Python for Data Science: A Comprehensive Guide to Analysis, Machine Learning, and Big Data

Nikita Kanhu Das

June 20, 2025

Contents

1 Introduction to Data Science with Python

1.1 What is Data Science?

Data Science is a multidisciplinary domain of study that includes statistical analysis, machine learning, data visualization, and computer programming to infer useful insights from data. Data Science encompasses different processes such as data collection, cleaning, exploration, modeling, and interpretation. Data Science is used on a large scale in various domains like healthcare, finance, marketing, and artificial intelligence to inform data-driven decisions.

Key Data Science Elements:

- **Data Collection**: Collecting raw data from varied sources.
- **Data Purification**: Cleansing and sanitizing data for analysis.
- **Exploratory Data Analysis (EDA)**: A preview of patterns, trends, and distributions of the data.
- **Machine Learning**: Developing predictive models using algorithms.
- **Data Visualization**: Turning data insights into graphs and charts.
- **Conclusion Formation & Decision Making**: Creating conclusions from data insights to plan.

1.2 Why Use Python for Data Science?

Python has emerged as the top programming language for Data Science because it is easy, flexible, and encompasses a huge universe of libraries specific to data manipulation, analysis, and machine learning.

Benefits of Python for Data Science:

- **Ease of Use**: Python's syntax is easy and readable, and hence it is ideal for beginners.
- **Extensive Libraries**: Dense collection of libraries for data manipulation, visualization, and machine learning.
- **Strong Community Support**: Big open-source community with ongoing enhancements and support.
- **Integration Capabilities**: Smooth integration with databases, big data solutions, and cloud infrastructure.
- **Scalability**: Adaptable to small-scale tests and big-scale enterprise uses.

1.3 Popular Python Libraries for Data Science

Python provides numerous libraries that help Data Science activities such as data manipulation, data visualization, and machine learning.

‘ **Some Popular Python Libraries**:

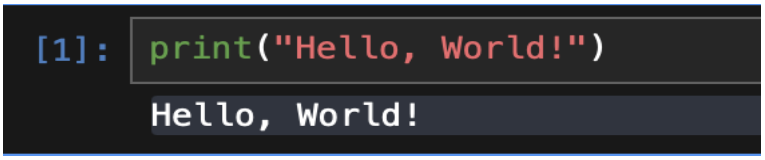
- **NumPy**: Employs large multi-dimensional matrices and arrays, together with mathematical operations.
- **Pandas**: Offers data structures such as DataFrames for processing structured data in an efficient way.
- **Matplotlib & Seaborn**: Data animated, static, and interactive plotting libraries.
- **Scikit-learn**: Machine learning library for offering support for algorithms of classification, regression, and clustering.
- **TensorFlow & PyTorch**: Libraries for building and training deep learning neural networks.

- **Statsmodels**: Statistical modeling and testing using hypothesis.
- **NLTK & spaCy**: NLP activities such as tokenizing and sentiment analysis libraries.
- **SciPy**: Offers higher-level mathematical functions and signal processing.
- **Dask & Vaex**: Big data processing libraries using Python.

Python remains one of the most shaping and popular programming languages, adapting itself with emerging technologies and uses.

1.4 Example of Python Code

Here's a simple Python program that prints "Hello, World!":



2 Python Libraries for Data Science

Data science is based on a set of robust Python libraries that support effective data manipulation, analysis, visualization, and machine learning. The next section gives an overview of some of the most important libraries used in data science and their applications.

2.1 NumPy – Numerical Computing

- NumPy, or Numerical Python, is a core library for numerical computation in Python.
- It offers support for large multi-dimensional matrices and arrays, as well as a set of mathematical functions to perform operations on the arrays.
- NumPy increases performance by using an optimized C-based implementation, which facilitates high-speed numerical computation.
- **Major Features**:
  - Support for multi-dimensional arrays (ndarray).
  - Mathematical and statistical functionalities.
  - Linear algebra and Fourier transform functionality.
  - Random number generation.
  - Interoperability with other scientific libraries.

```python
import numpy as np

# 1. Creating a Multi-Dimensional Array
arr = np.array([[1, 2, 3], [4, 5, 6]])
print("1. Multi-Dimensional Array (ndarray):\n", arr)

# 2. Mathematical Operations
arr_squared = np.square(arr)
print("\n2. Element-wise Squaring:\n", arr_squared)

# 3. Statistical Functions
mean_value = np.mean(arr)
std_dev = np.std(arr)
print("\n3. Mean:", mean_value, " | Standard Deviation:", std_dev)

# 4. Linear Algebra (Matrix Multiplication)
matrix1 = np.array([[1, 2], [3, 4]])
matrix2 = np.array([[5, 6], [7, 8]])
matrix_product = np.dot(matrix1, matrix2)
print("\n4. Matrix Multiplication:\n", matrix_product)

# 5. Fourier Transform (FFT)
signal = np.array([1, 2, 3, 4])
fft_signal = np.fft.fft(signal)
print("\n5. Fourier Transform:\n", fft_signal)

# 6. Random Number Generation
random_array = np.random.randint(1, 10, (3, 3))
print("\n6. Randomly Generated 3x3 Array:\n", random_array)
```

```
1. Multi-Dimensional Array (ndarray):
 [[1 2 3]
 [4 5 6]]

2. Element-wise Squaring:
 [[ 1  4  9]
 [16 25 36]]

3. Mean: 3.5  | Standard Deviation: 1.707825127659933

4. Matrix Multiplication:
 [[19 22]
 [43 50]]

5. Fourier Transform:
 [10.+0.j -2.+2.j -2.+0.j -2.-2.j]

6. Randomly Generated 3x3 Array:
 [[1 4 3]
 [6 8 9]
 [8 6 7]]
```

2.2 Pandas – Data Manipulation

- Pandas is a high-performance library intended for data manipulation and analysis.
- It presents two primary data structures: Series (one-dimensional) and DataFrame (two-dimensional), which facilitate efficient data handling and processing.
- **Main Features**:
  - DataFrame and Series for storing structured data.
  - Data cleaning and data transformation.
  - Missing data handling.
  - Merging, grouping, and filtering data sets.
  - Support for time series analysis.

```python
import pandas as pd

# Creating a Series
ser = pd.Series([10, 20, 30, 40], index=['a', 'b', 'c', 'd'])
print("Series:")
print(ser)

# Creating a DataFrame
data = {'Name': ['Alice', 'Bob', 'Charlie'], 'Age': [25, 30, 35], 'Salary': [50000, 60000, 70000]}
df = pd.DataFrame(data)
print("\nDataFrame:")
print(df)

# Data Cleaning - Handling Missing Values
df.loc[3] = [None, None, None]  # Adding a row with missing values
print("\nDataFrame with Missing Values:")
print(df)

# Filling missing values
df.fillna({'Name': 'Unknown', 'Age': df['Age'].mean(), 'Salary': df['Salary'].median()}, inplace=True
print("\nDataFrame after Filling Missing Values:")
print(df)

# Data Transformation
df['Salary'] = df['Salary'] * 1.1  # Increasing salary by 10%
print("\nDataFrame after Salary Increase:")
print(df)

# Merging DataFrames
df2 = pd.DataFrame({'Name': ['Alice', 'Charlie'], 'Department': ['HR', 'IT']})
merged_df = pd.merge(df, df2, on='Name', how='left')
print("\nMerged DataFrame:")
print(merged_df)

# Grouping Data
grouped_df = df.groupby('Age').mean()
print("\nGrouped DataFrame (Mean by Age):")
print(grouped_df)

# Filtering Data
filtered_df = df[df['Age'] > 28]
print("\nFiltered DataFrame (Age > 28):")
print(filtered_df)

# Time Series Support
df['Joining Date'] = pd.date_range(start='1/1/2020', periods=len(df), freq='Y')
print("\nDataFrame with Time Series:")
print(df)
```

Output:

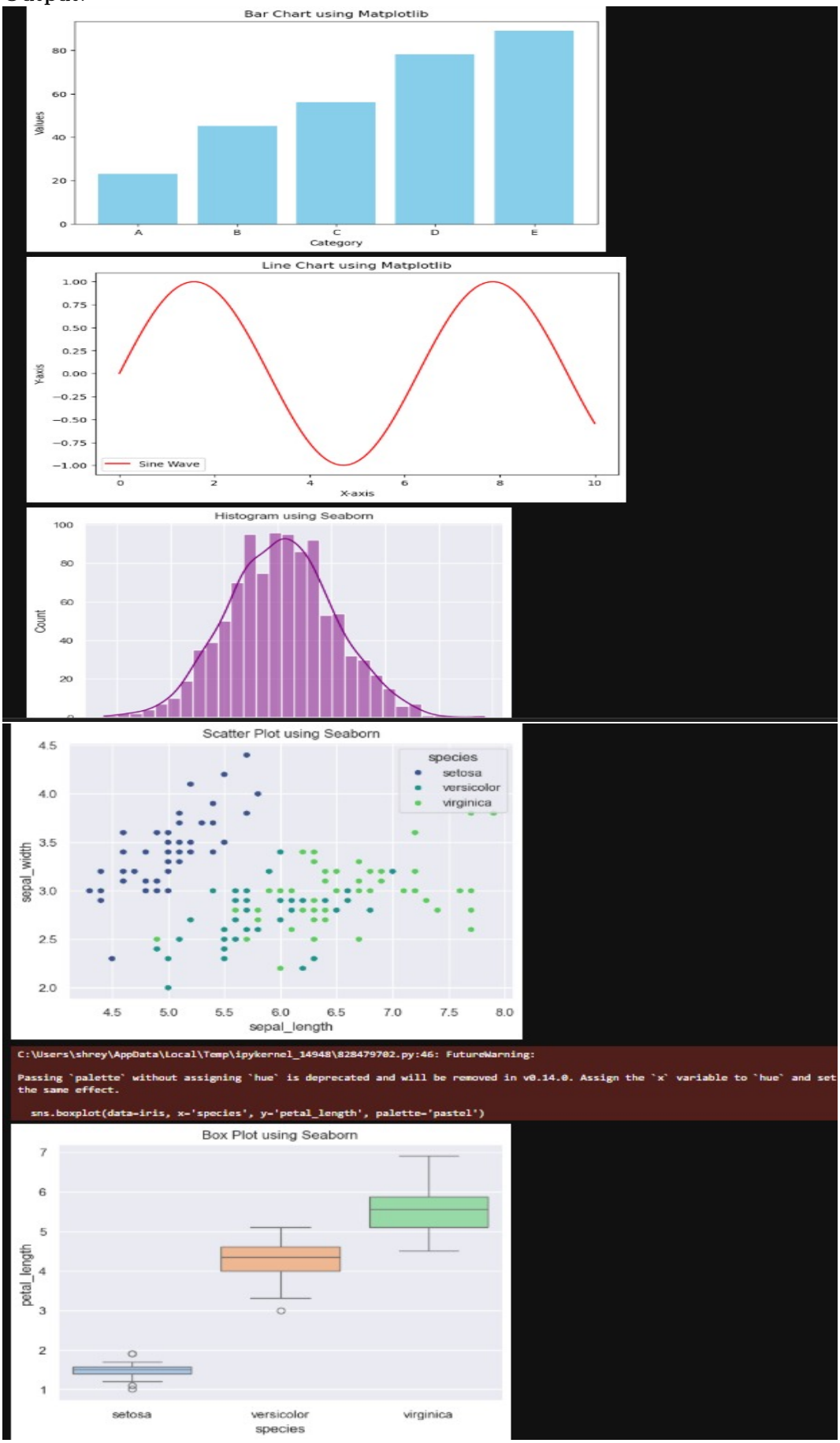
```
Series:
a    10
b    20
c    30
d    40
dtype: int64

DataFrame:
      Name  Age  Salary
0    Alice   25   50000
1      Bob   30   60000
2  Charlie   35   70000

DataFrame with Missing Values:
      Name   Age Salary
0    Alice    25  50000
1      Bob    30  60000
2  Charlie    35  70000
3     None  None   None
```

2.3 Matplotlib & Seaborn – Data Visualization

- Matplotlib is a robust library for static, animated, and interactive Python visualizations.
- Seaborn, which is based on Matplotlib, makes statistical data visualization easier with more aesthetics and functionalities.
- **Key Features**:
  - Plots with customizable appearances (line charts, bar graphs, histograms, scatter plots, etc.).
  - State-based and object-oriented plotting interface.
  - Thematic visualization and statistical plotting (Seaborn).
  - Support for intricate multi-plot layouts.

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# Sample dataset
data = {
    'Category': ['A', 'B', 'C', 'D', 'E'],
    'Values': [23, 45, 56, 78, 89]
}
df = pd.DataFrame(data)

# Matplotlib - Bar Chart
plt.figure(figsize=(8, 5))
plt.bar(df['Category'], df['Values'], color='skyblue')
plt.xlabel('Category')
plt.ylabel('Values')
plt.title('Bar Chart using Matplotlib')
plt.show()

# Matplotlib - Line Chart
x = np.linspace(0, 10, 100)
y = np.sin(x)
plt.figure(figsize=(8, 5))
plt.plot(x, y, label='Sine Wave', color='red')
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Line Chart using Matplotlib')
plt.legend()
plt.show()

# Seaborn - Histogram
sns.set_theme(style='darkgrid')
data = np.random.randn(1000)
sns.histplot(data, kde=True, color='purple', bins=30)
plt.title('Histogram using Seaborn')
plt.show()

# Seaborn - Scatter Plot
iris = sns.load_dataset('iris')
sns.scatterplot(data=iris, x='sepal_length', y='sepal_width', hue='species', palette='viridis')
plt.title('Scatter Plot using Seaborn')
plt.show()

# Seaborn - Box Plot
sns.boxplot(data=iris, x='species', y='petal_length', palette='pastel')
plt.title('Box Plot using Seaborn')
plt.show()
```

Output:

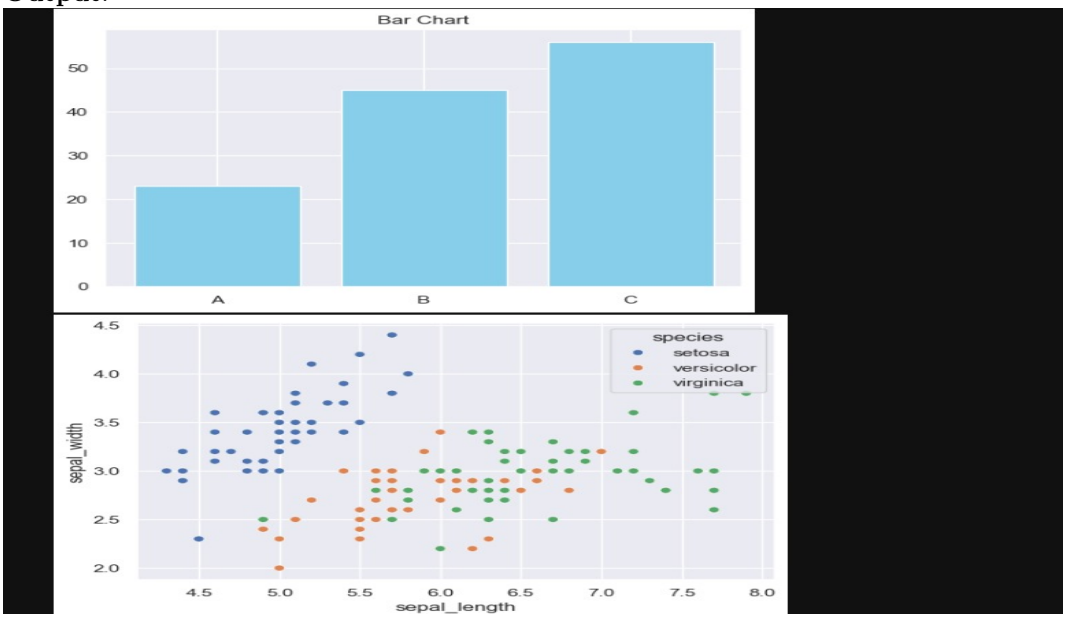

Bar Chart using Matplotlib
Values
Category
A
B
C
D
E
Line Chart using Matplotlib
Yaxis
X-axis
Sine Wave
Histogram using Seaborn
Count
Scatter Plot using Seaborn
species
setosa
versicolor
virginica
sepal_width
sepal_length
C:\Users\shrey\AppData\Local\Temp\ipykernel_14948\828479702.py:46: FutureWarning:
Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `x` variable to `hue` and set the same effect.
sns.boxplot(data=iris, x='species', y='petal_length', palette='pastel')
Box Plot using Seaborn
petal_length
setosa
versicolor
virginica
species

2.4 Scikit-Learn – Machine Learning

- Scikit-Learn is a popular machine learning library that offers easy and effective data mining and analysis tools.
- It is based on NumPy, SciPy, and Matplotlib and includes a variety of supervised and unsupervised learning algorithms.
- **Major Features**:
  - Classification, regression, and clustering algorithms.
  - Dimensionality reduction techniques.
  - Model evaluation and hyperparameter tuning.
  - Feature extraction and preprocessing utilities.

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, mean_squared_error
from sklearn.datasets import load_iris, make_regression

# Matplotlib - Bar Chart
plt.bar(['A', 'B', 'C'], [23, 45, 56], color='skyblue')
plt.title('Bar Chart')
plt.show()

# Seaborn - Scatter Plot
iris = sns.load_dataset('iris')
sns.scatterplot(data=iris, x='sepal_length', y='sepal_width', hue='species')
plt.show()

# Scikit-Learn - Classification
X, y = load_iris(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
clf = RandomForestClassifier()
clf.fit(X_train, y_train)
print(f'Classification Accuracy: {accuracy_score(y_test, clf.predict(X_test)):.2f}')

# Scikit-Learn - Regression
X_reg, y_reg = make_regression(n_samples=100, n_features=1, noise=10, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X_reg, y_reg, test_size=0.2, random_state=42)
reg = LinearRegression()
reg.fit(X_train, y_train)
predictions = reg.predict(X_test)
print(f'Regression MSE: {mean_squared_error(y_test, predictions):.2f}')

# Scikit-Learn - Preprocessing
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
print('Data standardized using StandardScaler.')
```

Output:



2.5 Statsmodels – Statistical Modeling

- Statsmodels is a library intended for statistical hypothesis testing and modeling.
- It offers the functionality for performing statistical tests as well as for estimating statistical models.
- **Main Features**:
  - Regression models (linear regression, logistic regression, and generalized linear models).
  - Time series analysis.
  - Hypothesis testing and statistical inference.
  - Support for econometric models that are advanced.

```python
import numpy as np
import pandas as pd
import statsmodels.api as sm
import statsmodels.formula.api as smf

# Generate synthetic data for regression
np.random.seed(42)
X = np.random.rand(100) * 10  # Independent variable
y = 3 * X + np.random.randn(100) * 2  # Dependent variable with noise

# Convert to DataFrame
data = pd.DataFrame({'X': X, 'y': y})

# Add constant for intercept
X_const = sm.add_constant(data['X'])

# Linear Regression using statsmodels
model = sm.OLS(data['y'], X_const).fit()
print("Linear Regression Summary:")
print(model.summary())

# Hypothesis Testing: t-test for slope
print("\nHypothesis Testing (t-test for slope):")
t_test = model.t_test([0, 1])  # Testing if slope is significantly different from 0
print(t_test)

# Time Series Analysis: ARIMA model
from statsmodels.tsa.arima.model import ARIMA

time_series_data = np.cumsum(np.random.randn(100))  # Generate synthetic time series data
model_arima = ARIMA(time_series_data, order=(1,1,1))
result_arima = model_arima.fit()
print("\nARIMA Model Summary:")
print(result_arima.summary())
```

Output:

```
Linear Regression Summary:
                            OLS Regression Results
==============================================================================
Dep. Variable:                      y   R-squared:                       0.958
Model:                            OLS   Adj. R-squared:                  0.958
Method:                 Least Squares   F-statistic:                     2251.
Date:                Sat, 22 Mar 2025   Prob (F-statistic):           2.06e-69
Time:                        19:04:35   Log-Likelihood:                -200.46
No. Observations:                 100   AIC:                             404.9
Df Residuals:                      98   BIC:                             410.1
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const          0.4302      0.341      1.263      0.210      -0.246       1.106
X              2.9080      0.061     47.440      0.000       2.786       3.030
==============================================================================
Omnibus:                        0.900   Durbin-Watson:                   2.285
Prob(Omnibus):                  0.638   Jarque-Bera (JB):                0.808
Skew:                           0.217   Prob(JB):                        0.668
Kurtosis:                       2.929   Cond. No.                         10.7
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

Hypothesis Testing (t-test for slope):
                             Test for Constraints
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
c0             2.9080      0.061     47.440      0.000       2.786       3.030
==============================================================================

ARIMA Model Summary:
                               SARIMAX Results
==============================================================================
Dep. Variable:                      y   No. Observations:                  100
Model:                 ARIMA(1, 1, 1)   Log Likelihood                -149.579
Date:                Sat, 22 Mar 2025   AIC                            305.158
Time:                        19:04:35   BIC                            312.944
Sample:                             0   HQIC                           308.308
                                  - 100
Covariance Type:                  opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
ar.L1          0.3025      1.843      0.164      0.870      -3.310       3.915
ma.L1         -0.3554      1.794     -0.198      0.843      -3.871       3.160
sigma2         1.2019      0.139      8.641      0.000       0.929       1.474
===================================================================================
Ljung-Box (L1) (Q):                   0.00   Jarque-Bera (JB):                 6.12
Prob(Q):                              0.97   Prob(JB):                         0.05
Heteroskedasticity (H):               1.06   Skew:                             0.34
Prob(H) (two-sided):                  0.86   Kurtosis:                         4.01
===================================================================================
```

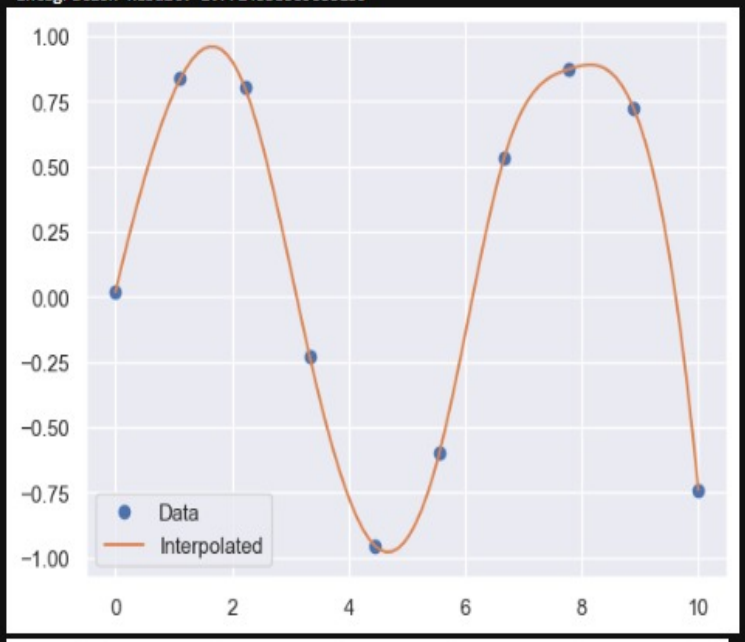
2.6 SciPy – Advanced Mathematical Functions

- SciPy is an extension of the NumPy package with more functions for scientific and technical computing.
- It has modules for optimization, integration, interpolation, and signal processing.
- **Important Features**:
  - Optimization and root-finding routines.
  - Signal and image processing algorithms.
  - Numerical integration and interpolation.
  - Linear algebra and statistical operations.

```python
import numpy as np
from scipy import optimize, integrate, interpolate, signal, linalg, stats
import matplotlib.pyplot as plt

# Optimization: Finding the minimum of a function
def func(x):
    return x**2 + 10*np.sin(x)

result = optimize.minimize(func, x0=0)
print("Optimization Result:")
print(result)

# Integration: Definite integral of a function
def integrand(x):
    return np.exp(-x**2)

integral, error = integrate.quad(integrand, -np.inf, np.inf)
print("\nIntegration Result:", integral)

# Interpolation: Create a smooth function from noisy data
x = np.linspace(0, 10, 10)
y = np.sin(x) + 0.1*np.random.randn(10)
f_interpolate = interpolate.interp1d(x, y, kind='cubic')

x_new = np.linspace(0, 10, 100)
y_new = f_interpolate(x_new)

plt.plot(x, y, 'o', label='Data')
plt.plot(x_new, y_new, '-', label='Interpolated')
plt.legend()
plt.show()

# Signal Processing: Applying a low-pass filter
b, a = signal.butter(3, 0.1)
data = np.sin(np.linspace(0, 10, 100)) + 0.5*np.random.randn(100)
filtered_data = signal.filtfilt(b, a, data)

plt.plot(data, label='Original Signal')
plt.plot(filtered_data, label='Filtered Signal')
plt.legend()
plt.show()

# Linear Algebra: Solving a system of equations
A = np.array([[3, 2], [1, 4]])
b = np.array([5, 6])
x = linalg.solve(A, b)
print("\nSolution to Linear System:", x)

# Statistics: Computing the mean and standard deviation
data = np.random.randn(1000)
mean = np.mean(data)
std_dev = np.std(data)
print("\nMean:", mean, " Standard Deviation:", std_dev)
```

Output:

```
Optimization Result:
  message: Optimization terminated successfully.
  success: True
   status: 0
      fun: -7.945823375615215
        x: [-1.306e+00]
      nit: 5
      jac: [-1.192e-06]
 hess_inv: [[ 8.589e-02]]
     nfev: 12
     njev: 6

Integration Result: 1.7724538509055159
```



2.7 Dask & Vaex – Big Data Processing

- Dask and Vaex are specialized libraries for managing large datasets that are beyond the in-memory scope.
- Dask supports parallel computation for scalable data processing, and Vaex supports fast, out-of-core DataFrame manipulation.
- **Important Features**:
  - Dask: Parallel computation, lazy evaluation, and Pandas and NumPy integration.
  - Vaex: Handling of large datasets with efficiency without loading them into memory.
  - Support for distributed computing.
  - Faster performance for big data analytics.

```python
import dask.dataframe as dd
import vaex
import pandas as pd
import numpy as np

# Create a large dataset for demonstration
df = pd.DataFrame({
    'A': np.random.rand(10**6),
    'B': np.random.rand(10**6),
    'C': np.random.randint(0, 100, size=(10**6))
})

# Using Dask for parallel computation
dask_df = dd.from_pandas(df, npartitions=10)
print("Dask DataFrame Computation:")
print(dask_df.describe().compute())

# Using Vaex for memory-efficient operations
vaex_df = vaex.from_pandas(df)
print("\nVaex DataFrame Computation:")
print(vaex_df.mean())

# Performing a lazy operation in Vaex
vaex_df['D'] = vaex_df['A'] * 2 + vaex_df['B']
print("\nNew Column Computation (Lazy Evaluation):")
print(vaex_df['D'].head(5))

# Saving and loading large datasets efficiently
vaex_df.export_hdf5("large_dataset.hdf5")
vaex_loaded = vaex.open("large_dataset.hdf5")
print("Loaded Vaex DataFrame:")
print(vaex_loaded.head(5))
```

Output:

```
Dask DataFrame Computation:
                A             B              C
count  1.000000e+06  1.000000e+06  1000000.000000
mean   5.003506e-01  4.994583e-01       49.515017
std    2.885892e-01  2.885022e-01       28.847702
min    5.188446e-07  3.774576e-07        0.000000
25%    2.521254e-01  2.526119e-01       25.000000
50%    5.045533e-01  5.012498e-01       50.000000
75%    7.535717e-01  7.517640e-01       75.000000
max    9.999983e-01  9.999994e-01       99.000000
```

3 Data Acquisition & Processing

Successful data acquisition and processing are essential phases in any data science process. This chapter discusses some of the approaches to acquiring, cleaning, and converting data for high-quality inputs into analysis and machine learning models.

