Data Transformation with Ranking Transformation

Machine learning algorithms do not understand strings. Hence, we need to convert the input data into numeric before passing it on to the algorithms for training. We can skip the numeric conversion of the string target variable while doing classification, as it is handled by the algorithms.

The predictor variables could be of two types,

- **Ordinal Variable**: Categorical strings which have some natural ordering, for example, the Size column can be ordered automatically like S<M<L. Or the priority of support tickets like P1>P2>P3 etc. Hence, while converting them to numeric, we must assign such numeric values that represent the natural ordering of the variables. Like S<M<L can be represented by 1<2<3.
- **Nominal Variable**: Categorical strings that do NOT have any natural ordering, for example, Gender, Colors, etc. Here if the number of unique values is two in a variable, then we call it a **Binary variable**, and special treatment is done by replacing values as 0 and 1. When there are more than two unique values, then we create a **dummy variable** for each unique type.

Convert ordinal categorical to numeric

Consider the below data, this contains three categorical string variables, Gender, Department, and Rating. Out of these, Rating is ordinal and the other two are nominal variables.

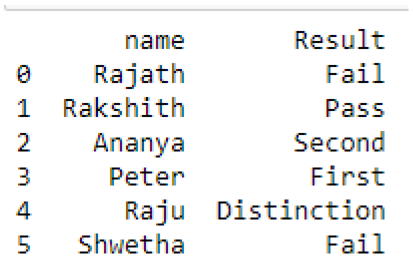
You can convert the ordinal variable to numeric by providing a mapping for each unique value. For example, here we know that Rating-A is better than Rating-B, and Rating-B is better than Rating-C

Hence, 3>2>1 can represent the order A>B>C.

This order must be known to you while converting any ordinal categorical data. Some times it will not be obvious, then you must use your business domain knowledge or consult with a business analyst to confirm it.

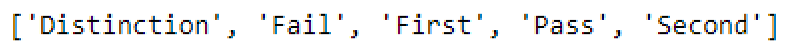
The mapping can be done using the **LabelEncoder() and replace()** function of a Pandas Series.

```
import pandas as pd

from sklearn import preprocessing

data = {"name": ["Rajath", "Rakshith", "Ananya", "Peter", "Raju", "Shwetha"],

        "Result": ["Fail", "Pass", "Second", "First", "Distinction", "Fail"]}

df = pd.DataFrame(data, columns = ["name","Result"])

print(df)
```



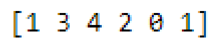
```
       name       Result
0    Rajath         Fail
1  Rakshith         Pass
2    Ananya       Second
3     Peter        First
4      Raju  Distinction
5   Shwetha         Fail
```

```
le = preprocessing.LabelEncoder()

le.fit(df["Result"])

print(); print(list(le.classes_))
```



```
['Distinction', 'Fail', 'First', 'Pass', 'Second']
```

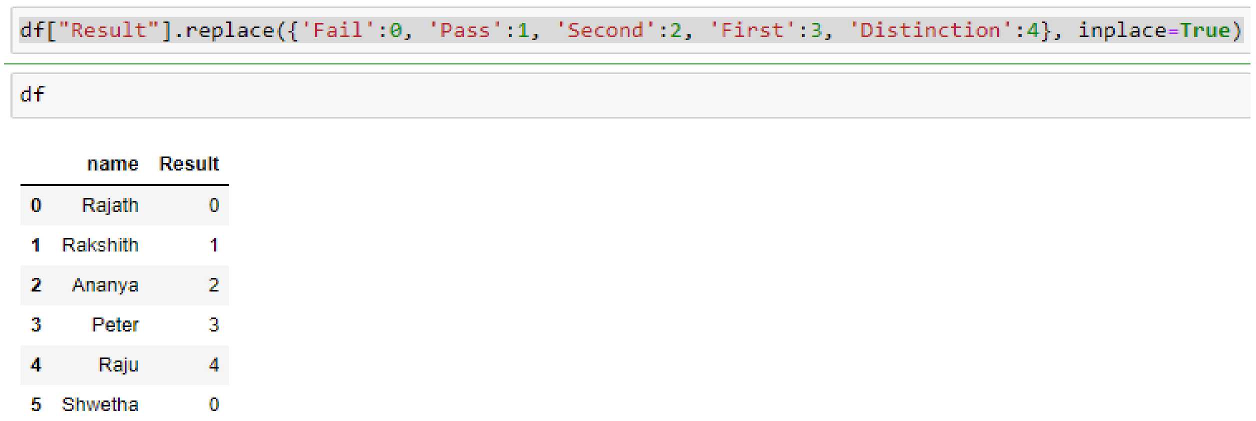
```
print(); print(le.transform(df["Result"]))
```



```
[1 3 4 2 0 1]
```

