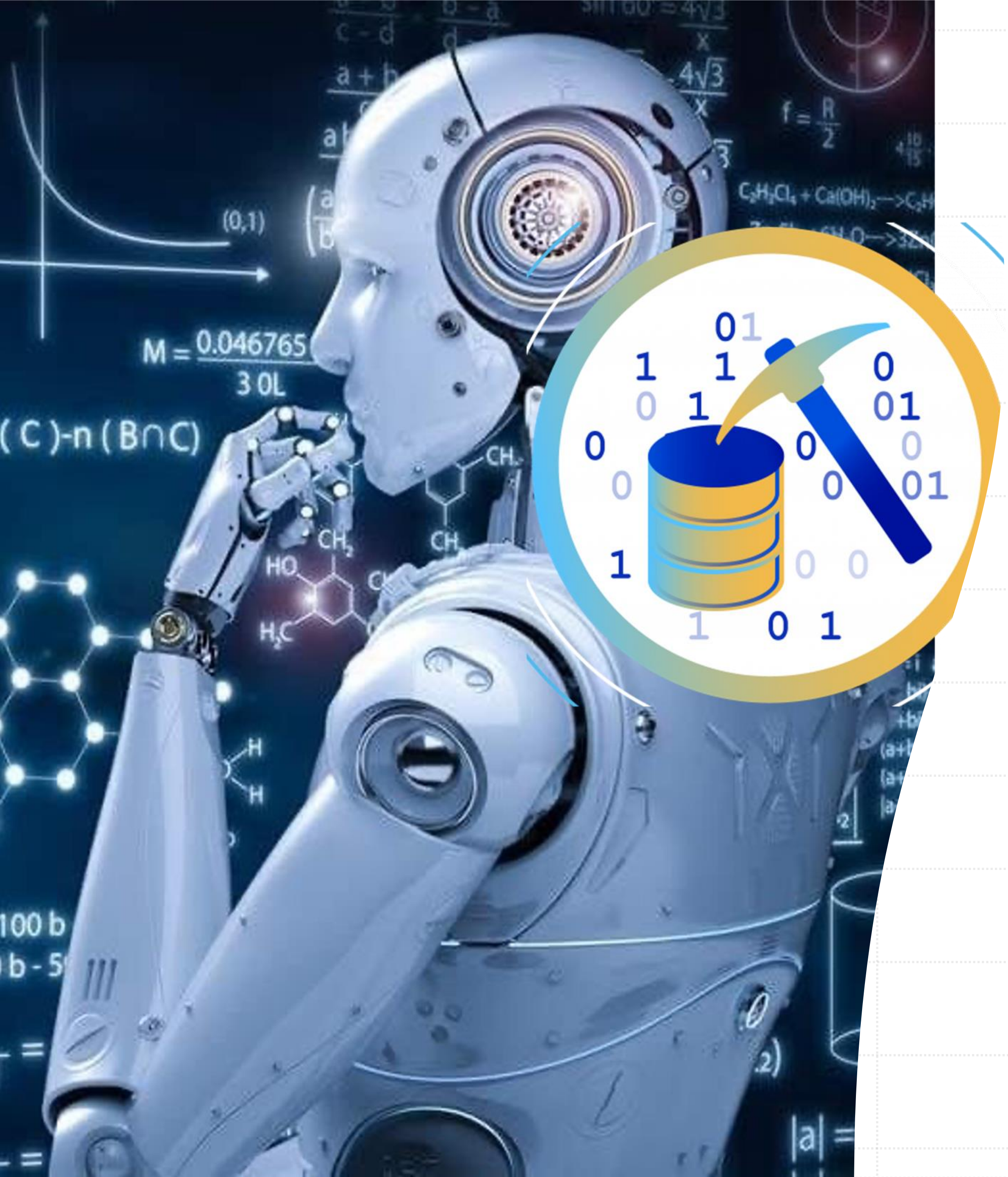


Module: Data Mining and Machine Learning

Lecture 2: Data Handling and Pre-processing

Dr. Eric Gyamfi

Assist. Prof. of Computing

School of Computing

egyamfi@staff.ncirl.ie

Lecture Outcome

- At the end of this lecture, student will:
  - Identify Issues in Data
  - Understand the causes of the issues
  - Understand their impact on the ML output
  - Understand Data Handling, preprocessing, and distribution

Sources of Data

- Internal Sources:
  - Sales records, CRM databases, inventory systems.
- External Sources:
  - Public datasets (e.g., Kaggle, UCI Machine Learning Repository).
  - APIs for real-time data (e.g., OpenWeather, Twitter).
  - Purchased datasets from third-party vendors.
- Generated Data:
  - IoT devices, sensors, and simulations

Data Repositories

- Data repository is a generic terminology that refers to a segmented dataset in a storage entity used for reporting or analysis.
- A data repository serves as a centralized storage facility for managing and storing various datasets
- Types of repository:
  - **Relational Databases (SQL):** Structured data stored in tables with relationships (e.g., MySQL, PostgreSQL)
  - **Non-Relational Databases (NoSQL):** Flexible schema for unstructured data (e.g., MongoDB, Cassandra)
  - **Cloud Storage**: Services for large-scale data storage (e.g., AWS S3, Google Cloud Storage)
  - **Data Lakes**: Centralized repositories for structured and unstructured data.

Relational Databases

- **Definition**: Databases that store data in structured tables with predefined relationships between them (based on rows and columns).
- **Characteristics**:
  - Structured Schema: Each table has a fixed structure.
  - Data Relationships: Managed using primary and foreign keys.
  - SQL (Structured Query Language): Used for querying and managing data.
  - ACID Compliance: Ensures reliability in transactions (Atomicity, Consistency, Isolation, Durability).
- **Popular Technologies**: MySQL, PostgreSQL, SQLite, Oracle DB, Microsoft SQL Server.
- **Use Cases**:
  - Financial systems (e.g., banking transactions).
  - Employee management systems (e.g., HR databases).
  - Inventory tracking.

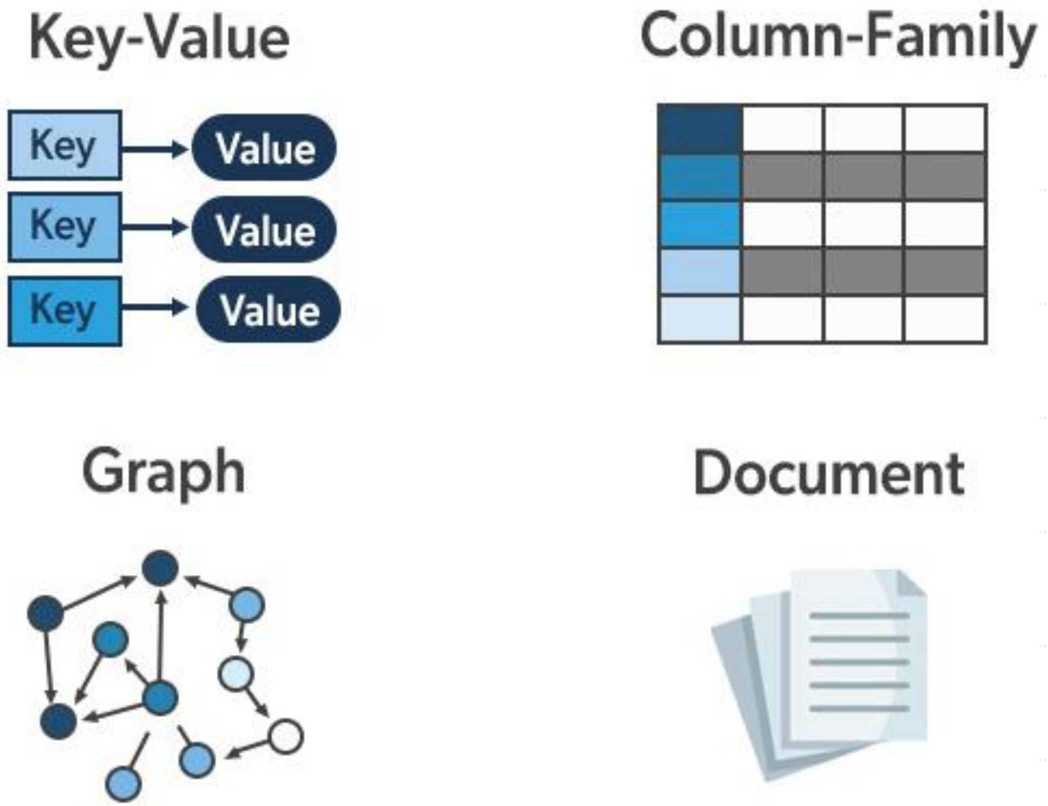
id	name	type	size
1	Rex	dog	big
2	Cleo	dog	small
3	Leroy	dog	medium
4	Sunny	cat	big
5	Snow Ball	cat	medium