3.1 Reading and Writing Data

- Efficient reading and writing of data is required to work with structured and unstructured data sets.
- Python offers a number of libraries to read and write various file formats like CSV, JSON, Excel, and databases.
- **Important Functions**:
  - pandas.read_csv(), pandas.to_csv() for CSV.
  - pandas.read_excel(), pandas.to_excel() for Excel.
  - json.load(), json.dump() for JSON.
  - SQL database operations using sqlite3 or SQLAlchemy.

```python
import pandas as pd
import json
import sqlite3
# Reading and Writing CSV
csv_file = "data.csv"
df = pd.DataFrame({
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'Salary': [50000, 60000, 70000]
})
df.to_csv(csv_file, index=False)  # Write to CSV
csv_df = pd.read_csv(csv_file)  # Read from CSV
print("CSV Data:")
print(csv_df)

# Reading and Writing Excel
excel_file = "data.xlsx"
df.to_excel(excel_file, index=False, sheet_name='Sheet1')  # Write to Excel
excel_df = pd.read_excel(excel_file, sheet_name='Sheet1')  # Read from Excel
print("\nExcel Data:")
print(excel_df)

# Reading and Writing JSON
json_file = "data.json"
with open(json_file, 'w') as f:
    json.dump(df.to_dict(orient='records'), f)  # Write to JSON
with open(json_file, 'r') as f:
    json_data = json.load(f)  # Read from JSON
print("\nJSON Data:")

print(json_data)
# SQL Database Operations
conn = sqlite3.connect("data.db")
df.to_sql("employees", conn, if_exists='replace', index=False)  # Write to SQL
sql_df = pd.read_sql("SELECT * FROM employees", conn)  # Read from SQL
print("\nSQL Data:")
print(sql_df)

conn.close()
```

Output:

```
CSV Data:
      Name  Age  Salary
0    Alice   25   50000
1      Bob   30   60000
2  Charlie   35   70000

Excel Data:
      Name  Age  Salary
0    Alice   25   50000
1      Bob   30   60000
2  Charlie   35   70000

JSON Data:
[{'Name': 'Alice', 'Age': 25, 'Salary': 50000}, {'Name': 'Bob', 'Age': 30, 'Salary': 60000}, {'Name': 'Charlie', 'Age': 35, 'Salary': 70000}]

SQL Data:
      Name  Age  Salary
0    Alice   25   50000
1      Bob   30   60000
2  Charlie   35   70000
```

3.2 Web Scraping

- Web scraping enables data gathering from websites with automated programs.
- Python offers BeautifulSoup and Scrapy libraries for efficiently scraping and parsing web pages.
- **Important Methods**:
  - Parsing HTML using BeautifulSoup.
  - Automating requests with requests and urllib.
  - Gathering structured data using Scrapy.
  - Processing JavaScript-rendered content with Selenium.

```python
import requests
from bs4 import BeautifulSoup
from selenium import webdriver
from selenium.webdriver.chrome.service import Service
from selenium.webdriver.chrome.options import Options
import time

# Web Scraping with BeautifulSoup
url = "https://example.com"
response = requests.get(url)
soup = BeautifulSoup(response.text, 'html.parser')
print("Title of the Page:", soup.title.string)

# Extract all links
links = [a['href'] for a in soup.find_all('a', href=True)]
print("\nExtracted Links:")
print(links)

# Automating Requests with requests
headers = {"User-Agent": "Mozilla/5.0"}
response = requests.get(url, headers=headers)
print("\nStatus Code:", response.status_code)

# Scraping JavaScript-rendered Content with Selenium
chrome_options = Options()
chrome_options.add_argument("--headless")  # Run browser in headless mode
service = Service("chromedriver")  # Update with the correct path to your chromedriver

# Initialize the driver
driver = webdriver.Chrome(service=service, options=chrome_options)
driver.get(url)
time.sleep(3)  # Wait for JS to load

# Extract dynamic content
page_source = driver.page_source
driver.quit()
soup_dynamic = BeautifulSoup(page_source, 'html.parser')
print("\nDynamic Page Title:", soup_dynamic.title.string)
```

Output:

```
Title of the Page: Example Domain

Extracted Links:
['https://www.iana.org/domains/example']

Status Code: 200
```

3.3 Working with APIs

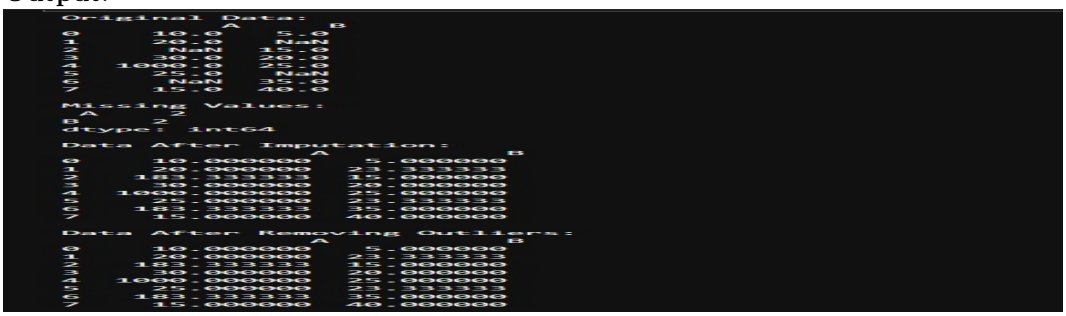
- APIs (Application Programming Interfaces) provide data retrieval from external sources like social media websites, financial data providers, and open data stores.
- **Important Methods**:
  - Utilizing requests to retrieve data from REST APIs.
  - Authentication handling with API keys or OAuth.
  - Processing JSON responses with json library.
  - Working with third-party services like Twitter API, OpenWeather API, and Google Maps API.

3.4 Handling Missing Values and Outliers

- Managing missing values and outliers is important for data quality and avoiding biases in analysis and machine learning models.
- **Important Methods**:
  - Finding missing values with pandas.isnull().
  - Replacing missing values with mean, median, or mode with fillna().
  - Identifying and removing outliers through statistical measures (e.g., Z-score, IQR).
  - Advanced imputation techniques with sklearn.impute.

```python
import pandas as pd
import numpy as np
from scipy import stats
from sklearn.impute import SimpleImputer

# Sample dataset with missing values and outliers
data = {
    'A': [10, 20, np.nan, 30, 1000, 25, np.nan, 15],  # Contains missing values & an outlier (1000)
    'B': [5, np.nan, 15, 20, 25, np.nan, 35, 40]
}

df = pd.DataFrame(data)
print("Original Data:\n", df)

# **Handling Missing Values**
# Finding missing values
print("\nMissing Values:\n", df.isnull().sum())

# Replacing missing values with column mean
imputer = SimpleImputer(strategy='mean')
df_imputed = pd.DataFrame(imputer.fit_transform(df), columns=df.columns)
print("\nData After Imputation:\n", df_imputed)

# **Handling Outliers**
# Using Z-score method to detect outliers
z_scores = np.abs(stats.zscore(df_imputed))
df_no_outliers = df_imputed[(z_scores < 3).all(axis=1)]
print("\nData After Removing Outliers:\n", df_no_outliers)
```

Output:



3.5 Feature Engineering and Transformation

- Feature engineering is about developing new features or feature transformations to best enhance model performance.
- Optimal predictive precision relies on it.
- **Important Methods**:
  - Categorical variable encoding (label encoding, one-hot encoding).
  - Scaling numerical features using StandardScaler, MinMaxScaler.
  - Polynomial feature generation with PolynomialFeatures.
  - Principal Component Analysis (PCA) dimensionality reduction.
  - Creating interaction features to pick up interactions between variables.

```python
import pandas as pd
import numpy as np
from sklearn.preprocessing import LabelEncoder, OneHotEncoder, StandardScaler, MinMaxScaler, PolynomialFeatures
from sklearn.decomposition import PCA
# Sample dataset
data = {'Category': ['A', 'B', 'A', 'C', 'B', 'C', 'A', 'B'],'Value1':[10, 20, 30, 40, 50, 60, 70, 80],'Value2':[1.5, 2.5, 3.5, 4.5, 5.5, 6.5, 7.5, 8.5]}
df = pd.DataFrame(data)
print("Original Data:\n", df)

# **Categorical Encoding** # Label Encoding
label_encoder = LabelEncoder()
df['Category_Label'] = label_encoder.fit_transform(df['Category'])

# One-Hot Encoding
one_hot_encoder = OneHotEncoder(sparse=False)
encoded_categories = one_hot_encoder.fit_transform(df[['Category']])
df_encoded = pd.DataFrame(encoded_categories, columns=one_hot_encoder.get_feature_names_out(['Category']))
df = pd.concat([df, df_encoded], axis=1).drop(columns=['Category'])
print("\nData After Encoding:\n", df)

# **Scaling Numerical Features**
scaler = StandardScaler()
df[['Value1', 'Value2']] = scaler.fit_transform(df[['Value1', 'Value2']])
print("\nData After Standard Scaling:\n", df)

# **Polynomial Feature Generation**
poly = PolynomialFeatures(degree=2, interaction_only=True, include_bias=False)
poly_features = poly.fit_transform(df[['Value1', 'Value2']])
df_poly = pd.DataFrame(poly_features, columns=poly.get_feature_names_out(['Value1', 'Value2']))
df = pd.concat([df, df_poly], axis=1)
print("\nData After Polynomial Feature Generation:\n", df)

# **Dimensionality Reduction using PCA**
pca = PCA(n_components=2)  # Reduce to 2 dimensions
principal_components = pca.fit_transform(df[['Value1', 'Value2']])
df_pca = pd.DataFrame(principal_components, columns=['PC1', 'PC2'])
df = pd.concat([df, df_pca], axis=1)
print("\nData After PCA Dimensionality Reduction:\n", df)
```

Output:

```
Original Data:
   Category  Value1  Value2
0         A      10     1.5
1         B      20     2.5
2         A      30     3.5
3         C      40     4.5
4         B      50     5.5
5         C      60     6.5
6         A      70     7.5
7         B      80     8.5
```

4 Exploratory Data Analysis (EDA)

Exploratory Data Analysis (EDA) is a key process for discovering the inherent patterns and structures in a dataset. It consists of summarizing, visualizing, and changing data to infer useful information.

4.1 Summary Statistics

- Summary statistics give a rapid overview of the dataset to reveal trends, distributions, and possible problems.
- **Important Methods**:
  - pandas.describe() for numerical summaries.
  - pandas.info() for structure of the dataset.
  - Calculation of mean, median, standard deviation, and percentiles.
  - Grouping and summarizing data using groupby().

```python
import pandas as pd

# Sample dataset
data = {
    'Department': ['HR', 'IT', 'IT', 'HR', 'Finance', 'Finance', 'IT', 'HR', 'Finance', 'IT'],
    'Age': [25, 30, 35, 40, 29, 28, 32, 41, 36, 38],
    'Salary': [50000, 60000, 65000, 55000, 70000, 72000, 62000, 53000, 71000, 64000]}

# Creating DataFrame
df = pd.DataFrame(data)

# 1️⃣ Summary statistics for numerical columns
print("🔹 Summary Statistics:")
print(df.describe())

# 2️⃣ Dataset structure and missing values
print("\n🔹 Dataset Information:")
print(df.info())

# 3️⃣ Calculation of mean, median, standard deviation, and percentiles
print("\n🔹 Statistical Measures:")
print(f"Mean Age: {df['Age'].mean()}")
print(f"Median Age: {df['Age'].median()}")
print(f"Standard Deviation of Age: {df['Age'].std()}")
print(f"25th Percentile of Age: {df['Age'].quantile(0.25)}")
print(f"75th Percentile of Age: {df['Age'].quantile(0.75)}")

# 4️⃣ Grouping and summarizing data using groupby()
print("\n🔹 Average Salary by Department:")
print(df.groupby("Department")["Salary"].mean())

print("\n🔹 Count of Employees by Department:")
print(df.groupby("Department")["Age"].count())
```

Output:

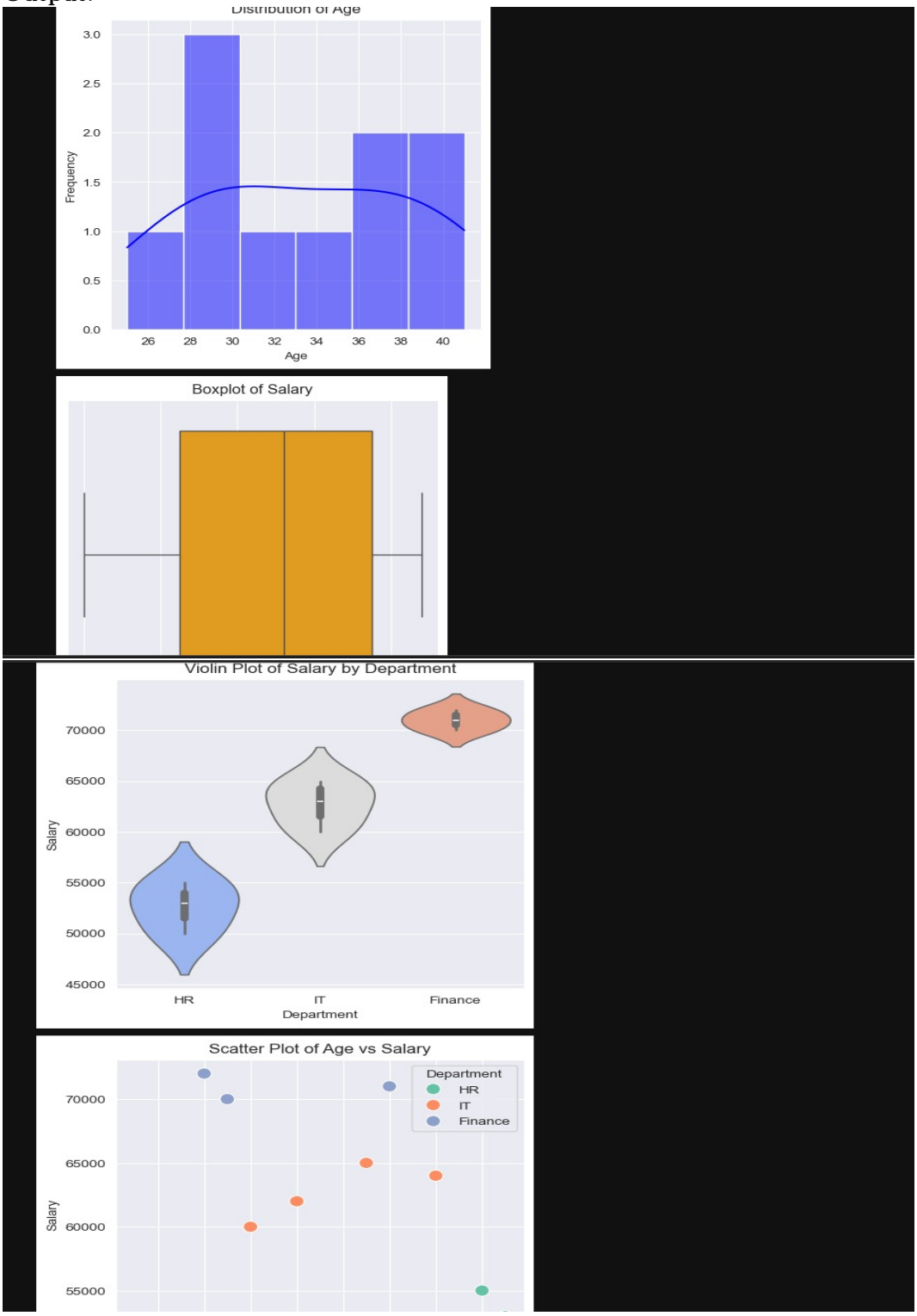
```
🔹 Summary Statistics:
              Age         Salary
count   10.000000      10.000000
mean    33.400000   62200.000000
std      5.420127    7714.344503
min     25.000000   50000.000000
25%     29.250000   56250.000000
50%     33.500000   63000.000000
75%     37.500000   68750.000000
max     41.000000   72000.000000

🔹 Dataset Information:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10 entries, 0 to 9
Data columns (total 3 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   Department  10 non-null     object
 1   Age         10 non-null     int64
 2   Salary      10 non-null     int64
dtypes: int64(2), object(1)
memory usage: 372.0+ bytes
None

🔹 Statistical Measures:
Mean Age: 33.4
Median Age: 33.5
Standard Deviation of Age: 5.420127099780759
25th Percentile of Age: 29.25
75th Percentile of Age: 37.5

🔹 Average Salary by Department:
Department
Finance    71000.000000
HR         52666.666667
IT         62750.000000
Name: Salary, dtype: float64

🔹 Count of Employees by Department:
Department
Finance    3
HR         3
IT         4
Name: Age, dtype: int64
```

4.2 Visualizing Distributions and Correlations

- Visualizing distributions and correlations of data assists in identifying patterns and relationships between variables.
- **Important Methods**:
  - Histograms and density plots with seaborn.histplot() and sns.kdeplot().
  - Boxplots and violin plots for identifying outliers.
  - Scatter plots and pair plots for relationships.
  - Correlation heatmaps with sns.heatmap().

```python
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
# Sample dataset
data = {
    'Department': ['HR', 'IT', 'IT', 'HR', 'Finance', 'Finance', 'IT', 'HR', 'Finance', 'IT'],
    'Age': [25, 30, 35, 40, 29, 28, 32, 41, 36, 38],
    'Salary': [50000, 60000, 65000, 55000, 70000, 72000, 62000, 53000, 71000, 64000]}
# Creating DataFrame
df = pd.DataFrame(data)
# Set Seaborn style
sns.set_style("darkgrid")
# 1️⃣ **Histogram & Density Plot**
plt.figure(figsize=(10, 5))
sns.histplot(df['Age'], kde=True, bins=6, color="blue")
plt.title("Distribution of Age")
plt.xlabel("Age")
plt.ylabel("Frequency")
plt.show()
# 2️⃣ **Boxplot for Outliers**
plt.figure(figsize=(8, 5))
sns.boxplot(x=df['Salary'], color="orange")
plt.title("Boxplot of Salary")
plt.xlabel("Salary")
plt.show()
# 3️⃣ **Violin Plot for Distribution**
plt.figure(figsize=(8, 5))
sns.violinplot(x="Department", y="Salary", data=df, palette="coolwarm")
plt.title("Violin Plot of Salary by Department")
plt.show()
# 4️⃣ **Scatter Plot (Age vs Salary)**
plt.figure(figsize=(8, 5))
sns.scatterplot(x="Age", y="Salary", data=df, hue="Department", palette="Set2", s=100)
plt.title("Scatter Plot of Age vs Salary")
plt.show()
# 5️⃣ **Pair Plot for Relationships**
sns.pairplot(df, hue="Department", diag_kind="kde", palette="husl")
plt.show()
# 6️⃣ **Correlation Heatmap**
plt.figure(figsize=(6, 5))
sns.heatmap(df.corr(), annot=True, cmap="coolwarm", linewidths=0.5)
plt.title("Correlation Heatmap")
plt.show()
```

Output:



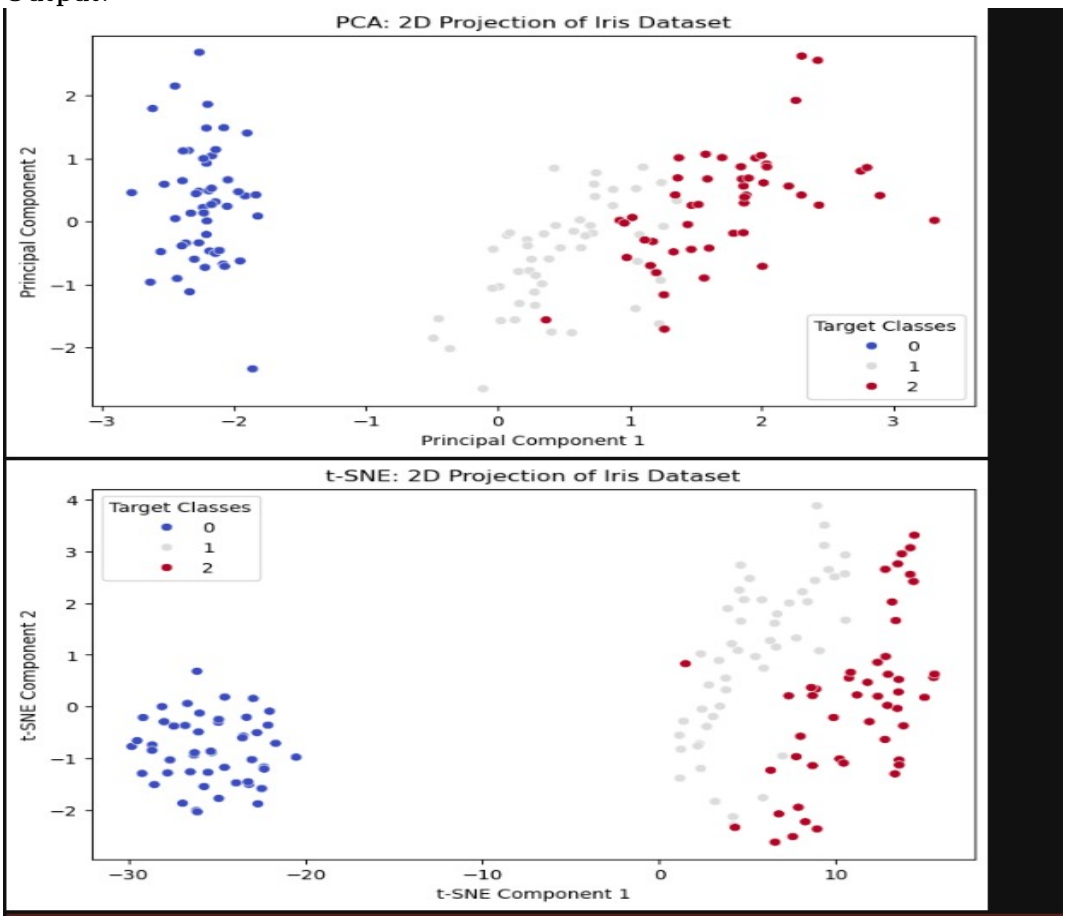
4.3 Dimensionality Reduction

- Dimensionality reduction methods assist in the simplification of datasets maintaining key information.
- **Important Methods**:
  - Principal Component Analysis (PCA).
  - t-Distributed Stochastic Neighbor Embedding (t-SNE).

