Sumo + Akki

Feature Engineering

Manoj C. Patil

Kavayitri Bahinabai Chaudhari North Maharashtra University, Jalgaon

March 10, 2023

Outline

Feature Scaling

- Feature scaling is a technique used to standardize the range of features or variables.
- Most algorithms use Euclidean distance between two data points in their computations, and if the features have different scales, then the distance will be dominated by the feature with the largest scale.
- It can improve the performance of ML algorithms.

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
mean	20.091	6.188	230.722	146.688	3.597	3.217	17.849	0.438	0.406	3.688	2.812
std	6.027	1.786	123.939	68.563	0.535	0.978	1.787	0.504	0.499	0.738	1.615
min	10.4	4.0	71.1	52.0	2.76	1.513	14.5	0.0	0.0	3.0	1.0
max	33.9	8.0	472.0	335.0	4.93	5.424	22.9	1.0	1.0	5.0	8.0

Descriptive Statistics of mtcars data

Why do we need Feature Scaling?

- When dealing with different units of measurement:
  If the input features of a dataset have different units of measurement (e.g., height in centimeters and weight in pounds), feature scaling can help to bring them to a common scale.
- When dealing with numerical features of different ranges:
  If the input features of a dataset have numerical values with different ranges (e.g., one feature ranges from 0 to 1 while another ranges from 0 to 1000), feature scaling can help to bring them to a similar range, which can aid in model training and evaluation.
- When visualizing data :
  Feature scaling can be useful when visualizing data, particularly when creating plots that involve multiple features. Scaling the features can help to make the visualization more informative and easier to interpret.

Min-Max scaling

Scales features to a fixed range, typically [0, 1] or [-1, 1].

- Formula: $x' = \frac{x - \min(x)}{\max(x) - \min(x)}$
- Advantages: easy to implement, preserve the shape of the original distribution.
- Disadvantages: sensitive to outliers, can be problematic if the minimum and maximum values change for new data.

Descriptive Statistics for IRIS data

Python Code

```
X=iris.iloc[:,:-1]
pd.DataFrame(X).describe().loc[['mean','std','min','max']]
```

	Sepal_length	Sepal_width	Petal_length	Petal_width
mean	5.843333	3.054000	3.758667	1.198667
std	0.828066	0.433594	1.764420	0.763161
min	4.300000	2.000000	1.000000	0.100000
max	7.900000	4.400000	6.900000	2.500000

Descriptive Statistics for IRIS data after Scaling

Python Code

```
# Apply MinMax scaling
scaler = MinMaxScaler()
X_minmax = scaler.fit_transform(X)
pd.DataFrame(X_minmax).describe().loc[['mean','std','min','max']]
```

	Sepal_length	Sepal_width	Petal_length	Petal_width
mean	0.428704	0.439167	0.467571	0.457778
std	0.230018	0.180664	0.299054	0.317984
min	0.000000	0.000000	0.000000	0.000000
max	1.000000	1.000000	1.000000	1.000000

Standardization

Scales features to have zero mean and unit variance or in range [0,1].

- Z- Score Normalization $\frac{x - mean(x)}{SD(x)}$ or Mean Normalization: $\frac{x - mean(x)}{max(x) - min(x)}$
- Advantages: less sensitive to outliers, works well with Gaussian distributions.
- Disadvantages: can result in negative values, changes the shape of the original distribution.

Descriptive Stats for IRIS data after Scaling

Python Code

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_standard = scaler.fit_transform(X)
pd.DataFrame(X_standard).describe().loc[['mean','std','min','max']]
```

	Sepal_length	Sepal_width	Petal_length	Petal_width
mean	-0.0000	-0.0000	0.0000	-0.0000
std	1.0034	1.0034	1.0034	1.0034
min	-1.8700	-2.4390	-1.5687	-1.4444
max	2.4920	3.1147	1.7863	1.7109

(-3,3)

Descriptive Stats for IRIS data after Scaling

Python Code