id_pet	weight	age
1	32,3	7
2	4,35	13,6
3	24,9	4
5	8,75	5

name	food	danger_level
Cleo	diet	5
Sunny	wet	1
Rex	dry	4
Leroy	wet	3

Non-Relational Databases (NoSQL)

- **Definition**: Databases that store data in a flexible, often schema-less format, suitable for unstructured or semi-structured data.
- **Characteristics**:
  - **Schema Flexibility**: Can store data in various formats (key-value pairs, documents, graphs, etc.).
  - **Scalability**: Horizontal scaling for large data volumes.
  - **High Performance**: Optimized for fast writes and distributed data storage.
- **Types**:
  - **Document Databases**: MongoDB, Couchbase (store JSON-like documents).
  - **Key-Value Stores**: Redis, DynamoDB.
  - **Column-Family Stores**: Cassandra, HBase.
  - **Graph Databases**: Neo4j, Amazon Neptune.
- **Use Cases:**
  - Real-time analytics (e.g., log processing, IoT data).
  - E-commerce platforms (e.g., product catalogs).
  - Social media networks (e.g., managing relationships).



Importing Local Data Files

- Importing data from local files is one of the most common tasks in data analytics. Python supports importing various file formats using built-in libraries and external libraries like pandas.
- **Common Local File Formats**
  - **CSV (Comma-Separated Values)**: Tabular data stored as plain text.
  - **Excel Files**: Data stored in spreadsheets with multiple sheets (e.g., .xls, .xlsx).
  - **Text Files**: Unstructured or semi-structured data in .txt.
  - **JSON Files**: Semi-structured data stored in a lightweight, human-readable format.
  - **SQL Databases**: Data exported from relational database management systems.

```
import pandas as pd

# Import a CSV file
data = pd.read_csv('data/sales_data.csv')

# Display the first 5 rows
print(data.head())
```

```
import pandas as pd

# Import an Excel file (specify sheet name)
data = pd.read_excel('data/financial_data.xlsx', sheet_name='Q1')

# Display summary statistics
print(data.describe())
```

```
import pandas as pd

# Import a JSON file
data = pd.read_json('data/customer_data.json')

# Display the structure of the DataFrame
print(data.info())
```

```
# Read a text file line by line
with open('data/logs.txt', 'r') as file:
    lines = file.readlines()

# Print the first 5 lines
print(lines[:5])
```

Importing Data from APIs

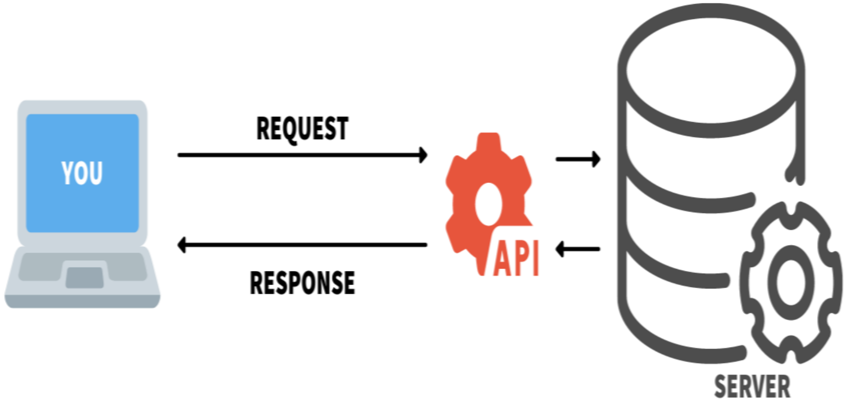
Definition: **An API (Application Programming Interface)** is a set of rules and protocols that allows one application to interact with another, enabling data exchange.

Types of APIs:

- **Public APIs**: Open to external developers (e.g., Twitter API, Google Maps API).
- **Private APIs**: Restricted access, often for internal use (e.g., company data systems).
- **RESTful APIs**: A common architectural style, typically HTTP-based, that uses standard methods like GET, POST, PUT, DELETE.

How APIs Work

- **Request**: The client sends a request to the server (e.g., via URL).
- **Response**: The server returns data in a structured format (usually JSON or XML).
- **Endpoints**: API paths that define what data or functionality is being requested (e.g., /users)



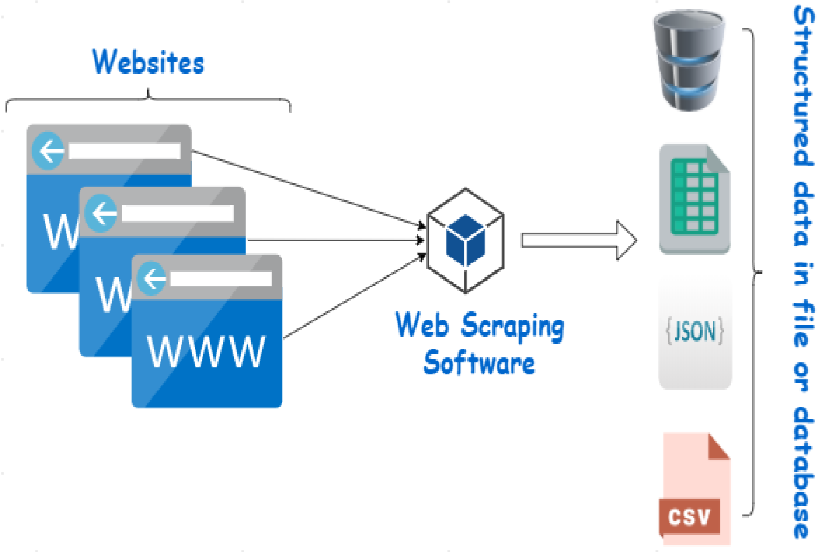

```
[1]: import requests
     import json
```

```
[2]: # Function to get live stock data for a symbol
     def get_stock_data():
         url = "https://www.alphavantage.co/query?function=TIME_SERIES_INTRADAY&symbol=IBM&interval=5min&outputsize=full&apikey=demo"
         response = requests.get(url)

         # Check if the response is successful
         if response.status_code == 200:
             data = response.json()
             last_refreshed = data["Meta Data"]["3. Last Refreshed"]
             price = data["Time Series (5min)"][last_refreshed]["1. open"]
             return price
         else:
             return None
```

```
[3]: stock_prices = {}
     price = get_stock_data()
     symbol = "IBM"
     if price is not None:
         stock_prices[symbol] = price

     print(f"{symbol}: {price}")

     IBM: 224.7800
```

Web Scraping

- **Definition**: Web scraping is the process of extracting data from websites by simulating human browsing behavior.
- **Use Cases**:
  - Collecting price data from e-commerce websites.
  - Gathering real-time news or social media data.
  - Extracting product information or reviews.
- **Web Scraping Workflow**
  - **Send a Request**: Use Python libraries (like requests) to request a web page.
  - **Parse the HTML**: Use libraries like BeautifulSoup to parse the HTML content.
  - **Extract Data**: Identify HTML tags (e.g., <div>, <span>, <a>) that contain the data of interest.
  - **Store Data**: Save the extracted data in a structured format (e.g., CSV, JSON, or database).