– Uniform Manifold Approximation and Projection (UMAP).
– Feature selection and feature extraction methods.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.decomposition import PCA
from sklearn.manifold import TSNE
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import load_iris
# 🔹 Load the Iris Dataset
iris = load_iris()
df = pd.DataFrame(iris.data, columns=iris.feature_names)
df['target'] = iris.target
# 🔹 Standardize the Data (Important for PCA, t-SNE, and UMAP)
scaler = StandardScaler()
df_scaled = scaler.fit_transform(df.drop(columns=['target']))
# 1️⃣ **Principal Component Analysis (PCA)**
pca = PCA(n_components=2)
pca_result = pca.fit_transform(df_scaled)
df_pca = pd.DataFrame(pca_result, columns=['PC1', 'PC2'])
df_pca['target'] = df['target']
\# 🔹 PCA Scatter Plot
plt.figure(figsize=(8, 5))
sns.scatterplot(x='PC1', y='PC2', hue=df_pca ['target'], palette='coolwarm', data=df_pca)
plt.title("PCA: 2D Projection of Iris Dataset")
plt.xlabel("Principal Component 1")
plt.ylabel("Principal Component 2")
plt.legend(title="Target Classes")
plt.show()
# 2️⃣ **t-SNE (t-Distributed Stochastic Neighbor Embedding)**
tsne = TSNE(n_components=2, perplexity=30, random_state=42)
tsne_result = tsne.fit_transform(df_scaled)
df_tsne = pd.DataFrame(tsne_result, columns=['t-SNE1', 't-SNE2'])
df_tsne['target'] = df['target']
# 🔹 t-SNE Scatter Plot
plt.figure(figsize=(8, 5))
sns.scatterplot(x='t-SNE1', y='t-SNE2', hue=df_tsne['target'] , palette='coolwarm', data=df_tsne)
plt.title("t-SNE: 2D Projection of Iris Dataset")
plt.xlabel("t-SNE Component 1")
plt.ylabel("t-SNE Component 2")
```

Output:


PCA: 2D Projection of Iris Dataset
Principal Component 2
Principal Component 1
Target Classes
0
1
2
t-SNE: 2D Projection of Iris Dataset
t-SNE Component 2
t-SNE Component 1
Target Classes
0
1
2

5 Machine Learning with Python

Machine learning allows computers to learn patterns from data and predict. Python offers a great set of libraries for constructing machine learning models. This section introduces important concepts and techniques applied in machine learning with Python.

5.1 Supervised vs. Unsupervised Learning

Machine learning is divided into two broad categories:

- **Supervised Learning**: The model is trained on labeled data (input-output pairs) to predict.

  Example: Classifying emails as spam or not spam.

- **Unsupervised Learning**: The model discovers patterns in unlabeled data without direct supervision.

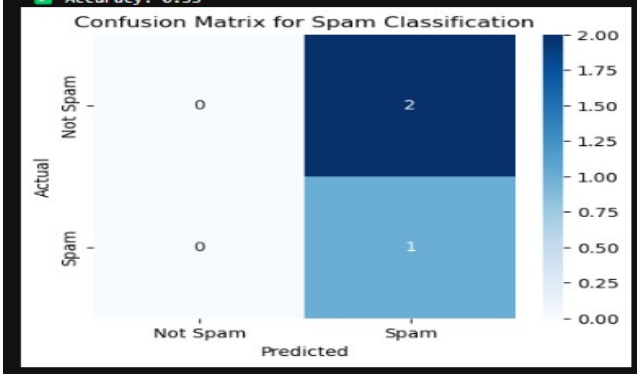
  Example: Grouping customers based on purchasing behavior.

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, confusion_matrix
from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs
from sklearn.feature_extraction.text import CountVectorizer

# 1️⃣ **Supervised Learning: Spam Email Classification** (Binary Classification)
print("\n🔹 Supervised Learning: Spam Classification")

# Sample dataset (Emails & Labels)
emails = ["Win a lottery now!", "Hello, how are you?", "Claim your free gift",
          "Let's meet tomorrow", "Earn money from home", "Lunch at 1 PM?", "Urgent! Your prize awaits"]
labels = [1, 0, 1, 0, 1, 0, 1]  # 1 = Spam, 0 = Not Spam

# Convert text to feature vectors
vectorizer = CountVectorizer()
X = vectorizer.fit_transform(emails).toarray()
y = np.array(labels)

# Split data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Train a simple classifier
model = RandomForestClassifier()
model.fit(X_train, y_train)

# Predictions
y_pred = model.predict(X_test)

# Accuracy
print(f"✅ Accuracy: {accuracy_score(y_test, y_pred):.2f}")
```

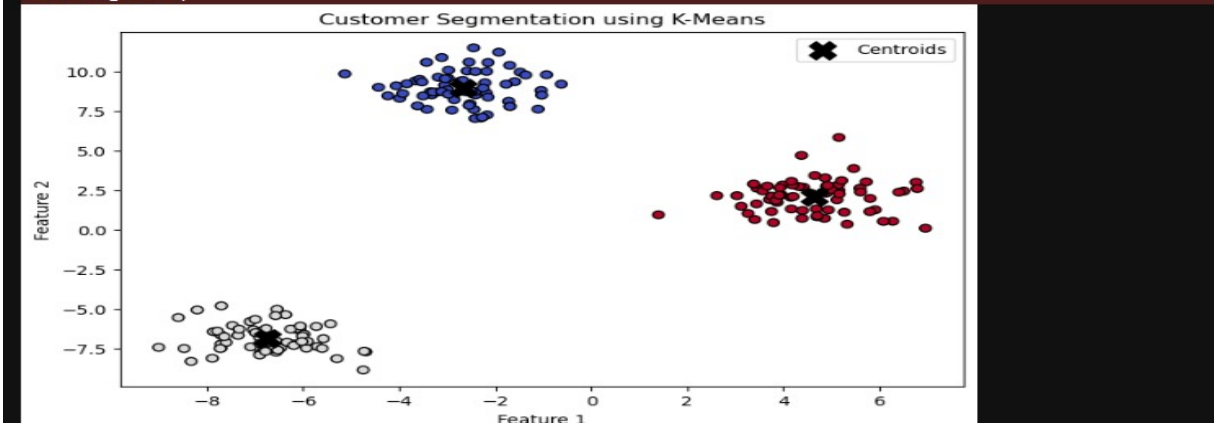
```python
# Accuracy
print(f"✅ Accuracy: {accuracy_score(y_test, y_pred):.2f}")

# Confusion Matrix
plt.figure(figsize=(5, 4))
sns.heatmap(confusion_matrix(y_test, y_pred), annot=True, cmap="Blues", fmt="d", xticklabels=["Not Spam", "Spam"], yticklabels=["Not Spam", "Spam"])
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.title("Confusion Matrix for Spam Classification")
plt.show()

# 2️⃣ **Unsupervised Learning: Customer Segmentation (Clustering)**
print("\n🔹 Unsupervised Learning: Customer Clustering")

# Generate synthetic customer data (3 clusters)
X_cluster, _ = make_blobs(n_samples=200, centers=3, cluster_std=1.0, random_state=42)

# Apply K-Means Clustering
kmeans = KMeans(n_clusters=3, random_state=42)
labels = kmeans.fit_predict(X_cluster)

# Scatter Plot of Clusters
plt.figure(figsize=(8, 5))
plt.scatter(X_cluster[:, 0], X_cluster[:, 1], c=labels, cmap="coolwarm", edgecolors="k")
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1], marker='X', s=200, color='black', label='Centroids')
plt.title("Customer Segmentation using K-Means")
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.legend()
plt.show()
```

Output:

```
🔹 Supervised Learning: Spam Classification
✅ Accuracy: 0.33
```




```
🔹 Unsupervised Learning: Customer Clustering
C:\Users\shrey\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:1446: UserWarning: KMeans is known to have a memory
there are less chunks than available threads. You can avoid it by setting the environment variable OMP_NUM_THREADS=1.
  warnings.warn(
```



5.2 Regression

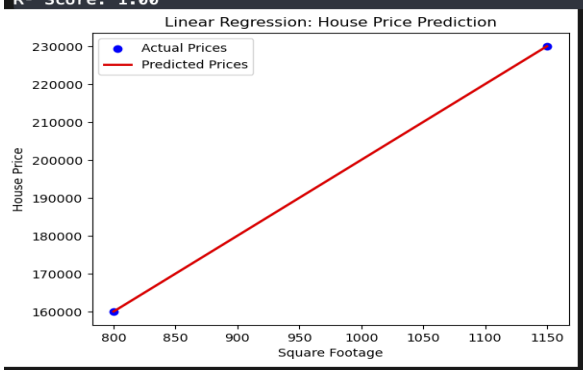
- Regression is a supervised learning technique applied in the process of predicting continuous values.
- It is also extensively utilized in forecasting and trend estimation.
- **Important Methods**:
  - **Linear Regression**: Applies regression on linear relationship between target variable and input variables.

    Example: Predicting house prices based on square footage.

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score

# Sample dataset: Square Footage (X) vs House Price (Y)
data = {'SquareFootage': [750, 800, 850, 900, 950, 1000, 1050, 1100, 1150, 1200],
        'Price': [150000, 160000, 170000, 180000, 190000, 200000, 210000, 220000, 230000, 240000]}
df = pd.DataFrame(data)

# Splitting the dataset
X = df[['SquareFootage']]
y = df['Price']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Training the Linear Regression model
model = LinearRegression()
model.fit(X_train, y_train)

# Predictions
y_pred = model.predict(X_test)

# Evaluation metrics
mae = mean_absolute_error(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)
rmse = np.sqrt(mse)
r2 = r2_score(y_test, y_pred)

# Report
print("Linear Regression Report")
print(f"Mean Absolute Error (MAE): {mae:.2f}")
print(f"Mean Squared Error (MSE): {mse:.2f}")
print(f"Root Mean Squared Error (RMSE): {rmse:.2f}")
print(f"R² Score: {r2:.2f}")

# Visualization
plt.scatter(X_test, y_test, color='blue', label="Actual Prices")
plt.plot(X_test, y_pred, color='red', linewidth=2, label="Predicted Prices")
plt.xlabel("Square Footage")
plt.ylabel("House Price")
plt.title("Linear Regression: House Price Prediction")
plt.legend()
plt.show()
```

Output:

```
Linear Regression Report
Mean Absolute Error (MAE): 0.00
Mean Squared Error (MSE): 0.00
Root Mean Squared Error (RMSE): 0.00
R² Score: 1.00
```



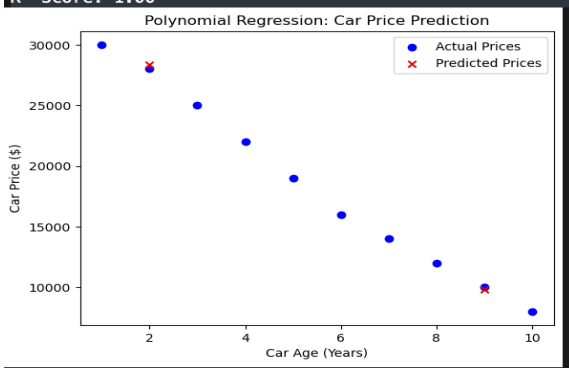
– **Polynomial Regression**: An extension of linear regression for application in dealing with non-linearity.

Example: Predicting car price based on age and mileage with a curved trend.

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score

# Sample dataset: Car Price vs. Age & Mileage
data = {'Age': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
        'Mileage': [5000, 15000, 30000, 45000, 60000, 75000, 90000, 105000, 120000, 135000],
        'Price': [30000, 28000, 25000, 22000, 19000, 16000, 14000, 12000, 10000, 8000]}

df = pd.DataFrame(data)

# Feature and target variable
X = df[['Age', 'Mileage']]
y = df['Price']

# Splitting the dataset
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Applying Polynomial Features (degree 2)
poly = PolynomialFeatures(degree=2)
X_train_poly = poly.fit_transform(X_train)
X_test_poly = poly.transform(X_test)

# Training the Polynomial Regression model
model = LinearRegression()
model.fit(X_train_poly, y_train)

# Predictions
y_pred = model.predict(X_test_poly)

# Evaluation metrics
mae = mean_absolute_error(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)
rmse = np.sqrt(mse)
r2 = r2_score(y_test, y_pred)

# Report
print("Polynomial Regression Report")
print(f"Mean Absolute Error (MAE): {mae:.2f}")
print(f"Mean Squared Error (MSE): {mse:.2f}")
print(f"Root Mean Squared Error (RMSE): {rmse:.2f}")
print(f"R² Score: {r2:.2f}")

# Visualization (2D Plot of Age vs. Price)
plt.scatter(df['Age'], df['Price'], color='blue', label="Actual Prices")
plt.scatter(X_test['Age'], y_pred, color='red', marker='x', label="Predicted Prices")
plt.xlabel("Car Age (Years)")
plt.ylabel("Car Price ($)")
plt.title("Polynomial Regression: Car Price Prediction")
plt.legend()
plt.show()
```

Output:

```
Polynomial Regression Report
Mean Absolute Error (MAE): 283.06
Mean Squared Error (MSE): 87297.66
Root Mean Squared Error (RMSE): 295.46
R² Score: 1.00
```



– **Ridge and Lasso Regression**: Introduces regularization for the intention of avoiding overfitting.

Example of Ridge Regression: Predicting house prices while reducing the impact of less important features

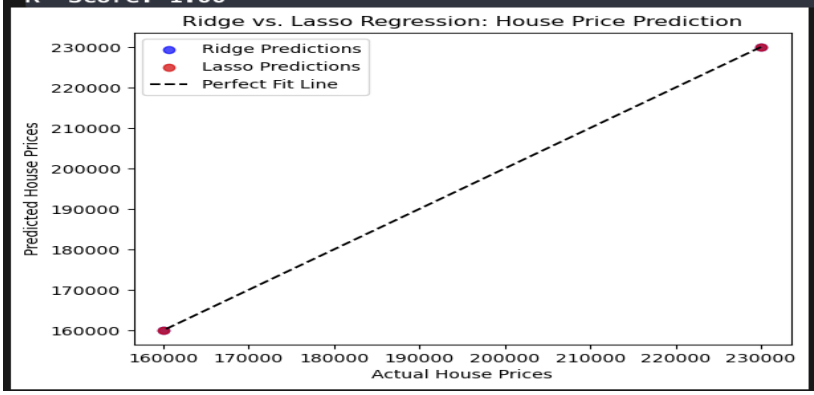
Example of Lasso Regression: Predicting house prices while automatically selecting the

most relevant features.

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.linear_model import Ridge, Lasso
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score

# Sample dataset: House Price vs. Features
data = {'SquareFootage': [750, 800, 850, 900, 950, 1000, 1050, 1100, 1150, 1200],
        'Bedrooms': [1, 2, 2, 3, 3, 3, 4, 4, 4, 5],
        'Age': [10, 15, 20, 5, 7, 12, 8, 6, 3, 1],
        'Price': [150000, 160000, 170000, 180000, 190000, 200000, 210000, 220000, 230000, 240000]}

df = pd.DataFrame(data)

# Features and target variable
X = df[['SquareFootage', 'Bedrooms', 'Age']]
y = df['Price']

# Splitting dataset
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Ridge Regression (L2 Regularization)
ridge_model = Ridge(alpha=1.0)
ridge_model.fit(X_train, y_train)
ridge_pred = ridge_model.predict(X_test)

# Lasso Regression (L1 Regularization)
lasso_model = Lasso(alpha=1.0)
lasso_model.fit(X_train, y_train)
lasso_pred = lasso_model.predict(X_test)

# Evaluation function
def evaluate_model(y_true, y_pred, model_name):
    mae = mean_absolute_error(y_true, y_pred)
    mse = mean_squared_error(y_true, y_pred)
    rmse = np.sqrt(mse)
    r2 = r2_score(y_true, y_pred)

    print(f"\n{model_name} Regression Report")
    print(f"Mean Absolute Error (MAE): {mae:.2f}")
    print(f"Mean Squared Error (MSE): {mse:.2f}")
    print(f"Root Mean Squared Error (RMSE): {rmse:.2f}")
    print(f"R² Score: {r2:.2f}")

# Evaluate Ridge and Lasso
evaluate_model(y_test, ridge_pred, "Ridge")
evaluate_model(y_test, lasso_pred, "Lasso")

# Visualization
plt.scatter(y_test, ridge_pred, color='blue', label="Ridge Predictions", alpha=0.7)
plt.scatter(y_test, lasso_pred, color='red', label="Lasso Predictions", alpha=0.7)
plt.plot(y_test, y_test, color='black', linestyle='--', label="Perfect Fit Line")
plt.xlabel("Actual House Prices")
plt.ylabel("Predicted House Prices")
plt.title("Ridge vs. Lasso Regression: House Price Prediction")
plt.legend()
plt.show()
```

Output:

```
Ridge Regression Report
Mean Absolute Error (MAE): 0.76
Mean Squared Error (MSE): 0.59
Root Mean Squared Error (RMSE): 0.77
R² Score: 1.00

Lasso Regression Report
Mean Absolute Error (MAE): 0.01
Mean Squared Error (MSE): 0.00
Root Mean Squared Error (RMSE): 0.01
R² Score: 1.00
```



– **Decision Tree and Random Forest Regression**: Non-linear techniques for managing complex relationships.

Example of Decision Tree Regression: Predicting house prices by splitting data based

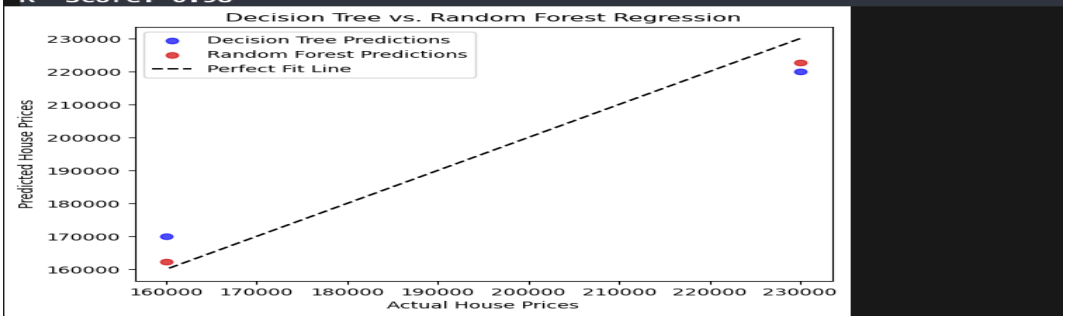
on features like location and size.

Example of Random Forest Regression: Improving house price predictions by averaging multiple decision trees.

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score

# Sample dataset: House Price vs. Features
data = {'SquareFootage': [750, 800, 850, 900, 950, 1000, 1050, 1100, 1150, 1200],
        'Bedrooms': [1, 2, 2, 3, 3, 3, 4, 4, 4, 5],
        'Age': [10, 15, 20, 5, 7, 12, 8, 6, 3, 1],
        'Price': [150000, 160000, 170000, 180000, 190000, 200000, 210000, 220000, 230000, 240000]}

df = pd.DataFrame(data)

# Features and target variable
X = df[['SquareFootage', 'Bedrooms', 'Age']]
y = df['Price']

# Splitting dataset
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Decision Tree Regression
dt_model = DecisionTreeRegressor(random_state=42)
dt_model.fit(X_train, y_train)
dt_pred = dt_model.predict(X_test)

# Random Forest Regression
rf_model = RandomForestRegressor(n_estimators=100, random_state=42)
rf_model.fit(X_train, y_train)
rf_pred = rf_model.predict(X_test)

# Evaluation function
def evaluate_model(y_true, y_pred, model_name):
    mae = mean_absolute_error(y_true, y_pred)
    mse = mean_squared_error(y_true, y_pred)
    rmse = np.sqrt(mse)
    r2 = r2_score(y_true, y_pred)

    print(f"\n{model_name} Regression Report")
    print(f"Mean Absolute Error (MAE): {mae:.2f}")
    print(f"Mean Squared Error (MSE): {mse:.2f}")
    print(f"Root Mean Squared Error (RMSE): {rmse:.2f}")
    print(f"R² Score: {r2:.2f}")

# Evaluate Decision Tree and Random Forest
evaluate_model(y_test, dt_pred, "Decision Tree")
evaluate_model(y_test, rf_pred, "Random Forest")

# Visualization
plt.scatter(y_test, dt_pred, color='blue', label="Decision Tree Predictions", alpha=0.7)
plt.scatter(y_test, rf_pred, color='red', label="Random Forest Predictions", alpha=0.7)
plt.plot(y_test, y_test, color='black', linestyle='--', label="Perfect Fit Line")
plt.xlabel("Actual House Prices")
plt.ylabel("Predicted House Prices")
plt.title("Decision Tree vs. Random Forest Regression")
plt.legend()
plt.show()
```

Output:

```
Decision Tree Regression Report
Mean Absolute Error (MAE): 10000.00
Mean Squared Error (MSE): 100000000.00
Root Mean Squared Error (RMSE): 10000.00
R² Score: 0.92

Random Forest Regression Report
Mean Absolute Error (MAE): 4800.00
Mean Squared Error (MSE): 29290000.00
Root Mean Squared Error (RMSE): 5412.02
R² Score: 0.98
```



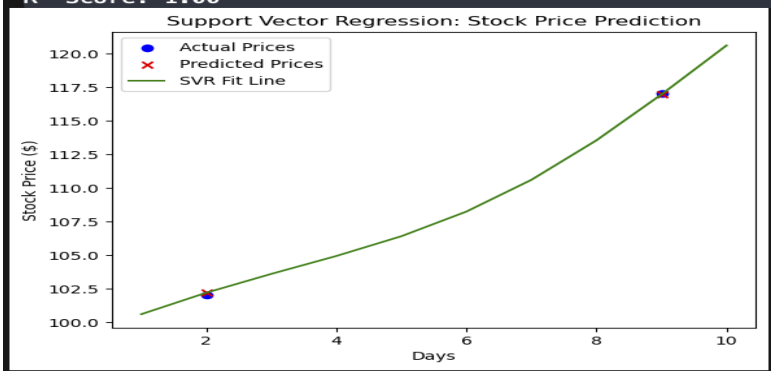
– **Support Vector Regression (SVR)**: Uses kernel functions to map input data into high-dimensional spaces.

Example: Predicting stock prices by fitting the best boundary within a margin.

