Introduction to Data Science

Categorical Feature Encoding

Muhammad Affan Alim

Encoding Categorical data

- Categorical variables are those values which are selected from a group of categories or labels.
- For example, the variable **Gender** with the values of **male** or **female** is categorical
- The variable **marital status** with the values of **never married**, **married**, **divorced** or **widowed**

Cardinal, Ordinal and Nominal Numbers

Type of categorical variables

1. Cardinal: how many
2. Ordinal: position
3. Nominal: name

Cardinal Numbers

- A Cardinal Number says how many of something, such as one, two, three, four, five, etc.
- Example: here are **five** coins:

- It does not have fractions or decimals, it is only used for counting.
- How to remember: "Cardinal is Counting"

Ordinal Numbers

- An Ordinal Number tells us the **position** of something in a list.
- **1st, 2nd, 3rd, 4th, 5th** and so on
- Example: In this picture the girl is 2nd:

- How to remember: "Ordinal says what Order things are in".

Nominal Numbers

- A Nominal Number is a number used only as a **name**, or to identify something (not as an actual value or position)

- Examples:

- the number on the back of a player: "8"
- a zip code: "91210"
- a model number: "380“

- How to remember: "Nominal is a Name".

Encoding Techniques

Following are the encoding techniques

- Creating binary variables through one-hot encoding
- Performing one-hot encoding of frequent categories
- Replacing categories with ordinal numbers
- Replacing categories with counts or frequency of observations
- Encoding with integers in an ordered manner
- Encoding with the mean of the target

Encoding Techniques cont…

Following are the encoding techniques

- Encoding with the Weight of Evidence
- Grouping rare or infrequent categories
- Performing binary encoding
- Performing feature hashing

Encoding Techniques cont…

- We will use the Credit Approval Dataset available in the UCI Machine Learning
- Repository, available at https:/ / archive. ics. uci. edu/ ml/ datasets/ credit+approval

Encoding Categorical Data

- In one-hot encoding, we represent a categorical variable as a group of binary variables, where each binary variable represents one category.
- The binary variable indicates whether the category is present in an observation (1) or not (0).

Encoding Categorical Data

- The following table shows the onehot encoded representation of the **Gender** variable with the categories of **Male** and **Female:**

Gender	Female	Male
Female	1	0
Male	0	1
Male	0	1
Female	1	0
Female	1	0

- Make sure that you have **already imputed missing data** with any of the recipes from

Encoding Categorical data

- Dataset contains some categorical data in qualitative nature
- It is easily understandable by human but encode these into numeric values for machine learning model
- Example
- Color, place of birth, fruit etc.

Index	Country
1	Pak
2	US
3	Pak
4	UK
5	Us
6	Singapore
7	UK

Encoding Categorical data

- these categories are categorical string and it is required to assign a numeric number like pak= 1, US = 2 etc.

```
>> dataset['Origin'] = dataset['Origin'].map({‘Pak’:1, 'US' :2, ‘UK‘: 3})
>> dataset['Origin'].replace([‘Pak’, ‘US’, ‘UK’], [0, 1, 2], inplace=True)
```

Index	Country	encoding
1	Pak	1
2	US	2:
3	Pak	1
4	UK	3
5	US	2
6	Singapore	4
7	UK	3

Encoding Categorical data

- values can encode the additional binary features corresponding the each value respect or not
- Example

Index	Country
1	' Pak '
2	'USA'
3	'UK'
4	'UK'
5	'France'
…	…

Index	C_ Pak	C_USA	C_UK	C_France
1	1	0	0	0
2	0	1	0	0
3	0	0	1	0
4	0	0	1	0
5	0	0	0	1
…	…	…	…	…

Encoding Categorical data

- In doing so your model can leverage the information that what country is given without inference any order between the different options
- There are two main methods for encoding techniques
  1. One-Hot encoding: *n*-category with *n*-feature
  2. Dummy encoding: *n*-category with (*n*-1)-feature

Encoding Categorical data

- Both models create huge number of columns being created
- Example:

One-hot encoding

```
dfNew1 = pd.get_dummies(dfNew,columns=['Embarked'], prefix='E')
```

Dummy encoding

```
dfNew2 = pd.get_dummies(dfNew,columns=['Embarked'], drop_first =
True, prefix = 'E')
```