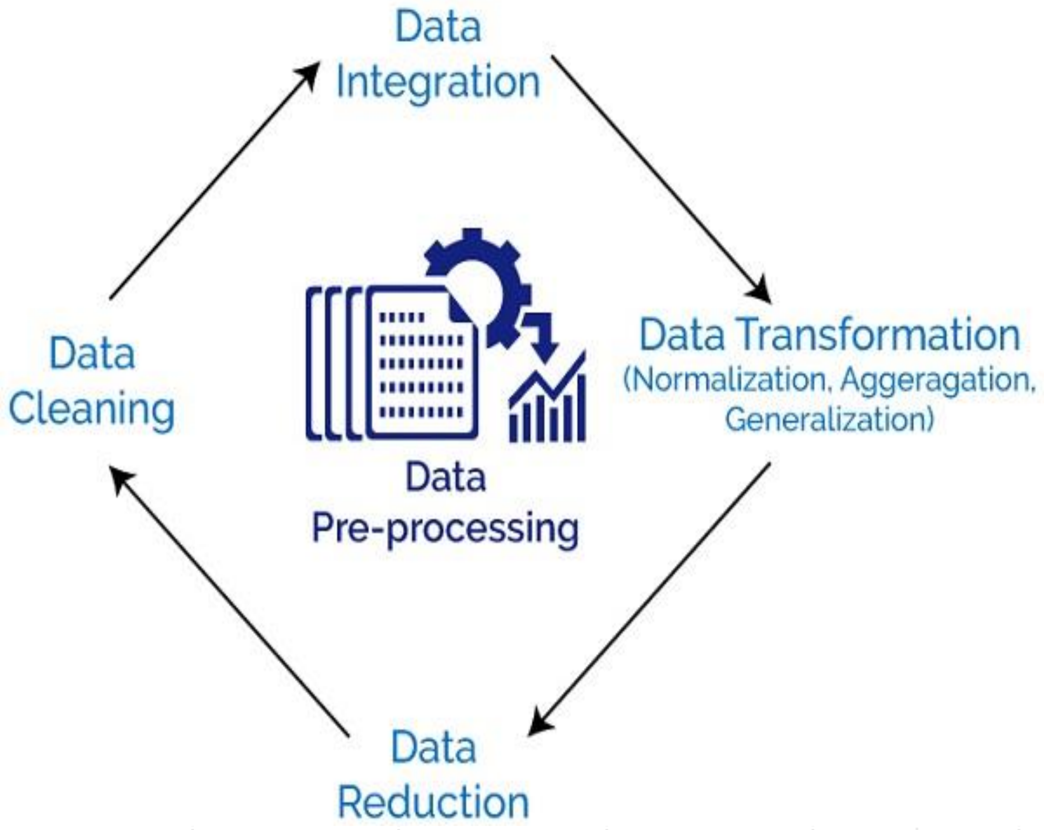



```
[4]: import requests
     from bs4 import BeautifulSoup
```

```
[6]: # Making a GET request
     r = requests.get('https://www.geeksforgeeks.org/python-programming-language/')

     # Parsing the HTML
     soup = BeautifulSoup(r.content, 'html.parser')

     s = soup.find('div', class_='entry-content')
     content = soup.find_all('p')

     print(content)
```

Data Preprocessing

- **Definition**: Data preprocessing is a crucial step in the data analysis pipeline that involves preparing and cleaning data for further analysis or machine learning models.
- **Purpose**:
  - Clean and format raw data.
  - Handle missing values, incorrect data types, and inconsistencies.
  - Standardize the data to make it suitable for analysis.
- **Why is it important?**
  - Ensures data integrity.
  - Reduces the noise and improves the accuracy of machine learning models.
  - Helps in transforming data into the right format (e.g., numerical, categorical).



Source: Electronics Media

Handling Missing Values in Data

- **Common Causes**:
  - Data entry errors.
  - Incomplete data collection.
  - Data corruption during transfer.

- **Methods to Handle Missing Data:**
- **Removing Missing Data**: If the missing data is insignificant.
- **Imputing Missing Data**: Filling in missing values with the mean, median, or mode of the column, or using advanced imputation methods (e.g., KNN, regression).
- **Leaving Missing Data**: Some machine learning models (like decision trees) can handle missing data.
- Note:
- .dropna() removes rows with any missing values.
- .fillna(df.mean()) fills missing values with the mean of the respective column.

[7]:
```
import pandas as pd
import numpy as np
```

[9]:
```
# Sample DataFrame with missing values
data = {'Name': ['Alice', 'Bob', 'Charlie', np.nan, 'Eve'],
        'Age': [25, np.nan, 30, 22, np.nan],
        'Salary': [50000, 55000, np.nan, 45000, 60000]}
# The DataFrame with missing values
print("Original Data:")
pd.DataFrame(data)
```

Original Data:

[9]:

	Name	Age	Salary
0	Alice	25.0	50000.0
1	Bob	NaN	55000.0
2	Charlie	30.0	NaN
3	NaN	22.0	45000.0
4	Eve	NaN	60000.0

[12]:
```
# 1. Drop rows with missing values
df_cleaned = df.dropna()
# Display results
print("\nData After Dropping Missing Values:")
df_cleaned
```

Data After Dropping Missing Values:

[12]:

	Name	Age	Salary
0	Alice	25.0	50000.0

[14]:
```
# 2. Impute missing values with mean for numerical columns
df_filled = df.fillna(df.mean(numeric_only=True))
print("\nData After Imputing Missing Values:")
df_filled
```

Data After Imputing Missing Values:

[14]:

	Name	Age	Salary
0	Alice	25.000000	50000.0
1	Bob	25.666667	55000.0
2	Charlie	30.000000	52500.0
3	NaN	22.000000	45000.0
4	Eve	25.666667	60000.0

Removing Duplicates

- *.drop_duplicates() removes rows with identical values across all columns.*

```
[25]: # Sample DataFrame with duplicate rows
data = {'Name': ['Alice', 'Bob', 'Alice', 'Charlie', 'Bob'],
        'Age': [25, 30, 25, 30, 30]}

df = pd.DataFrame(data)

# Show original data
print("Original Data:")
df
```

Original Data:

[25]:

	Name	Age
0	Alice	25
1	Bob	30
2	Alice	25
3	Charlie	30
4	Bob	30

```
[26]: # Remove duplicate rows
df_no_duplicates = df.drop_duplicates()

# Display cleaned data
print("\nData After Removing Duplicates:")
df_no_duplicates
```

Data After Removing Duplicates:

[26]:

	Name	Age
0	Alice	25
1	Bob	30
3	Charlie	30

Types of Missing Data

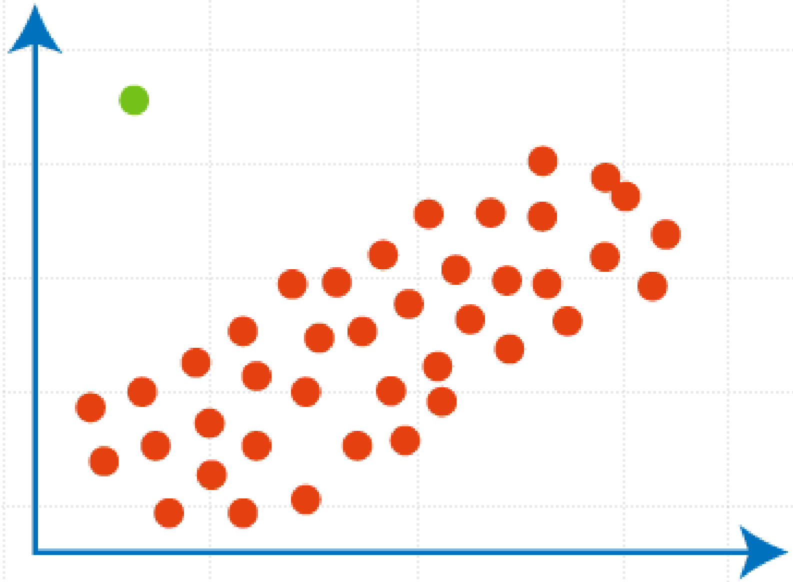
- There are three main types of missing data.
  - Missing Completely At Random (MCAR):
    - When data are MCAR, the fact that the data are missing is independent of the observed and unobserved data.
  - Missing At Random (MAR):
    - When data are MAR, the fact that the data are missing is systematically related to the observed but not the unobserved data
  - Missing Not At Random (MNAR):
    - When data are MNAR, the fact that the data are missing is systematically related to the unobserved data, that is, the missingness is related to events or factors which are not measured by the user.

Handling Outliers

- **Definition**: Outliers are data points significantly different from other observations in a dataset.
- **Characteristics**:
  - They deviate markedly from the central values.
  - Can be caused by measurement errors, data entry errors, or genuine variability in data.
- **Why Handle Outliers?**
  - Outliers can skew statistical measures like mean and standard deviation.
  - They can negatively affect machine learning models, especially those sensitive to scale (e.g., linear regression, KNN).

- **Methods to Detect Outliers**

1. **Statistical Methods**:
   1. **Z-Score**:
      1. Measures how many standard deviations a data point is from the mean.
      2. Threshold: |Z| > 3 is often considered an outlier.
   2. **IQR (Interquartile Range)**:
      1. Based on the spread of the middle 50% of data (Q1–Q3).
      2. Formula:. $\text{Outlier} > Q3 + 1.5 \times IQR$ or $\text{Outlier} < Q1 - 1.5 \times IQR$.



2. **Visualization**:

    1. Boxplots.
    2. Scatterplots.

Methods to Handle Outliers

- **Removal**: Drop the outliers if they are due to errors or irrelevant to analysis.
- **Transformation**: Apply logarithmic or square root transformations to reduce the effect of outliers.
- **Capping/Truncation**: Replace extreme values with the maximum/minimum acceptable range.

Sample Code

Detecting Outliers with Z-Score

[16]:
```python
import numpy as np
import pandas as pd

# Sample data
data = {'Value': [10, 12, 15, 18, 20, 22, 100]}  # 100 is an outlier
df = pd.DataFrame(data)

# Calculate Z-scores
df['Z-Score'] = (df['Value'] - df['Value'].mean()) / df['Value'].std()

# Identify outliers
outliers = df[df['Z-Score'].abs() > 3]

print("Outliers detected using Z-Score:")
print(outliers)
```

```
Outliers detected using Z-Score:
Empty DataFrame
Columns: [Value, Z-Score]
Index: []
```

Detecting Outliers with IQR

[18]:
```python
Q1 = df['Value'].quantile(0.25)  # 25th percentile
Q3 = df['Value'].quantile(0.75)  # 75th percentile
IQR = Q3 - Q1  # Interquartile range

# Define outlier boundaries
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR

# Identify outliers
outliers = df[(df['Value'] < lower_bound) | (df['Value'] > upper_bound)]

print("\nOutliers detected using IQR:")
outliers
```

Outliers detected using IQR:

[18]:

	Value	Z-Score
6	100	2.247575

Handling Outliers by Capping

[19]:
```python
# Replace outliers with boundary values
df['Capped_Value'] = np.where(df['Value'] > upper_bound, upper_bound,
                              np.where(df['Value'] < lower_bound, lower_bound, df['Value']))

print("\nData after capping outliers:")
df
```

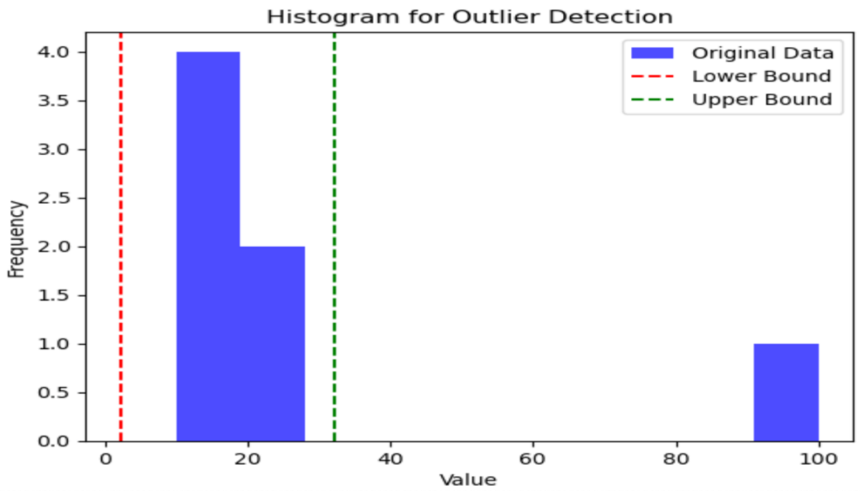
Data after capping outliers:

[19]:

	Value	Z-Score	Capped_Value
0	10	-0.567479	10.00
1	12	-0.504922	12.00
2	15	-0.411087	15.00
3	18	-0.317252	18.00
4	20	-0.254695	20.00
5	22	-0.192139	22.00
6	100	2.247575	32.25

Visualization

[21]:
```python
# Plot histogram to visualize the data distribution
plt.hist(df['Value'], bins=10, alpha=0.7, color='blue', label='Original Data')
plt.axvline(lower_bound, color='red', linestyle='--', label='Lower Bound')
plt.axvline(upper_bound, color='green', linestyle='--', label='Upper Bound')
plt.title("Histogram for Outlier Detection")
plt.xlabel("Value")
plt.ylabel("Frequency")
plt.legend()
plt.show()
```



Data Transformation Techniques

- **Categorical Data Encoding**:
  - **Label Encoding**: Convert categories into numeric labels (useful for ordinal categories).
  - **One-Hot Encoding**: Create binary columns for each category (useful for nominal categories).
- **Feature Extraction**: Converting raw data into meaningful features, such as extracting year, month, and day from a date.

