Module 2

Data Quality Assessment

Data quality assessment is the process of evaluating the accuracy, completeness, reliability, and overall suitability of data for a specific purpose or use case.

Ensuring data quality is crucial because poor-quality data can lead to incorrect insights, flawed decision-making, and wasted resources.

Here are the key steps and considerations involved in data quality assessment:

Data Profiling:

Conduct data profiling to gain a better understanding of your data. This involves summarizing key statistics and characteristics of the data, such as data types, distributions, and value ranges.

Data Accuracy:

Evaluate the accuracy of the data by comparing it to trusted sources or known standards. Check for errors, inconsistencies, and outliers in the data.

Data Completeness:

Assess whether the data is complete and whether it contains all the required information for your analysis or application. Look for missing values and determine how to handle them.

Data Consistency:

Ensure that the data is consistent within itself and across different data sources or datasets.

Check for naming conventions, data formats, and data definitions.

Data Reliability:

Evaluate the reliability of the data sources and the methods used to collect and process the data. Consider the reputation and trustworthiness of the sources.

Data Relevance:

Determine if the data is relevant to your specific use case. Irrelevant or outdated data can lead to incorrect conclusions.

Data Timeliness:

Assess whether the data is up-to-date and whether it meets your required timeframes. Outdated data may not reflect current conditions.

Data Integrity:

Ensure that data has not been tampered with or corrupted during storage or transmission. Implement data integrity checks, such as checksums or hashing.

Data Documentation:

Maintain comprehensive documentation of the data, including metadata, data lineage, and data transformation processes. This documentation helps users understand the data's context and history.

Data Quality Metrics:

Define specific data quality metrics and thresholds that align with your objectives. Common metrics include accuracy, completeness, consistency, and reliability.

Data Cleaning:

If you identify data quality issues, implement data cleaning and data transformation processes to correct or remove problematic data points.

Data Validation and Verification:

Develop validation and verification procedures to ensure that data quality is maintained over time. Regularly monitor and validate data as it is collected or updated.

Data Quality Tools:

Utilize data quality tools and software to automate data quality checks and validations. These tools can help streamline the assessment process.

Data Governance:

Establish data governance policies and procedures to maintain data quality standards consistent across the organization. Continuously work on improving data quality by addressing identified issues, refining data collection processes, and enhancing data quality controls.

Data quality assessment is an ongoing process that should be integrated into the data management lifecycle. It requires collaboration between data analysts, data engineers, data scientists, and business stakeholders to ensure that data is fit for its intended purpose. Regularly monitoring and improving data quality is essential for data-driven decision-making and successful business outcomes.

Handling Missing Values

Program 1: It will show the features which has null values

```
# importing pandas package

import pandas as pd

# making data frame from csv file

data = pd.read_csv("C:/Users/sraba/OneDrive/Desktop/applied data science/module2/employees.csv")

# creating  subset of dataset, bool series with gender value  NaN

# filtering data

bool_series = pd.isnull(data["Gender"])

missing_values = pd.isnull(data["Gender"]).sum()

Print(missing_values)

# displaying data only with Gender = NaN

data[bool_series]

# display the details of dataset
data.info()
```

output:

	First Name	Gender	Start Date	Last Login Time	Salary	Bonus %	Senior Management	Team
20	Lois	NaN	4/22/1995	7:18 PM	64714	4.934	True	Legal
22	Joshua	NaN	3/8/2012	1:58 AM	90816	18.816	True	Client Services
27	Scott	NaN	7/11/1991	6:58 PM	122367	5.218	False	Legal
31	Joyce	NaN	2/20/2005	2:40 PM	88657	12.752	False	Product
41	Christine	NaN	6/28/2015	1:08 AM	66582	11.308	True	Business Development
49	Chris	NaN	1/24/1980	12:13 PM	113590	3.055	False	Sales
51	NaN	NaN	12/17/2011	8:29 AM	41126	14.009	NaN	Sales
53	Alan	NaN	3/3/2014	1:28 PM	40341	17.578	True	Finance
60	Paula	NaN	11/23/2005	2:01 PM	48866	4.271	False	Distribution
64	Kathleen	NaN	4/11/1990	6:46 PM	77834	18.771	False	Business Development
69	Irene	NaN	7/14/2015	4:31 PM	100863	4.382	True	Finance
70	Todd	NaN	6/10/2003	2:26 PM	84692	6.617	False	Client Services
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
939	Ralph	NaN	7/28/1995	6:53 PM	70635	2.147	False	Client Services
945	Gerald	NaN	4/15/1989	12:44 PM	93712	17.426	True	Distribution
961	Antonio	NaN	6/18/1989	9:37 PM	103050	3.050	False	Legal
972	Victor	NaN	7/28/2006	2:49 PM	76381	11.159	True	Sales
985	Stephen	NaN	7/10/1983	8:10 PM	85668	1.909	False	Legal
989	Justin	NaN	2/10/1991	4:58 PM	38344	3.794	False	Legal
995	Henry	NaN	11/23/2014	6:09 AM	132483	16.655	False	Distribution

145 rows × 8 columns

Program: 2

#Heatmap Visualization of Null value data

```
import seaborn as sns
import matplotlib.pyplot as plt
# importing pandas package

import pandas as pd

# making data frame from csv file

data = pd.read_csv("C:/Users/sraba/OneDrive/Desktop/applied data science/module2/employees.csv")

print(data.isnull())
sns.heatmap(data.isnull(), cbar=True, cmap='viridis')
plt.show()
```

Description:

The code uses Seaborn and Matplotlib to create a heatmap visualization of missing values in a DataFrame named data. Here's a breakdown of what the code does:

import seaborn as sns and import matplotlib.pyplot as plt: These lines import the Seaborn and Matplotlib libraries, which are used for data visualization.

sns.heatmap(data.isnull(), cbar=False, cmap='viridis'):

data.isnull(): This part of the code creates a DataFrame of the same shape as your original DataFrame data, where each cell is True if the corresponding cell in data is missing (i.e., contains a NaN or None value) and False otherwise.

sns.heatmap(): This function from Seaborn is used to create a heatmap. It takes several parameters:

data.isnull(): The DataFrame of boolean values indicating missing values.

cbar=False: This parameter specifies whether to show the colorbar. In this case, it's set to False to hide the colorbar.

cmap='viridis': This parameter specifies the colormap to use for coloring the heatmap. 'viridis' is a popular choice, but you can choose other colormaps as well.

plt.show(): This line displays the heatmap using Matplotlib.

Program 3: It will show the features which doesn’t has any null values.

```
# importing pandas package
import pandas as pd

# making data frame from csv file
data = pd.read_csv("C:/Users/sraba/OneDrive/Desktop/applied data
science/module2/employees.csv")

# creating bool series True for NaN values
bool_series = pd.notnull(data["Gender"])

# display the no of not null value of specific feature
without_missing_values=pd.notnull(data["Gender"]).sum()

print(without_missing_values)

# filtering data
# displaying data only with Gender = Not NaN
data[bool_series]
# display the details of dataset
data.info()
```

Output: As shown in the output image, only the rows having Gender = NOT NULL are displayed.