Replacing categories with ordinal numbers

Replacing categories with ordinal numbers

- Ordinal encoding consists of replacing the categories with digits from 1 to *k* (or 0 to *k-1*, depending on the implementation), where *k* is the number of distinct categories of the variable.
- The numbers are assigned arbitrarily.
- Ordinal encoding is better suited for nonlinear machine learning models, which can navigate through the arbitrarily assigned digits to try and find patterns that relate to the target

Replacing categories with counts or frequency of observations

- In count or frequency encoding, we replace the categories with the count or the percentage of observations with that category
- That is, if 10 out of 100 observations show the category **blue** for the variable **color**, we would replace **blue** with 10 when doing count encoding, or by 0.1 if performing frequency encoding

Replacing categories with counts or frequency of observations

- Note that if two different categories are present in the same percentage of observations, they will be replaced by the same value, which may lead to information loss.

Replacing categories with counts or frequency of observations

```
>> data = pd.read_csv('./dataset/creditApprovalUCI_new.csv')
>> count_map = data['A7'].value_counts().to_dict()
>> count_map
```

{'v': 399, 'h': 138, 'bb': 59, 'ff': 57, 'Missing': 9, 'j': 8, 'z': 8, 'dd': 6, 'n': 4, 'o': 2}

```
>> data['A7'] = data['A7'].map(count_map)
```

{399: 399, 138: 138, 59: 59, 57: 57, 8: 16, 9: 9, 6: 6, 4: 4, 2: 2}

Limit the columns (replace the less repeated value)

- For categorical columns, the labels with low frequencies probably affect the robustness of statistical models negatively.
- Thus, assigning a general category to these less frequent values helps to keep the robustness of the model.
- For example, if your data size is 100,000 rows, it might be a good option to unite the labels with a count less than 100 to a new category like “Other”.

Limit the columns (replace the less repeated value)

- **Limit your columns**
- What values to included? First creating the mask of the values which is less than n times in values.

```
>> Counts = dfNew['Embarked'].value_counts()
```

Limit the columns (replace the less repeated value)

First create the mask

```
>> mask = dfNew['Embarked'].isin(counts[counts<170].index)
>> dfNew['Embarked'][mask] = 'other'
>> print(pd.value_count(dfNew['Embarked']))
```

Transform Numerical Variables

- Linear and logistic regression assume that the variables are normally distributed.
- If they are not, we can often apply a mathematical transformation to change their distribution into Gaussian, and sometimes even unmask linear relationships between variables and their targets

Transform Numerical Variables

- Commonly used mathematical transformations include the logarithm, reciprocal, power, square and cube root transformations, as well as the Box-Cox and Yeo-Johnson transformations.

Log Transform

- Logarithm transformation (or log transform) is one of the most commonly used mathematical transformations in feature engineering. What are the benefits of log transform:

i. It helps to handle skewed data and after transformation, the distribution becomes more approximate to normal

ii. It also decreases the effect of the outliers, due to the normalization of magnitude differences and the model become more robust

Log Transform cont…

iii. In most of the cases the magnitude order of the data changes within the range of the data. For instance, the difference between ages 15 and 20 is not equal to the ages 65 and 70. In terms of years, yes, they are identical, but for all other aspects, 5 years of difference in young ages mean a higher magnitude difference. This type of data comes from a multiplicative process and log transform normalizes the magnitude differences like that

Log Transform cont…

A critical note:

The data you apply log transform must have only positive values, otherwise you receive an error. Also, you can add 1 to your data before transform it. Thus, you ensure the output of the transformation to be positive.

Log Transform example-1

```
#Log Transform Example
data = pd.DataFrame({'value':[2,45, -23, 85, 28, 2, 35, -12]})

data['log+1'] = (data['value']+1).transform(np.log)

#Negative Values Handling
#Note that the values are different
data['log'] = (data['value']-data['value'].min()+1) .transform(np.log)
```

	value	log(x+1)	log(x-min(x)+1)
0	2	1.09861	3.25810
1	45	3.82864	4.23411
2	-23	nan	0.00000
3	85	4.45435	4.69135
4	28	3.36730	3.95124
5	2	1.09861	3.25810
6	35	3.58352	4.07754
7	-12	nan	2.48491