Now let us do the same operation using **replace()** function.

```
df["Result"].replace({'Fail':0, 'Pass':1, 'Second':2, 'First':3,
'Distinction':4}, inplace=True)
```



```
df["Result"].replace({'Fail':0, 'Pass':1, 'Second':2, 'First':3, 'Distinction':4}, inplace=True)
```

```
df
```

	name	Result
0	Rajath	0
1	Rakshith	1
2	Ananya	2
3	Peter	3
4	Raju	4
5	Shwetha	0

Data Transformation with Discretization

Many machine learning algorithms perform better when they are trained with discrete variables. For that reason, it is common for feature engineering procedures to incorporate discretization. You may find that many of the winning submissions in machine learning competitions leverage this technique.

Don’t be too intimidated by the 5-syllable term. Discretization simply entails transforming continuous values into discrete categories. It’s a common concept in statistics, often referred to as ‘binning’ or ‘bucketing’.

Discretization has numerous merits in machine learning and is easy to execute in Python, as will be explained in detail.

Benefits

The notion of transforming continuous variables into discrete variables may seem unnecessary at a glance, but there are many advantages to including this step in a feature engineering procedure.

1. **Reduces noise**

Continuous variables tend to store information with minute fluctuations that provide no added value for the machine learning task of interest. Dealing with such values will lead a model to account for a lot of noise, which inevitably hampers performance.

Discretization resolves this issue by enabling users to limit this noise by creating features with distributions comprising fewer unique values.

2. Provides intuitive features

Depending on the use case, discrete variables can be more intuitive than continuous variables.

For example, if one were to build a causal model that compares young and old people, it might be worth recording age as a discrete variable (e.g., “child”, “adult”, and “elderly”) instead of a continuous variable.

This would make the results of the model much easier to interpret since the feature is customized for the application.

3. Minimizes outlier influence

Finally, discretization mitigates the influence of outliers. Regardless of how much an outlier skews a distribution of values, it will be converted into the upper or lower groups after the transformation.

Drawback

Despite its many benefits, discretization has one noteworthy disadvantage: information loss.

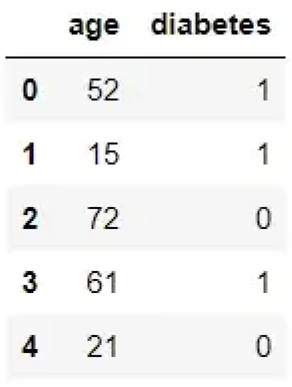
This is, of course, a characteristic of many transformations in feature engineering. Techniques like normalization and principal component analysis (PCA) will naturally incur some information loss. Overall, the information loss in discretization won’t necessarily be substantial, but it’s worth keeping in mind when considering how many discrete values should be present after the transformation.

Types of Discretization

There are a number of approaches for executing discretization that can be implemented in Python. The two most prominent Python packages that can facilitate this transformation are: scikit-learn and feature_engine.

To showcase some of the transformers provided by these packages, let’s work with the following made-up data, composed of the predictive feature “age” and the target label “diabetes”.

```
import pandas as pd
import numpy as np
# create fake data

age = [np.random.randint(1,80) for _ in range(100)]
df = pd.DataFrame(age, columns=['age'])

df['diabetes'] = [np.random.randint(0,2) for _ in range(100)]
X = pd.DataFrame(df['age'])
y = df['diabetes']
df.head()
```



	age	diabetes
0	52	1
1	15	1
2	72	0
3	61	1
4	21	0

The goal will be to transform the values in the age feature into 5 discrete groups using the various discretization methods available.