	First Name	Gender	Start Date	Last Login Time	Salary	Bonus %	Senior Management	Team
0	Douglas	Male	8/6/1993	12:42 PM	97308	6.945	True	Marketing
1	Thomas	Male	3/31/1996	6:53 AM	61933	4.170	True	NaN
2	Maria	Female	4/23/1993	11:17 AM	130590	11.858	False	Finance
3	Jerry	Male	3/4/2005	1:00 PM	138705	9.340	True	Finance
4	Larry	Male	1/24/1998	4:47 PM	101004	1.389	True	Client Services
5	Dennis	Male	4/18/1987	1:35 AM	115163	10.125	False	Legal
6	Ruby	Female	8/17/1987	4:20 PM	65476	10.012	True	Product
7	NaN	Female	7/20/2015	10:43 AM	45906	11.598	NaN	Finance
8	Angela	Female	11/22/2005	6:29 AM	95570	18.523	True	Engineering
9	Frances	Female	8/8/2002	6:51 AM	139852	7.524	True	Business Development
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
994	George	Male	6/21/2013	5:47 PM	98874	4.479	True	Marketing
996	Phillip	Male	1/31/1984	6:30 AM	42392	19.675	False	Finance
997	Russell	Male	5/20/2013	12:39 PM	96914	1.421	False	Product
998	Larry	Male	4/20/2013	4:45 PM	60500	11.985	False	Business Development
999	Albert	Male	5/15/2012	6:24 PM	129949	10.169	True	Sales

855 rows × 8 columns

Handling Missing Values

Once you've identified the missing values, you can choose one or more of the following methods to handle them:

1. Removing Rows with Missing Values

If the missing values are relatively few and won't significantly affect your analysis:

program -4

Dropping Rows with at least 1 null value in CSV file

```
# importing pandas module
import pandas as pd
```

```
# making data frame from csv file

data = pd.read_csv("C:/Users/sraba/OneDrive/Desktop/applied data
science/module2/employees.csv")


# making new data frame with dropped NA values

new_data = data.dropna(axis = 0, how ='any')

new_data


print("Old data frame length:", len(data))

print("New data frame length:", len(new_data))

print("Number of rows with at least 1 NA value: ", (len(data)-len(new_data)))
```

Description: Using Dropna function. data: This is assumed to be a Pandas DataFrame containing your data.

.dropna(axis=0, how='any'): This is a DataFrame method that removes rows containing missing values from the original DataFrame.

axis=0: This parameter specifies that you want to remove rows (axis 0) containing missing values. Rows are dropped when at least one NaN (missing value) is found in any of the row's cells.

how='any': This parameter specifies that you want to drop rows if any missing value is found in the row. In other words, if any cell in a row contains a NaN, that entire row will be removed from the DataFrame.

2. Filling Missing Values

You can fill missing values with a specific value, such as the mean, median, or a constant

Program -5

Filling null values in CSV File

```
# importing pandas package
import pandas as pd

# making data frame from csv file
data = pd.read_csv("C:/Users/sraba/OneDrive/Desktop/applied data
science/module2/employees.csv")
```

```
# Printing the first 10 to 24 rows of
# the data frame for visualization
data[10:25]

# Fill missing values with the mean/mode/median of the column
#print(data['Bonus %'].fillna(data['Bonus %'].mean(), inplace=True))


# Fill missing values with a constant
#data.fillna(0, inplace=True)

# will replace  Nan value in dataframe with value -99
#print(data.replace(to_replace = np.nan, value = -99))
```

3. Filling with Categorical Missing Values

For categorical variables, you can fill missing values with the most frequent category or a specific label:

Program 6

```
# importing pandas package
import pandas as pd

# making data frame from csv file
data = pd.read_csv("C:/Users/sraba/OneDrive/Desktop/applied data
science/module2/employees.csv")

# filling a null values using fillna() with a specific value
data["Gender"].fillna("No Gender", inplace = True)

missing_values = pd.isnull(data["Gender"]).sum()
print(missing_values)
```

program 7

```
# importing pandas package
import pandas as pd

# reading data  from csv file
data = pd.read_csv("C:/Users/sraba/OneDrive/Desktop/applied data
science/module2/employees.csv")


# Fill missing categorical values with the most frequent category
```

```
data['Gender'].fillna(data['Gender'].mode()[0], inplace=True)

missing_values = pd.isnull(data["Gender"]).sum()
print(missing_values)
```

4. Handling Time Series Data

When dealing with time series data, you may need to handle missing values differently.

- **Methods like forward fill,**
- **backward fill, and**
- **interpolation are often appropriate.**

Forward fill (ffill) and backward fill (bfill) are techniques used to fill missing values in a DataFrame by propagating values from neighboring rows in a specific direction. These methods are often used when dealing with time-series data or any data where values have a natural order or sequence.

You can choose between forward fill and backward fill based on your specific needs and the nature of your data. These methods are useful for preserving temporal or sequential patterns in your data when filling missing values.

Here's how to use forward fill and backward fill in Pandas:

Forward Fill (ffill):

Forward fill replaces missing values with the most recent non-missing value in the column. It propagates values forward in the column.

Program :8

```
import pandas as pd
import numpy as np

# Sample DataFrame with missing values
data = pd.DataFrame({'A': [1, 2, np.nan, 4, np.nan, 6]})

# Forward fill missing values in column 'A'
data['A'].fillna(method='ffill', inplace=True)

# Display the DataFrame after forward fill
print(data)
```

Backward Fill (bfill):

Backward fill replaces missing values with the next non-missing value in the column. It propagates values backward in the column.

Program :9

```
import pandas as pd
import numpy as np

# Sample DataFrame with missing values
data = pd.DataFrame({'A': [1, 2, np.nan, 4, np.nan, 6]})

# Backward fill missing values in column 'A'
data['A'].fillna(method='bfill', inplace=True)

# Display the DataFrame after backward fill
print(data)
```

b. Interpolation

Interpolation is a method for filling missing values in a DataFrame by estimating them based on the values of adjacent data points. It can be a useful technique when dealing with time-series or continuous data where the order of data points is significant. In Python, you can perform interpolation using the interpolate() method in Pandas.

Here's how to use interpolation to fill missing values in a DataFrame:

Program :10

```
import pandas as pd
import numpy as np

# Sample DataFrame with missing values
data = pd.DataFrame({'A': [1, 2, np.nan, 4, np.nan, 6]})

# Interpolate missing values in column 'A' using linear interpolation
data['A'].interpolate(method='linear', inplace=True)

# Display the DataFrame after interpolation
```

```
print(data)

output:
     A
0  1.0
1  2.0
2  3.0
3  4.0
4  5.0
5  6.0
```

In this example, we have a DataFrame with missing values in column 'A'. We use the interpolate() method with the method='linear' parameter to perform linear interpolation specifically on column 'A'. The inplace=True parameter modifies the DataFrame in place.

