yes | RareLabelCategoricalEncoder() | No | | No | |
| BaseN | No | - | No | | Yes | BaseNEncoder() |
| Binary | No | | No | | Yes | BinaryEncoder() |
| CaatBoostEncoder | No | | No | | Yes | CatBoostEncoder() |
| GLM encoder | No | | No | | Yes | GLMMEncoder() |
| Hashing | No | | No | | Yes | HashingEncoder() |
| Helmet Coding | No | | No | | Yes | HelmertEncoder() |
| James-Stein Encoder | No | | No | | Yes | JamesSteinEncoder() |
| Leave One Out | No | | No | | Yes | LeaveOneOutEncoder() |
| M-estimte | No | | No | | Yes | MEstimateEncoder() |
| Polynomial Coding | No | | No | | Yes | PolynomialEncoder() |
| Sum Coding | No | | No | | Yes | SumEncoder() |

*Table 5: Categorical Encoding Techniques supported by the different Open Source Python Libraries*

# Ordinal Encoding

Also referred to as Label Encoding, Ordinal Encoding numerically labels the categories into the number of unique classes. For example, for a categorical variable with *n* number of unique categories, Ordinal Encoding will replace the categories by numerical digits from *0* to *n-1*.

The inherent characteristic of Ordinal Encoding is that it assumes there is a relationship (some kind of order or hierarchy) between each of the unique classes. Given that Sickit-learn, Feature-engine, and Category encoders all offer ordinal encoding implementation – let's review some of those demonstrations below.

## Feature-engine

Feature-engine's OrdinalCategoricalEncoder() works only with categorical variables, where a list of variables can be indicated, or the encoder will automatically select all categorical variables in the train set. It replaces the categories by numbers, starting from 0 to n-1, where n is the number of different categories. If we select "arbitrary" as the encoding method, then the encoder will assign numbers in the sequence that the labels appear in the variable (i.e. first-come first-served). If "ordered" is selected, the encoder will assign numbers following the mean of the target value for that label. The labels for which the mean of the target is higher will get assigned the number 0, and those where the mean

of the target is smallest will be assigned n-1.

```
import pandas as pd
from sklearn.model_selection import train_test_split
from feature_engine.categorical_encoders import OrdinalCat

# Load dataset
def load_titanic():
 data = pd.read_csv('https://www.openml.org/data/get_csv/1
 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 = OrdinalCategoricalEncoder(encoding_method='arbit
  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)
```

The output of the precedent code block returns the original pandas dataframe where the selected categorical variables were transformed.

## Scikit-learn

Scitkit-learn's OrdinalEncoder() requires the input to be sliced for the categorical variables. During the encoding process, the numbers are simply assigned per the alphabetical order of the labels.

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

# Load dataset
def load_titanic():
 data = pd.read_csv('https://www.openml.org/data/get_csv/1
 data = data.replace('?', np.nan)
 data['cabin'] = data['cabin'].astype(str).str[0]
  data['pclass'] = data['pclass'].astype('O')
 data['embarked'].fillna('C', inplace=True)
 return data

data = load_titanic()

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

# set up the encoder
encoder = OrdinalEncoder()

# fit the encoder
encoder.fit(X_train[['pclass', 'cabin', 'embarked']], y_tra

# transform the data
train_t= encoder.transform(X_train[['pclass', 'cabin', 'emb
test_t= encoder.transform(X_test[['pclass', 'cabin', 'embar
```

The output of the precedent code block is a NumPy array with (only) 3 columns.

## Category encoders

Category encoders' OrdinalEncoder() allows us to specify the variables/columns as a parameter. An optional mapping dict can be passed as well, in cases where we have the knowledge that there is some true order to the classes themselves. Otherwise, the classes are assumed to have no true order and the numbers are assigned to the labels at random.

```
import pandas as pd
from sklearn.model_selection import train_test_split
from category_encoders.ordinal import OrdinalEncoder

# Load dataset
def load_titanic():
 data = pd.read_csv('https://www.openml.org/data/get_csv/1
 data = data.replace('?', np.nan)
 data['cabin'] = data['cabin'].astype(str).str[0]
 data['pclass'] = data['pclass'].astype('O')
 data['embarked'].fillna('C', inplace=True)
 return data

data = load_titanic()

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

# set up the encoder
encoder = OrdinalEncoder(cols=['pclass', 'cabin', 'embarke

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

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

# Transformation

We transform numerical variables with various mathematical transformation functions e.g. logarithmic, linear, power, and reciprocal with a general aim of obtaining a more "Gaussian", or "Normal" looking distribution of the original variable(s).

Scikit-learn offers the FunctionTransformer() which, in principle, can apply any function desired and defined by the user. It takes the function as an argument, either as a NumPy method, or as a lambda function. Through transformers such as LogTransformer() and ReciprocalTransformer(), Feature-engine, instead, supports mathematical transformations with individual specific transformers.

When it comes to "automatic" transformations, both Scikit-learn and Feature-engine packages support Yeo-Johnson and Box Cox transformations. While Scikit-learn centralizes the transformations within the PowerTransformer() just by changing the 'method' argument, Feature-engine has 2 individual Yeo-Johnson and Box Cox transformers.