```
from sklearn.preprocessing import Normalizer
scaler = Normalizer()
X_normalized = scaler.fit_transform(X)
pd.DataFrame(X_normalized).describe().loc[['mean','std','min','max']]
```

$$\frac{X - \text{mean}(x)}{\max(x) - \min(x)}$$

	Sepal_length	Sepal_width	Petal_length	Petal_width
mean	0.7516	0.4048	0.4550	0.1410
std	0.0446	0.1051	0.1597	0.0781
min	0.6539	0.2384	0.1678	0.0147
max	0.8609	0.6071	0.6370	0.2804

Robust scaling

Scales features using statistics that are robust to outliers.

- Formula: $x' = \frac{x - \text{median}(x)}{\text{IQR}(x)}$
- Advantages: works well with data that has outliers, preserves the shape of the distribution.
- Disadvantages: not as commonly used, can be sensitive to extreme outliers.

Descriptive Stats for IRIS data after Scaling

Python Code

```
from sklearn.preprocessing import RobustScaler
scaler = RobustScaler()
X_robust = scaler.fit_transform(X)
pd.DataFrame(X_normalized).describe().loc[['mean','std','min','max']]
```

	Sepal_length	Sepal_width	Petal_length	Petal_width
mean	0.7516	0.4048	0.4550	0.1410
std	0.0446	0.1051	0.1597	0.0781
min	0.6539	0.2384	0.1678	0.0147
max	0.8609	0.6071	0.6370	0.2804

Situations where Feature Scaling Techniques useful

Standardization

- Useful when using distance-based algorithms such as KNN or SVM
- Useful when dealing with features with different units of measurement
- Centers the data around 0 and scales it to have a standard deviation of 1

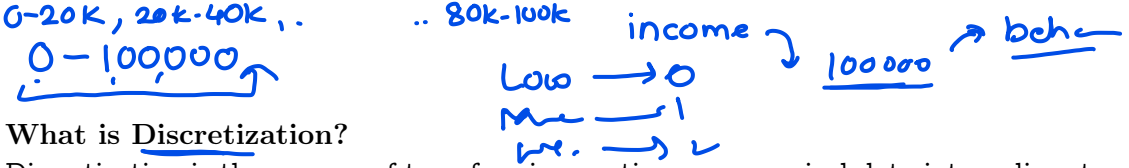
Min-Max Scaling

- Useful when dealing with numerical features of different ranges
- Scales the data to a fixed range (usually 0 to 1)

Robust Scaling

- Useful when dealing with outliers in the data
- Scales the data based on percentiles rather than the mean and standard deviation

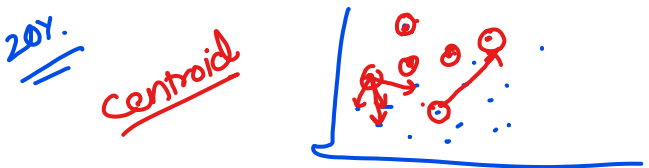
Discretization in Feature Scaling



What is Discretization?

Discretization is the process of transforming continuous numerical data into a discrete form by dividing the data into intervals or bins.

- Discretization can be done by equal width binning, equal frequency binning, or k-means clustering.
- Discretization is often used to simplify complex data or to prepare data for machine learning algorithms that can only handle discrete values.



Why is Discretization Needed?

Discretization can be useful in the following situations:

- When dealing with continuous data that cannot be easily quantified or compared.
- When dealing with data that has a large number of unique values.
- When preparing data for machine learning algorithms that can only handle discrete values, such as decision trees or association rule mining.

Applications of Discretization

- Customer segmentation based on spending behavior, income, or other demographic variables.
- Fraud detection by identifying anomalous patterns in financial transactions.
- Medical diagnosis by discretizing medical test results and identifying patterns that indicate the presence of a particular condition.

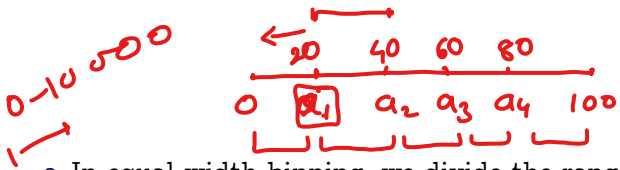
Python

```
from sklearn.preprocessing import KBinsDiscretizer
discretizer = KBinsDiscretizer(n_bins=5, strategy='quantile')
X_binned = discretizer.fit_transform(X)
print(X_binned[:10])
```

The above code uses the KBinsDiscretizer from the scikit-learn library to discretize the data into 5 equal-width bins.

0	5.1	1
2	4.7	0
3	4.6	0
4	5.0	1
5	5.4	1
6	4.6	0
7	5.0	1
8	4.4	0

Equal Width Binning



- In equal width binning, we divide the range of the feature into equally spaced bins of the same width.
- It is a simple and straightforward method, but it can be sensitive to outliers.
- This method is useful when the distribution of the data is approximately uniform.
- Here's an example Python code for equal width binning:

Python Code