We will use the following functions to visualize the transformation and highlight the features of each method.

```
from collections import Counter
import matplotlib.pyplot as plt
def plot_bins(values):

    """visualizes the number of observations in each bin"""

    counter = Counter(values)
    plt.bar(counter.keys(), counter.values())
    plt.xlabel('Bins')
    plt.ylabel('Frequency')
    plt.show()


def show_bins(bins):
    """Shows the ranges of values the bins comprises"""

    for i in range(len(bins)-1):
        print(f'Bin {i}: {str(bins[i])} -> {str(bins[i+1])}')
```

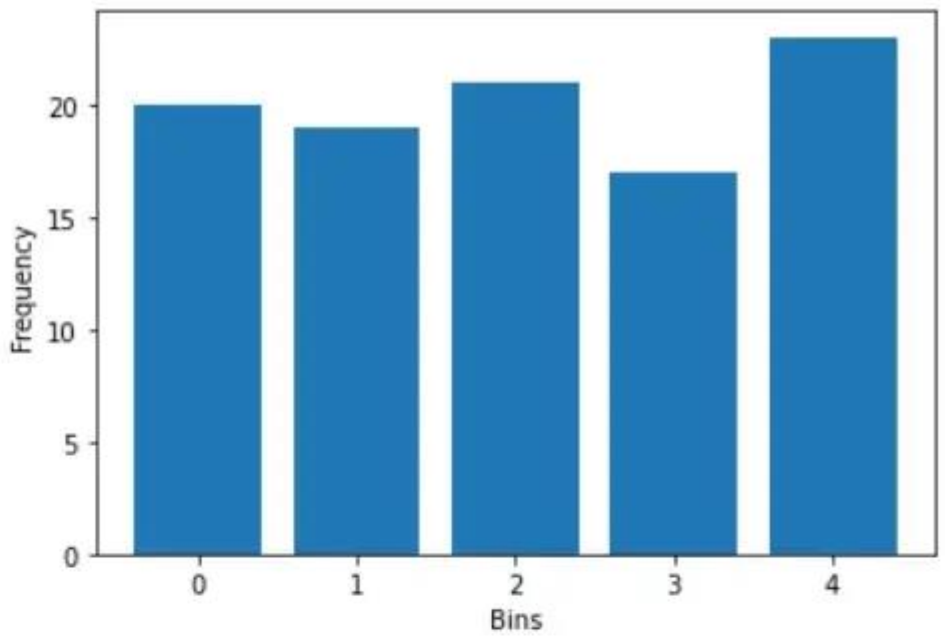
1. Equal-Frequency Discretization

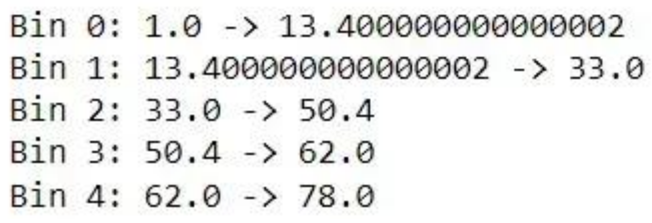
Equal frequency discretization entails transforming continuous data into bins, with each bin having the same (or similar) number of records.

To carry out this method in Python, we can use the scikit-learn package's KBinsDiscretizer, where the strategy hyperparameter is set to 'quantile'.

```
from sklearn.preprocessing import KBinsDiscretizer
# create discretizer

kbins      =      KBinsDiscretizer(n_bins=5,      strategy='quantile',
encode='ordinal')
age_bin = kbins.fit_transform(np.array(df['age']).reshape(-1,1))
```

```
# show results
plot_bins(Counter(list(age_bin.flatten())))
show_bins(kbins.bin_edges_[0])
```





```
Bin 0: 1.0 -> 13.400000000000002
Bin 1: 13.400000000000002 -> 33.0
Bin 2: 33.0 -> 50.4
Bin 3: 50.4 -> 62.0
Bin 4: 62.0 -> 78.0
```

The 5 bins are relatively close in terms of the number of observations. However, in order to attain this uniformity, the width of each bin has to be uneven.