- Time 1: 0 minutes, Temperature: 20°C
- Time 2: 10 minutes, Temperature: 30°C

Example:

The formula for linear interpolation is:

$$\text{Estimated Value} = \text{Value at Time 1} + \left(\frac{\text{Time Interval}}{\text{Total Time Interval}}\right) \times \text{Change in Value}$$

In this case:

- Value at Time 1 = 20°C
- Time Interval = 5 minutes (the time for which you want to estimate the temperature)
- Total Time Interval = 10 minutes (the time difference between the two known data points)
- Change in Value = Temperature at Time 2 - Temperature at Time 1 = 30°C - 20°C = 10°C

Now, plug these values into the formula:

$$\text{Estimated Temperature at 5 minutes} = 20^\circ C + \left(\frac{5 \text{ minutes}}{10 \text{ minutes}}\right) \times 10^\circ C = 20^\circ C + 0.5 \times 10^\circ C = 25^\circ C$$

5.Imputation with Machine Learning Models

For more advanced scenarios, you can use machine learning models like k-Nearest Neighbors (KNN) or regression to impute missing values based on other features in your dataset.

After running this code, the DataFrame will contain imputed values for missing data, where each missing value has been replaced by an estimated value based on **the two nearest neighbors in the**

feature space. This imputation technique can be useful when you want to preserve the overall structure and relationships in your data while filling in missing values.

Program :11

```
# importing pandas package
import pandas as pd

from sklearn.impute import KNNImputer

# making data frame from csv file
data = pd.read_csv("C:/Users/sraba/OneDrive/Desktop/applied data
science/module2/salary.csv")
print(data.info)

imputer = KNNImputer(n_neighbors=2)
data = pd.DataFrame(imputer.fit_transform(data), columns=data.columns)
data.info
```

The code uses scikit-learn's KNNImputer to impute missing values in a Pandas DataFrame df using a k-Nearest Neighbors (KNN) imputation approach. Here's a breakdown of the code:

from sklearn.impute import KNNImputer: This line imports the KNNImputer class from scikit-learn's impute module. The KNNImputer is a machine learning-based imputation method that replaces missing values with estimates based on the values of their nearest neighbors.

imputer = KNNImputer(n_neighbors=2): This line creates an instance of the KNNImputer class with n_neighbors set to 2. This means that it will consider the values of the two nearest neighbors to impute missing values.

df = pd.DataFrame(imputer.fit_transform(df), columns=df.columns): This line applies the KNN imputation to the DataFrame df and replaces the original DataFrame with the imputed one. Here's what's happening within this line:

imputer.fit_transform(df): This fits the KNNImputer model to the data in df and then transforms the DataFrame to impute missing values. The fit_transform method returns a NumPy array with imputed values.

pd.DataFrame(...): It converts the NumPy array with imputed values back into a Pandas DataFrame.

columns=df.columns: This sets the column names of the new DataFrame to be the same as the original DataFrame df.

Using LinearRegression

Regression-based imputation is a technique that uses regression models to predict missing values in a dataset based on the relationships between the missing variable and other variables in the dataset. In Python, you can use various regression models from libraries like Scikit-Learn or StatsModels for this purpose.

Here's an example using Scikit-Learn's LinearRegression for regression-based imputation:

Program 12

```
import pandas as pd
from sklearn.linear_model import LinearRegression
# Sample dataset with missing values
data = {
    'Feature1': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
    'Target': [2, 4, 6, 8, 10, None, None, None, None, None]
}
df = pd.DataFrame(data)
print(df)
# Separate rows with and without missing values
df_missing = df[df['Target'].isna()]
df_not_missing = df[df['Target'].notna()]
# Create a linear regression model
model = LinearRegression()
# Fit the model on data with no missing values
model.fit(df_not_missing[['Feature1']], df_not_missing['Target'])
# Predict missing values
```

```
predicted_values = model.predict(df_missing[['Feature1']])
# Fill missing values with predicted values
df_missing['Target'] = predicted_values
# Combine the filled and non-missing rows
df = pd.concat([df_not_missing, df_missing], ignore_index=True)
print(df)
```

In this example:

We have a dataset with a missing target variable ('Target').

We split the dataset into two parts: one with missing values (df_missing) and one without missing values (df_not_missing).

We create a linear regression model and fit it using the available data in df_not_missing.

We then use the trained model to predict missing values in df_missing.

Finally, we combine the filled rows and non-missing rows to obtain the complete dataset.

Data imputation methods :

Data imputation methods are techniques used to fill in missing values in a dataset. Handling missing data is a crucial step in data preprocessing because many machine learning algorithms and statistical analyses require complete datasets. Here are some common data imputation methods:

Mean/Median/Mode Imputation:

Mean Imputation: Fill missing values with the mean (average) of the non-missing values in the column.

Median Imputation: Fill missing values with the median value of the non-missing values in the column. This is less sensitive to outliers than the mean.

Mode Imputation: Fill missing values with the mode (most frequent value) of the column for categorical data.

Forward Fill and Backward Fill:

Forward Fill (ffill): Replace missing values with the most recent non-missing value before them (propagating values forward).

Backward Fill (bfill): Replace missing values with the next non-missing value after them (propagating values backward). These methods are useful for time-series data.

Interpolation:

Linear Interpolation: Estimate missing values based on a linear relationship between neighboring data points.

Polynomial Interpolation: Estimate values using polynomial functions fitted to the surrounding data points.

Spline Interpolation: Use piecewise-defined polynomials or splines to fill missing values.

Time-based Interpolation: Interpolate values based on time intervals or timestamps.

K-Nearest Neighbors (KNN) Imputation:

Estimate missing values by averaging or weighting values from the k-nearest neighbors in a multidimensional space.

Regression Imputation:

Use regression models to predict missing values based on relationships with other variables in the dataset.

Multiple Imputation:

Generate multiple complete datasets with imputed values and then perform analyses on each dataset. This accounts for uncertainty in imputed values and can provide more robust results.

Deep Learning Imputation:

Train neural networks or deep learning models to predict missing values based on the relationships in the data. Techniques like autoencoders can be used for this purpose.

Domain-specific Imputation:

Use domain knowledge or specific rules to impute missing values. For example, imputing missing age values based on common age-group characteristics.

Random Sampling Imputation:

Fill missing values by randomly selecting values from the observed data. This can introduce some randomness but is useful in certain scenarios.

Listwise Deletion:

Discard rows with missing values entirely. This should be used cautiously, as it can lead to a loss of information and potentially biased results.

The choice of imputation method depends on the nature of your data, the underlying assumptions, and the specific goals of your analysis. It's essential to understand the characteristics of your data and carefully consider the potential impact of the chosen imputation method on your results. It's also good practice to document and report any imputation methods used in your data analysis to maintain transparency.

Feature Aggregation:

Feature aggregation is a data preprocessing technique used in machine learning and data analysis to reduce the dimensionality of a dataset by combining or summarizing multiple related features (variables) into a single feature or a set of new features. Feature aggregation is often employed to simplify complex datasets, improve model performance, and reduce computational complexity.

When performing feature aggregation, it's essential to consider the domain knowledge, the goals of the analysis or modelling task, and the potential impact on the interpretability of the data. Careful feature selection and aggregation can lead to more efficient and accurate machine learning models while avoiding information loss. However, inappropriate aggregation can introduce biases or obscure valuable insights, so it should be done judiciously.

Here are some common methods and scenarios where feature aggregation is applied:

Summation or Aggregation Functions:

Calculate the sum, mean, median, maximum, minimum, or other statistical measures of a set of related features.

For example, if you have daily sales data, you can aggregate it into monthly or yearly totals.