```
sampledata = pd.DataFrame({'A': np.random.normal(0, 1, 100)})
sampledata['A_binned'] = pd.cut(sampledata['A'], 5, labels=False)
sampledata.head()
```

A	A_binned
-0.700232	1
0.947235	3
-1.157644	1
0.938526	3
-1.791459	0

Equal Frequency Binning

- In equal frequency binning, we divide the data into equally sized bins based on the number of observations in each bin.
- This method is less sensitive to outliers compared to equal width binning.
- It is useful when the distribution of the data is skewed.
- Here's an example Python code for equal frequency binning:

Equal Frequency Binning...

pandas

Python code

```
data = pd.DataFrame({'A': np.random.normal(0, 1, 100)})
data['A_binned'] = pd.qcut(data['A'], 5, labels=False)
data.head()
```

A	A_binned
0.777220	3
-1.612469	0
1.716547	4
0.826012	3
-0.131053	2

K-means Binning

- Applies K-means clustering to the feature values
- Useful when the distribution of the feature is not uniform and not skewed

Python

```
from sklearn.cluster import KMeans
data = {'A':[1,2,9,4,5,2,7,8,9,10]}
df = pd.DataFrame(data)
K = 3 # number of clusters (K)
kmeans = KMeans(n_clusters=K, random_state=0).fit(df[['A']])
df['A_bins'] = kmeans.labels_
print(df.T)
```

A	1	2	9	4	5	2	7	8	9	10
A_bins	1	1	0	2	2	1	0	0	0	0

Situations where each Discretization will be used?

- Equal Width Binning: useful when the feature has a uniform distribution
- Equal Frequency Binning: useful when the feature has a skewed distribution
- K-means Binning: useful when the feature has a non-uniform and non-skewed distribution

Feature Encoding

- Feature encoding is the process of converting categorical variables into numerical representations that can be used as input to machine learning models.
- This is necessary because most machine learning algorithms can only handle numerical data.
- There are several techniques for feature encoding, each with its own strengths and weaknesses.

Python Code

```
# Create a sample dataset
data = pd.DataFrame({
    'gender': ['M','F','M','M','F','F','M','F','M','F'],
    'edu': ['Bachelors','Masters','PhD','Bachelors',
            'Masters','PhD','Bachelors','Masters','PhD','Bachelors'],
    'salary': [50000,75000,100000,60000,80000,120000,70000,
               90000,110000,65000],
    'churn': [0,1,0,1,0,1,0,1,0,1]
})
```

Cat

Cat

gender

Cont.

Cat

One-Hot Encoding

- Creates a binary vector for each category in a categorical variable.
- The vector has a length equal to the number of categories, and each element is 0 except for the element corresponding to the category, which is 1.
- Useful for nominal categorical variables with no inherent order or ranking.
- Example: Encoding the colors of a car (red, blue, green) as [1 0 0], [0 1 0], [0 0 1].

Python Code

```
# Apply one-hot encoding to color and size
onehot = OneHotEncoder()
data_onehot = onehot.fit_transform(data[['color',]])
print(data['color'])
print(data_onehot.toarray())
```

color	OneHot		
red	0	0	1
green	0	1	0
blue	1	0	0
red	0	0	1
green	0	1	0
green	0	1	0
blue	1	0	0

Ordinal Encoding

- Assigns a numerical value to each category in a categorical variable based on its order or rank.
- The numerical values can be assigned in any order, but they must be consistent across all data points.
- Useful for categorical variables with an inherent order or ranking.
- Example: Encoding education levels (high school, some college, bachelor's degree, master's degree) as 1, 2, 3, 4.

Python Code