- **Note**:
  - **Label Encoding**: Converts 'Red' to 0, 'Blue' to 1, and 'Green' to 2.
  - **One-Hot Encoding**: Creates separate columns for each category, indicating presence with a 1 or 0.

```python
from sklearn.preprocessing import LabelEncoder, OneHotEncoder
import pandas as pd
# Sample DataFrame with categorical data
data = {'Color': ['Red', 'Blue', 'Green', 'Blue', 'Red']}
df = pd.DataFrame(data)

# Label Encoding
label_encoder = LabelEncoder()
df['Color_Label'] = label_encoder.fit_transform(df['Color'])

# One-Hot Encoding
df_encoded = pd.get_dummies(df, columns=['Color'], prefix='Color')

# Display results
print("\nLabel Encoded Data:")
print(df)

print("\nOne-Hot Encoded Data:")
print(df_encoded)
```

```
Label Encoded Data:
   Color  Color_Label
0    Red            2
1   Blue            0
2  Green            1
3   Blue            0
4    Red            2

One-Hot Encoded Data:
   Color_Label  Color_Blue  Color_Green  Color_Red
0            2       False        False       True
1            0        True        False      False
2            1       False         True      False
3            0        True        False      False
4            2       False        False       True
```

Feature Scaling Techniques

- **Feature Scaling** is a technique to standardize the independent features present in the data. It is performed during the data pre-processing to handle highly varying values.
- Algorithms like **KNN** and **SVM** are sensitive to the scale of features.
- Features with larger ranges can dominate the learning process.
- **Methods**:
  - **Standardization (Z-score normalization)**: Converts data to have a mean of 0 and a standard deviation of 1.
  - **Normalization (Min-Max scaling)**: Scales data to a fixed range (e.g., between 0 and 1).

```python
from sklearn.preprocessing import StandardScaler, MinMaxScaler
import pandas as pd

# Sample DataFrame with numerical data
data = {'Height': [5.5, 6.0, 5.8, 6.2, 5.4],
        'Weight': [150, 180, 160, 190, 145]}

df = pd.DataFrame(data)

# Standardization
scaler = StandardScaler()
df_standardized = scaler.fit_transform(df)

# Min-Max Scaling
scaler_minmax = MinMaxScaler()
df_normalized = scaler_minmax.fit_transform(df)

# Display results
print("\nStandardized Data:")
print(pd.DataFrame(df_standardized, columns=['Height', 'Weight']))

print("\nNormalized Data:")
print(pd.DataFrame(df_normalized, columns=['Height', 'Weight']))
```

```
Standardized Data:
     Height    Weight
0 -0.935414 -0.866025
1  0.734968  0.866025
2  0.066815 -0.288675
3  1.403122  1.443376
4 -1.269491 -1.154701

Normalized Data:
   Height    Weight
0   0.125  0.111111
1   0.750  0.777778
2   0.500  0.333333
3   1.000  1.000000
4   0.000  0.000000
```

Feature Reduction

- **Definition**: Reducing the number of features while retaining as much information as possible.
- **Purpose**:
  - Reduce computational cost.
  - Prevent overfitting by eliminating irrelevant features.
  - Visualize high-dimensional data in 2D or 3D.
- **Principal Component Analysis (PCA)**:Reduces dimensionality by transforming data into a smaller number of uncorrelated components.
- **Linear Discriminant Analysis (LDA)**:Focuses on finding linear combinations of features that best separate classes.
- **Feature Extraction**: Create new features that capture the essence of existing data.

```python
from sklearn.decomposition import PCA
import pandas as pd

# Sample data
data = {'Feature1': [2, 4, 6, 8],
        'Feature2': [1, 3, 5, 7],
        'Feature3': [2, 4, 6, 8]}

df = pd.DataFrame(data)

# Apply PCA to reduce dimensions to 2 components
pca = PCA(n_components=2)
reduced_data = pca.fit_transform(df)

# Display results
print("Original Data:")
print(df)

print("\nReduced Data:")
print(reduced_data)
```

```
Original Data:
   Feature1  Feature2  Feature3
0         2         1         2
1         4         3         4
2         6         5         6
3         8         7         8

Reduced Data:
[[ 5.19615242  0.        ]
 [ 1.73205081  0.        ]
 [-1.73205081  0.        ]
 [-5.19615242 -0.        ]]
```

Feature Selection

- **Feature selection** is the process by which we select a subset (Most important) of input features from the data for a model to reduce noise.
- **Purpose**:
  - Improve model interpretability.
  - Reduce training time.
  - Eliminate noise from irrelevant features.
- **Common Feature Selection Techniques**

1. **Filter Methods**:
   1. Statistical tests (e.g., chi-squared test, ANOVA).
2. **Wrapper Methods**:
   1. Recursive Feature Elimination (RFE).
   2. Stepwise feature selection.
3. **Embedded Methods**:
   1. Feature importance from models like decision trees or random forests.

```python
from sklearn.ensemble import RandomForestClassifier
import pandas as pd

# Sample data
data = {'Age': [25, 32, 47, 51, 62],
        'Salary': [50000, 60000, 80000, 85000, 120000],
        'Purchased': [0, 1, 0, 1, 1]}

df = pd.DataFrame(data)

# Split features and target
X = df[['Age', 'Salary']]
y = df['Purchased']

# Fit Random Forest to calculate feature importance
model = RandomForestClassifier()
model.fit(X, y)

# Display feature importance
importance = model.feature_importances_
for feature, score in zip(X.columns, importance):
    print(f"{feature}: {score}")
```

```
Age: 0.5221631205673759
Salary: 0.47783687943262415
```

Descriptive Statistics

- **Definition**: Descriptive statistics summarize and describe the main features of a dataset.
- **Purpose**:
  - Provide insights into data distribution.
  - Measure central tendency and variability.

Key Metrics:

1. **Mean**: Average value of the data.

$$\text{Mean} = \frac{\text{Sum of all values}}{\text{Number of values}}$$

2. **Median**: Middle value in sorted data.
3. **Mode**: Most frequently occurring value.
4. **Variance**: Measures data spread (average squared deviation from the mean).

$$\text{Variance} = \frac{\sum (x - \text{Mean})^2}{N}$$

5. **Standard Deviation**: Square root of variance (how much data deviates from the mean).

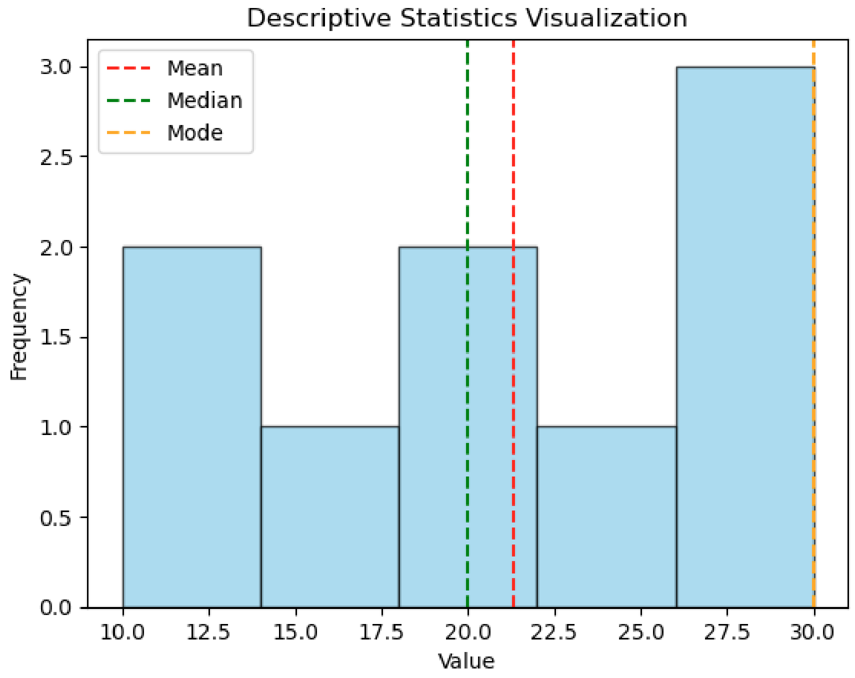
$$\text{Standard Deviation} = \sqrt{\text{Variance}}$$