Time-Based Aggregation:

Aggregate time-series data into different time intervals (e.g., hourly, daily, weekly) to capture patterns at different levels of granularity. This is useful for trend analysis and forecasting.

Categorical Feature Aggregation:

Combine categories within a categorical feature to create broader categories or reduce the number of distinct values. For example, group rare categories into an "Other" category.

Feature Engineering:

Create new features by combining or transforming existing features to capture specific relationships or interactions in the data. Feature engineering can include operations like adding ratios, creating interaction terms, or using mathematical functions.

Text Data Aggregation:

In natural language processing (NLP) tasks, aggregate text data by calculating various statistics such as term frequency-inverse document frequency (TF-IDF) scores, word embeddings, or topic modelling representations.

Feature Selection and Dimensionality Reduction:

Use feature aggregation as a dimensionality reduction technique to reduce the number of features in high-dimensional datasets. Dimensionality reduction can help prevent overfitting and reduce computational complexity.

Image and Sensor Data Aggregation:

Aggregate pixel values or sensor readings within regions of interest to extract meaningful features from images or sensor data. For example, compute histograms or texture features from image regions.

Feature Scaling and Normalization:

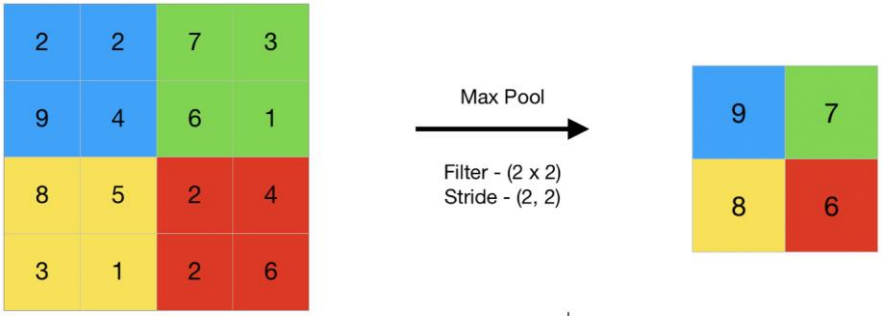
Combine feature scaling and normalization techniques to ensure that features are on a consistent scale. Aggregating features and then scaling them can help in some cases.

Principal Component Analysis (PCA):

PCA can be considered a form of feature aggregation that combines the original features into orthogonal components while preserving as much variance as possible. It's a powerful dimensionality reduction technique.

Feature Extraction in Deep Learning:

In deep learning, feature aggregation layers, such as pooling layers in convolutional neural networks (CNNs), aggregate features within local regions of input data to reduce spatial dimensions while retaining essential information.



Feature encoding:

Feature encoding is a crucial step in preparing data for machine learning models, especially when dealing with categorical variables.

Encoding categorical variables means converts them into numerical representations that machine learning algorithms can work with.

Python provides several methods and libraries for feature encoding.

Few common techniques are there, and they can be implemented using libraries like Pandas and Scikit-Learn.

Here's how you can perform feature encoding in Python:

1. **Label Encoding:** Label encoding assigns a unique integer to each category in a categorical variable. In this example, 'Red' gets encoded as 2, 'Green' as 1, and 'Blue' as 0.

```
import pandas as pd
from sklearn.preprocessing import LabelEncoder
data = {'Color': ['Red', 'Green', 'Blue', 'Red', 'Green']}
df = pd.DataFrame(data)
print(df)
label_encoder = LabelEncoder()
df['Color_encoded'] = label_encoder.fit_transform(df['Color'])
print(df['Color_encoded'])
```

2. **One-Hot Encoding**: One-hot encoding creates binary columns for each category in the categorical variable, where each column represents the presence (1) or absence (0) of a category. This will create three columns: 'Color_Red', 'Color_Green', and 'Color_Blue', where each row will have 1 in the corresponding column for the color and 0 in the others.

```
import pandas as pd
data = {'Color': ['Red', 'Green', 'Blue', 'Red', 'Green']}
df = pd.DataFrame(data)
print(df)
df_encoded = pd.get_dummies(df, columns=['Color'], prefix=['Color'])
print(df_encoded)
```

3. **Custom Encoding**: You can also perform custom encoding for categorical variables if needed, especially if the variable has some ordinal relationship. For example, you might assign values based on the order of importance.
   This custom encoding maps 'Small' to 1, 'Medium' to 2, and 'Large' to 3.

```
import pandas as pd
data = {'Size': ['Small', 'Medium', 'Large', 'Medium', 'Small']}
df = pd.DataFrame(data)
size_mapping = {'Small': 1, 'Medium': 2, 'Large': 3}
df['Size_encoded'] = df['Size'].map(size_mapping)
print(df['Size_encoded'])
```

4. **Binary Encoding:** Binary encoding is an efficient method for encoding categorical features by converting the categories into binary code.
5. **Count Encoding:** Count encoding replaces each category with the count of occurrences in the dataset. It's useful when the frequency of each category is informative.
6. **Target Encoding (Mean Encoding):** Target encoding is used for encoding categorical features based on the mean of the target variable within each category.
7. **Feature hashing**, also known as the hashing trick, is a technique used for dimensionality reduction and feature encoding in machine learning
   It's particularly useful when you have a large number of categorical features, and you want to convert them into a fixed number of numerical features, reducing the dimensionality of the data.
   Feature hashing works by applying a hash function to the original features and then using the hash values as new features.

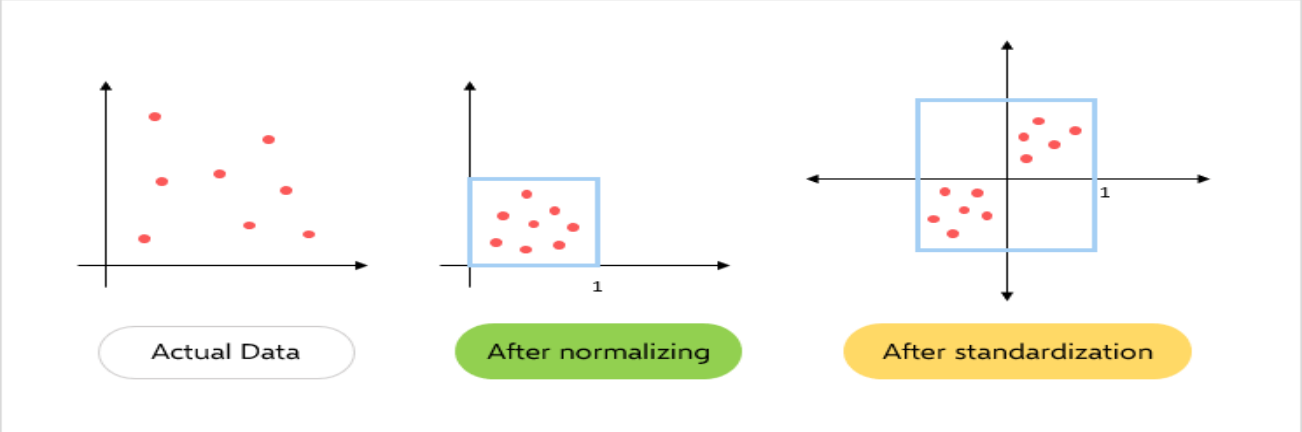
Remember that the choice of encoding method depends on the nature of your data and the machine learning algorithm you intend to use.
Label encoding may introduce unintended ordinal relationships, so it's generally safer to use one-hot encoding for non-ordinal categorical variables.
Custom encoding can be useful when you have domain knowledge that suggests a specific encoding scheme.
Additionally, Scikit-Learn provides tools like LabelEncoder and OneHotEncoder for these purposes, making it convenient to integrate feature encoding into your machine learning pipelines.