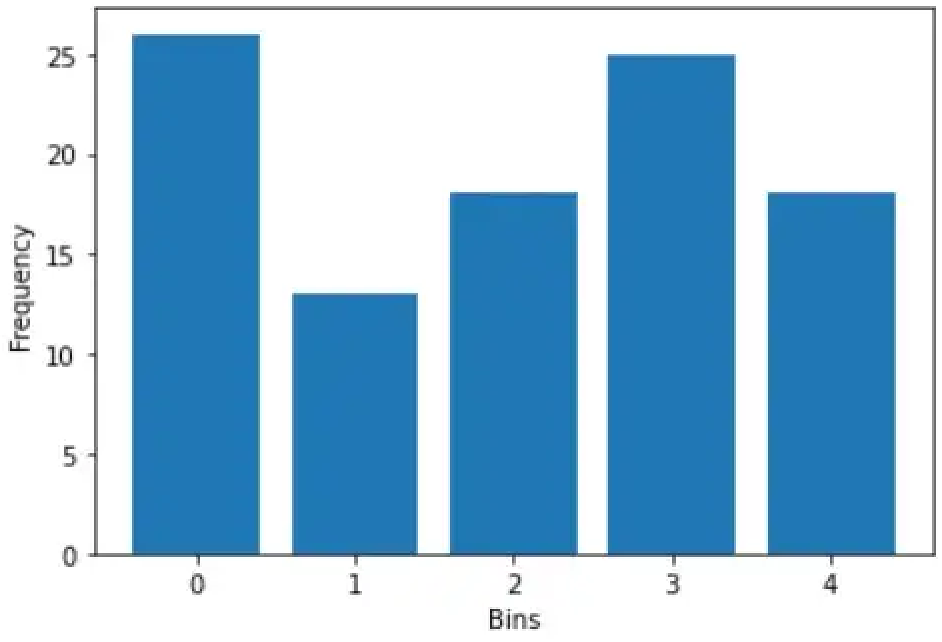
We can also execute this transformation with the EqualFrequencyDiscretiser in the feature_engine package.

2. Equal-Width Discretization

As the name suggests, the equal-width discretization transforms data into bins with the same width. Much like the equal-frequency discretization, we can use this technique with the sci-kit learn package's KBinsDiscretizer. However, the strategy hyperparameter should be set to 'uniform'.

```
from sklearn.preprocessing import KBinsDiscretizer
# create discretizer
```

```
kbins = KBinsDiscretizer(n_bins=5, strategy='uniform', encode='ordinal')
age_bin = kbins.fit_transform(np.array(df['age']).reshape(-1,1))


# show results
plot_bins(Counter(list(age_bin.flatten())))
show_bins(kbins.bin_edges_[0])
```



```
Bin 0: 1.0 -> 16.4
Bin 1: 16.4 -> 31.8
Bin 2: 31.8 -> 47.2
Bin 3: 47.2 -> 62.6
Bin 4: 62.6 -> 78.0
```

As shown by the output, all of the created bins have a width of 15.4.

We can also execute this transformation by using the feature_engine package’s EqualWidthDiscretiser.