Sample Code

[27]:
```python
import numpy as np
import statistics as stats
import pandas as pd

# Sample data
data = [10, 12, 15, 20, 20, 25, 30, 30, 30]

# Convert to DataFrame
df = pd.DataFrame(data, columns=['Value'])

# Mean
mean = np.mean(df['Value'])

# Median
median = np.median(df['Value'])

# Mode
mode = stats.mode(df['Value'])

# Variance
variance = np.var(df['Value'], ddof=0)  # Population variance

# Standard Deviation
std_dev = np.std(df['Value'], ddof=0)  # Population standard deviation

# Print Results
print(f"Mean: {mean}")
print(f"Median: {median}")
print(f"Mode: {mode}")
print(f"Variance: {variance}")
print(f"Standard Deviation: {std_dev}")
```

```
Mean: 21.333333333333332
Median: 20.0
Mode: 30
Variance: 55.333333333333334
Standard Deviation: 7.438637868140466
```

[29]:
```python
import matplotlib.pyplot as plt

# Plot histogram
plt.hist(df['Value'], bins=5, color='skyblue', alpha=0.7, edgecolor='black')

# Add Mean, Median, and Mode lines
plt.axvline(mean, color='red', linestyle='--', label='Mean')
plt.axvline(median, color='green', linestyle='--', label='Median')
plt.axvline(mode, color='orange', linestyle='--', label='Mode')

# Add labels
plt.title('Descriptive Statistics Visualization')
plt.xlabel('Value')
plt.ylabel('Frequency')
plt.legend()
plt.show()
```

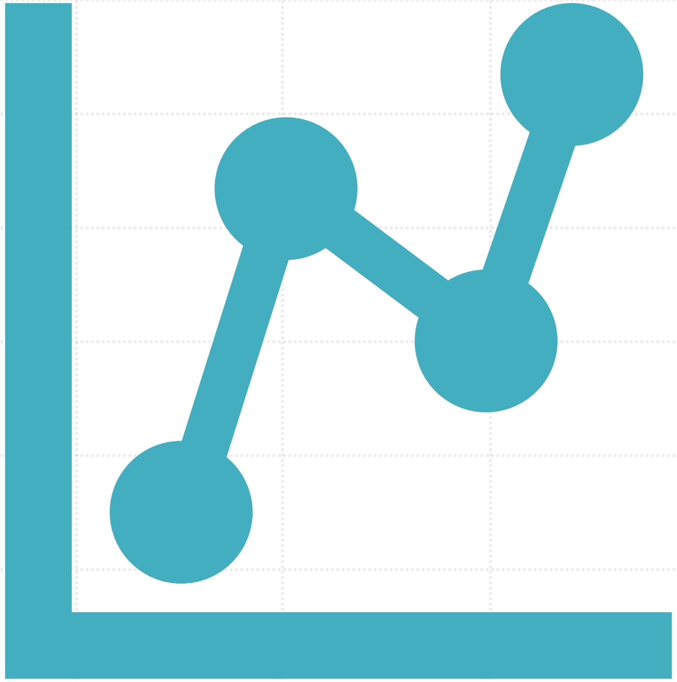


Visualizing Data Distributions

Importance: Helps understand data spread, skewness, and patterns.

Key Visualization Methods:

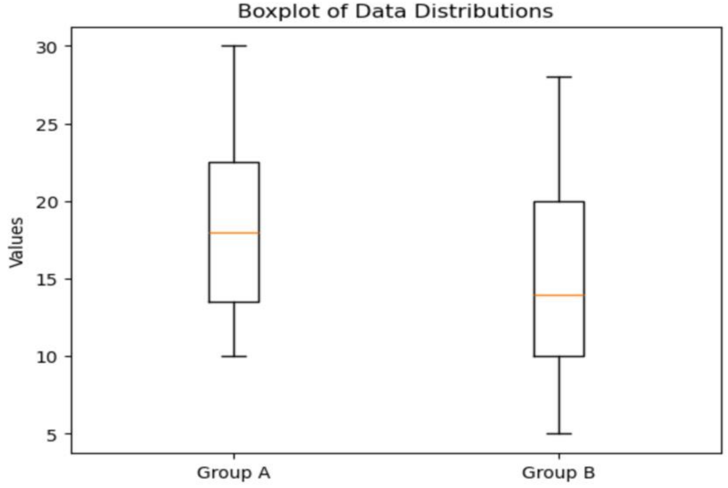
1. **Boxplots**: Show data spread, quartiles, and outliers.
2. **Histograms**: Represent frequency distribution of data.
3. **Density Plots**: Smooth approximation of the data distribution.
4. **Comparing Distributions**: Overlay plots or use side-by-side comparisons.



Sample Code

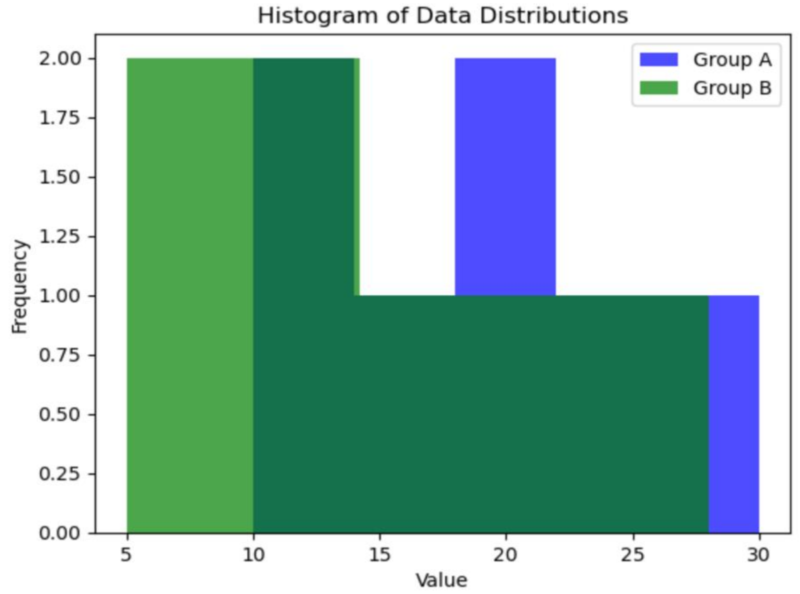
Box Plot

```
[30]: import matplotlib.pyplot as plt
      import pandas as pd

      # Sample data
      data = {'Group A': [10, 12, 15, 18, 20, 25, 30],
              'Group B': [5, 8, 12, 14, 18, 22, 28]}
      df = pd.DataFrame(data)

      # Create Boxplot
      plt.boxplot([df['Group A'], df['Group B']], labels=['Group A', 'Group B'])
      plt.title('Boxplot of Data Distributions')
      plt.ylabel('Values')
      plt.show()
```



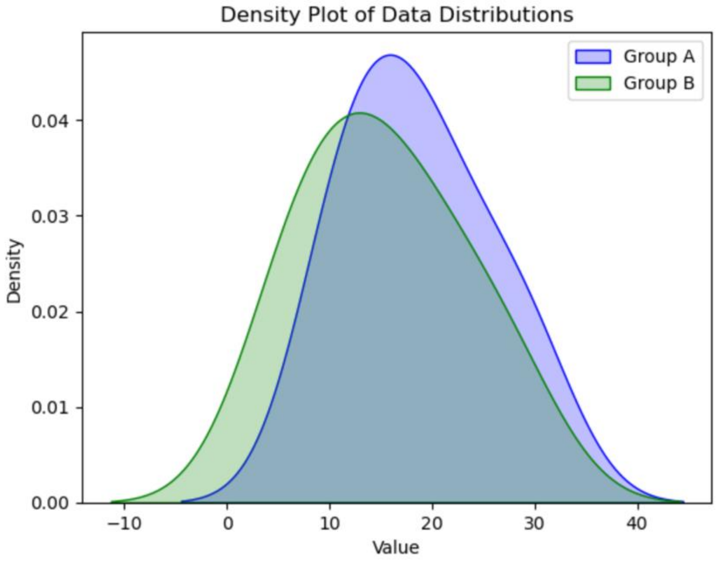
Histogram

```
[31]: plt.hist(df['Group A'], bins=5, alpha=0.7, color='blue', label='Group A')
      plt.hist(df['Group B'], bins=5, alpha=0.7, color='green', label='Group B')
      plt.title('Histogram of Data Distributions')
      plt.xlabel('Value')
      plt.ylabel('Frequency')
      plt.legend()
      plt.show()
```