Normalization techniques in python

Normalization, also known as feature scaling, is a crucial preprocessing step in data science and machine learning.
It involves scaling numerical features to a standard range to ensure that they have similar magnitudes.
Normalization helps prevent certain features from dominating others and can lead to more stable and effective machine learning models.

Choose the appropriate normalization technique based on the nature of your data and the requirements of your machine learning algorithm.

Some algorithms, like k-means clustering, are sensitive to the scale of features and may require standardization, while others, like decision trees or random forests, are less affected by feature scaling.



Three common techniques for normalization in Python are Min-Max scaling , Z-score scaling (Standardization) and custom scaling.

1. Min-Max Scaling:

Min-Max scaling scales features to a specific range, typically [0, 1]. It's especially useful when you want to transform your data into a bounded range.

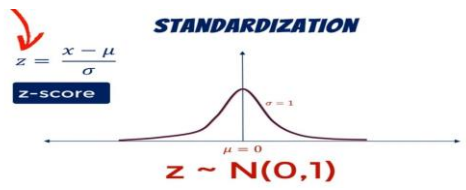
python
After applying Min-Max scaling, both features will be in the range [0, 1].

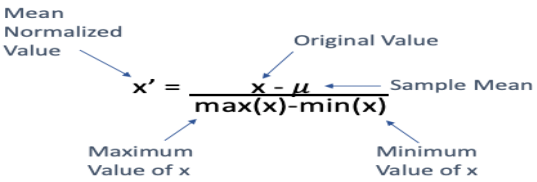
```
import pandas as pd

from sklearn.preprocessing import MinMaxScaler

# Sample data
```

```
data = {'Feature1': [10, 20, 30, 40],
        'Feature2': [1, 2, 3, 4]}
df = pd.DataFrame(data)
print(df)
# Create a MinMaxScaler instance
scaler = MinMaxScaler()

# Fit the scaler to your data and transform it
df[['Feature1', 'Feature2']] = scaler.fit_transform(df[['Feature1', 'Feature2']])
print(df[['Feature1', 'Feature2']])
```

2. Z-Score Scaling (Standardization):

Z-score scaling (Standardization) scales features to have a mean of 0 and a standard deviation of 1. It's particularly useful when your data follows a normal distribution.





After applying Z-score scaling, both features will have a mean of 0 and a standard deviation of 1.

```
import pandas as pd
from sklearn.preprocessing import StandardScaler
# Sample data
data = {'Feature1': [10, 20, 30, 40],
        'Feature2': [1, 2, 3, 4]}
df = pd.DataFrame(data)
```

```
print(df)

# Create a StandardScaler instance

scaler = StandardScaler()

# Fit the scaler to your data and transform it

df[['Feature1', 'Feature2']] = scaler.fit_transform(df[['Feature1', 'Feature2']])

print(df[['Feature1', 'Feature2']])
```

3. Custom Scaling

You can also perform custom scaling if needed, depending on the specific requirements of your dataset and analysis. This may involve scaling to a different range or using domain-specific knowledge to determine the scaling factor.

Custom scaling allows you to define the scaling logic based on your domain knowledge or specific data characteristics.

```
import pandas as pd

# Sample data

data = {'Feature1': [10, 20, 30, 40],

        'Feature2': [1, 2, 3, 4]}

df = pd.DataFrame(data)

print(df)

# Custom scaling (for example, scaling to a range of [0, 100])

df['Feature1'] = (df['Feature1'] - df['Feature1'].min()) / (df['Feature1'].max() - df['Feature1'].min()) * 100

print(df[['Feature1', 'Feature2']])
```

Standardization is commonly used when the distribution of features is not known, while Min-Max scaling is suitable when you want to map features to a specific range. Custom scaling can be useful when you have domain-specific constraints or knowledge about the data.

Data Visualization using matplotlib

In 2002, John Hunter was introduced the Matplotlib.

It is a popular Python library for creating static, animated, and interactive visualizations in various formats.

It provides a wide range of tools for creating plots and charts, making it a valuable tool for data visualization and scientific computing.

Matplotlib offers many additional features, such as 3D plotting, annotations, and interactive widgets when used in Jupyter Notebooks.

Matplotlib works well with other Python libraries like Seaborn and Pandas for enhanced data visualization capabilities.

Step to use Matplotlib

Installation: Use pip to install Matplotlib

pip install matplotlib

Import Matplotlib: Import the library in your Python script or Jupyter Notebook

```
import matplotlib.pyplot as plt
```

Different Plot Types: Matplotlib supports various plot types, including scatter plots, bar plots, histograms, pie charts, and more.

You can choose the appropriate type depending on your data and visualization needs.

Code to do simple line chart

```
x = [1, 2, 3, 4, 5]

y = [2, 4, 6, 8, 10]

plt.plot(x, y)

plt.xlabel('X-axis')

plt.ylabel('Y-axis')

plt.title('Simple Line Plot')

plt.show()
```

Subplots: You can create multiple subplots within a single figure using plt.subplot() or plt.subplots().

plt.subplot(2, 2, 4) means:

nrows = 2: There are 2 rows in the grid.

ncols = 2: There are 2 columns in the grid.

index = 4: You are creating a subplot in the fourth position (bottom-right) of the 2x2 grid.

```
plt.subplot(2, 2, 1)

plt.plot(x, y)

plt.title('Subplot 1')


plt.subplot(2, 2, 2)

plt.scatter(x, y)

plt.title('Subplot 2')
```

```
plt.subplot(2, 2, 3)
plt.bar(x, y)
plt.title('Subplot 3')

plt.subplot(2, 2, 4)
plt.hist(y, bins=5)
plt.title('Subplot 4')
plt.tight_layout()  # Ensures proper spacing
plt.show()
```

Saving Plots: You can save your plots to various file formats (e.g., PNG, PDF, SVG) using plt.savefig().

```
plt.savefig('my_plot.png')
```

Functions or Submodules:

Matplotlib is organized into several submodules, each of which serves a specific purpose or provides a particular set of functionality.

matplotlib.pyplot (pyplot): This submodule provides a high-level interface for creating and customizing plots and charts.