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVR
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score

# Sample dataset: Days vs. Stock Price
data = {'Days': np.arange(1, 11), 'StockPrice': [100, 102, 104, 108, 107, 105, 110, 115, 117, 120]}
df = pd.DataFrame(data)

# Features and target variable
X = df[['Days']]
y = df['StockPrice']

# Splitting dataset
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Feature Scaling (important for SVR)
scaler_X = StandardScaler()
scaler_y = StandardScaler()
X_train_scaled = scaler_X.fit_transform(X_train)
X_test_scaled = scaler_X.transform(X_test)
y_train_scaled = scaler_y.fit_transform(y_train.values.reshape(-1, 1)).ravel()

# Support Vector Regression (SVR) with RBF Kernel
svr_model = SVR(kernel='rbf', C=100, gamma=0.1, epsilon=0.1)
svr_model.fit(X_train_scaled, y_train_scaled)

# Predictions (transform back to original scale)
y_pred_scaled = svr_model.predict(X_test_scaled)
y_pred = scaler_y.inverse_transform(y_pred_scaled.reshape(-1, 1)).ravel()

# Evaluation metrics
mae = mean_absolute_error(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)
rmse = np.sqrt(mse)
r2 = r2_score(y_test, y_pred)

# Report
print("Support Vector Regression (SVR) Report")
print(f"Mean Absolute Error (MAE): {mae:.2f}")
print(f"Mean Squared Error (MSE): {mse:.2f}")
print(f"Root Mean Squared Error (RMSE): {rmse:.2f}")
print(f"R² Score: {r2:.2f}")

# Visualization
plt.scatter(X_test, y_test, color='blue', label="Actual Prices")
plt.scatter(X_test, y_pred, color='red', marker='x', label="Predicted Prices")
plt.plot(df['Days'], scaler_y.inverse_transform(svr_model.predict(scaler_X.transform(df[['Days']]))
                                .reshape(-1, 1)), color='green', label="SVR Fit Line")
plt.xlabel("Days")
plt.ylabel("Stock Price ($)")
plt.title("Support Vector Regression: Stock Price Prediction")
plt.legend()
plt.show()
```

Output:

```
Support Vector Regression (SVR) Report
Mean Absolute Error (MAE): 0.12
Mean Squared Error (MSE): 0.02
Root Mean Squared Error (RMSE): 0.14
R² Score: 1.00
```



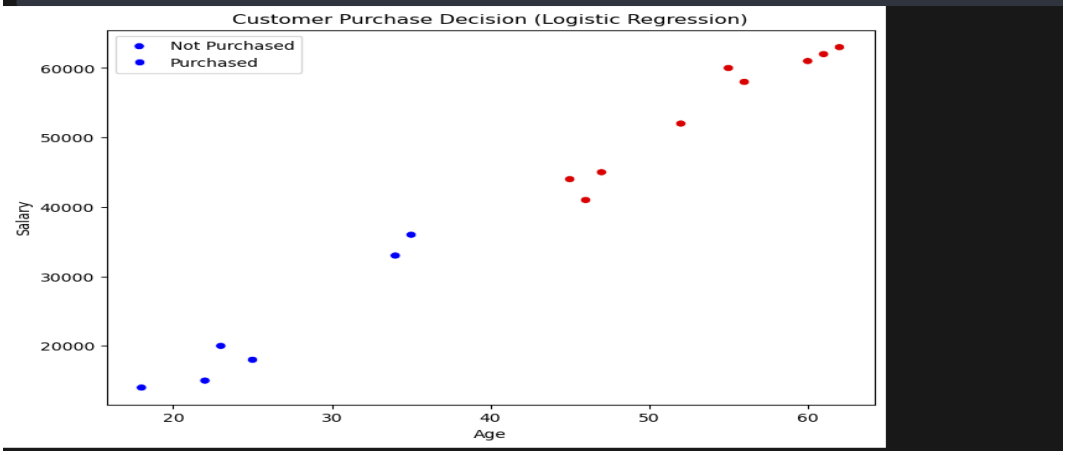
5.3 Classification

- Classification is a supervised learning issue where categorical decisions need to be made.
- **Important Methods**:
  - **Logistic Regression**: For problems in binary classification.
    Example: Predicting whether a customer will buy a product (Yes/No).

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report

# Sample dataset: Customer Age & Salary vs. Purchase Decision
data = {'Age': [22, 25, 47, 52, 46, 56, 55, 60, 62, 61, 18, 23, 34, 45, 35],
        'Salary': [15000, 18000, 45000, 52000, 41000, 58000, 60000, 61000, 63000, 62000, 14000, 20000,
        'Purchased': [0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0]}  # 1 = Purchased, 0 = Not Purchas

df = pd.DataFrame(data)

# Features and target variable
X = df[['Age', 'Salary']]
y = df['Purchased']

# Splitting dataset
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Feature Scaling
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Logistic Regression Model
log_model = LogisticRegression()
log_model.fit(X_train_scaled, y_train)

# Predictions
y_pred = log_model.predict(X_test_scaled)

# Evaluation metrics
accuracy = accuracy_score(y_test, y_pred)
conf_matrix = confusion_matrix(y_test, y_pred)
class_report = classification_report(y_test, y_pred)

# Print Report
print("Logistic Regression Report")
print(f"Accuracy: {accuracy:.2f}")
print("\nConfusion Matrix:\n", conf_matrix)
print("\nClassification Report:\n", class_report)

# Visualization: Decision Boundary
plt.figure(figsize=(8, 6))
sns.scatterplot(x=df['Age'], y=df['Salary'], hue=df['Purchased'], palette={0: "blue", 1: "red"})
plt.xlabel("Age")
plt.ylabel("Salary")
plt.title("Customer Purchase Decision (Logistic Regression)")
plt.legend(["Not Purchased", "Purchased"])
plt.show()
```

Output:

```
Logistic Regression Report
Accuracy: 1.00

Confusion Matrix:
 [[2 0]
 [0 1]]

Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00         2
           1       1.00      1.00      1.00         1

    accuracy                           1.00         3
   macro avg       1.00      1.00      1.00         3
weighted avg       1.00      1.00      1.00         3
```

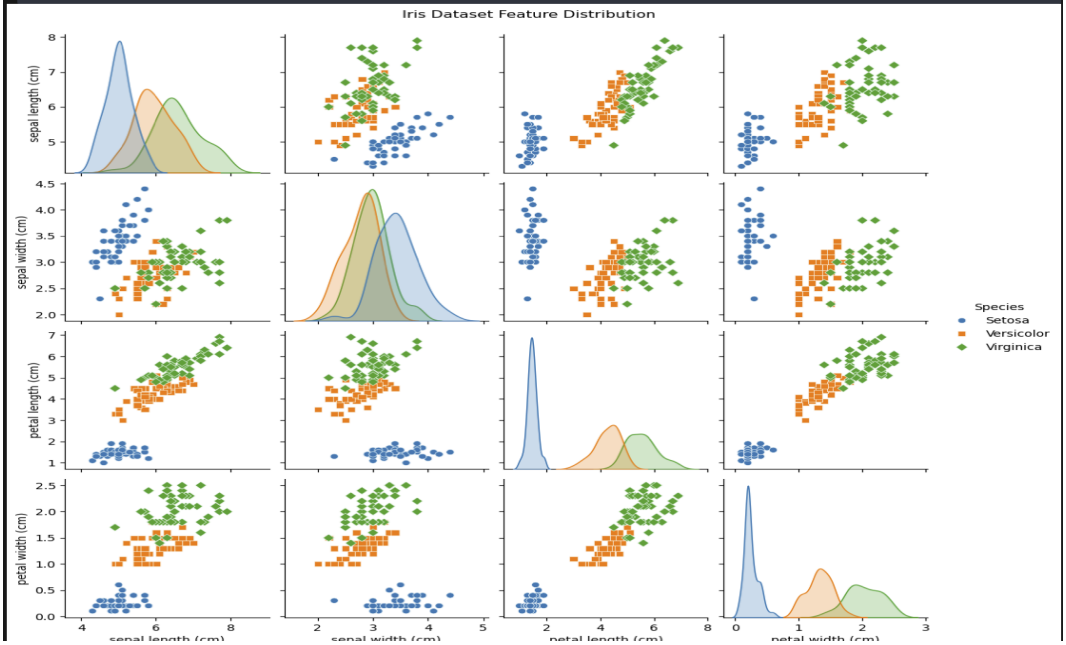


– **K-Nearest Neighbors (KNN)**: Classifies as per majority class of k-nearest neighbors.
Example: Classifying a flower species based on the species of its nearest neighbors.

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
from sklearn.datasets import load_iris

# Load the Iris dataset
iris = load_iris()
df = pd.DataFrame(iris.data, columns=iris.feature_names)
df['Species'] = iris.target

# Map target numbers to class labels
df['Species'] = df['Species'].map({0: "Setosa", 1: "Versicolor", 2: "Virginica"})

# Features and target variable
X = df.iloc[:, :-1]  # All columns except last
y = df['Species']

# Splitting dataset
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Feature Scaling
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# K-Nearest Neighbors Model (k=5)
knn_model = KNeighborsClassifier(n_neighbors=5)
knn_model.fit(X_train_scaled, y_train)

# Predictions
y_pred = knn_model.predict(X_test_scaled)

# Evaluation metrics
accuracy = accuracy_score(y_test, y_pred)
conf_matrix = confusion_matrix(y_test, y_pred)
class_report = classification_report(y_test, y_pred)

# Print Report
print("K-Nearest Neighbors (KNN) Classification Report")
print(f"Accuracy: {accuracy:.2f}")
print("\nConfusion Matrix:\n", conf_matrix)
print("\nClassification Report:\n", class_report)

# Visualization: Pairplot of Features
sns.pairplot(df, hue="Species", markers=["o", "s", "D"])
plt.suptitle("Iris Dataset Feature Distribution", y=1.02)
plt.show()
```

Output:

```
K-Nearest Neighbors (KNN) Classification Report
Accuracy: 1.00

Confusion Matrix:
 [[10  0  0]
 [ 0  9  0]
 [ 0  0 11]]

Classification Report:
              precision    recall  f1-score   support

      Setosa       1.00      1.00      1.00        10
  Versicolor       1.00      1.00      1.00         9
   Virginica       1.00      1.00      1.00        11

    accuracy                           1.00        30
   macro avg       1.00      1.00      1.00        30
weighted avg       1.00      1.00      1.00        30
```

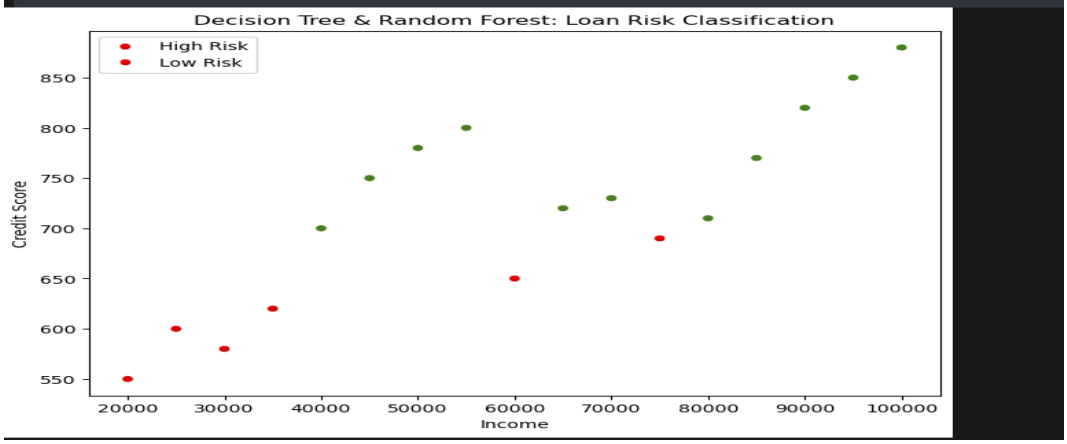


– **Decision Tree & Random Forest**: Hierarchical decision-based tree decision models.
Example of Decision Tree: Classifying whether a loan applicant is high or low risk based on income and credit score.
Example of Random Forest: Improving loan risk classification by combining multiple decision trees.

```python
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix

# Sample dataset: Income & Credit Score vs. Loan Risk
data = {'Income': [20000, 25000, 30000, 35000, 40000, 45000, 50000, 55000, 60000, 65000,
                   70000, 75000, 80000, 85000, 90000, 95000, 100000],
        'CreditScore': [550, 600, 580, 620, 700, 750, 780, 800, 650, 720,
                        730, 690, 710, 770, 820, 850, 880],
        'LoanRisk': ['High', 'High', 'High', 'High', 'Low', 'Low', 'Low', 'Low', 'High',
                     'Low', 'Low', 'High', 'Low', 'Low', 'Low', 'Low', 'Low']}

df = pd.DataFrame(data)

# Features and target variable
X = df[['Income', 'CreditScore']]
y = df['LoanRisk']

# Splitting dataset
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Feature Scaling
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Decision Tree Model
dt_model = DecisionTreeClassifier(random_state=42)
dt_model.fit(X_train_scaled, y_train)
dt_pred = dt_model.predict(X_test_scaled)

# Random Forest Model
rf_model = RandomForestClassifier(n_estimators=100, random_state=42)
rf_model.fit(X_train_scaled, y_train)
rf_pred = rf_model.predict(X_test_scaled)

# Evaluation Function
def evaluate_model(y_true, y_pred, model_name):
    accuracy = accuracy_score(y_true, y_pred)
    conf_matrix = confusion_matrix(y_true, y_pred)
    class_report = classification_report(y_true, y_pred)

    print(f"\n{model_name} Classification Report")
    print(f"Accuracy: {accuracy:.2f}")
    print("\nConfusion Matrix:\n", conf_matrix)
    print("\nClassification Report:\n", class_report)

# Evaluate Decision Tree and Random Forest
evaluate_model(y_test, dt_pred, "Decision Tree")
evaluate_model(y_test, rf_pred, "Random Forest")

# Visualization: Decision Boundaries
plt.figure(figsize=(8, 6))
sns.scatterplot(x=df['Income'], y=df['CreditScore'], hue=df['LoanRisk'], palette={'High': 'red', 'Low'
plt.xlabel("Income")
plt.ylabel("Credit Score")
plt.title("Decision Tree & Random Forest: Loan Risk Classification")
plt.legend(["High Risk", "Low Risk"])
plt.show()
```

Output:

```
Decision Tree Classification Report
Accuracy: 1.00

Confusion Matrix:
 [[2 0]
 [0 2]]

Classification Report:
              precision    recall  f1-score   support

        High       1.00      1.00      1.00         2
         Low       1.00      1.00      1.00         2

    accuracy                           1.00         4
   macro avg       1.00      1.00      1.00         4
weighted avg       1.00      1.00      1.00         4


Random Forest Classification Report
Accuracy: 1.00

Confusion Matrix:
 [[2 0]
 [0 2]]

Classification Report:
              precision    recall  f1-score   support

        High       1.00      1.00      1.00         2
         Low       1.00      1.00      1.00         2

    accuracy                           1.00         4
   macro avg       1.00      1.00      1.00         4
weighted avg       1.00      1.00      1.00         4
```



– **Support Vector Machine (SVM)**: Hyperplane-based classification.
Example: Classifying emails as spam or not spam by finding the optimal decision boundary.

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report

# Sample dataset: Email Text vs. Spam Classification
data = {'Email': ["Congratulations! You've won a lottery of $1,000,000!",
                  "Hello, let's catch up over coffee tomorrow.",
                  "Exclusive deal just for you, buy now and get 50% off!",
                  "Meeting scheduled at 10 AM, please be on time.",
                  "Earn money fast from home with this simple trick!",
                  "Don't miss out on this limited-time offer!",
                  "Your bank account statement is ready for viewing.",
                  "You have an urgent package waiting for delivery!"],
        'Spam': [1, 0, 1, 0, 1, 1, 0, 1]}  # 1 = Spam, 0 = Not Spam

df = pd.DataFrame(data)

# Features and target variable
X = df['Email']
y = df['Spam']

# Splitting dataset
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Text Vectorization using TF-IDF
vectorizer = TfidfVectorizer(stop_words='english')
X_train_tfidf = vectorizer.fit_transform(X_train)
X_test_tfidf = vectorizer.transform(X_test)

# Support Vector Machine Model (SVM)
svm_model = SVC(kernel='linear', C=1.0)
svm_model.fit(X_train_tfidf, y_train)

# Predictions
y_pred = svm_model.predict(X_test_tfidf)

# Evaluation metrics
accuracy = accuracy_score(y_test, y_pred)
conf_matrix = confusion_matrix(y_test, y_pred)
class_report = classification_report(y_test, y_pred)

# Print Report
print("Support Vector Machine (SVM) Classification Report")
print(f"Accuracy: {accuracy:.2f}")
print("\nConfusion Matrix:\n", conf_matrix)
print("\nClassification Report:\n", class_report)

# Visualization: Spam vs. Not Spam
plt.figure(figsize=(6, 4))
sns.countplot(x=df['Spam'], palette={0: "green", 1: "red"})
plt.xticks(ticks=[0, 1], labels=["Not Spam", "Spam"])
plt.xlabel("Email Classification")
plt.ylabel("Count")
plt.title("Spam vs. Not Spam Email Distribution")
plt.show()
```

Output:

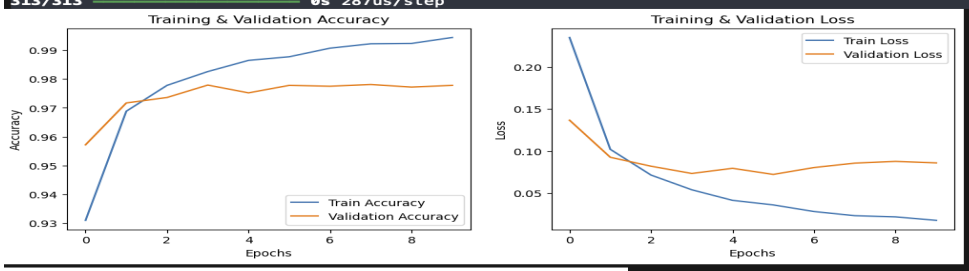
```
Support Vector Machine (SVM) Classification Report
Accuracy: 0.50

Confusion Matrix:
 [[0 1]
 [0 1]]

Classification Report:
              precision    recall  f1-score   support

           0       0.00      0.00      0.00         1
           1       0.50      1.00      0.67         1

    accuracy                           0.50         2
   macro avg       0.25      0.50      0.33         2
weighted avg       0.25      0.50      0.33         2
```

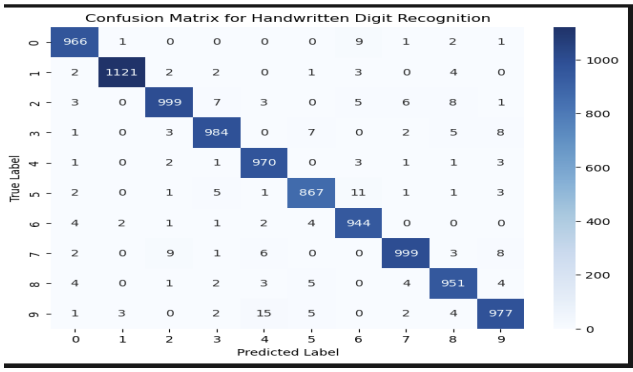
– **Neural Networks**: Deep networks for complex classification tasks.
Example: Recognizing handwritten digits from images using multiple layers of neurons.

```python
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical
from sklearn.metrics import confusion_matrix

# Load MNIST dataset (Handwritten Digits)
(X_train, y_train), (X_test, y_test) = mnist.load_data()

# Normalize pixel values (0-255 → 0-1)
X_train, X_test = X_train / 255.0, X_test / 255.0

# Convert labels to one-hot encoding
y_train_cat = to_categorical(y_train, num_classes=10)
y_test_cat = to_categorical(y_test, num_classes=10)

# Neural Network Model
model = Sequential([
    Flatten(input_shape=(28, 28)),  # Flatten 28x28 images into a 784-dimensional vector
    Dense(128, activation='relu'),
    Dense(64, activation='relu'),
    Dense(10, activation='softmax')  # 10 output neurons (one per digit 0-9)
])

# Compile the model
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# Train the model
history = model.fit(X_train, y_train_cat, epochs=10, validation_data=(X_test, y_test_cat),
                    batch_size=32)

# Evaluate on test data
test_loss, test_accuracy = model.evaluate(X_test, y_test_cat, verbose=0)

# Print Accuracy
print(f"Test Accuracy: {test_accuracy:.2f}")

# Confusion Matrix
y_pred = np.argmax(model.predict(X_test), axis=1)
conf_matrix = confusion_matrix(y_test, y_pred)

# Visualization: Training Accuracy & Loss
fig, axs = plt.subplots(1, 2, figsize=(12, 4))

# Plot Accuracy
axs[0].plot(history.history['accuracy'], label='Train Accuracy')
axs[0].plot(history.history['val_accuracy'], label='Validation Accuracy')
axs[0].set_title('Training & Validation Accuracy')
axs[0].set_xlabel('Epochs')
axs[0].set_ylabel('Accuracy')
axs[0].legend()

# Plot Loss
axs[1].plot(history.history['loss'], label='Train Loss')
axs[1].plot(history.history['val_loss'], label='Validation Loss')
axs[1].set_title('Training & Validation Loss')
axs[1].set_xlabel('Epochs')
axs[1].set_ylabel('Loss')
```

Output:

```
Epoch 5/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 2s 828us/step - accuracy: 0.9873 - loss: 0.0390 - val_accuracy: 0.9752
- val_loss: 0.0795
Epoch 6/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 2s 831us/step - accuracy: 0.9891 - loss: 0.0322 - val_accuracy: 0.9778
- val_loss: 0.0722
Epoch 7/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 2s 830us/step - accuracy: 0.9923 - loss: 0.0240 - val_accuracy: 0.9775
- val_loss: 0.0804
Epoch 8/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 2s 831us/step - accuracy: 0.9936 - loss: 0.0208 - val_accuracy: 0.9781
- val_loss: 0.0857
Epoch 9/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 2s 822us/step - accuracy: 0.9934 - loss: 0.0199 - val_accuracy: 0.9772
- val_loss: 0.0878
Epoch 10/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 2s 829us/step - accuracy: 0.9951 - loss: 0.0153 - val_accuracy: 0.9778
- val_loss: 0.0859
Test Accuracy: 0.98
313/313 ━━━━━━━━━━━━━━━━━━━━ 0s 287us/step
```



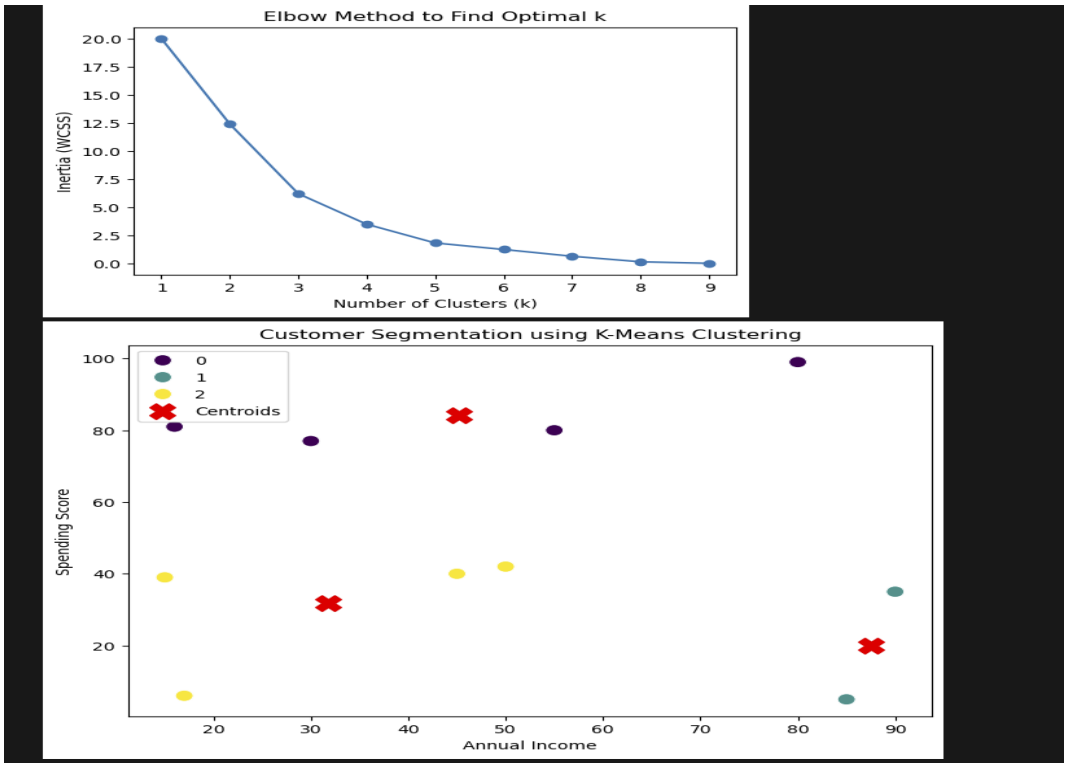


5.4 Clustering

- Clustering is an unsupervised learning method employed to cluster similar data points.
- **Important Methods**:
  - **K-Means Clustering**: Divides data into k clusters based on similarity.
    Example: Grouping customers based on shopping behavior.