Log Transform example -2

1. Import the required Python libraries, classes, and functions:

```
>> import numpy as np
>> import pandas as pd
>> import matplotlib.pyplot as plt
>> import scipy.stats as stats
>> from sklearn.datasets import load_boston
>> from sklearn.preprocessing import FunctionTransformer
>> from feature_engine.variable_transformers import LogTransformer
```

Log Transform example -2

2. Let's load the Boston House Prices dataset into a pandas dataframe:

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

Log Transform example -2

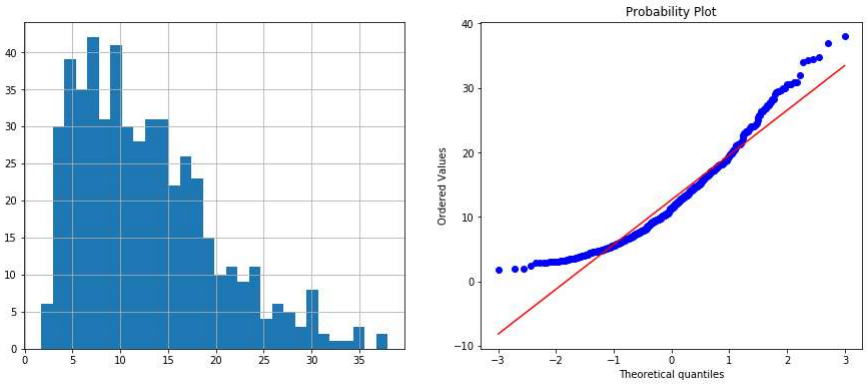
- **To evaluate the effect of the transformation on the variable distribution, we'll create a function that takes a dataframe and a variable name as inputs and plots a histogram next to a Q-Q plot:**

```
>> def diagnostic_plots(df, variable):
>>     plt.figure(figsize=(15,6))
>>     plt.subplot(1, 2, 1)
>>     df[variable].hist(bins=30)
>>     plt.subplot(1, 2, 2)
>>     stats.probplot(df[variable], dist="norm", plot=plt)
>>     plt.show()
```

Log Transform example -2

4. Now, let's plot the distribution of the LSTAT variable:

```
>> diagnostic_plots(data, 'LSTAT')
```



Log Transform example -2

- **First, let's make a copy of the original dataframe using pandas copy():**

```
>> data_tf = data.copy()
```

- We've created a copy so that we can modify the values in the copy and not in the original dataframe, which we need for the rest of the recipe.

Log Transform example -2

6. **Let's apply the logarithmic transformation with NumPy to a subset of positive variables to capture the transformed variables in the new dataframe:**

```
>>  data_tf[['LSTAT', 'NOX', 'DIS', 'RM']] = np.log(data[['LSTAT',
    'NOX', 'DIS', 'RM']])
```

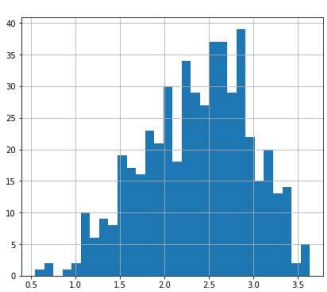
Log Transform example -2

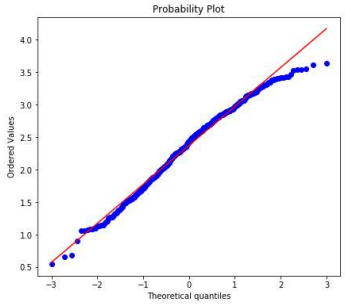
7. Let's check the distribution of LSTAT after the transformation with the diagnostic function we created in step 3:

>> diagnostic_plots(data_tf, 'LSTAT')

We can see the effect of the transformation in the following output:

Now, let's





Performing Variable Discretization

- Discretization, or binning, is the process of transforming continuous variables into discrete variables by creating a set of contiguous intervals, also called bins.
- Discretization is used to change the distribution of skewed variables and to minimize the influence of outliers, and hence improve the performance of some machine learning models.