The usual differences between the two libraries, as we discussed in earlier sections, translates onto transformations as well. Feature-engine outputs a pandas dataframe and automatically selects numerical variables or allows us to declare selected variables, while Scikit-learn applies the transformation to the entire dataframe and returns a NumPy array.

Feature-engine returns an error if a transformation is not mathematically possible, for example log(0), or reciprocal of 0, while Scikit-learn will introduce NaNs instead, necessitating you to do a rationality check afterwards.

# Log Transformer

This transformer applies the natural logarithm or the base 10 logarithm to numerical variables. Log transformers are used to reduce the skewness of data and also help better visualize underlying patterns

## Feature-engine

Feature-engine's LogTransformer() only works with numerical non-negative values. If the variable contains a 0 or a negative value, the transformer will return an error. A list of variables can be passed as an argument, or alternatively, the transformer will automatically select and transform all numerical variables.

```
import pandas as pd
from sklearn.model_selection import train_test_split
from feature_engine.variable_transformers import LogTransf


# 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 = 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)
```

## Scikit-learn

Scikit-learn applies the logarithmic transformation through its FunctionTransformer() by passing the logarithmic function as a NumPy method into the transformer, as shown below.

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

# 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 = FunctionTransformer(np.log)

# fit the transformer
tf.fit(X_train[['LotArea', 'GrLivArea']])

# transform the data
train_t= tf.transform(X_train[['LotArea', 'GrLivArea']])
test_t= tf.transform(X_test[['LotArea', 'GrLivArea']])
```

# Box Cox Transformation

Box Cox transformation is a method of transforming non-normal variables into a normal distribution shape, using a shift, or transformation parameter λ, to find out the best transformation. Normality is an important assumption for many statistical techniques; and if your data isn't normally distributed, applying a Box Cox transformation allows you to run a broader number of tests.

## Feature-engine

The BoxCoxTransformer() applies the Box Cox transformation to numerical variables and works only with non-negative variables. Similar to the other Feature-engine variable transformers, a list of variables can be passed as an argument, or it will automatically select and transform all numerical variables.

```
import pandas as pd
from sklearn.model_selection import train_test_split
from feature_engine.variable_transformers import BoxCoxTra

# 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 = 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)
```

The transformation implemented by this transformer is that of scipy.stats.boxcox and returned as a pandas dataframe.

## Scikit-learn

Scikit-learn offers both Box Cox and Yeo-Johnson transformation through its PowerTransformer(). Box Cox requires the input data to be strictly positive values.

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

# 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 = PowerTransformer(method="box-cox")

# fit the transformer
tf.fit(X_train[['LotArea', 'GrLivArea']])
```

```
# transform the data
train_t= tf.transform(X_train[['LotArea', 'GrLivArea']])
test_t= tf.transform(X_test[['LotArea', 'GrLivArea']])
```

The optimal parameter for stabilizing variance and minimizing the skewness is estimated through maximum likelihood. As with all Scikit-learn transformers, the results are returned as a NumPy array.

# Discretization

Discretization converts, or partitions, continuous numerical variables into discrete variables of contiguous intervals, or bins, that span across the full range of the variable values. Discretization is often implemented to improve the signal to noise ratio for a given variable and reduce the effects of outliers.

| | Feature-engine | Scikit-learn |
|---|---|---|
| output | dataframe | numpy array |
| select variables | yes | no |
| Can encode bins | no, but can be used paired with categorical transformers | yes, one hot encoding of intervals |

*Table 6: Main characteristics of the discretization transformers of Feature-engine and Scikit-learn*

The differences in output type and variable selection methods between the two packages, as we discussed earlier, remain valid for this transformation as well. One of the major differences between Scikit-learn and Feature-engine's discretization offerings lies in the fact that Scikit-learn offers KBinsDiscretizer() as a centralized transformer through which we can do equal-width, equal-frequency, and k-means discretization allowing us to optimize the model through grid search of all techniques; wherein with Feature-engine, we would need to do this manually given that they are offered as separate transformers – EqualFrequencyDiscretiser() and EqualWidthDiscretiser().

Additionally, Scikit-learn allows us to one hot encode the bins straightaway from the transformer, just by setting up the encoding parameter. For Feature-engine, if we wish to treat the bins as categories, we would need to set "return_object=True", and then we can run any of the categorical encoders at the back end of the discretization transformer.

| | Feature-engine | | Scikit-learn | |
|---|---|---|---|---|
| | Supported | Transformer | Supported | How to do it |
| Equal-width discretization | Yes | EqualFrequencyDiscretiser() | Yes | KBinsDiscretizer( strategy='uniform') |
| Equal-frequency discretization | Yes | EqualWidthDiscretiser() | Yes | KBinsDiscretizer( strategy='quantiles') |
| K-means Discretization | No | | Yes | KBinsDiscretizer( strategy='kmeans') |
| Discretization with Decision trees | Yes | DecisionTreeDiscretiser() | No | |

*Table 7: Discretization techniques supported by Scikit-learn and Feature-engine*

# Equal Frequency Discretization

This type of discretization bins variables into a predefined number of contiguous intervals. The bin intervals are normally the percentiles.