Density Plot

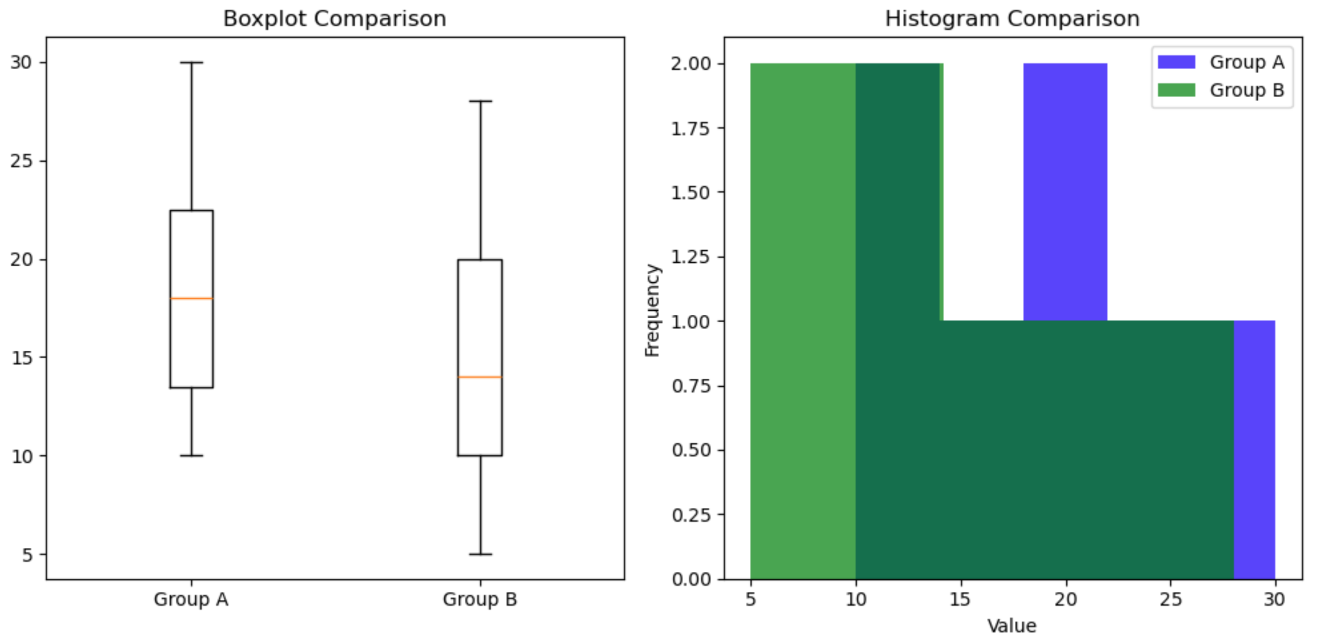
```
[32]: import seaborn as sns

      # Create Density Plot
      sns.kdeplot(df['Group A'], color='blue', label='Group A', shade=True)
      sns.kdeplot(df['Group B'], color='green', label='Group B', shade=True)
      plt.title('Density Plot of Data Distributions')
      plt.xlabel('Value')
      plt.ylabel('Density')
      plt.legend()
      plt.show()
```



Techniques for Comparing Distributions

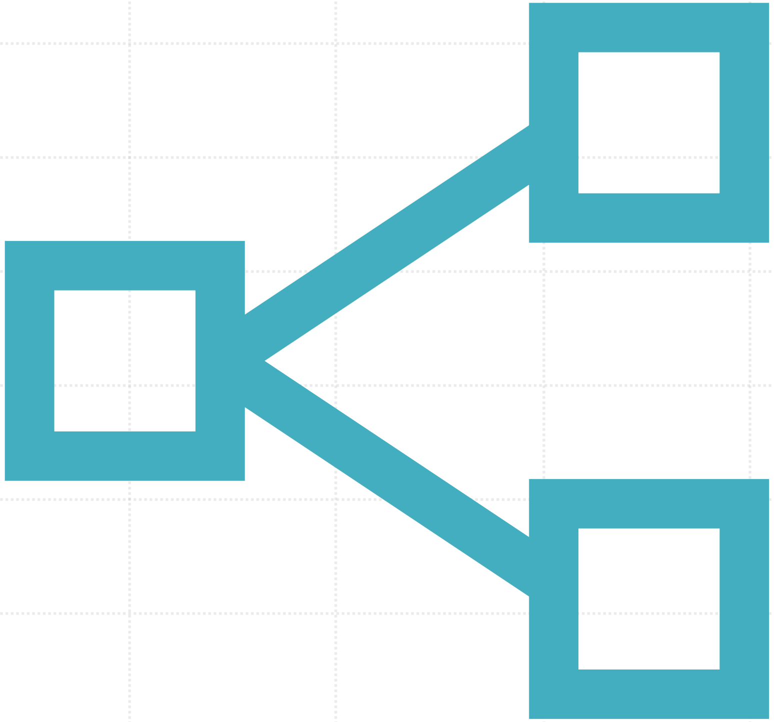
- **Overlapping Visualizations**
  - Combine histograms or density plots of multiple datasets to highlight similarities or differences.
- **Side-by-Side Boxplots**
  - Useful for comparing central tendencies and variability across groups.

```
# Boxplot
plt.subplot(1, 2, 1)
plt.boxplot([df['Group A'], df['Group B']], labels=['Group A', 'Group B'])
plt.title('Boxplot Comparison')

# Histogram
plt.subplot(1, 2, 2)
plt.hist(df['Group A'], bins=5, alpha=0.7, color='blue', label='Group A')
plt.hist(df['Group B'], bins=5, alpha=0.7, color='green', label='Group B')
plt.title('Histogram Comparison')
plt.xlabel('Value')
plt.ylabel('Frequency')
plt.legend()

plt.tight_layout()
plt.show()
```



Data Relationships

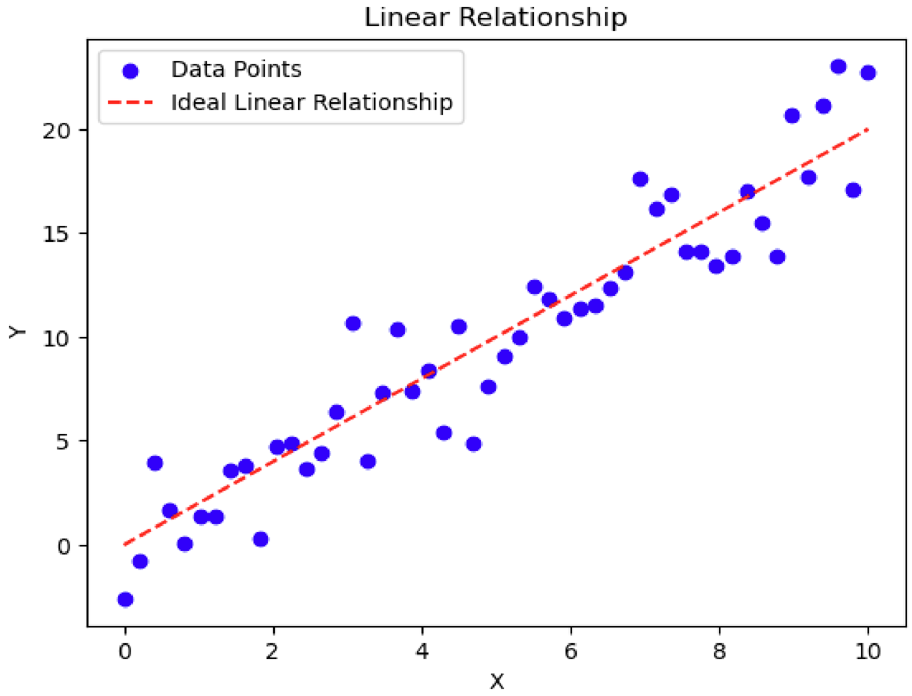
- **Definition**: Relationships show how variables influence each other.
- **Importance**:
  - Identify patterns and correlations.
  - Guide decision-making and predictive modeling.
- **Types of Relationships**:
  1. **Linear Relationship**: Straight-line correlation (positive or negative).
  2. **Non-Linear Relationship**: Variables relate in a curved or complex way.
  3. **Categorical Relationship**: Interaction between categorical variables.
  4. **Quantifying Relationships**: Use metrics like correlation coefficients.
     - Use **Pearson Correlation** for linear relationships.
     - Explore heatmaps for multi-variable correlation.