It is commonly used for most basic plotting tasks and simplifies the process of creating visualizations.

matplotlib.figure: The figure module is responsible for creating and managing figure objects, which serve as the top-level container for all the elements of a plot.

matplotlib.axes: The axes module contains classes for creating and customizing the coordinate systems (i.e., the x and y axes) within a figure. You can create subplots and customize them using this submodule.

matplotlib.axis: This submodule handles the properties and formatting of axis objects, including tick marks, labels, and scales.

matplotlib.lines: The lines module provides classes and functions for working with line plots. It allows you to create and customize lines in your plots.

matplotlib.patches: The patches module offers various geometric shapes (e.g., rectangles, circles, polygons) that you can use to annotate or highlight regions in your plots.

matplotlib.text: This submodule deals with adding text annotations to your plots. It includes classes for working with text elements and labels.

matplotlib.image: The image module is used for displaying and manipulating images in Matplotlib plots.

matplotlib.colors: This submodule provides classes and functions for working with colors, including colormaps and color conversions.

matplotlib.colorbar: The colorbar module allows you to add colorbars to your plots, which are useful for interpreting the color mapping in various types of plots.

matplotlib.ticker: The ticker module is responsible for controlling the formatting of tick locations and labels on the axes.

matplotlib.gridspec: This submodule provides a flexible way to create complex subplot layouts.

matplotlib.transforms: The transforms module is used for coordinate transformations and is often used internally by Matplotlib.

matplotlib.widgets: The widgets module allows you to add interactive widgets to your Matplotlib figures in Jupyter notebooks and other interactive environments.

matplotlib.backends: The backends module contains code for handling different rendering backends, such as rendering to a window, saving to an image file, or embedding plots in GUI applications.

```
import matplotlib

matplotlib.use('TkAgg')
```

matplotlib.dates: This submodule is used for working with date and time data in Matplotlib.

mpl_toolkits: This is not a single module but a directory containing various toolkit submodules. One of the commonly used toolkits is mpl_toolkits.mplot3d, which provides 3D plotting functionality**.**

Descriptive Statistics

Descriptive statistics are measures that summarize important features of data, often with a single number. Producing descriptive statistics is a common first step to take after cleaning and preparing a data set for analysis.

Measures of center

Measures of center are statistics that give us a sense of the "middle" value of a numeric variable.

Common measures of center include the mean, median and mode.

The mean is simply an average: the sum of the values divided by the total number of records. We can use df.mean() to get the mean of each column in a DataFrame.

```
%matplotlib inline

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
mtcars = pd.read_csv("../input/mtcars/mtcars.csv")
mtcars =  mtcars.rename(columns={'Unnamed: 0': 'model'})
mtcars.index = mtcars.model
del mtcars["model"]

mtcars.head()
```

mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
model										
Mazda RX4	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4
Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4
Datsun 710	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4
Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3
Hornet Sportabout	18.7	8	360.0	175	3.15	3.440	17.02	0	0	3

MEAN & Median

```
mtcars.mean()             # Get the mean of each column
```

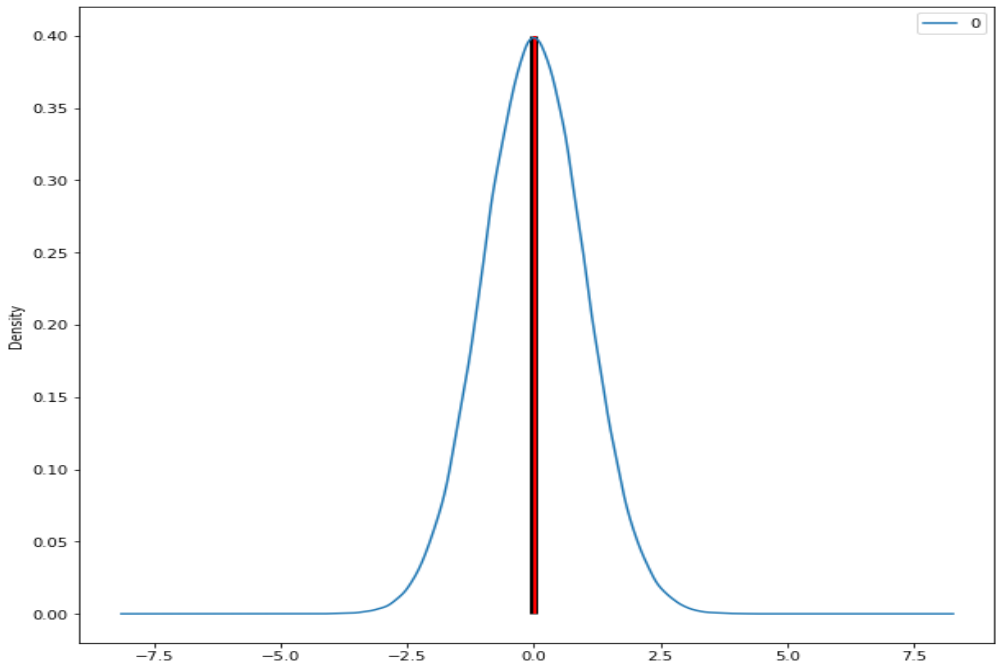
```
mpg     20.090625
cyl      6.187500
disp   230.721875
hp     146.687500
drat     3.596563
wt       3.217250
qsec    17.848750
vs       0.437500
am       0.406250
gear     3.687500
carb     2.812500
dtype: float64
```

```
mtcars.mean(axis=1)       # Get the mean of each row

mtcars.median()           # Get the median of each column
```

```
mpg     19.200
cyl      6.000
disp   196.300
hp     123.000
drat     3.695
wt       3.325
qsec    17.710
vs       0.000
am       0.000
gear     4.000
carb     2.000
dtype: float64
```

Although the mean and median both give us some sense of the center of a distribution, they aren't always the same. The median always gives us a value that splits the data into two halves while the mean is a numeric average so extreme values can have a significant impact on the mean. In a symmetric distribution, the mean and median will be the same.

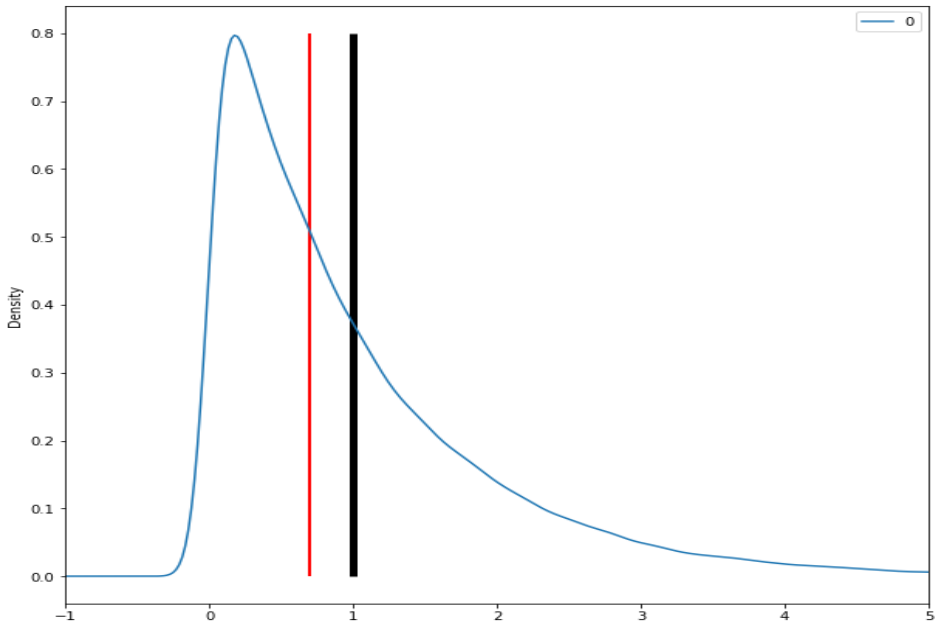
```
norm_data = pd.DataFrame(np.random.normal(size=100000))

norm_data.plot(kind="density",
           figsize=(10,10));


plt.vlines(norm_data.mean(),    # Plot black line at mean
        ymin=0,
        ymax=0.4,
        linewidth=5.0);

plt.vlines(norm_data.median(),  # Plot red line at median
        ymin=0,
        ymax=0.4,
        linewidth=2.0,
        color="red");
```



In the plot above the mean and median are both so close to zero that the red median line lies on top of the thicker black line drawn at the mean.