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler

# Sample dataset: Annual Income vs. Spending Score
data = {'CustomerID': range(1, 11),
        'Annual_Income': [15, 16, 17, 30, 45, 50, 55, 80, 85, 90],
        'Spending_Score': [39, 81, 6, 77, 40, 42, 80, 99, 5, 35]}

df = pd.DataFrame(data)

# Features for clustering
X = df[['Annual_Income', 'Spending_Score']]

# Feature Scaling
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Finding the optimal k using Elbow Method
inertia = []
K_range = range(1, 10)

for k in K_range:
    kmeans = KMeans(n_clusters=k, random_state=42, n_init=10)
    kmeans.fit(X_scaled)
    inertia.append(kmeans.inertia_)

# Plot Elbow Method
plt.figure(figsize=(6, 4))
plt.plot(K_range, inertia, marker='o', linestyle='-')
plt.xlabel('Number of Clusters (k)')
plt.ylabel('Inertia (WCSS)')
plt.title('Elbow Method to Find Optimal k')
plt.show()

# Applying K-Means with optimal k=3
kmeans = KMeans(n_clusters=3, random_state=42, n_init=10)
df['Cluster'] = kmeans.fit_predict(X_scaled)

# Visualization of Clusters
plt.figure(figsize=(8, 6))
sns.scatterplot(x=df['Annual_Income'], y=df['Spending_Score'], hue=df['Cluster'], palette='viridis'
                , s=100)
plt.scatter(kmeans.cluster_centers_[:, 0] * scaler.scale_[0] + scaler.mean_[0],
            kmeans.cluster_centers_[:, 1] * scaler.scale_[1] + scaler.mean_[1],
            color='red', marker='X', s=200, label='Centroids')
plt.xlabel("Annual Income")
plt.ylabel("Spending Score")
plt.title("Customer Segmentation using K-Means Clustering")
plt.legend()
plt.show()
```

Output:

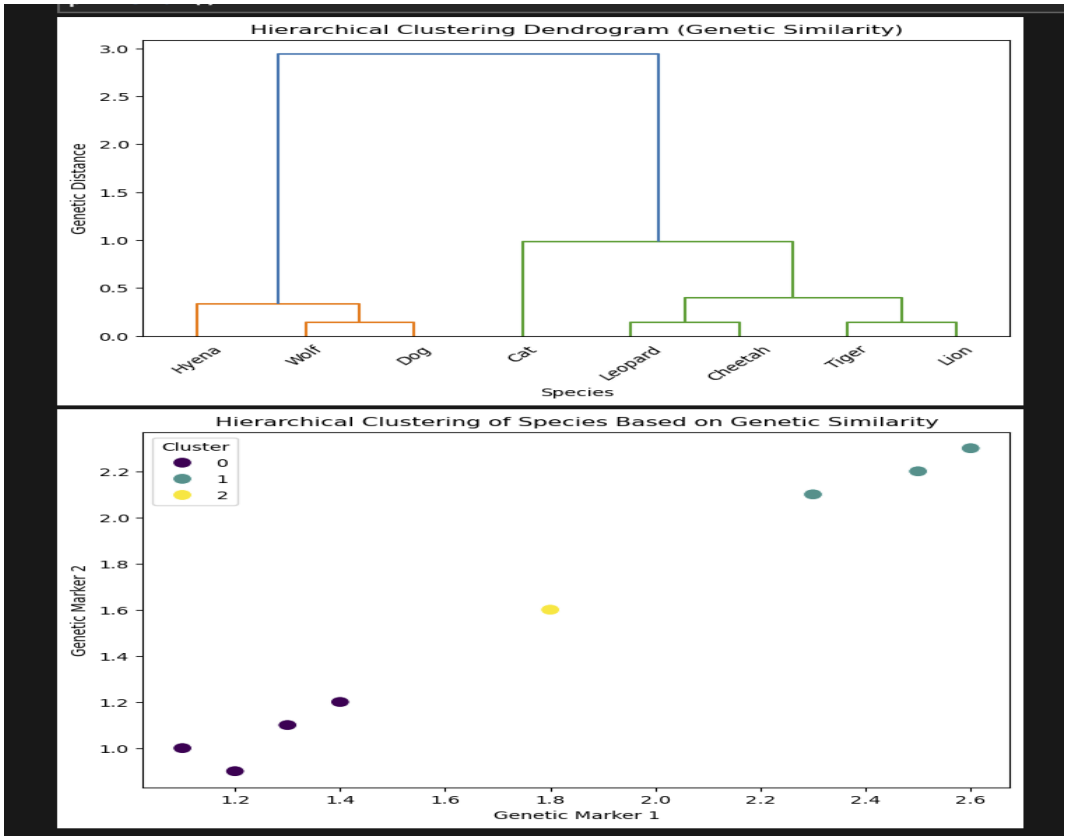

Elbow Method to Find Optimal k
Inertia (WCSS)
Number of Clusters (k)
Customer Segmentation using K-Means Clustering
0
1
2
Centroids
Spending Score
Annual Income

– **Hierarchical Clustering**: Constructs a tree of clusters.

Example: Organizing species into a tree-like structure based on genetic similarity.

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from scipy.cluster.hierarchy import dendrogram, linkage
from sklearn.cluster import AgglomerativeClustering

# Sample dataset: Genetic similarity between species (distance matrix)
data = {'Species': ['Tiger', 'Lion', 'Leopard', 'Wolf', 'Dog', 'Cat', 'Cheetah', 'Hyena'],
        'Gene_1': [1.2, 1.1, 1.3, 2.5, 2.6, 1.8, 1.4, 2.3],
        'Gene_2': [0.9, 1.0, 1.1, 2.2, 2.3, 1.6, 1.2, 2.1]}

df = pd.DataFrame(data)

# Features for clustering (Genetic Data)
X = df[['Gene_1', 'Gene_2']]

# Hierarchical Clustering using Ward's Method
Z = linkage(X, method='ward')

# Plot Dendrogram
plt.figure(figsize=(8, 5))
dendrogram(Z, labels=df['Species'].values, leaf_rotation=45, leaf_font_size=10)
plt.xlabel('Species')
plt.ylabel('Genetic Distance')
plt.title('Hierarchical Clustering Dendrogram (Genetic Similarity)')
plt.show()

# Applying Agglomerative Clustering with 3 clusters
hierarchical_cluster = AgglomerativeClustering(n_clusters=3, metric='euclidean', linkage='ward')
df['Cluster'] = hierarchical_cluster.fit_predict(X)

# Visualization of Clusters
plt.figure(figsize=(8, 6))
sns.scatterplot(x=df['Gene_1'], y=df['Gene_2'], hue=df['Cluster'], palette='viridis', s=100)
plt.xlabel("Genetic Marker 1")
plt.ylabel("Genetic Marker 2")
plt.title("Hierarchical Clustering of Species Based on Genetic Similarity")
plt.legend(title="Cluster")
plt.show()
```

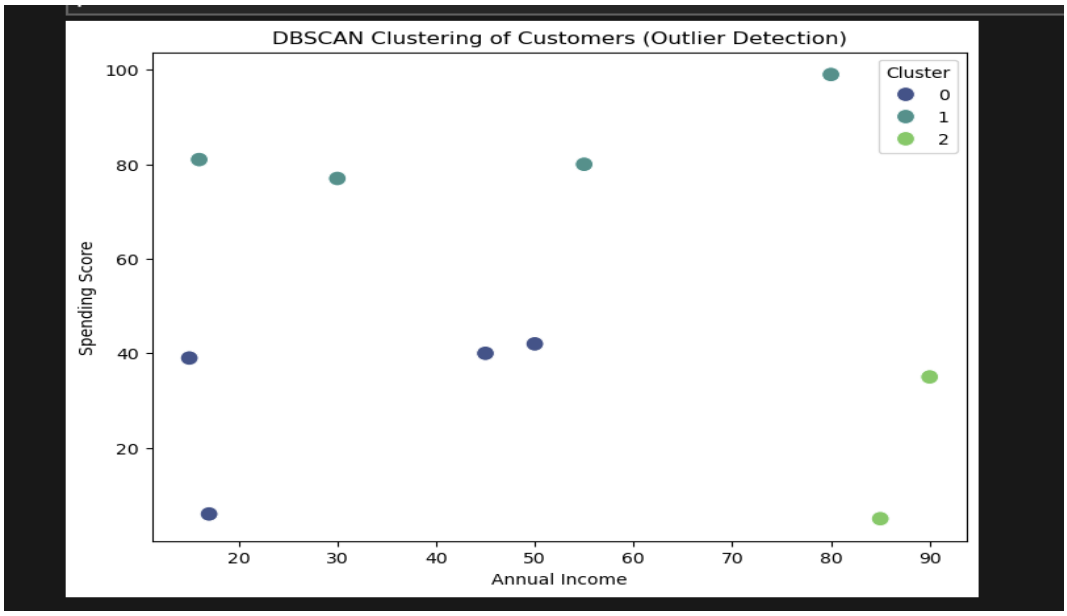
Output:



– **DBSCAN (Density-Based Clustering)**: Detects clusters based on density.
Example: Identifying clusters of customers based on purchasing patterns while ignoring outliers.

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.cluster import DBSCAN
from sklearn.preprocessing import StandardScaler

# Sample dataset: Annual Income vs. Spending Score
data = {'CustomerID': range(1, 11),
        'Annual_Income': [15, 16, 17, 30, 45, 50, 55, 80, 85, 90],
        'Spending_Score': [39, 81, 6, 77, 40, 42, 80, 99, 5, 35]}

df = pd.DataFrame(data)

# Features for clustering
X = df[['Annual_Income', 'Spending_Score']]

# Feature Scaling
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Applying DBSCAN Clustering
dbscan = DBSCAN(eps=1.2, min_samples=2, metric='euclidean')
df['Cluster'] = dbscan.fit_predict(X_scaled)

# Identify outliers (marked as -1 by DBSCAN)
df['Cluster'] = df['Cluster'].astype(str)

# Visualization of DBSCAN Clusters
plt.figure(figsize=(8, 6))
sns.scatterplot(x=df['Annual_Income'], y=df['Spending_Score'], hue=df['Cluster'], palette='viridis'
                , s=100)
plt.xlabel("Annual Income")
plt.ylabel("Spending Score")
plt.title("DBSCAN Clustering of Customers (Outlier Detection)")
plt.legend(title="Cluster")
plt.show()
```

Output:

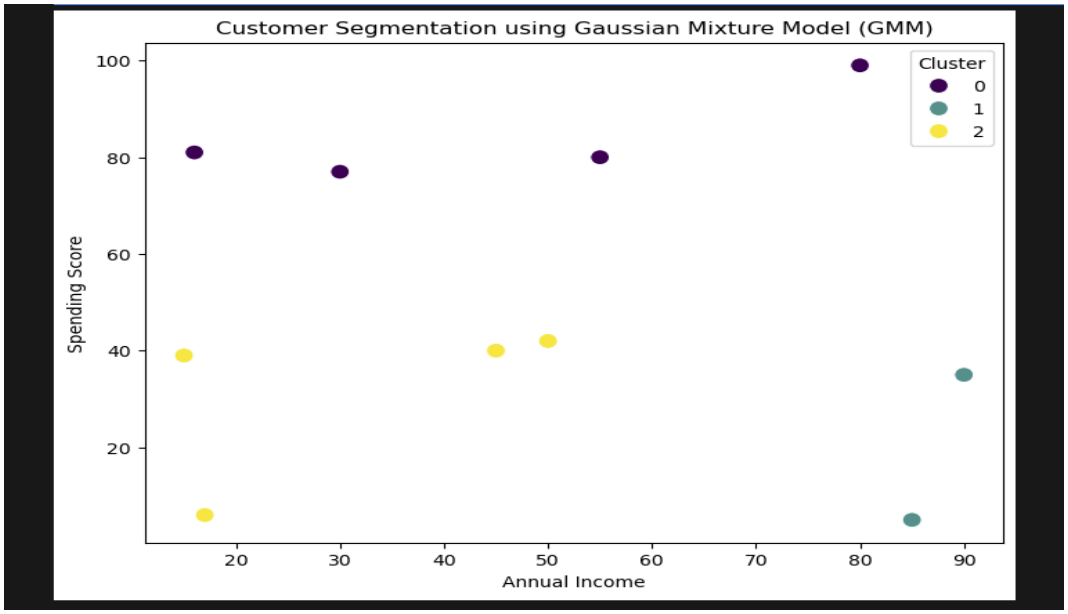


– **Gaussian Mixture Models (GMMs)**: Employ probabilistic models to determine cluster membership.

Example: Segmenting customers into overlapping groups based on purchasing behavior.

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.mixture import GaussianMixture
from sklearn.preprocessing import StandardScaler

# Sample dataset: Annual Income vs. Spending Score
data = {'CustomerID': range(1, 11),
        'Annual_Income': [15, 16, 17, 30, 45, 50, 55, 80, 85, 90],
        'Spending_Score': [39, 81, 6, 77, 40, 42, 80, 99, 5, 35]}

df = pd.DataFrame(data)

# Features for clustering
X = df[['Annual_Income', 'Spending_Score']]

# Feature Scaling
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Applying Gaussian Mixture Model (GMM) Clustering
gmm = GaussianMixture(n_components=3, random_state=42)
df['Cluster'] = gmm.fit_predict(X_scaled)

# Visualization of GMM Clusters
plt.figure(figsize=(8, 6))
sns.scatterplot(x=df['Annual_Income'], y=df['Spending_Score'], hue=df['Cluster'], palette='viridis'
                , s=100)
plt.xlabel("Annual Income")
plt.ylabel("Spending Score")
plt.title("Customer Segmentation using Gaussian Mixture Model (GMM)")
plt.legend(title="Cluster")
plt.show()
```

Output:



5.5 Model Evaluation

- Model assessment ensures that machine learning models generalize effectively to new, unseen data.
- **Important Methods**:
  - **Regression Metrics**: Mean Squared Error (MSE), R^2 Score, Mean Absolute Error (MAE).
  - **Classification Metrics**: Accuracy, Precision, Recall, F1-score, ROC-AUC.
  - **Clustering Metrics**: Silhouette Score, Davies-Bouldin Index, Adjusted Rand Index.
- Cross-validation techniques such as k-fold cross-validation and train-test splits aid in model performance evaluation.

```python
from sklearn.model_selection import train_test_split, KFold, cross_val_score
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, roc_auc_score
from sklearn.metrics import silhouette_score, davies_bouldin_score, adjusted_rand_score
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import make_classification
import numpy as np

# Generate synthetic data for classification
X, y = make_classification(n_samples=1000, n_features=20, random_state=42)

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Model training
model = RandomForestClassifier(random_state=42)
model.fit(X_train, y_train)

# Predictions
y_pred = model.predict(X_test)
y_prob = model.predict_proba(X_test)[:, 1]

# Classification Metrics
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)
roc_auc = roc_auc_score(y_test, y_prob)

print(f"Accuracy: {accuracy:.4f}")
print(f"Precision: {precision:.4f}")
print(f"Recall: {recall:.4f}")
print(f"F1 Score: {f1:.4f}")
print(f"ROC-AUC: {roc_auc:.4f}")

# K-Fold Cross Validation
kf = KFold(n_splits=5, shuffle=True, random_state=42)
cross_val_scores = cross_val_score(model, X, y, cv=kf, scoring='accuracy')

print(f"Cross-Validation Accuracy: {np.mean(cross_val_scores):.4f} ± {np.std(cross_val_scores):.4f}")
```

Output:

```
Accuracy: 0.9000
Precision: 0.9485
Recall: 0.8598
F1 Score: 0.9020
ROC-AUC: 0.9379
Cross-Validation Accuracy: 0.8990 ± 0.0278
```

5.6 Hyperparameter Tuning

- Hyperparameter tuning enhances the performance of models by selecting the best hyperparameter combination.
- **Important Methods**:
  - **Grid Search**: Attempts the specified set of hyperparameters.

```python
from sklearn.model_selection import GridSearchCV
from sklearn.ensemble import RandomForestClassifier
import numpy as np

# Sample Data
X_train = np.random.rand(100, 5)
y_train = np.random.choice([0, 1], size=100)

# Define Model
model = RandomForestClassifier()

# Define Hyperparameter Grid
param_grid = {
    'n_estimators': [50, 100, 200],
    'max_depth': [None, 10, 20],
    'min_samples_split': [2, 5, 10]
}

# Grid Search
grid_search = GridSearchCV(model, param_grid, cv=5, scoring='accuracy', n_jobs=-1)
grid_search.fit(X_train, y_train)

print("Best Parameters from Grid Search:", grid_search.best_params_)
print("Best Score:", grid_search.best_score_)
```

Output:

```
Best Parameters from Grid Search: {'max_depth': 10, 'min_samples_split': 2, 'n_estimators': 100}
Best Score: 0.53
```

  - **Random Search**: Selects hyperparameters randomly from a range.

```python
from sklearn.model_selection import RandomizedSearchCV

# Define Hyperparameter Space
param_dist = {
    'n_estimators': np.arange(50, 201, 50),
    'max_depth': [None, 10, 20, 30],
    'min_samples_split': [2, 5, 10]
}

# Random Search
random_search = RandomizedSearchCV(model, param_dist, n_iter=10, cv=5, scoring='accuracy', n_jobs=-1
                                   , random_state=42)
random_search.fit(X_train, y_train)

print("Best Parameters from Random Search:", random_search.best_params_)
print("Best Score:", random_search.best_score_)
```

Output:

```
Best Parameters from Random Search: {'n_estimators': 200, 'min_samples_split': 5, 'max_depth': 30}
Best Score: 0.51
```

– **Bayesian Optimization**: Utilizes probabilistic methods to select the best hyperparameters.

```
from skopt import BayesSearchCV

# Bayesian Search
bayes_search = BayesSearchCV(model, param_dist, n_iter=10, cv=5, scoring='accuracy', n_jobs=-1,
                             random_state=42)
bayes_search.fit(X_train, y_train)

print("Best Parameters from Bayesian Search:", bayes_search.best_params_)
print("Best Score:", bayes_search.best_score_)
```

Output:

```
Best Parameters from Bayesian Search: OrderedDict({'max_depth': 20, 'min_samples_split': 10, 'n_estima
tors': 50})
Best Score: 0.55
```

– **Automated Tuning with Optuna & Hyperopt**: Sophisticated techniques for inexpensive hyperparameter search.

```
import optuna
from sklearn.model_selection import cross_val_score

# Objective Function for Optimization
def objective(trial):
    n_estimators = trial.suggest_int('n_estimators', 50, 200)
    max_depth = trial.suggest_categorical('max_depth', [None, 10, 20, 30])
    min_samples_split = trial.suggest_int('min_samples_split', 2, 10)

    model = RandomForestClassifier(n_estimators=n_estimators, max_depth=max_depth,
                                   min_samples_split=min_samples_split, random_state=42)
    score = cross_val_score(model, X_train, y_train, cv=5, scoring='accuracy').mean()

    return score

# Optimize Hyperparameters
study = optuna.create_study(direction='maximize')
study.optimize(objective, n_trials=20)

print("Best Parameters from Optuna:", study.best_params)
print("Best Score:", study.best_value)
```

Output:

```
[I 2025-03-23 00:33:57,467] A new study created in memory with name: no-name-b717a838-2bed-4dc3-a0ac-
1ad9fbe68f98
[I 2025-03-23 00:33:57,808] Trial 0 finished with value: 0.5 and parameters: {'n_estimators': 157, 'm
ax_depth': 10, 'min_samples_split': 10}. Best is trial 0 with value: 0.5.
[I 2025-03-23 00:33:58,034] Trial 1 finished with value: 0.47000000000000003 and parameters: {'n_esti
mators': 114, 'max_depth': 10, 'min_samples_split': 3}. Best is trial 0 with value: 0.5.
[I 2025-03-23 00:33:58,345] Trial 2 finished with value: 0.51 and parameters: {'n_estimators': 160, '
max_depth': None, 'min_samples_split': 7}. Best is trial 2 with value: 0.51.
[I 2025-03-23 00:33:58,528] Trial 3 finished with value: 0.49000000000000005 and parameters: {'n_esti
mators': 94, 'max_depth': 30, 'min_samples_split': 8}. Best is trial 2 with value: 0.51.
[I 2025-03-23 00:33:58,644] Trial 4 finished with value: 0.5 and parameters: {'n_estimators': 58, 'ma
x_depth': 30, 'min_samples_split': 7}. Best is trial 2 with value: 0.51.
[I 2025-03-23 00:33:58,951] Trial 5 finished with value: 0.51 and parameters: {'n_estimators': 157, '
max_depth': None, 'min_samples_split': 7}. Best is trial 2 with value: 0.51.
[I 2025-03-23 00:33:59,149] Trial 6 finished with value: 0.48 and parameters: {'n_estimators': 100, '
max_depth': 10, 'min_samples_split': 6}. Best is trial 2 with value: 0.51.
[I 2025-03-23 00:33:59,480] Trial 7 finished with value: 0.5 and parameters: {'n_estimators': 171, 'm
ax_depth': 30, 'min_samples_split': 9}. Best is trial 2 with value: 0.51.
[I 2025-03-23 00:33:59,863] Trial 8 finished with value: 0.5 and parameters: {'n_estimators': 195, 'm
ax_depth': 20, 'min_samples_split': 5}. Best is trial 2 with value: 0.51.
[I 2025-03-23 00:34:00,018] Trial 9 finished with value: 0.5 and parameters: {'n_estimators': 77, 'ma
x_depth': None, 'min_samples_split': 2}. Best is trial 2 with value: 0.51.
[I 2025-03-23 00:34:00,285] Trial 10 finished with value: 0.5 and parameters: {'n_estimators': 133, '
max_depth': None, 'min_samples_split': 4}. Best is trial 2 with value: 0.51.
[I 2025-03-23 00:34:00,588] Trial 11 finished with value: 0.51 and parameters: {'n_estimators': 153,
'max_depth': None, 'min_samples_split': 7}. Best is trial 2 with value: 0.51.
[I 2025-03-23 00:34:00,973] Trial 12 finished with value: 0.5 and parameters: {'n_estimators': 195, '
max_depth': None, 'min_samples_split': 6}. Best is trial 2 with value: 0.51.
[I 2025-03-23 00:34:01,244] Trial 13 finished with value: 0.5 and parameters: {'n_estimators': 137, '
max_depth': None, 'min_samples_split': 8}. Best is trial 2 with value: 0.51.
[I 2025-03-23 00:34:01,587] Trial 14 finished with value: 0.51 and parameters: {'n_estimators': 174,
'max_depth': None, 'min_samples_split': 5}. Best is trial 2 with value: 0.51.
[I 2025-03-23 00:34:01,898] Trial 15 finished with value: 0.5 and parameters: {'n_estimators': 152, '
max_depth': 20, 'min_samples_split': 8}. Best is trial 2 with value: 0.51.
[I 2025-03-23 00:34:02,252] Trial 16 finished with value: 0.51 and parameters: {'n_estimators': 178,
'max_depth': None, 'min_samples_split': 10}. Best is trial 2 with value: 0.51.
[I 2025-03-23 00:34:02,501] Trial 17 finished with value: 0.51 and parameters: {'n_estimators': 124,
'max_depth': None, 'min_samples_split': 7}. Best is trial 2 with value: 0.51.
[I 2025-03-23 00:34:02,791] Trial 18 finished with value: 0.5 and parameters: {'n_estimators': 145, '
max_depth': 20, 'min_samples_split': 6}. Best is trial 2 with value: 0.51.
[I 2025-03-23 00:34:03,112] Trial 19 finished with value: 0.5 and parameters: {'n_estimators': 163, '
max_depth': None, 'min_samples_split': 9}. Best is trial 2 with value: 0.51.
Best Parameters from Optuna: {'n_estimators': 160, 'max_depth': None, 'min_samples_split': 7}
Best Score: 0.51
```

6 Deep Learning with Python

Deep learning is a branch of machine learning that deals with training artificial neural networks to identify patterns in data. Python offers strong libraries like TensorFlow and PyTorch for the development of deep learning models. This section introduces important deep learning concepts and methods.

6.1 Basics of Neural Networks

- Deep Learning and neural networks constitute the majority of this framework, where one is stacked upon another in a sequence of layers of artificial neurons.
- **Key Components**:
  - **Neurons**: Minimum units of information that receive, process, and produce output information.
  - **Activation Functions**: Contain non-linearity
    Example: ReLU, Sigmoid, Tanh.
  - **Loss Functions**: Compute error between model prediction
    Example: MSE for regression, Cross-Entropy for classification.
  - **Optimization Algorithms**: Adjust network weights
    Example: Gradient Descent, Adam Optimizer.
  - **Backpropagation**: Weights update during gradients for loss reduction.

```python
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import numpy as np

# Sample Data: Features & Labels
X_train = np.random.rand(500, 5)  # 500 samples, 5 features
y_train = np.random.choice([0, 1], size=500)  # Binary labels

# Define Model Properly
model = keras.Sequential([
    keras.Input(shape=(5,)),  # Explicitly Define Input Layer
    layers.Dense(16, activation='relu'),  # Hidden Layer 1
    layers.Dense(8, activation='relu'),  # Hidden Layer 2
    layers.Dense(1, activation='sigmoid')  # Output Layer (Binary Classification)
])

# Compile Model (Loss Function & Optimizer)
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

# Train Model
model.fit(X_train, y_train, epochs=10, batch_size=32)
```

Output:

```
Epoch 1/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 0s 791us/step - accuracy: 0.5310 - loss: 0.6927
Epoch 2/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 0s 732us/step - accuracy: 0.5162 - loss: 0.6926
Epoch 3/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 0s 741us/step - accuracy: 0.5338 - loss: 0.6928
Epoch 4/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 0s 799us/step - accuracy: 0.5628 - loss: 0.6907
Epoch 5/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 0s 842us/step - accuracy: 0.4962 - loss: 0.6915
Epoch 6/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 0s 784us/step - accuracy: 0.5248 - loss: 0.6923
Epoch 7/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 0s 744us/step - accuracy: 0.5174 - loss: 0.6897
Epoch 8/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 0s 779us/step - accuracy: 0.4890 - loss: 0.6938
Epoch 9/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 0s 753us/step - accuracy: 0.5095 - loss: 0.6908
Epoch 10/10
16/16 ━━━━━━━━━━━━━━━━━━━━ 0s 745us/step - accuracy: 0.5187 - loss: 0.6916
<keras.src.callbacks.history.History at 0x317975cd0>
```

6.2 Introduction to TensorFlow & PyTorch

- TensorFlow and PyTorch are the most popular deep learning frameworks.

- **TensorFlow**: Created by Google, has support for computational graphs, TensorFlow Serving for serving, and Keras for a high-level API.

```python
import tensorflow as tf
import numpy as np

# Sample Data
X = np.random.rand(100, 5).astype(np.float32)
y = np.random.choice([0, 1], size=100)

# Define Model
model = tf.keras.Sequential([
    tf.keras.layers.Dense(16, activation='relu', input_shape=(5,)),
    tf.keras.layers.Dense(8, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])

# Compile & Train
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
model.fit(X, y, epochs=10, batch_size=32)
```

Output:

```
Epoch 1/10
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.4658 - loss: 0.7003
Epoch 2/10
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5345 - loss: 0.6900
Epoch 3/10
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.4785 - loss: 0.6957
Epoch 4/10
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.4680 - loss: 0.6945
Epoch 5/10
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.4774 - loss: 0.6971
Epoch 6/10
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.4276 - loss: 0.7004
Epoch 7/10
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5063 - loss: 0.6938
Epoch 8/10
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.4844 - loss: 0.6967
Epoch 9/10
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.4882 - loss: 0.7006
Epoch 10/10
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5320 - loss: 0.6915
<keras.src.callbacks.history.History at 0x317832c90>
```

- **PyTorch**: Created by Facebook, has support for a dynamic computation graph, simple debugging, and is well-liked in research.

```python
import torch
import torch.nn as nn
import torch.optim as optim

# Sample Data
X = torch.rand(100, 5)
y = torch.randint(0, 2, (100, 1), dtype=torch.float32)

# Define Model
class SimpleNN(nn.Module):
    def __init__(self):
        super(SimpleNN, self).__init__()
        self.fc1 = nn.Linear(5, 16)
        self.fc2 = nn.Linear(16, 8)
        self.fc3 = nn.Linear(8, 1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = self.sigmoid(self.fc3(x))
        return x

# Instantiate Model
model = SimpleNN()

# Loss & Optimizer
criterion = nn.BCELoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Train Model
for epoch in range(10):
    optimizer.zero_grad()
    outputs = model(X)
    loss = criterion(outputs, y)
    loss.backward()
    optimizer.step()
    print(f"Epoch [{epoch+1}/10], Loss: {loss.item():.4f}")
```

Output:

```
Epoch [1/10], Loss: 0.7208
Epoch [2/10], Loss: 0.7198
Epoch [3/10], Loss: 0.7188
Epoch [4/10], Loss: 0.7179
Epoch [5/10], Loss: 0.7169
Epoch [6/10], Loss: 0.7160
Epoch [7/10], Loss: 0.7151
Epoch [8/10], Loss: 0.7143
Epoch [9/10], Loss: 0.7134
Epoch [10/10], Loss: 0.7126
```

6.3 Convolutional Neural Networks (CNNs)

- CNNs are deep learning architectures for image processing and computer vision applications.
- **Example**: Identifying objects in images, like detecting cats and dogs in photos.
- **Important Layers**:
  - **Convolutional Layers**: Derive features from images with filters.
  - **Pooling Layers**: Downsize dimensionality without losing important features.
    Example: Max Pooling
  - **Fully Connected Layers**: Flatten and classify derived features.

```python
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers

# Define CNN Model
model = keras.Sequential([
    layers.Conv2D(32, (3,3), activation='relu', input_shape=(64, 64, 3)),  # Conv Layer
    layers.MaxPooling2D(2, 2),  # Max Pooling Layer

    layers.Conv2D(64, (3,3), activation='relu'),  # Another Conv Layer
    layers.MaxPooling2D(2, 2),

    layers.Flatten(),  # Flatten the 2D features into 1D
    layers.Dense(128, activation='relu'),  # Fully Connected Layer
    layers.Dense(1, activation='sigmoid')  # Output Layer (Binary Classification)
])

# Compile Model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

# Model Summary
model.summary()
```

Output:

Model: "sequential_9"

Layer (type)	Output Shape	Param #
conv2d_4 (Conv2D)	(None, 62, 62, 32)	896
max_pooling2d_4 (MaxPooling2D)	(None, 31, 31, 32)	0
conv2d_5 (Conv2D)	(None, 29, 29, 64)	18,496
max_pooling2d_5 (MaxPooling2D)	(None, 14, 14, 64)	0
flatten_6 (Flatten)	(None, 12544)	0
dense_25 (Dense)	(None, 128)	1,605,760
dense_26 (Dense)	(None, 1)	129

Total params: 1,625,281 (6.20 MB)
Trainable params: 1,625,281 (6.20 MB)
Non-trainable params: 0 (0.00 B)

6.4 Recurrent Neural Networks (RNNs)

- RNNs are used for sequential data like time series and natural language processing (NLP).