Sample Code

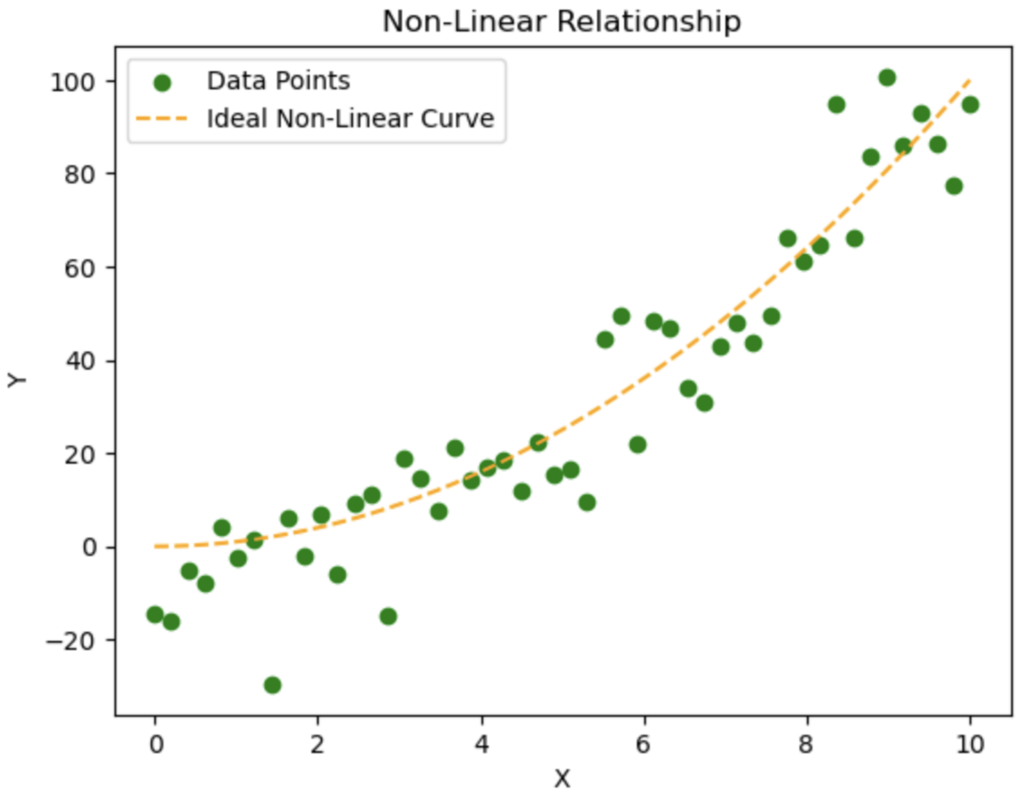
Linear Relationship

```python
import numpy as np
import matplotlib.pyplot as plt

# Generate synthetic linear data
x = np.linspace(0, 10, 50)
y = 2 * x + np.random.normal(0, 2, 50)

# Scatterplot
plt.scatter(x, y, color='blue', label='Data Points')
plt.plot(x, 2 * x, color='red', linestyle='--', label='Ideal Linear Relationship')
plt.title('Linear Relationship')
plt.xlabel('X')
plt.ylabel('Y')
plt.legend()
plt.show()
```



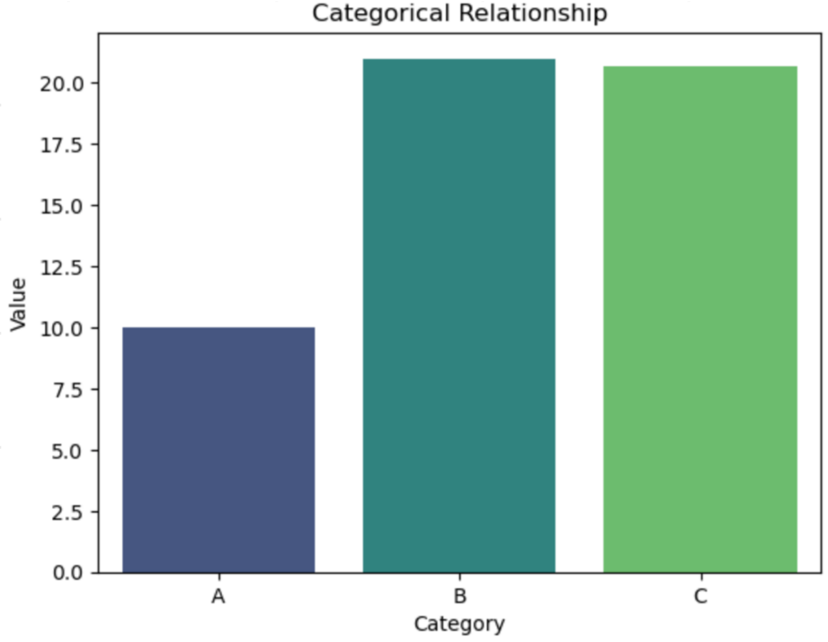
Non-Linear Relationship

```python
x = np.linspace(0, 10, 50)
y = x**2 + np.random.normal(0, 10, 50)

# Scatterplot
plt.scatter(x, y, color='green', label='Data Points')
plt.plot(x, x**2, color='orange', linestyle='--', label='Ideal Non-Linear Curve')
plt.title('Non-Linear Relationship')
plt.xlabel('X')
plt.ylabel('Y')
plt.legend()
plt.show()
```



Categorical Relationship

```
import seaborn as sns
import pandas as pd

# Create sample categorical data
data = {'Category': ['A', 'B', 'A', 'C', 'B', 'C', 'A', 'B', 'C'],
        'Value': [10, 20, 15, 10, 25, 30, 5, 18, 22]}
df = pd.DataFrame(data)

# Bar Plot
sns.barplot(x='Category', y='Value', data=df, ci=None, palette='viridis')
plt.title('Categorical Relationship')
plt.xlabel('Category')
plt.ylabel('Value')
plt.show()
```



Quantifying Data Relationship with Correlation

```
np.random.seed(42)
x = np.random.rand(50) * 10
y = 2 * x + np.random.normal(0, 5, 50)
z = np.random.rand(50) * 20

# Create DataFrame
df = pd.DataFrame({'X': x, 'Y': y, 'Z': z})

# Correlation Matrix
correlation_matrix = df.corr()

# Heatmap
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm')
plt.title('Correlation Matrix')
plt.show()
```