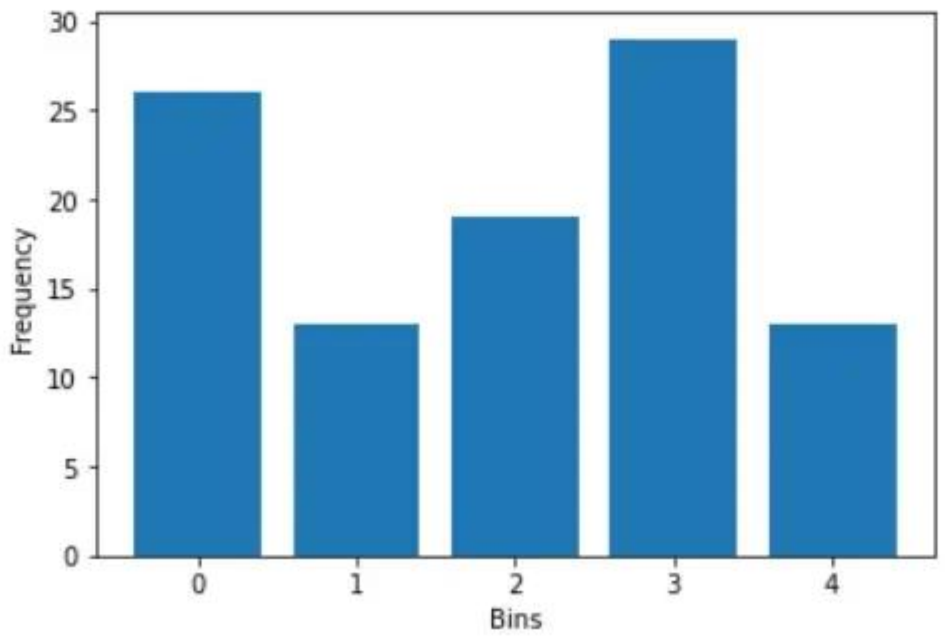
3. K-means Discretization

The k-means discretization entails using the k-means clustering algorithm to assign data points to bins.

It can also be executed with the scikit-learn package’s KBinsDiscretizer, where the strategy hyperparameter is set to ‘kmeans’.

```
from sklearn.preprocessing import KBinsDiscretizer
```

```
# create discretizer

kbins = KBinsDiscretizer(n_bins=5, strategy='kmeans',
encode='ordinal')
age_bin = kbins.fit_transform(np.array(df['age']).reshape(-1,1))

# show results
plot_bins(Counter(list(age_bin.flatten())))
show_bins(kbins.bin_edges_[0])
```



```
Bin 0: 1.0 -> 15.403846153846146
Bin 1: 15.403846153846146 -> 31.61336032388664
Bin 2: 31.61336032388664 -> 48.92105263157895
Bin 3: 48.92105263157895 -> 65.92307692307693
Bin 4: 65.92307692307693 -> 78.0
```

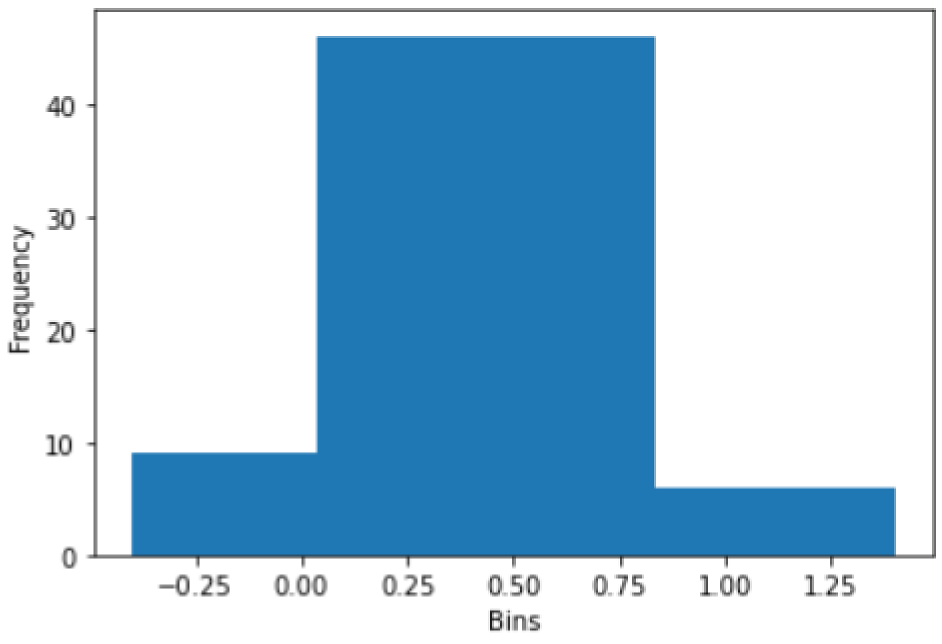
4. Decision Tree Discretization

The decision tree discretization is different from the previous methods in that it is a *supervised* learning technique, meaning that it requires the use of a target label to transform continuous variables.