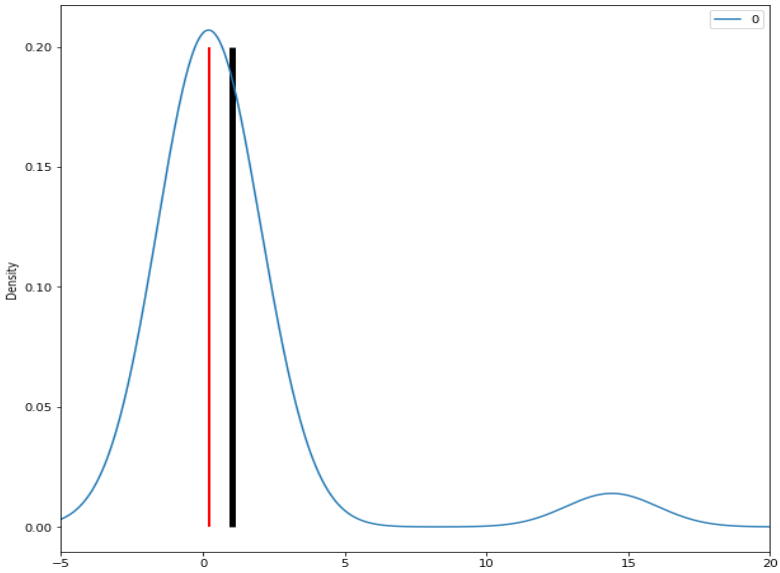
In skewed distributions, the mean tends to get pulled in the direction of the skew, while the median tends to resist the effects of skew:

```
skewed_data = pd.DataFrame(np.random.exponential(size=100000))

skewed_data.plot(kind="density",
           figsize=(10,10),
           xlim=(-1,5));


plt.vlines(skewed_data.mean(),     # Plot black line at mean
         ymin=0,
         ymax=0.8,
         linewidth=5.0);

plt.vlines(skewed_data.median(),   # Plot red line at median
         ymin=0,
         ymax=0.8,
         linewidth=2.0,
         color="red");
```



The mean is also influenced heavily by outliers, while the median resists the influence of outliers:

```
norm_data = np.random.normal(size=50)
outliers = np.random.normal(15, size=3)
combined_data = pd.DataFrame(np.concatenate((norm_data, outliers), axis=0))

combined_data.plot(kind="density",
           figsize=(10,10),
           xlim=(-5,20));


plt.vlines(combined_data.mean(),     # Plot black line at mean
          ymin=0,
          ymax=0.2,
          linewidth=5.0);

plt.vlines(combined_data.median(),   # Plot red line at median
          ymin=0,
          ymax=0.2,
          linewidth=2.0,
          color="red");
```



Since the median tends to resist the effects of skewness and outliers, it is known a "robust" statistic. The median generally gives a better sense of the typical value in a distribution with significant skew or outliers.

MODE:

The mode of a variable is simply the value that appears most frequently. Unlike mean and median, you can take the mode of a categorical variable and it is possible to have multiple modes. Find the mode with df.mode():

mtcars.mode()

mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
0	10.4	8.0	275.8	110.0	3.07	3.44	17.02	0.0	0.0	3.0
1	15.2	NaN	NaN	175.0	3.92	NaN	18.90	NaN	NaN	NaN
2	19.2	NaN	NaN	180.0	NaN	NaN	NaN	NaN	NaN	NaN
3	21.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
4	21.4	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
5	22.8	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
6	30.4	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

The columns with multiple modes (multiple values with the same count) return multiple values as the mode. Columns with no mode (no value that appears more than once) return NaN.

Measures of Spread

Measures of spread (dispersion) are statistics that describe how data varies. While measures of center give us an idea of the typical value, measures of spread give us a sense of how much the data tends to diverge from the typical value.

One of the simplest measures of spread is the range. Range is the distance between the maximum and minimum observations:

```
max(mtcars["mpg"]) - min(mtcars["mpg"])
```

```
23.5
```

As noted earlier, the median represents the 50th percentile of a data set. A summary of several percentiles can be used to describe a variable's spread. We can extract the minimum value (0th percentile), first quartile (25th percentile), median, third quartile(75th percentile) and maximum value (100th percentile) using the quantile() function:

```
five_num = [mtcars["mpg"].quantile(0),
        mtcars["mpg"].quantile(0.25),
        mtcars["mpg"].quantile(0.50),
        mtcars["mpg"].quantile(0.75),
        mtcars["mpg"].quantile(1)]

five_num
```

```
[10.4, 15.425, 19.2, 22.8, 33.9]
```

Since these values are so commonly used to describe data, they are known as the "five number summary". They are the same percentile values returned by df.describe():

```
mtcars["mpg"].describe()
```

```
count   32.000000
mean    20.090625
std      6.026948
min     10.400000
25%     15.425000
50%     19.200000
75%     22.800000
```

```
max     33.900000
Name: mpg, dtype: float64
```

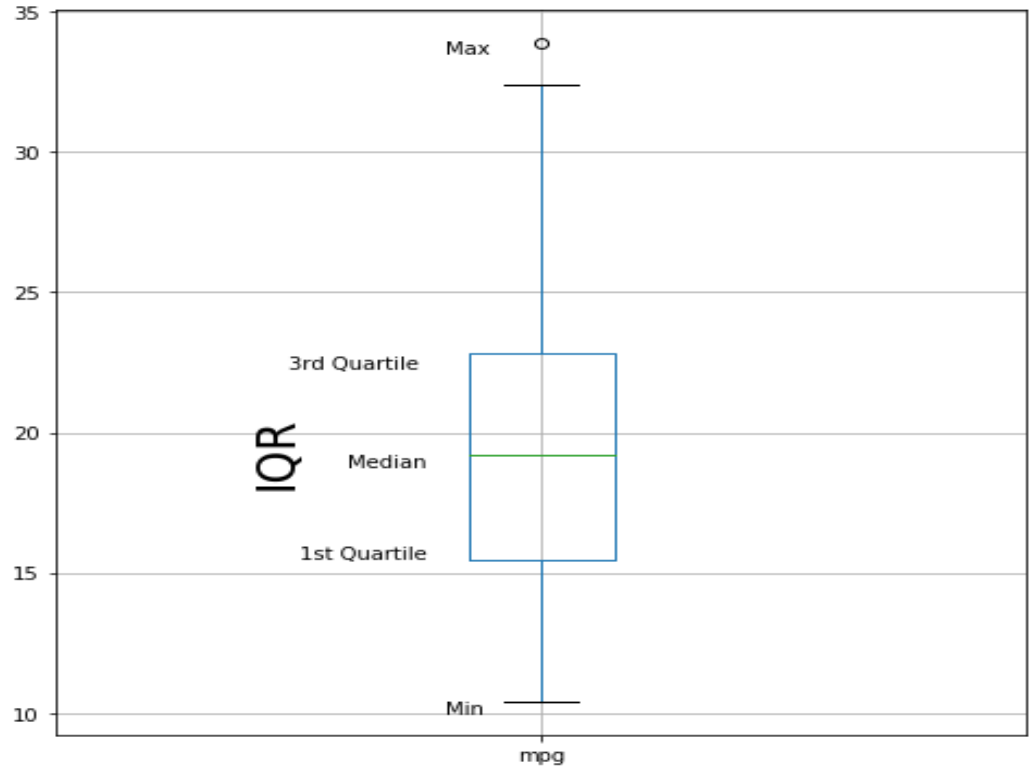
Interquartile (IQR) range is another common measure of spread. IQR is the distance between the 3rd quartile and the 1st quartile:

```
mtcars["mpg"].quantile(0.75) - mtcars["mpg"].quantile(0.25)
```

```
7.375
```