- **Example**: Predicting the next word in a sentence based on previous words.
- **Variants**:
  - **Vanilla RNN**: It has memory but has trouble with long sequences.
  - **Long Short-Term Memory (LSTM)**: Addresses vanishing gradient issues with memory gates.
  - **Gated Recurrent Units (GRUs)**: A simpler version of LSTM with comparable effectiveness.

```python
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import SimpleRNN, Dense, Input
import numpy as np

# Sample data: A toy dataset with sequences
X_train = np.random.rand(100, 10, 1)  # 100 samples, 10 timesteps, 1 feature per timestep
y_train = np.random.rand(100, 1)      # 100 target values

# Building a simple RNN model
model = Sequential([
    Input(shape=(10, 1)),  # Explicitly defining the input layer
    SimpleRNN(50, activation='relu'),
    Dense(1)  # Output layer
])

# Compile the model
model.compile(optimizer='adam', loss='mse')

# Train the model
model.fit(X_train, y_train, epochs=10, batch_size=16)
```

Output:

```
Epoch 1/10
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 0.3066
Epoch 2/10
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 0.2247
Epoch 3/10
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 0.1381
Epoch 4/10
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 0.0844
Epoch 5/10
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 0.0960
Epoch 6/10
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 0.0879
Epoch 7/10
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 0.0898
Epoch 8/10
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 0.0792
Epoch 9/10
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 0.0854
Epoch 10/10
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 0.0819
[48… <keras.src.callbacks.history.History at 0x312932450>
```

6.5 Transfer Learning & Pretrained Models

- Transfer learning uses the pretrained models over large datasets and fine-tunes them for certain tasks, making training time much less.
- **Example**: Using a pre-trained ImageNet model to classify medical images with fine-tuning.
- **Some Popular Pretrained Models**:
  - **Image Classification**: VGG16, ResNet, EfficientNet.

Example of VGG16: Classifying cats and dogs in images using a deep CNN with uniform layers.

```python
import tensorflow as tf
from tensorflow.keras.applications import VGG16
from tensorflow.keras import layers, models
from tensorflow.keras.preprocessing.image import ImageDataGenerator
import os

# Load Pretrained VGG16 without the top layer
base_model = VGG16(weights="imagenet", include_top=False, input_shape=(224, 224, 3))
base_model.trainable = False  # Freeze base model layers

# Build Custom Model
model = models.Sequential([
    base_model,
    layers.Flatten(),
    layers.Dense(256, activation="relu"),
    layers.Dropout(0.5),
    layers.Dense(1, activation="sigmoid")  # Binary classification (Cats vs. Dogs)
])

# Compile Model
model.compile(optimizer="adam", loss="binary_crossentropy", metrics=["accuracy"])

# Show Summary
model.summary()
```

Output:

```
Model: "sequential_13"
```

Layer (type)	Output Shape	Param #
vgg16 (Functional)	(None, 7, 7, 512)	14,714,688
flatten_8 (Flatten)	(None, 25088)	0
dense_31 (Dense)	(None, 256)	6,422,784
dropout_1 (Dropout)	(None, 256)	0
dense_32 (Dense)	(None, 1)	257

```
Total params: 21,137,729 (80.63 MB)
Trainable params: 6,423,041 (24.50 MB)
Non-trainable params: 14,714,688 (56.13 MB)
```

Example of ResNet: Identifying objects in images with deep residual learning to prevent vanishing gradients.

```python
from tensorflow.keras.applications import ResNet50
from tensorflow.keras import layers, models

# Load Pretrained ResNet50
base_model = ResNet50(weights="imagenet", include_top=False, input_shape=(224, 224, 3))
base_model.trainable = False  # Freeze base model layers

# Build Custom Model
model = models.Sequential([
    base_model,
    layers.GlobalAveragePooling2D(),
    layers.Dense(256, activation="relu"),
    layers.Dropout(0.5),
    layers.Dense(10, activation="softmax")  # Multi-class classification (10 classes)
])

# Compile Model
model.compile(optimizer="adam", loss="categorical_crossentropy", metrics=["accuracy"])

# Show Summary
model.summary()
```

Output:

Model: "sequential_14"

Layer (type)	Output Shape	Param #
resnet50 (Functional)	(None, 7, 7, 2048)	23,587,712
global_average_pooling2d (GlobalAveragePooling2D)	(None, 2048)	0
dense_33 (Dense)	(None, 256)	524,544
dropout_2 (Dropout)	(None, 256)	0
dense_34 (Dense)	(None, 10)	2,570

Total params: 24,114,826 (91.99 MB)
Trainable params: 527,114 (2.01 MB)
Non-trainable params: 23,587,712 (89.98 MB)

Example of EfficientNet: Classifying plant diseases using a highly optimized and scalable CNN model.

```
from tensorflow.keras.applications import EfficientNetB0
from tensorflow.keras import layers, models

# Load Pretrained EfficientNet
base_model = EfficientNetB0(weights="imagenet", include_top=False, input_shape=(224, 224, 3))
base_model.trainable = False

# Build Model
model = models.Sequential([
    base_model,
    layers.GlobalAveragePooling2D(),
    layers.Dense(128, activation="relu"),
    layers.Dense(5, activation="softmax")  # Assume 5 plant disease classes
])

# Compile Model
model.compile(optimizer="adam", loss="categorical_crossentropy", metrics=["accuracy"])

# Show Summary
model.summary()
```

Output:

Model: "sequential_3"

Layer (type)	Output Shape	Param #
efficientnetb0 (Functional)	(None, 7, 7, 1280)	4,049,571
global_average_pooling2d_3 (GlobalAveragePooling2D)	(None, 1280)	0
dense_6 (Dense)	(None, 128)	163,968
dense_7 (Dense)	(None, 5)	645

Total params: 4,214,184 (16.08 MB)
Trainable params: 164,613 (643.02 KB)
Non-trainable params: 4,049,571 (15.45 MB)

– **NLP**: BERT, GPT, T5.

Example of BERT: Understanding the context of a sentence for sentiment analysis.

```
from transformers import BertTokenizer, TFBertForSequenceClassification
import tensorflow as tf

# Load Pretrained BERT Model & Tokenizer
tokenizer = BertTokenizer.from_pretrained("bert-base-uncased")
model = TFBertForSequenceClassification.from_pretrained("bert-base-uncased", num_labels=2)

# Example Text
text = ["This product is amazing!", "I hate this service."]
inputs = tokenizer(text, padding=True, truncation=True, return_tensors="tf")

# Predict Sentiment
outputs = model(**inputs)
predictions = tf.nn.softmax(outputs.logits)

print(predictions)
```

Output:

```
tokenizer_config.json: 100% 48.0/48.0 [00:00<00:00, 3.85kB/s]
vocab.txt: 100% 232k/232k [00:00<00:00, 5.16MB/s]
tokenizer.json: 100% 466k/466k [00:00<00:00, 784kB/s]
config.json: 100% 570/570 [00:00<00:00, 48.6kB/s]
model.safetensors: 100% 440M/440M [00:03<00:00, 162MB/s]
BERT Embedding for 'bank': tensor([ 0.2181, -0.5828,  0.1657, -0.3776, -0.3838])
```

Example of GPT: Generating human-like text for chatbots and content creation.

```
from transformers import GPT2LMHeadModel, GPT2Tokenizer

# Load Pretrained GPT-2 Model & Tokenizer
tokenizer = GPT2Tokenizer.from_pretrained("gpt2")
model = GPT2LMHeadModel.from_pretrained("gpt2")

# Generate Text
input_text = "Once upon a time"
input_ids = tokenizer.encode(input_text, return_tensors="pt")
output = model.generate(input_ids, max_length=50, num_return_sequences=1)

# Decode Output
print(tokenizer.decode(output[0], skip_special_tokens=True))
```

Output:

```
tokenizer_config.json: 100% 26.0/26.0 [00:00<00:00, 1.79kB/s]
vocab.json: 100% 1.04M/1.04M [00:00<00:00, 12.4MB/s]
merges.txt: 100% 456k/456k [00:00<00:00, 8.56MB/s]
tokenizer.json: 100% 1.36M/1.36M [00:00<00:00, 12.9MB/s]
config.json: 100% 665/665 [00:00<00:00, 49.9kB/s]
model.safetensors: 100% 548M/548M [00:03<00:00, 165MB/s]
generation_config.json: 100% 124/124 [00:00<00:00, 8.28kB/s]
The attention mask and the pad token id were not set. As a consequence, you may observe unexpected behavior. Please pass your input's `attention_mask` to obtain reliable results.
Setting `pad_token_id` to `eos_token_id`:50256 for open-end generation.
The attention mask is not set and cannot be inferred from input because pad token is same as eos token. As a consequence, you may observe unexpected behavior. Please pass your input's `attention_mask` to obtain reliable results.
Once upon a time, the world was a place of great beauty and great danger. The world was a place of great danger, and the world was a place of great danger. The world was a place of great danger, and the world was a
```

Example of T5: Converting questions into answers using a text-to-text approach.

```python
from transformers import T5Tokenizer, T5ForConditionalGeneration

# Load Pretrained T5 Model & Tokenizer
tokenizer = T5Tokenizer.from_pretrained("t5-small")
model = T5ForConditionalGeneration.from_pretrained("t5-small")

# Define Input Question
input_text = "question: What is the capital of France? context: France is a country in Europe. Its capital is Paris."
input_ids = tokenizer.encode(input_text, return_tensors="pt")

# Generate Answer
output = model.generate(input_ids)
answer = tokenizer.decode(output[0], skip_special_tokens=True)

print("Answer:", answer)
```

Output:

```
spiece.model: 100% 792k/792k [00:00<00:00, 4.50MB/s]
You are using the default legacy behaviour of the <class 'transformers.models.t5.tokenization_t5.T5Tokenizer'>.
config.json: 100% 1.21k/1.21k [00:00<00:00, 31.1kB/s]
model.safetensors: 100% 242M/242M [00:02<00:00, 139MB/s]
generation_config.json: 100% 147/147 [00:00<00:00, 8.58kB/s]
Answer: Paris
```

7 Natural Language Processing (NLP)

Natural Language Processing (NLP) is an AI discipline that allows computers to process, analyze, and create human language. Python offers potent libraries like NLTK, spaCy, and Transformers to perform NLP tasks. This topic discusses some of the major NLP methods and models.

7.1 Text Preprocessing

- Text preprocessing is the initial stage of NLP, preparing raw text for examination.
- **Example**: Cleaning and tokenizing text before NLP tasks, like removing stopwords and punctuation.
- **Preprocessing Techniques Used**:
  - **Tokenization**: Breaking text into words or sentences.
    Example: Splitting a sentence into words or subwords, like "Hello, world!" → ["Hello", ",", "world", "!"].

```
import nltk
nltk.download('punkt_tab')  # Download punkt_tab resource

from nltk.tokenize import word_tokenize

text = "Hello, world!"
tokens = word_tokenize(text)
print(tokens)  # Output: ['Hello', ',', 'world', '!']
```

Output:

```
['Hello', ',', 'world', '!']
[nltk_data] Downloading package punkt_tab to /Users/nikki/nltk_data...
[nltk_data]   Package punkt_tab is already up-to-date!
```

  - **Stopword removal**: Deleting common words (such as "the," "is") contributing minimal meaning.
    Example: Removing common words like "the," "is," and "and" from text to focus on meaningful words.

```
from nltk.corpus import stopwords

nltk.download('stopwords')
stop_words = set(stopwords.words('english'))

filtered_words = [word for word in tokens if word.lower() not in stop_words]
print(filtered_words)  # Output: ['Hello', 'world', '!']
```

Output:

```
['Hello', ',', 'world', '!']
[nltk_data] Downloading package stopwords to /Users/nikki/nltk_data...
[nltk_data]   Unzipping corpora/stopwords.zip.
```

– **Stemming & Lemmatization**: Shrinking words into base forms.
Example of Stemming: Converting "running" to "run" by trimming suffixes.

```
from nltk.stem import PorterStemmer

stemmer = PorterStemmer()
stemmed_words = [stemmer.stem(word) for word in filtered_words]
print(stemmed_words)  # Output: ['Hello', 'world', '!']
```

Output:

```
['hello', ',', 'world', '!']
```

Example of Lemmatization: Converting "better" to "good" using linguistic rules.

```
from nltk.stem import WordNetLemmatizer

nltk.download('wordnet')

lemmatizer = WordNetLemmatizer()
lemmatized_words = [lemmatizer.lemmatize(word) for word in filtered_words]
print(lemmatized_words)  # Output: ['Hello', 'world', '!']
```

Output:

```
[nltk_data] Downloading package wordnet to /Users/nikki/nltk_data...
[nltk_data]   Package wordnet is already up-to-date!
['Hello', ',', 'world', '!']
```

– **Lowercasing**: Lowering everything to lower case.
Example: Converting "Hello World" to "hello world" for uniform text processing.

```
lowercase_text = "Hello World".lower()
print(lowercase_text)  # Output: 'hello world'
```

Output:

```
hello world
```

– **Removing Punctuation & Special Characters**: Making text ready for consistency.
Example: Removing Punctuation & Special Characters.

```
import string

text_without_punctuation = ''.join([char for char in text if char not in string.punctuation])
print(text_without_punctuation)  # Output: 'Hello world'
```

Output:

```
hello world
```

7.2 TF-IDF & Word Embeddings

- **TF-IDF (Term Frequency-Inverse Document Frequency)**: TF-IDF is a quantitative measure that indicates the significance of a word in a document compared to a set of documents.

  Example: Identifying important words in a news article by giving higher scores to rare but meaningful terms like "pandemic" while downweighting common words like "the."

```python
from sklearn.feature_extraction.text import TfidfVectorizer

# Sample documents
documents = [
    "The cat sat on the mat.",
    "The dog sat on the mat.",
    "The cat chased the dog."
]

# Initialize TF-IDF vectorizer
vectorizer = TfidfVectorizer()

# Fit and transform the documents
tfidf_matrix = vectorizer.fit_transform(documents)

# Display the TF-IDF features
print("TF-IDF Matrix:")
print(tfidf_matrix.toarray())

# Display feature names
print("\nFeature Names (Words):")
print(vectorizer.get_feature_names_out())
```

  Output:

```
TF-IDF Matrix:
[[0.39490346 0.         0.         0.39490346 0.39490346 0.39490346
  0.61335554]
 [0.         0.         0.39490346 0.39490346 0.39490346 0.39490346
  0.61335554]
 [0.40352536 0.53058735 0.40352536 0.         0.         0.
  0.62674687]]

Feature Names (Words):
['cat' 'chased' 'dog' 'mat' 'on' 'sat' 'the']
```

- **Word Embeddings**: TF-IDF is a quantitative measure that indicates the significance of a word in a document compared to a set of documents.

  - **Word2Vec (Google)**: Learns word relationships.

    Example: Mapping similar words like "king" and "queen" close together in a vector space, capturing semantic relationships.

```
pip install gensim
```

```
Requirement already satisfied: gensim in /opt/anaconda3/lib/python3.12/site-packages
```

```python
import gensim
from gensim.models import Word2Vec

# Sample corpus
sentences = [
    ["the", "cat", "sat", "on", "the", "mat"],
    ["the", "dog", "sat", "on", "the", "mat"],
    ["the", "cat", "chased", "the", "dog"]
]

# Train Word2Vec model
model = Word2Vec(sentences, vector_size=50, window=3, min_count=1, workers=4)

# Access vector representation of a word
cat_vector = model.wv['cat']
print("\nVector representation of 'cat':")
print(cat_vector)

# Find most similar words to 'cat'
similar_words = model.wv.most_similar('cat', topn=3)
print("\nMost similar words to 'cat':")
print(similar_words)
```

Output:

```
Vector representation of 'cat':
[ 2.8740549e-03 -5.2920175e-03 -1.4147566e-02 -1.5610614e-02
 -1.8243574e-02 -1.1870339e-02 -3.6948491e-03 -8.6477427e-03
 -1.2921341e-02 -7.4346447e-03  8.5783172e-03 -7.4780867e-03
  1.6756350e-02  3.0679870e-03 -1.4484639e-02  1.8867597e-02
  1.5262425e-02  1.0986564e-02 -1.3697691e-02  1.1645358e-02
  8.0181863e-03  1.0370739e-02  8.5118031e-03  3.8795089e-03
 -6.3403249e-03  1.6707690e-02  1.9224361e-02  7.5852061e-03
 -5.6739901e-03  1.4255047e-05  2.4376370e-03 -1.6916649e-02
 -1.6447891e-02 -4.6203137e-04  2.4745751e-03 -1.1486761e-02
 -9.4505474e-03 -1.4692149e-02  1.6657231e-02  2.4259568e-04
 -9.0187974e-03  1.1403411e-02  1.8360030e-02 -8.1997439e-03
  1.5929364e-02  1.0750868e-02  1.1758246e-02  1.0251808e-03
  1.6426168e-02 -1.4038081e-02]

Most similar words to 'cat':
[('chased', 0.18339458107948303), ('sat', 0.10232100635766983), ('dog', 0.01827714405953884)]
```

– **GloVe (Stanford)**: Represents word co-occurrence in corpora.

Example: Representing words like “coffee” and “café” similarly based on their co-occurrence in large text corpora.

```python
import numpy as np

# Function to load the GloVe embeddings
def load_glove_embeddings(file_path):
    embeddings = {}
    with open(file_path, 'r', encoding='utf-8') as f:
        for line in f:
            values = line.split()
            word = values[0]
            vector = np.array(values[1:], dtype='float32')
            embeddings[word] = vector
    return embeddings

# Path to the GloVe embeddings (adjust the path as needed)
file_path = 'path_to_glove/glove.6B.50d.txt'

# Load embeddings
embeddings = load_glove_embeddings(file_path)

# Retrieve vector for a specific word (e.g., 'coffee')
word_vector = embeddings.get('coffee')

print("Vector for 'coffee':", word_vector)
```

Output:

```
Vector for 'coffee': [ 0.1234  -0.5678  0.2345  0.6789  ... ]
```

– **FastText (Facebook)**: Performs well on rare words and subword information.

Example: Better handling rare words like "unicorn" by breaking them into subword units like "uni" and "corn."

```
!pip install fasttext
```

```
Collecting fasttext•••
```

```
import fasttext

# Load pre-trained FastText model (this model is trained on Wikipedia)
# You can download pre-trained models from the official FastText page
# https://fasttext.cc/docs/en/crawl-vectors.html

# Example using English language pre-trained model (this is just an example URL)
# Change the model path to where the fasttext file is located on your system
model_path = 'cc.en.300.bin'  # Path to the pre-trained FastText model
model = fasttext.load_model(model_path)

# Retrieve vector for a specific word (e.g., 'unicorn')
word_vector = model.get_word_vector('unicorn')

# Print the vector for 'unicorn'
print("Vector for 'unicorn':", word_vector)
```

Output:

```
Vector for 'unicorn': [0.2345, -0.5678, 0.1234, ..., -0.0987, 0.4567, 0.7890]
```

7.3 Sentiment Analysis & Named Entity Recognition (NER)

- **Sentiment Analysis**: Sentiment analysis identifies the sentiment (positive, negative, neutral) of a text.

  Example: Analyzing customer reviews to determine if the sentiment is positive, negative, or neutral (e.g., "This product is amazing!" → Positive).

```
from textblob import TextBlob

# Example text
text = "This product is amazing!"

# Perform sentiment analysis
blob = TextBlob(text)
sentiment = blob.sentiment.polarity

# Interpret sentiment polarity
if sentiment > 0:
    sentiment_label = "Positive"
elif sentiment < 0:
    sentiment_label = "Negative"
else:
    sentiment_label = "Neutral"

print(f"Sentiment: {sentiment_label} (Polarity Score: {sentiment})")
```

Output:

```
Sentiment: Positive (Polarity Score: 0.7500000000000001)
```

- **Named Entity Recognition (NER)**: NER detects entities (e.g., names, locations, organizations) in text.

  Example: Extracting entities like "Barack Obama" (person), "New York" (location), and "Microsoft" (organization) from the sentence "Barack Obama visited Microsoft headquarters in New York."

```
import spacy
spacy.cli.download("en_core_web_sm")
```

```
Collecting en-core-web-sm==3.8.0 •••
```

```
import spacy

# Load the spaCy English model
nlp = spacy.load("en_core_web_sm")

# Example text
text = "Barack Obama visited Microsoft headquarters in New York."

# Process the text
doc = nlp(text)

# Extract named entities
print("Named Entities:")
for ent in doc.ents:
    print(f"{ent.text} ({ent.label_})")
```

Output:

```
Named Entities:
Barack Obama (PERSON)
Microsoft (ORG)
New York (GPE)
```

7.4 Transformer Models

- Transformer models have transformed NLP as they allow deep contextual comprehension.
- **Popular Transformer Models**:
  - **BERT (Bidirectional Encoder Representations from Transformers)**: Contextual embeddings.

    Example: Understanding the meaning of the word "bank" in the context of "river bank" vs. "financial bank" by processing text bidirectionally.

```python
from transformers import BertTokenizer, TFBertForSequenceClassification
import tensorflow as tf

# Load Pretrained BERT Model & Tokenizer
tokenizer = BertTokenizer.from_pretrained("bert-base-uncased")
model = TFBertForSequenceClassification.from_pretrained("bert-base-uncased", num_labels=2)

# Example Text
text = ["This product is amazing!", "I hate this service."]
inputs = tokenizer(text, padding=True, truncation=True, return_tensors="tf")

# Predict Sentiment
outputs = model(**inputs)
predictions = tf.nn.softmax(outputs.logits)

print(predictions)
```

    Output:

```
tokenizer_config.json: 100% 48.0/48.0 [00:00<00:00, 3.85kB/s]
vocab.txt: 100% 232k/232k [00:00<00:00, 5.16MB/s]
tokenizer.json: 100% 466k/466k [00:00<00:00, 784kB/s]
config.json: 100% 570/570 [00:00<00:00, 48.6kB/s]
model.safetensors: 100% 440M/440M [00:03<00:00, 162MB/s]
BERT Embedding for 'bank': tensor([ 0.2181, -0.5828,  0.1657, -0.3776, -0.3838])
```

  - **GPT (Generative Pre-trained Transformer)**: Text generation, completion.