```
# Apply ordinal encoding to color and size
ordinal = OrdinalEncoder()
data_ordinal = ordinal.fit_transform(data[['color', 'size']])
print(data[['color', 'size']])
print(data_ordinal)
```

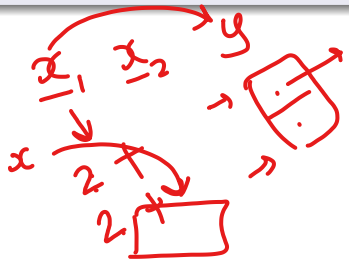
color	size	data_ordinal	
red	small	2	2
green	medium	1	1
blue	large	0	0
red	medium	2	1
green	small	1	2
green	medium	1	1
blue	large	0	0

Count/Frequency Encoding

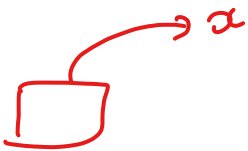
- Replaces each category in a categorical variable with the number of times it appears in the dataset (count encoding) or the proportion of times it appears (frequency encoding).
- Count/frequency encoding is useful for categorical variables with a large number of categories or categories with similar frequencies.
- Example: Encoding the countries in a dataset as the number of times they appear or their frequency of appearance.

Python Code

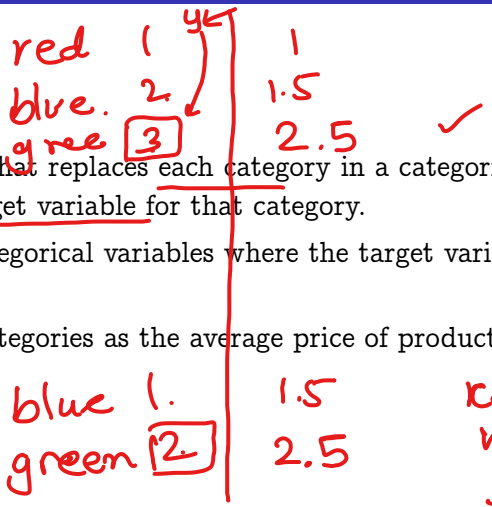
```
# Apply count/frequency encoding to color
count = CountEncoder()
data_count = count.fit_transform(data['color'])
print(data['color'])
print(data_count)
```



Color	color
red	2
green	3
blue	2
red	2
green	3
green	3
blue	2



Mean Encoding



- Mean encoding is a technique that replaces each category in a categorical variable with the mean value of the target variable for that category.
- Mean encoding is useful for categorical variables where the target variable varies significantly across categories.
- Example: Encoding product categories as the average price of products in that category.

Python Code

```
# Apply mean encoding to color
mean = TargetEncoder()
data_mean = mean.fit_transform(data['color'], data['label'])
print(data[['color','label']])
print(data_mean)
```

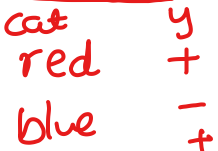
Color	Label		MeanEncode
red	0		0.51921
green	0		0.361715
blue	1		0.884739
red	1		0.51921
green	0		0.361715
green	1		0.361715
blue	1		0.884739

pd.mean(y)

0.50

df.groupby(Color).mean(key)

Weight of Evidence (WOE) Encoding



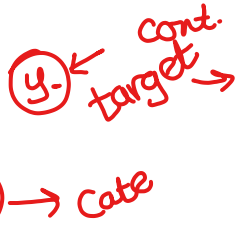
red $\longleftrightarrow$ $\log\left(\#(+)/\#(-)\right)$

- Replaces each category in a categorical variable with the logarithm of the ratio of the proportion of positive target variable instances to the proportion of negative target variable instances.
- Useful for categorical variables in binary classification problems.
- Example: Encoding credit risk categories (low, medium, high) as their respective WOE values based on the proportion of good credit instances to bad credit instances.

green $\log\left(\#(+ve)/\#(-ve)\right)$

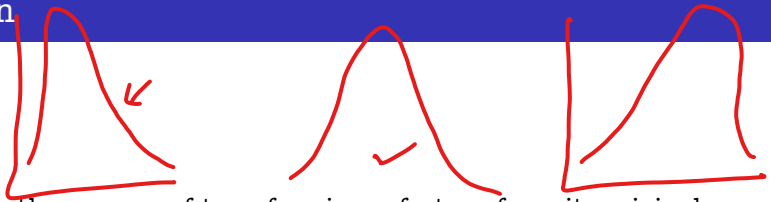
Python Code

```
woe = WOEEncoder()
data_woe = woe.fit_transform(data['color'], data['label'])
print(data[['color','label']])
print(data_woe)
```



Color	Label		WOE
red	0		-0.18232
green	0		-0.58779
blue	1		0.916291
red	1		-0.18232
green	0		-0.58779
green	1		-0.58779
blue	1		0.916291

Feature Transformation



- Feature transformation is the process of transforming a feature from its original distribution to a more desirable distribution, which can improve the performance of machine learning models.
- There are several techniques for feature transformation, including logarithmic transformation, reciprocal transformation, square root transformation, exponential transformation, Box-Cox transformation, and quintile transformation.

Python Code