The boxplots we learned to create in the lesson on plotting are just visual representations of the five number summary and IQR:

```
mtcars.boxplot(column="mpg",
               return_type='axes',
               figsize=(8,8))

plt.text(x=0.74, y=22.25, s="3rd Quartile")
plt.text(x=0.8, y=18.75, s="Median")
plt.text(x=0.75, y=15.5, s="1st Quartile")
plt.text(x=0.9, y=10, s="Min")
plt.text(x=0.9, y=33.5, s="Max")
plt.text(x=0.7, y=19.5, s="IQR", rotation=90, size=25);
```



Measures of spread with Variance and standard deviation

Variance

Variance and standard deviation are two other common measures of spread.

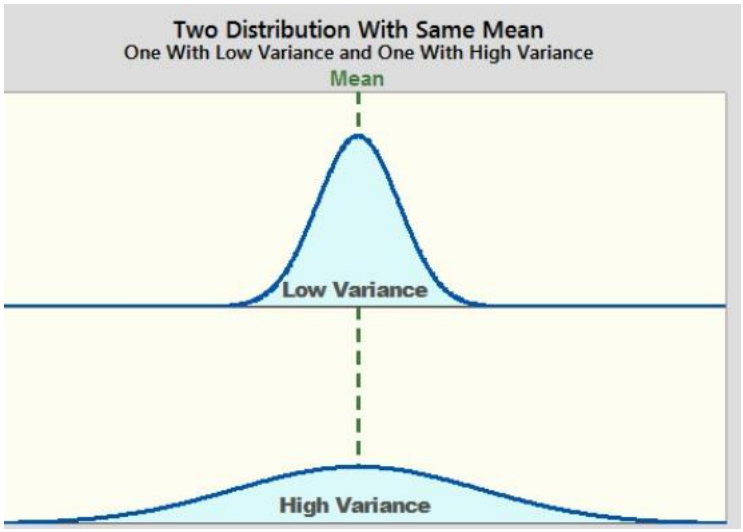
The variance of a distribution is the average of the squared deviations (differences) from the mean.

It quantifies the amount of variation or dispersion in a set of data points. It represents the average squared deviation of data points from the mean.

Use df.var() to check variance:

mtcars["mpg"].var()

6.026948052089105



Standard deviation

The standard deviation is the square root of the variance. Standard deviation can be more interpretable than variance, since the standard deviation is expressed in terms of the same units as the variable in question while variance is expressed in terms of units squared. It represents the average deviation or spread of data points from the mean.

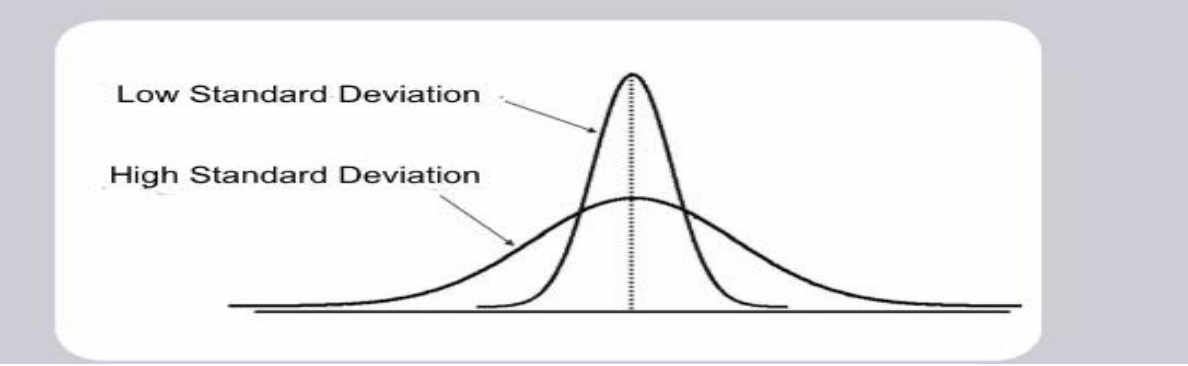
The standard deviation is a statistical measure that quantifies the amount of variation or dispersion in a set of data points.

A low standard deviation indicates that the data points tend to be close to the mean, while a high standard deviation suggests that the data points are more widely dispersed from the mean.

Use df.std() to check the standard deviation:

```
mtcars["mpg"].std()
```

6.026948052089105



Median absolute deviation

Since variance and standard deviation are both derived from the mean, they are susceptible to the influence of data skew and outliers. Median absolute deviation is an alternative measure of spread based on the median, which inherits the median's robustness against the influence of skew and outliers. It is the median of the absolute value of the deviations from the median:

```
abs_median_devs = abs(mtcars["mpg"] - mtcars["mpg"].median())
```

```
abs_median_devs.median() * 1.4826
```

```
5.411490000000001
```

Skewness and Kurtosis

descriptive statistics include measures that give you a sense of the shape of a distribution. Skewness measures the skew or asymmetry of a distribution while kurtosis measures how much data is in the tails of a distribution v.s. the center.

If the skewness is close to 0, it suggests that the data is approximately symmetric (normally distributed).

If the skewness is negative, it indicates a left-skewed (negatively skewed) distribution, meaning the tail on the left side is longer or fatter than the right side.

If the skewness is positive, it indicates a right-skewed (positively skewed) distribution, #meaning the tail on the right side is longer or fatter than the left side.

Kurtosis values can provide information about the shape of the data distribution:

If the kurtosis is close to 0, it indicates that the distribution has a similar shape to a normal distribution (mesokurtic).

If the kurtosis is negative (less than 0), it indicates a flatter distribution with lighter tails (platykurtic).

#If the kurtosis is positive (greater than 0), it indicates a more peaked distribution with #heavier tails (leptokurtic).

Pandas has built in functions for checking skewness and kurtosis, df.skew() and df.kurt() or df.kurtosis() respectively:

```
mtcars["mpg"].skew()  # Check skewness
```

Out[18]:

0.6723771376290805

In [19]:

```
mtcars["mpg"].kurt()  # Check kurtosis
```

Out[19]:

-0.0220062914240855