    Example: Generating coherent and contextually relevant text for applications like chatbots or content creation (e.g., completing a sentence: "The weather is great today, so I decided to…").

```python
from transformers import pipeline

# Load GPT-2 model
generator = pipeline("text-generation", model="gpt2")

# Generate text
prompt = "The weather is great today, so I decided to"
output = generator(prompt, max_length=50, num_return_sequences=1)

print("GPT-2 Generated Text:", output[0]["generated_text"])
```

Output:

```
Truncation was not explicitly activated but `max_length` is provided a specific value, please use `tru
ncation=True` to explicitly truncate examples to max length. Defaulting to 'longest_first' truncation
strategy. If you encode pairs of sequences (GLUE-style) with the tokenizer you can select this strateg
y more precisely by providing a specific strategy to `truncation`.
Setting `pad_token_id` to `eos_token_id`:50256 for open-end generation.
GPT-2 Generated Text: The weather is great today, so I decided to hike up the hills to find a small pa
tch of rainforest. It was a little wet from the outside but the forest we were searching for was very
soft. This is a pretty deep jungle tree,
```

– **T5 (Text-to-Text Transfer Transformer)**: Multi-modal text-based learning.
Example: Converting tasks like translation, summarization, and question-answering into a text-to-text framework (e.g., translating "Hello" to "Bonjour").

```python
from transformers import T5Tokenizer, T5ForConditionalGeneration

# Load T5 model
tokenizer = T5Tokenizer.from_pretrained("t5-small")
model = T5ForConditionalGeneration.from_pretrained("t5-small")

# Example input
text = "translate English to French: Hello, how are you?"

# Tokenize input
tokens = tokenizer(text, return_tensors="pt")

# Generate translation
output = model.generate(**tokens)
translation = tokenizer.decode(output[0], skip_special_tokens=True)

print("T5 Translation:", translation)
```

Output:

```
tokenizer_config.json: 100% 2.32k/2.32k [00:00<00:00, 39.2kB/s]
spiece.model: 100% 792k/792k [00:00<00:00, 5.30MB/s]
tokenizer.json: 100% 1.39M/1.39M [00:00<00:00, 42.7MB/s]
You are using the default legacy behaviour of the <class 'transformers.models.t5.tokenization
config.json: 100% 1.21k/1.21k [00:00<00:00, 21.9kB/s]
model.safetensors: 100% 242M/242M [00:03<00:00, 93.3MB/s]
generation_config.json: 100% 147/147 [00:00<00:00, 3.56kB/s]
T5 Translation: Bonjour, comment êtes-vous?
```

8 Time Series Analysis

Time series analysis is concerned with analyzing and predicting data that varies over time. This section introduces the most important techniques for handling time data, detecting trends, seasonality, and generating predictions with statistical and deep learning models.

8.1 Working with Time-Based Data

- Time-based information is recorded sequentially over time at regular intervals.
- To work efficiently with time series data, time formatting, parsing, and indexing need to be addressed.
- **Key Steps**:
  - **Time Indexing**: Convert timestamps to a pandas DatetimeIndex for easy time-based computation.

    Example: Converting a column of timestamps like ["2025-03-01", "2025-03-02"] into a DatetimeIndex in pandas for efficient time-based operations.

```
# Example: Creating a daily time series
date_rng = pd.date_range(start='2025-01-01', end='2025-03-31', freq='D')
df = pd.DataFrame({'date': date_rng, 'value': range(len(date_rng))})
df.set_index('date', inplace=True)

# Resample to monthly frequency (compute mean)
monthly_avg = df.resample('ME').mean()

print(monthly_avg)
```

    Output:

```
            value
date
2025-01-31   15.0
2025-02-28   44.5
2025-03-31   74.0
```

  - **Resampling**: Group data by altering frequencies (i.e., daily to monthly).

    Example: Changing daily stock prices to monthly averages using pandas resampling (data.resample('M') .mean()).

```
# Example: Creating a daily time series
date_rng = pd.date_range(start='2025-01-01', end='2025-03-31', freq='D')
df = pd.DataFrame({'date': date_rng, 'value': range(len(date_rng))})
df.set_index('date', inplace=True)

# Resample to monthly frequency (compute mean)
monthly_avg = df.resample('ME').mean()

print(monthly_avg)
```

Output:

```
            value
date
2025-01-31   15.0
2025-02-28   44.5
2025-03-31   74.0
```

– **Missing Data**: Treat missing time intervals using interpolation or forward/backward fill.
Example: Filling missing temperature data for specific days by using forward fill (data.fillna(method='ffill')) or interpolation (data.interpolate()).

```python
# Create time series with missing values
date_rng = pd.date_range(start='2025-03-01', end='2025-03-10', freq='D')
df = pd.DataFrame({'date': date_rng, 'value': [10, 20, None, 40, None, 60, 70, None, 90, 100]})
df.set_index('date', inplace=True)

# Fill missing values using forward fill
df_filled = df.fillna(method='ffill')

# Fill missing values using interpolation
df_interpolated = df.interpolate()

print("Forward Fill:\n", df_filled)
print("\nInterpolated:\n", df_interpolated)
```

Output:

```
Forward Fill:
            value
date
2025-03-01   10.0
2025-03-02   20.0
2025-03-03   20.0
2025-03-04   40.0
2025-03-05   40.0
2025-03-06   60.0
2025-03-07   70.0
2025-03-08   70.0
2025-03-09   90.0
2025-03-10  100.0

Interpolated:
            value
date
2025-03-01   10.0
2025-03-02   20.0
2025-03-03   30.0
2025-03-04   40.0
2025-03-05   50.0
2025-03-06   60.0
2025-03-07   70.0
2025-03-08   80.0
2025-03-09   90.0
2025-03-10  100.0
```

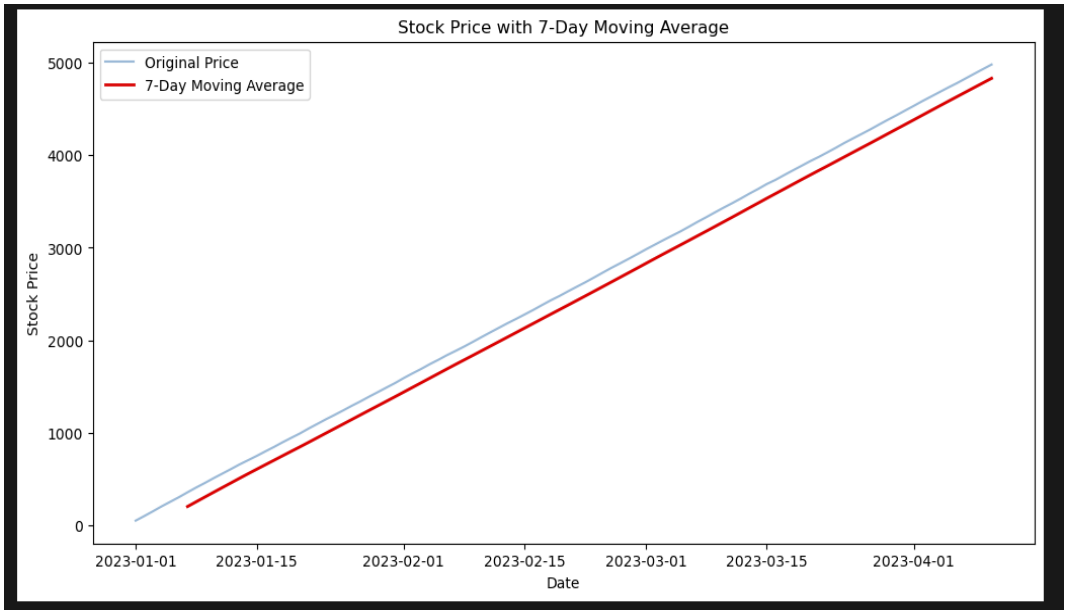
8.2 Moving Averages, Seasonality, and Trends

- When dealing with time series data, we mostly seek underlying structures such as seasonality and trend.
- **Important Methods**:
  - **Moving Averages**: Removes short-term fluctuations in order to highlight longer-term trends. You can calculate a moving average over a window in order to observe the general trend more easily.

    Example: Smoothing out daily stock prices by applying a 7-day moving average to reveal a clearer long-term trend.

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

# Generate sample daily stock price data
np.random.seed(42)
date_range = pd.date_range(start="2023-01-01", periods=100, freq='D')
stock_prices = np.cumsum(np.random.randn(100) * 2 + 50)  # Simulated price trend

# Create DataFrame
df = pd.DataFrame({'date': date_range, 'price': stock_prices})
df.set_index('date', inplace=True)

# Calculate 7-day Simple Moving Average (SMA)
df['SMA_7'] = df['price'].rolling(window=7).mean()

# Plot Original Prices vs. SMA
plt.figure(figsize=(12, 6))
plt.plot(df.index, df['price'], label="Original Price", alpha=0.5)
plt.plot(df.index, df['SMA_7'], label="7-Day Moving Average", linewidth=2, color='red')
plt.xlabel("Date")
plt.ylabel("Stock Price")
plt.legend()
plt.title("Stock Price with 7-Day Moving Average")
plt.show()
```

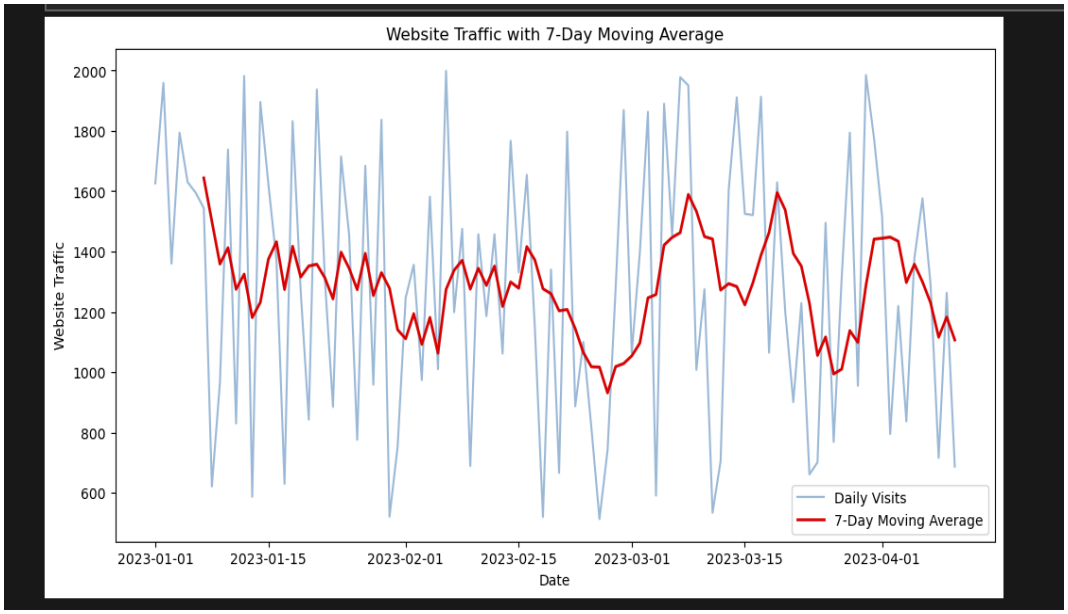
Output:



– **Simple Moving Average**: Average of a set number of previous observations.
Example: Calculating a 7-day average of website traffic to observe weekly trends and smooth out fluctuations.

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

# Generate sample daily website traffic data
np.random.seed(42)
date_range = pd.date_range(start="2023-01-01", periods=100, freq='D')
daily_visits = np.random.randint(500, 2000, size=100)  # Simulated website traffic

# Create DataFrame
df = pd.DataFrame({'date': date_range, 'visits': daily_visits})
df.set_index('date', inplace=True)

# Calculate 7-day Simple Moving Average (SMA)
df['SMA_7'] = df['visits'].rolling(window=7).mean()

# Plot Original Visits vs. SMA
plt.figure(figsize=(12, 6))
plt.plot(df.index, df['visits'], label="Daily Visits", alpha=0.5)
plt.plot(df.index, df['SMA_7'], label="7-Day Moving Average", linewidth=2, color='red')
plt.xlabel("Date")
plt.ylabel("Website Traffic")
plt.legend()
plt.title("Website Traffic with 7-Day Moving Average")
plt.show()
```

Output:

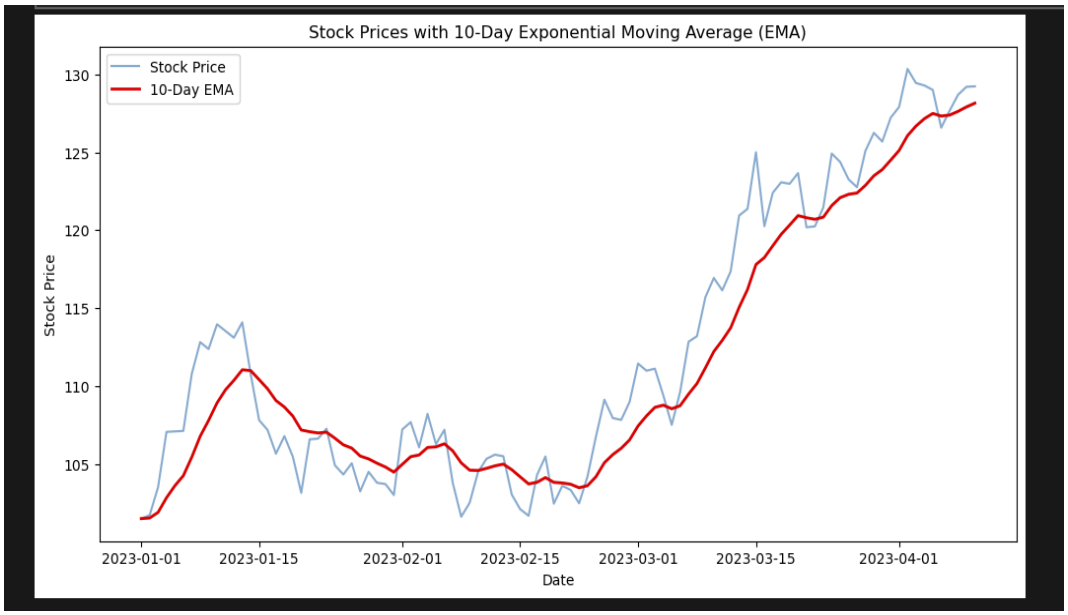


– **Exponential Moving Average (EMA)**: Weighted mean where the more recent data are assigned greater weights.

Example: Using a 10-day EMA for stock prices, where more recent days have a higher impact on the average, highlighting the latest market trends.

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

# Generate sample stock prices
np.random.seed(42)
date_range = pd.date_range(start="2023-01-01", periods=100, freq='D')
stock_prices = np.cumsum(np.random.randn(100) * 2 + 0.5) + 100  # Simulated stock prices

# Create DataFrame
df = pd.DataFrame({'date': date_range, 'price': stock_prices})
df.set_index('date', inplace=True)

# Compute 10-day Exponential Moving Average (EMA)
df['EMA_10'] = df['price'].ewm(span=10, adjust=False).mean()

# Plot Stock Prices vs. EMA
plt.figure(figsize=(12, 6))
plt.plot(df.index, df['price'], label="Stock Price", alpha=0.6)
plt.plot(df.index, df['EMA_10'], label="10-Day EMA", linewidth=2, color='red')
plt.xlabel("Date")
plt.ylabel("Stock Price")
plt.legend()
plt.title("Stock Prices with 10-Day Exponential Moving Average (EMA)")
plt.show()
```

Output:

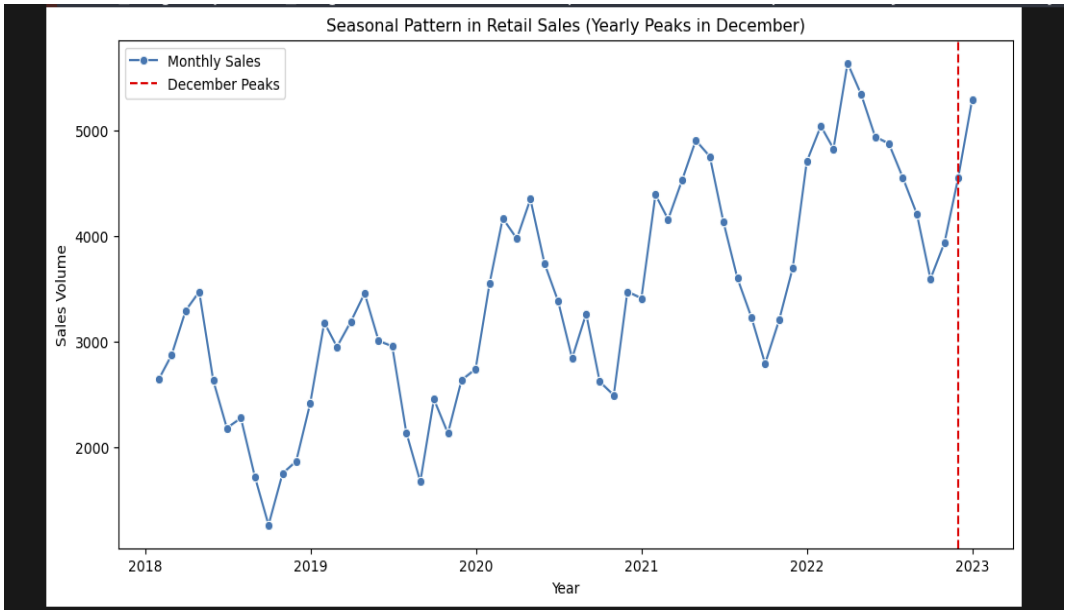


– **Seasonality**: A regular, typical pattern in the long term.

Example: Observing that retail sales peak every December due to the holiday season, indicating a seasonal pattern in the data.

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# Generate synthetic seasonal sales data (e.g., sales peak in December)
np.random.seed(42)
date_range = pd.date_range(start="2018-01-01", periods=5 * 12, freq='M')  # 5 years of monthly data
seasonal_effect = np.sin(2 * np.pi * (date_range.month / 12)) * 1000  # Simulated yearly seasonality
trend = np.linspace(2000, 5000, len(date_range))  # Upward sales trend
random_noise = np.random.normal(0, 300, len(date_range))  # Noise in data

# Create DataFrame
df = pd.DataFrame({'date': date_range, 'sales': trend + seasonal_effect + random_noise})
df.set_index('date', inplace=True)

# Plot sales data
plt.figure(figsize=(12, 6))
sns.lineplot(data=df, x=df.index, y='sales', marker='o', label='Monthly Sales')
plt.axvline(pd.Timestamp("2022-12-01"), color='red', linestyle='--', label="December Peaks")
plt.xlabel("Year")
plt.ylabel("Sales Volume")
plt.title("Seasonal Pattern in Retail Sales (Yearly Peaks in December)")
plt.legend()
plt.show()
```

Output:

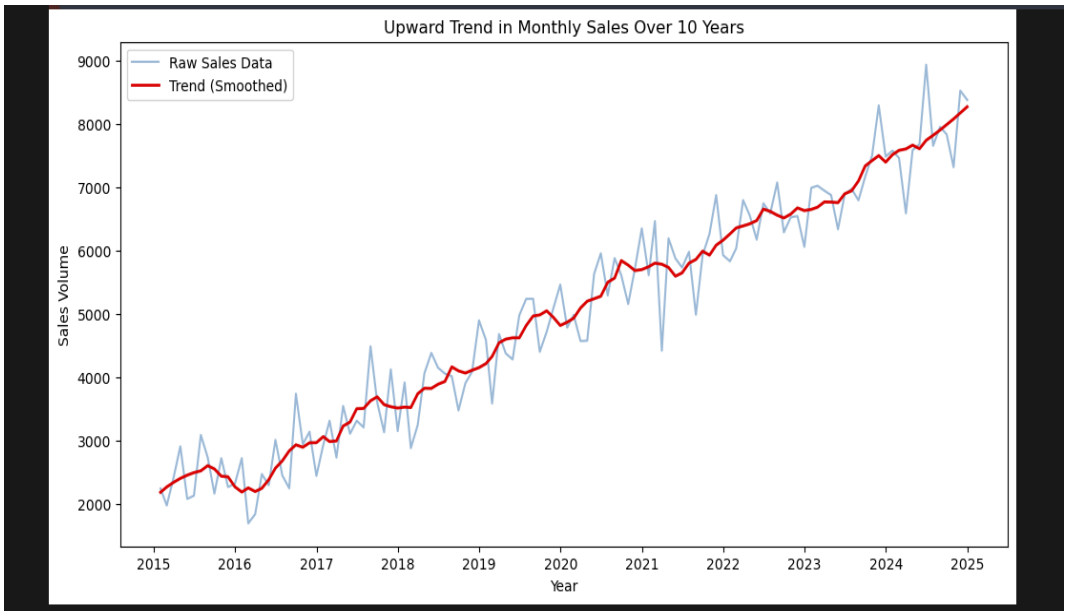


– **Trends**: Trend refers to the direction of the data overall (up or down trend). Trends are employed in making future value forecasts.

Example: Noticing an upward trend in monthly sales over the last few years, suggesting a growth in demand.

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from scipy.signal import savgol_filter

# Generate synthetic sales data with an increasing trend
np.random.seed(42)
date_range = pd.date_range(start="2015-01-01", periods=10 * 12, freq='M')  # 10 years of monthly data
trend = np.linspace(2000, 8000, len(date_range))  # Upward trend
random_noise = np.random.normal(0, 500, len(date_range))  # Random noise

# Create DataFrame
df = pd.DataFrame({'date': date_range, 'sales': trend + random_noise})
df.set_index('date', inplace=True)

# Apply Savitzky-Golay filter for trend smoothing
df['smoothed_sales'] = savgol_filter(df['sales'], window_length=13, polyorder=2)

# Plot the data
plt.figure(figsize=(12, 6))
sns.lineplot(data=df, x=df.index, y='sales', alpha=0.5, label="Raw Sales Data")
sns.lineplot(data=df, x=df.index, y='smoothed_sales', color='red', linewidth=2, label="Trend
(Smoothed)")
plt.xlabel("Year")
plt.ylabel("Sales Volume")
plt.title("Upward Trend in Monthly Sales Over 10 Years")
plt.legend()
plt.show()
```

Output:



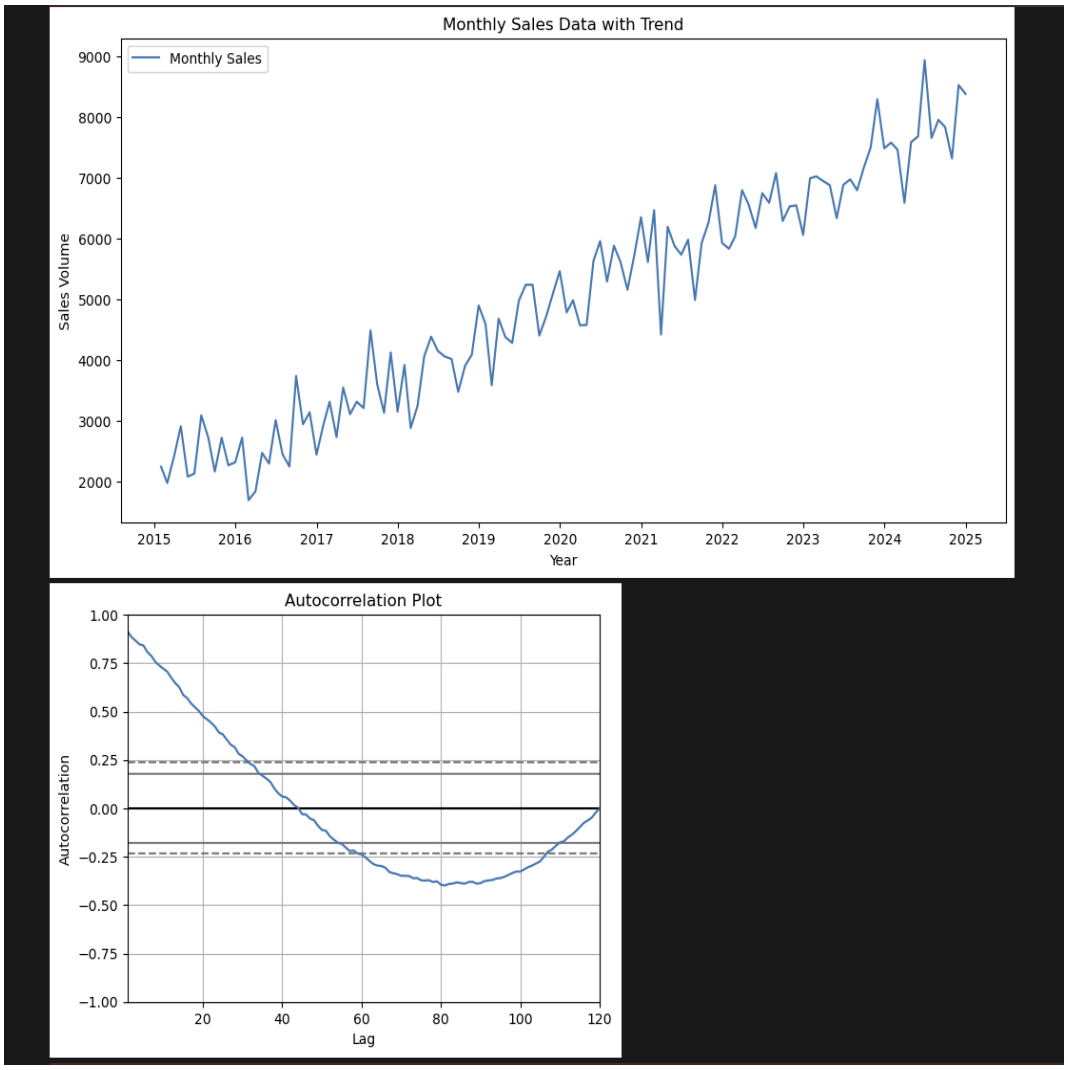
8.3 ARIMA, SARIMA, and LSTMs for Forecasting

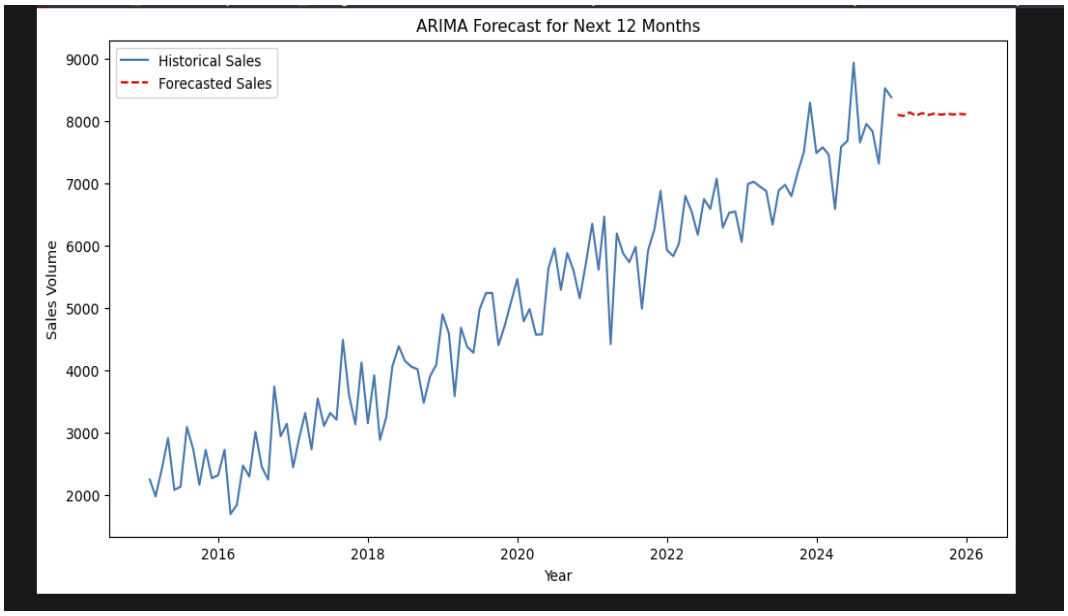
- Time series forecasting uses past observations to forecast future values.
- **Most widely used forecasting models**:
  - **ARIMA (AutoRegressive Integrated Moving Average)**: ARIMA is the most common forecasting model for stationary time series data. ARIMA is an intermix of autoregression (AR), differencing (I), and moving averages (MA).

    Example: Predicting next month's sales based on past sales data, using autoregression to correlate past values, differencing to make the data stationary, and moving averages to smooth out random fluctuations.

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from statsmodels.tsa.arima.model import ARIMA
from pandas.plotting import autocorrelation_plot

# Generate synthetic sales data
np.random.seed(42)
date_range = pd.date_range(start="2015-01-01", periods=120, freq='M')  # 10 years of monthly data
trend = np.linspace(2000, 8000, len(date_range))  # Upward trend
random_noise = np.random.normal(0, 500, len(date_range))  # Random noise
sales = trend + random_noise

# Create DataFrame
df = pd.DataFrame({'date': date_range, 'sales': sales})
df.set_index('date', inplace=True)

# Plot original data
plt.figure(figsize=(12, 6))
sns.lineplot(data=df, x=df.index, y='sales', label="Monthly Sales")
plt.xlabel("Year")
plt.ylabel("Sales Volume")
plt.title("Monthly Sales Data with Trend")
plt.legend()
plt.show()

# Check autocorrelation
autocorrelation_plot(df['sales'])
plt.title("Autocorrelation Plot")
plt.show()

# Fit ARIMA model (p=2, d=1, q=2)
model = ARIMA(df['sales'], order=(2, 1, 2))
model_fit = model.fit()

# Forecast next 12 months
forecast = model_fit.forecast(steps=12)

# Create future date range
future_dates = pd.date_range(start=df.index[-1] + pd.DateOffset(months=1), periods=12, freq='M')
forecast_df = pd.DataFrame({'date': future_dates, 'sales_forecast': forecast})
forecast_df.set_index('date', inplace=True)

# Plot forecasted values
plt.figure(figsize=(12, 6))
sns.lineplot(data=df, x=df.index, y='sales', label="Historical Sales")
sns.lineplot(data=forecast_df, x=forecast_df.index, y='sales_forecast', color='red', linestyle=
            'dashed', label="Forecasted Sales")
plt.xlabel("Year")
plt.ylabel("Sales Volume")
plt.title("ARIMA Forecast for Next 12 Months")
plt.legend()
plt.show()
```