Performing Variable Discretization

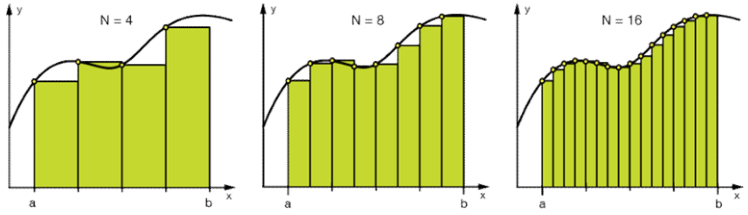
- Binning can be applied on both categorical and numerical data:

```
#Numerical Binning Example

Value      Bin
0-30   ->  Low
31-70  ->  Mid
71-100 ->  High

#Categorical Binning Example

Value      Bin
Spain  ->  Europe
Italy  ->  Europe
Chile  ->  South America
Brazil ->  South America
```



Dividing the variable into intervals of equal width

- In equal-width discretization, the variable values are sorted into intervals of the same width.
- The number of intervals is decided arbitrarily and the width is determined by the range of values of the variable and the number of bins to create, so for the variable X, the interval width is given as follows

$$Width = \frac{Max(X) - Min(X)}{Bins}$$

Dividing the variable into intervals of equal width

```
>> dfS['RM_bin_2'] = pd.cut(dfS['RM'], 4 , labels=[1,2,3,4] ,
   include_lowest=True) # fix bining

>> dfS['RM_bin_3'] = pd.cut(dfS['RM'], [2,3.8,5.2,6.9,8.8],
   labels=[1,2,3,4] , include_lowest=True) # Vriable bining

>>
```

Dividing the variable with equal count into unequal intervals

```
>> dfS['RM_bin_2'] = pd.qcut(dfS['RM'], 5, labels=[0,1,2,3,4])
```

Scaling

- In most cases, the numerical features of the dataset do not have a certain range and they differ from each other
- In real life, it is nonsense to expect age and income columns to have the same range
- But from the machine learning point of view, how these two columns can be compared?

Scaling

- Scaling solves this problem. The continuous features become identical in terms of the range, after a scaling process
- This process is not mandatory for many algorithms, but it might be still nice to apply. However, the algorithms based on distance calculations such as k-NN, k-Means or PCA need to have scaled continuous features as model input.

Scaling

- Basically, there are two common ways of scaling:

i. Normalization

ii. Standardization

Scaling - Normalization

- Normalization (or min-max normalization) scale all values in a fixed range between 0 and 1.
- This transformation does not change the distribution of the feature and due to the decreased standard deviations, the effects of the outliers increases.
- Therefore, before normalization, it is recommended to handle the outliers.

$$X_{norm} = \frac{X - X_{min}}{X_{max} - X_{min}}$$

Scaling - Normalization

```
>> data = pd.DataFrame({'value':[2,45, -23, 85, 28, 2, 35, -12]})

>> data['normalized'] = (data['value'] - data['value'].min()) /
    (data['value'].max() - data['value'].min())
```

	value	normalized
0	2	0.23
1	45	0.63
2	-23	0.00
3	85	1.00
4	28	0.47
5	2	0.23
6	35	0.54
7	-12	0.10

Scaling - Normalization

```
>> from sklearn.preprocessing import MinMaxScaler
>> data = [[-1, 2], [-0.5, 6], [0, 10], [1, 18]]
>> scaler = MinMaxScaler()
>> print(scaler.fit(data))
MinMaxScaler()    # output
>> print(scaler.data_max_)
[ 1. 18.]     # output
>>> print(scaler.transform(data))
[[0.   0.  ]
 [0.25 0.25]
 [0.5  0.5 ]
 [1.   1.  ]]
>>> print(scaler.transform([[2, 2]]))
[[1.5 0. ]]
```

Scaling-Standardization

- Standardization (or z-score normalization) scales the values while taking into account standard deviation.
- If the standard deviation of features is different, their range also would differ from each other.
- This reduces the effect of the outliers in the features.
- In the following formula of standardization, the **mean** is shown as μ and the **standard deviation** is shown as σ.

$$z = \frac{x - \mu}{\sigma}$$

Scaling-Standardization

```
>> data = pd.DataFrame({'value':[2,45, -23, 85, 28, 2, 35, -12]})
>> data['standardized'] = (data['value'] - data['value'].mean()) /
   data['value'].std()
```

	value	standardized
0	2	-0.52
1	45	0.70
2	-23	-1.23
3	85	1.84
4	28	0.22
5	2	-0.52
6	35	0.42
7	-12	-0.92

Scaling-Standardization

```
>> from sklearn.preprocessing import StandardScaler
>> ss = StandardScaler()
>> tips_ds_scaled = ss.fit_transform(tips_ds_numeric)
```