## Feature-engine

EqualFrequencyDiscretiser() sorts the numerical variable values into contiguous intervals of equal proportion of observations, where the interval limits are calculated according to the quantiles. This number of intervals, i.e. the number of quantiles in which the variable should be divided, is determined by the user. The transformer can return the variable as either numeric or object (default being numeric).

Inherent to Feature-engine, a list of variables can be indicated, or the discretizer will automatically select all numerical variables in the train set.

```
import pandas as pd
from sklearn.model_selection import train_test_split
from feature_engine.discretisers import EqualFrequencyDisc

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

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

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

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

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

The EqualFrequencyDiscretiser() first finds the boundaries for the intervals or quantiles for each variable as it fits the data. Then it transforms the variables, by sorting the values into the intervals and returns the pandas dataframe.

## Scikit-learn

The Scikit-learn package can implement equal frequency discretization through its KBinsDiscretizer() transformer by setting the "strategy" parameter to "quantiles".

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

# 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 = KBinsDiscretizer(n_bins=10, strategy='quantile')

# fit the transformer
disc.fit(X_train[['LotArea', 'GrLivArea']])

# transform the data
train_t= disc.transform(X_train[['LotArea', 'GrLivArea']])
test_t= disc.transform(X_test[['LotArea', 'GrLivArea']])
```

By default, the NumPy array output is one-hot encoded into a sparse matrix. This can be further configured, such as setting to an ordinal encoding method instead, with the "encode" parameter.

# Outlier Handling

Machine learning algorithms are sensitive to the range and distribution of variable data points where outliers, or anomalies, can deceive the model training process. They can often be the result of measurement/experimental errors or exceptional system conditions and therefore do not relay the statistical characteristic of the underlying system. Therefore, we must treat and manage outliers, by either capping them at maximum or minimum values or by removing them altogether, if using algorithms that are sensitive to them.

Feature-engine exclusively offers outlier handling abilities through Winsorizer() by capping or censoring at maximum or minimum values of a variable at an arbitrary or derived value, and through OutlierTrimmer() by removing the outliers altogether from the data set. It can do so based on Gaussian approximation, the inter-quartile range proximity rule, or percentiles. You can refer to the Python Feature Engineering Cookbook for further in-depth information on outlier handling as well as a host of other information on feature engineering for building machine learning models.

# Text

Alongside the usual go-to library for most Natural Language Processing (NLP) needs – Natural Language Toolkit (NLTK), Scikit-learn also hosts a selected range of a few straightforward feature engineering vectorizers for NLP applications. This includes the TF-IDF vectorizer, HashingVectorizer(), as well as the Bags of Words vectorizer implemented through CountVectorizer(), each of which comes with stop word removal capabilities, adding to the versatility that Scikit-learn brings to feature engineering.

# Transaction Data and Time Series

For relational and transaction data, where there could be multiple transaction records for each specific entry, usually accompanied by a timestamp, Featuretools offers an exclusive robust framework geared towards automatic feature engineering. Featuretools uses Deep Feature Synthesis (DFS) to carefully select relevant data and engineer features automatically at the transaction level, with the ability the add a cut-off time for each time period and adding a secondary time index.

# Final Thoughts

We sure have covered a lot of ground in this article and hopefully were able to put forth a valuable comparison of all the major open source Python libraries for feature engineering. Once you get your hands dirty and try them out as well, you will have a much greater appreciation of the little nuances and advantages each of the packages brings to your pipeline.

Featuretools, Category encoders, Scikit-learn, and Feature-engine – each of these libraries will help you streamline your data preparation pipelines in their own way.

Feature engineering is an essential component in end-to-end data science and machine learning pipelines. It is meant to be an iterative process that every data scientist should master in order to optimize model performance — even simpler ones. It is a computationally expensive and time-consuming portion of your pipeline, and gaining those little efficiencies by knowing the advantages and edges of each package will definitely stack up through your workflow.



OPEN SOURCE

Previous
← Paquetes Python de código abierto para la ingeniería de atributos: Comparaciones y conclusiones

Next
Feature Selection for Machine Learning: A Comprehensive Overview →

## SUBSCRIBE

Email *

Get the latest articles, new releases and demos!

Submit

## CATEGORIES

Select Category

## RECENT POSTS

Feature Selection For Machine Learning: A Comprehensive Overview

Practical Code Implementations Of Feature Engineering For Machine Learning With Python

Paquetes Python De Código Abierto Para La Ingeniería De Atributos: Comparaciones Y Conclusiones

Best Resources To Learn Feature Engineering For Machine Learning

Los Mejores Recursos Para Aprender Ingeniería De Atributos Para Machine Learning

## CATEGORIES

Advanced Resources About Machine Learning

All Posts

Best Resources To Learn About Machine Learning

Español

Open Source

Recommended

## Subscribe

**Email** *

Get the latest articles, new releases and demos!

Submit

Main Website

Privacy