```
import pandas as pd
mtcars = pd.read_csv('mtcars.csv')
mtcars.head()
```

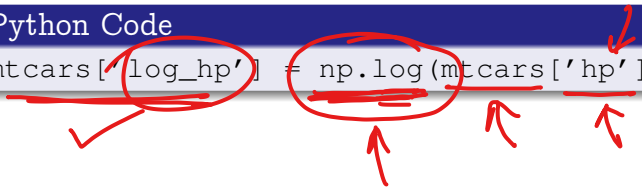
	model	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
0	Mazda RX4	21.0	6	160.0	110	3.9	2.62	16.46	0	1	4	4
1	Mazda RX4 Wag	21.0	6	160.0	110	3.9	2.875	17.02	0	1	4	4
2	Datsun 710	22.8	4	108.0	93	3.85	2.32	18.61	1	1	4	1
3	Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1
4	Hornet Sportabout	18.7	8	360.0	175	3.15	3.44	17.02	0	0	3	2

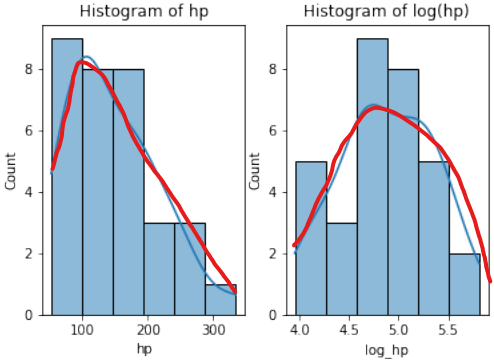
Logarithmic Transformation

- Used to transform data that is highly skewed or has a large range of values.
- Compresses the range of the data and reduces the impact of outliers, making it more suitable for machine learning models.
- Commonly used for financial data, such as stock prices/income data.

Python Code

```
mtcars['log_hp'] = np.log(mtcars['hp'])
```





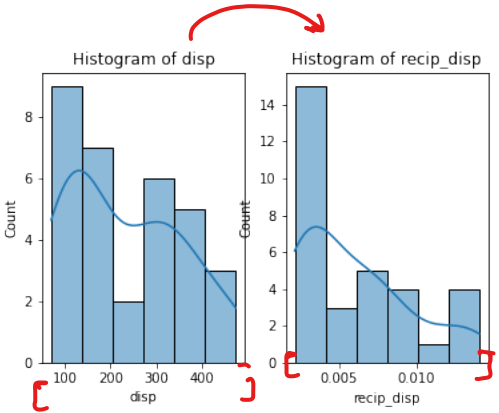
Reason: We might use a logarithmic transformation on the "hp" (horsepower) variable to reduce the skewness in its distribution. This could be useful if we are planning to use horsepower as a predictor variable in a linear regression model, as a more symmetric distribution can help improve the accuracy of the model.

Reciprocal Transformation

- Used to transform data that has a large range of values or is highly skewed.
- It compresses the range of the data and can improve the performance of machine learning models that assume linear relationships.
- Commonly used for data that is measured in units per time, such as speed/flow rates.

Python Code

```
mtcars['recip_disp'] = 1 / mtcars['disp']
```



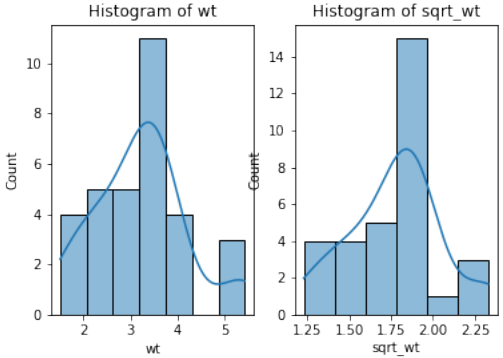
Reason: We might use a reciprocal transformation on the "disp" (displacement) variable if we want to put more emphasis on smaller engine sizes. By taking the reciprocal, we amplify the importance of smaller values of "disp", while de-emphasizing larger values.

Square Root Transformation

- Used to transform data that has a skewed distribution or contains negative values.
- It compresses the range of the data and can improve the performance of machine learning models that assume linear relationships.
- Commonly used for data that is measured in counts or frequencies, such as the number of website visits or sales per day.

Python Code