As suggested by the name, it uses the decision tree algorithm to find the ideal cut-off points to segment the observations.

We can carry out this technique with the DecisionTreeDiscretiser in the feature_engine package.

```
from feature_engine.discretisation import DecisionTreeDiscretiser
# create discretizer

dtd = DecisionTreeDiscretiser(random_state=42, variables=['age'],
scoring='f1_micro',regression=False)
age_bin = dtd.fit_transform(X, y)
# show results
plot_bins(age_bin['age'])
```



5. Custom Discretization

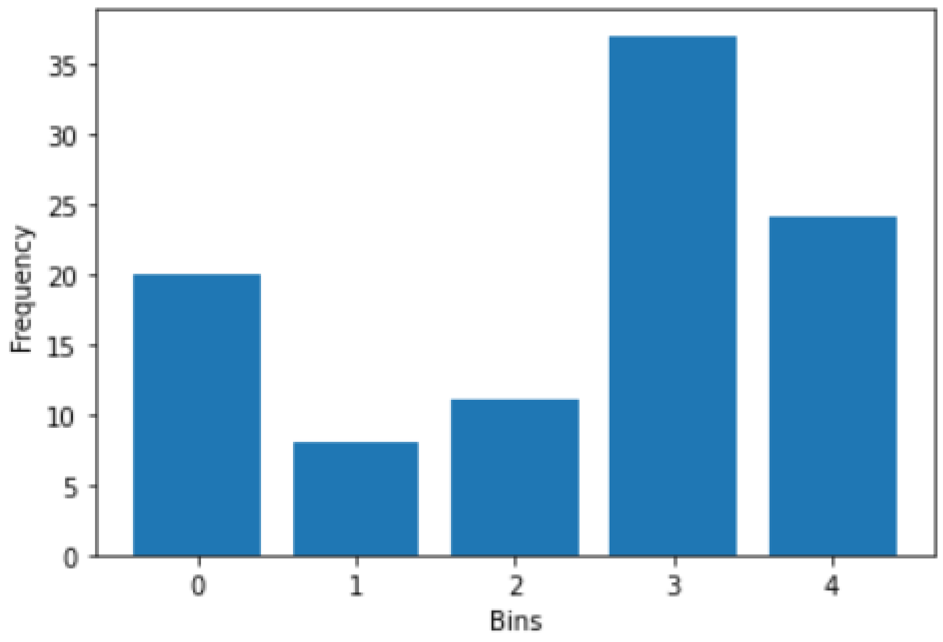
Finally, users have the option to transform their continuous variables into discrete values based on custom rules. Users with certain domain knowledge might benefit from creating bins with predefined widths.

In our example, suppose we wish to transform the age feature into predefined groups composing the age groups “1–12”, “13–18”, “19–30”, “31–60”, and “60–80”.

We can achieve this in Python with the feature_engine package’s ArbitraryDiscretiser.

```
from feature_engine.discretisation import ArbitraryDiscretiser
# create discretizer

age_dict = {'age': [0, 12, 18, 30, 60, np.Inf]}
ad = ArbitraryDiscretiser(binning_dict=age_dict)
```

```
age_bin = ad.fit_transform(X)
# show results
plot_bins(age_bin['age'])
show_bins(ad.binner_dict_['age'])
```



```
Bin 0: 0 -> 12
Bin 1: 12 -> 18
Bin 2: 18 -> 30
Bin 3: 30 -> 60
Bin 4: 60 -> inf
```