```
mtcars['sqrt_wt'] = np.sqrt(mtcars['wt'])
```



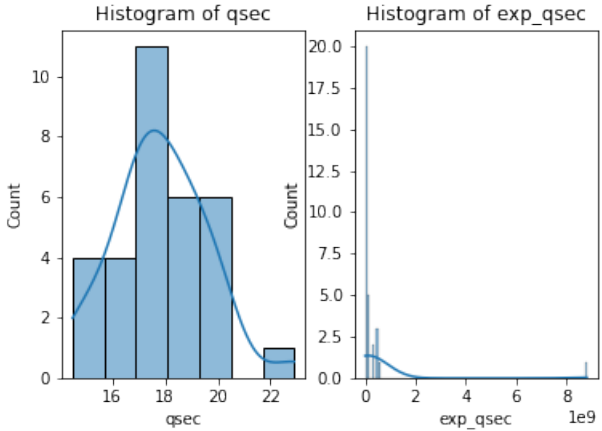
Reason: We might use a square root transformation on the "wt" (weight) variable to reduce the skewness in its distribution. This could be useful if we are planning to use weight as a predictor variable in a linear regression model, as a more symmetric distribution can help improve the accuracy of the model.

Exponential Transformation

- Used to transform data that has an exponential distribution or contains negative values.
- It can improve the performance of ML models that assume linear relationships and can be useful for forecasting future values.
- Commonly used for financial data, such as stock prices or interest rates.

Python Code

```
mtcars['exp_qsec'] = np.exp(mtcars['qsec'])
```



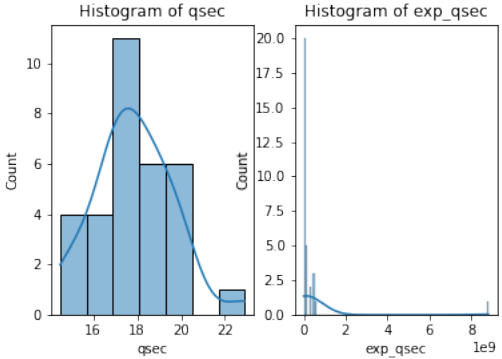
Reason: We might use an exponential transformation on the "qsec" (quarter mile time) variable if we want to amplify the importance of smaller values. By taking the exponential, we amplify the importance of smaller values of "qsec", while de-emphasizing larger values.

Box-Cox Transformation

- Used to transform data that has a skewed distribution and contains both positive and negative values.
- It can improve the performance of ML models that assume normal distributions and can be useful for forecasting future values.
- Commonly used for financial data, such as stock prices or interest rates.

Python Code

```
from scipy.stats import boxcox
mtcars['boxcox_mpg'], _ = boxcox(mtcars['mpg'])
```



Reason: To transform "mpg" (miles/gallon)into a standard distribution, we can use Box-Cox transformation which finds the optimal power transformation for a feature. This method is useful when dealing with non-standard distribution and allows for trying multiple transformations to compare their effectiveness.

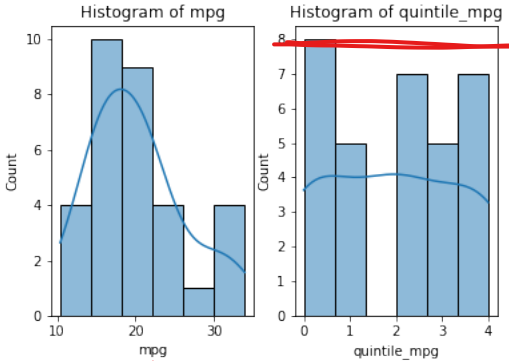
Quintile Transformation

pd.cut
pd.qcut

- Used to transform data that has a non-normal distribution and contains outliers.
- It transforms the data into quintiles, with each quintile containing an equal number of observations.
- Can improve the performance of ML models that are sensitive to outliers and can be useful for categorical data, such as customer segments or geographic regions.

Python Code

```
mtcars['quintile_mpg'] = pd.qcut(mtcars['mpg'], q=5,labels=False)
```



Reason: To transform "mpg" (miles per gallon) into a categorical variable, we can use quintile transformation by dividing the data into 5 equal groups based on rank or percentiles. This helps reduce outliers and creates categories for further analysis.