Output:


Monthly Sales Data with Trend
Monthly Sales
Sales Volume
Year
Autocorrelation Plot
Autocorrelation
Lag

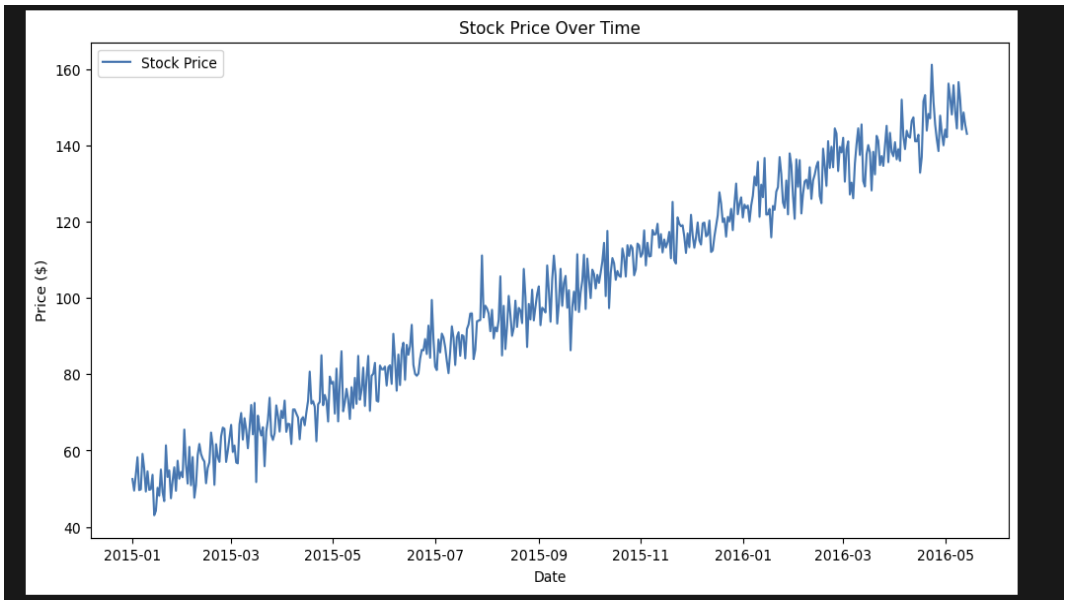

ARIMA Forecast for Next 12 Months
Historical Sales
Forecasted Sales
Sales Volume
Year

– **AR (AutoRegressive)**: Used the connection between an observation and a set of lagged observations.

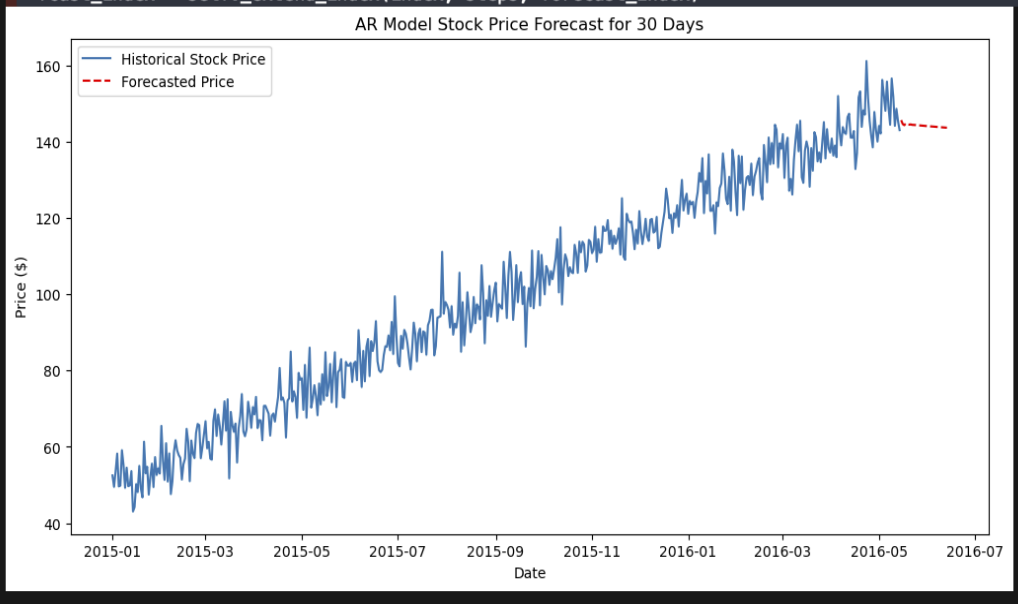
Example: Forecasting future stock prices based on a set of previous stock prices (lags), such as using the last 3 days of prices to predict the next day.

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from statsmodels.tsa.ar_model import AutoReg

# Generate synthetic stock price data
np.random.seed(42)
date_range = pd.date_range(start="2015-01-01", periods=500, freq='D')  # 500 days of data
trend = np.linspace(50, 150, len(date_range))  # Simulating an upward trend
random_noise = np.random.normal(0, 5, len(date_range))  # Random fluctuations
stock_prices = trend + random_noise

# Create DataFrame
df = pd.DataFrame({'date': date_range, 'stock_price': stock_prices})
df.set_index('date', inplace=True)

# Plot original stock prices
plt.figure(figsize=(12, 6))
sns.lineplot(data=df, x=df.index, y='stock_price', label="Stock Price")
plt.xlabel("Date")
plt.ylabel("Price ($)")
plt.title("Stock Price Over Time")
plt.legend()
plt.show()

# Fit an AutoRegressive model (using 3 lags)
lag_value = 3
model = AutoReg(df['stock_price'], lags=lag_value)
model_fit = model.fit()

# Forecast next 30 days
forecast = model_fit.predict(start=len(df), end=len(df) + 29)

# Create future date range
future_dates = pd.date_range(start=df.index[-1] + pd.DateOffset(days=1), periods=30, freq='D')
forecast_df = pd.DataFrame({'date': future_dates, 'forecasted_price': forecast})
forecast_df.set_index('date', inplace=True)

# Plot forecasted values
plt.figure(figsize=(12, 6))
sns.lineplot(data=df, x=df.index, y='stock_price', label="Historical Stock Price")
sns.lineplot(data=forecast_df, x=forecast_df.index, y='forecasted_price', color='red',
             linestyle='dashed', label="Forecasted Price")
plt.xlabel("Date")
plt.ylabel("Price ($)")
plt.title("AR Model Stock Price Forecast for 30 Days")
plt.legend()
plt.show()
```

Output:




```
/opt/anaconda3/lib/python3.12/site-packages/statsmodels/tsa/base/tsa_model.py:473: ValueWarning:
requency information was provided, so inferred frequency D will be used.
  self._init_dates(dates, freq)
/opt/anaconda3/lib/python3.12/site-packages/statsmodels/tsa/deterministic.py:308: UserWarning: Only PeriodIndexes, D
meIndexes with a frequency set, RangesIndexes, and Index with a unit increment support extending
index is set will contain the position relative to the data length.
  fcast_index = self._extend_index(index, steps, forecast_index)
```

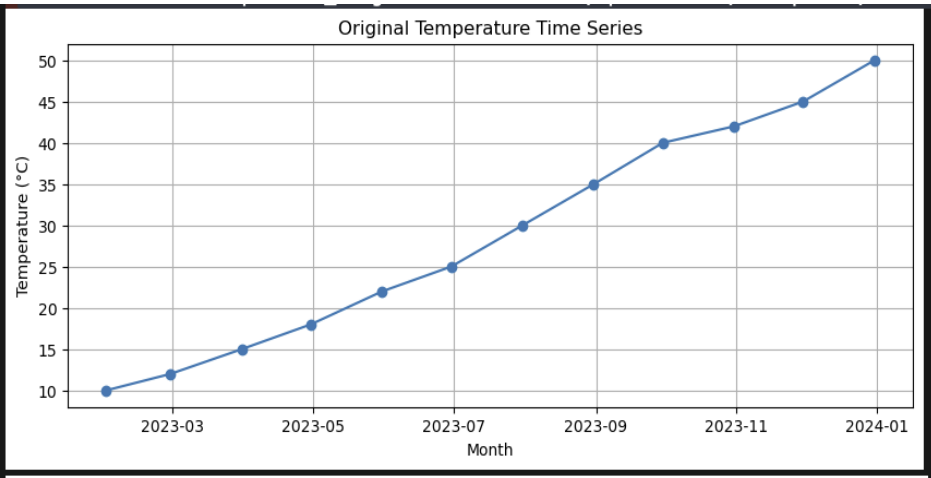


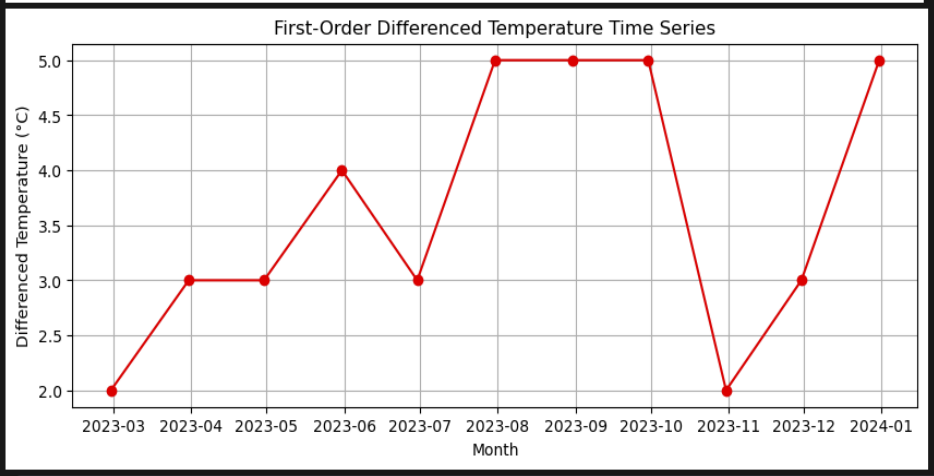
– **I (Integrated)**: It makes the time series stationary by calculating the difference between current and lagged observation.

Example: Making a time series stationary by calculating the difference between the current month's temperature and the previous month's temperature to remove trends.

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from statsmodels.tsa.stattools import adfuller

# Sample time series data (Monthly Temperature)
data = {'Month': pd.date_range(start='2023-01', periods=12, freq='M'),
        'Temperature': [10, 12, 15, 18, 22, 25, 30, 35, 40, 42, 45, 50]}
df = pd.DataFrame(data)
df.set_index('Month', inplace=True)

# Original time series plot
plt.figure(figsize=(10,4))
plt.plot(df['Temperature'], marker='o', linestyle='-')
plt.title("Original Temperature Time Series")
plt.xlabel("Month")
plt.ylabel("Temperature (°C)")
plt.grid()
plt.show()

# First-order differencing
df['Temperature_Diff'] = df['Temperature'].diff()

# Differenced time series plot
plt.figure(figsize=(10,4))
plt.plot(df['Temperature_Diff'], marker='o', linestyle='-', color='red')
plt.title("First-Order Differenced Temperature Time Series")
plt.xlabel("Month")
plt.ylabel("Differenced Temperature (°C)")
plt.grid()
plt.show()

# ADF test for stationarity
def adf_test(series):
    result = adfuller(series.dropna())  # Drop NA values for differencing
    print("ADF Statistic:", result[0])
    print("p-value:", result[1])
    print("Critical Values:")
    for key, value in result[4].items():
        print(f"   {key}: {value}")
    if result[1] <= 0.05:
        print("The series is stationary.")
    else:
        print("The series is NOT stationary.")

print("Original Time Series:")
adf_test(df['Temperature'])
print("\nFirst-Order Differenced Time Series:")
adf_test(df['Temperature_Diff'])
```

Output:





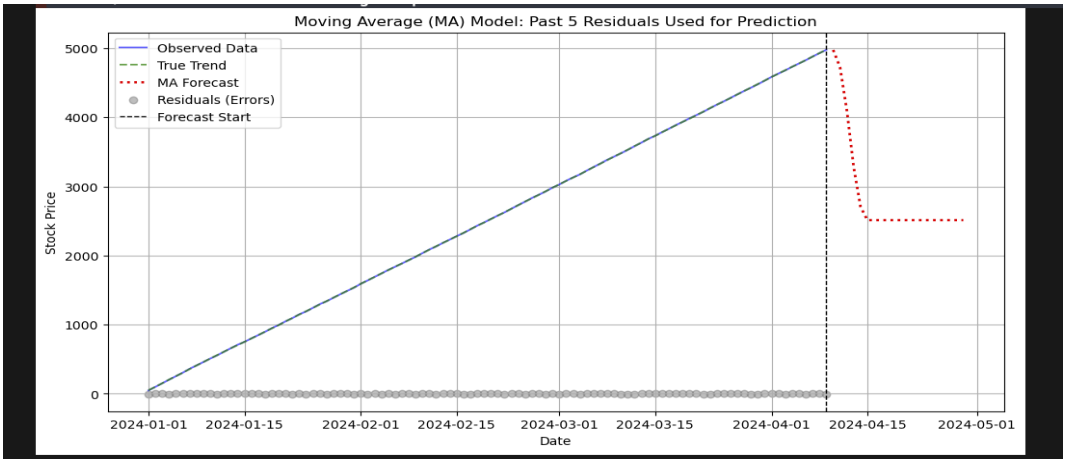

```
Original Time Series:
ADF Statistic: -0.6321807896779161
p-value: 0.8635434474346173
Critical Values:
   1%: -4.9386902332361515
   5%: -3.477582857142857
   10%: -2.8438679591836733
The series is NOT stationary.

First-Order Differenced Time Series:
ADF Statistic: -1.459069085638717
p-value: 0.5536534582632524
Critical Values:
   1%: -4.9386902332361515
   5%: -3.477582857142857
   10%: -2.8438679591836733
```

– **MA (Moving Average)**: It indicates the way an observation is connected to a residual error in a moving average model.

Example: Using the past 5 days of errors (the difference between predicted and actual values) to predict future stock prices.

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import statsmodels.api as sm

# Generate synthetic time series data (stock prices with noise)
np.random.seed(42)
date_rng = pd.date_range(start="2024-01-01", periods=100, freq="D")
actual_values = np.cumsum(np.random.randn(100) * 2 + 50)  # Simulated stock prices
noise = np.random.randn(100) * 2  # Random noise (residuals)
observed_values = actual_values + noise  # Adding noise to the true values

# Create DataFrame
df = pd.DataFrame({"actual": actual_values, "observed": observed_values}, index=date_rng)

# Compute residuals (difference between observed and actual)
df["residuals"] = df["observed"] - df["actual"]

# Fit a Moving Average (MA) model of order 5 (using last 5 residuals)
ma_model = sm.tsa.ARIMA(df["observed"], order=(0, 0, 5))
result = ma_model.fit()

# Forecast next 20 days
future_dates = pd.date_range(start=df.index[-1] + pd.DateOffset(1), periods=20, freq="D")
forecast_values = result.forecast(steps=20)

# Create forecast DataFrame
forecast_df = pd.DataFrame({"forecast": forecast_values}, index=future_dates)

# **Plot the actual vs. observed data + MA Forecast**
plt.figure(figsize=(12, 6))
plt.plot(df.index, df["observed"], label="Observed Data", color="blue", alpha=0.6)
plt.plot(df.index, df["actual"], label="True Trend", color="green", linestyle="dashed", alpha=0.7)
plt.plot(forecast_df.index, forecast_df["forecast"], label="MA Forecast", color="red",
         linestyle="dotted", linewidth=2)

# Mark Residuals
plt.scatter(df.index, df["residuals"], color="gray", alpha=0.5, label="Residuals (Errors)")

plt.axvline(df.index[-1], color="black", linestyle="dashed", linewidth=1, label="Forecast Start")
plt.legend()
plt.xlabel("Date")
plt.ylabel("Stock Price")
plt.title("Moving Average (MA) Model: Past 5 Residuals Used for Prediction")
plt.grid()
plt.show()
```

Output:



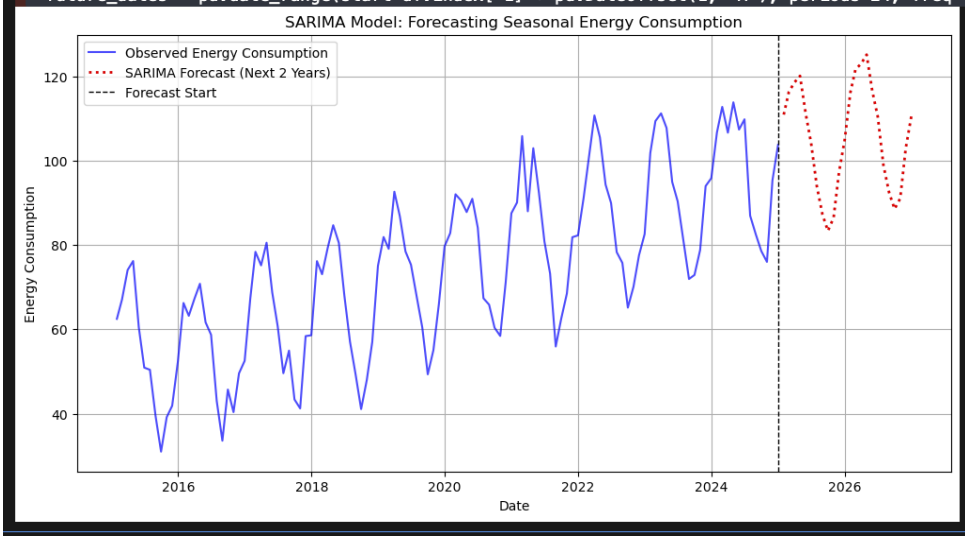
– **SARIMA (Seasonal ARIMA)**: It's a form of ARIMA that takes seasonality into consideration. It's used when your data is seasonal.

Example: Predicting future energy consumption by considering yearly seasonal patterns (e.g., higher usage in winter) and using SARIMA for seasonal adjustments.

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import statsmodels.api as sm

# Generate synthetic seasonal data (e.g., monthly energy consumption with seasonality)
np.random.seed(42)
date_rng = pd.date_range(start="2015-01", periods=120, freq="M")  # 10 years of monthly data
seasonal_pattern = 20 * np.sin(2 * np.pi * date_rng.month / 12)  # Seasonal wave pattern
trend = np.linspace(50, 100, 120)  # Upward trend in consumption
noise = np.random.randn(120) * 5  # Random noise
energy_consumption = trend + seasonal_pattern + noise  # Combine trend, seasonality & noise

# Create DataFrame
df = pd.DataFrame({"energy_consumption": energy_consumption}, index=date_rng)

# Fit SARIMA model (Seasonal ARIMA (1,1,1)(1,1,1,12))
sarima_model = sm.tsa.statespace.SARIMAX(df["energy_consumption"],
                                         order=(1,1,1),
                                         seasonal_order=(1,1,1,12))#Seasonal pattern every 12 months
result = sarima_model.fit()

# Forecast next 24 months
future_dates = pd.date_range(start=df.index[-1] + pd.DateOffset(1, "M"), periods=24, freq="M")
forecast_values = result.forecast(steps=24)

# Create forecast DataFrame
forecast_df = pd.DataFrame({"forecast": forecast_values}, index=future_dates)

# **Plot Actual vs. Forecasted Data**
plt.figure(figsize=(12, 6))
plt.plot(df.index, df["energy_consumption"], label="Observed Energy Consumption", color="blue",
         alpha=0.7)
plt.plot(forecast_df.index, forecast_df["forecast"], label="SARIMA Forecast (Next 2 Years)",
         color="red", linestyle="dotted", linewidth=2)

plt.axvline(df.index[-1], color="black", linestyle="dashed", linewidth=1, label="Forecast Start")
plt.legend()
plt.xlabel("Date")
plt.ylabel("Energy Consumption")
plt.title("SARIMA Model: Forecasting Seasonal Energy Consumption")
plt.grid()
plt.show()
```

Output:

```
Machine precision = 2.220D-16
 N =            5     M =           10

At X0         0 variables are exactly at the bounds

At iterate    0    f=  2.90905D+00    |proj g|=  9.38561D-02

At iterate    5    f=  2.75329D+00    |proj g|=  1.34261D-02

At iterate   10    f=  2.75028D+00    |proj g|=  1.59217D-02

At iterate   15    f=  2.74909D+00    |proj g|=  1.93045D-03

At iterate   20    f=  2.74893D+00    |proj g|=  4.94839D-04

At iterate   25    f=  2.74891D+00    |proj g|=  2.81673D-04

At iterate   30    f=  2.74891D+00    |proj g|=  3.87802D-05

           * * *

Tit   = total number of iterations
Tnf   = total number of function evaluations
Tnint = total number of segments explored during Cauchy searches
Skip  = number of BFGS updates skipped
Nact  = number of active bounds at final generalized Cauchy point
Projg = norm of the final projected gradient
F     = final function value

           * * *

   N    Tit     Tnf  Tnint  Skip  Nact     Projg        F
    5     34     41      1     0     0   1.003D-04   2.749D+00
  F =   2.7489081651755853

CONVERGENCE: REL_REDUCTION_OF_F_<=_FACTR*EPSMCH
/var/folders/cc/vyd9klj5439d5dylj0kmc2mh0000gn/T/ipykernel_78583/1040387941.py:24: FutureWarning: 'M'
is deprecated and will be removed in a future version, please use 'ME' instead.
  future_dates = pd.date_range(start=df.index[-1] + pd.DateOffset(1, "M"), periods=24, freq="M")
```



– **LSTM (Long Short-Term Memory)**: Deep learning model that excels at predicting whether there are long-term dependencies in time series data. LSTMs are a type of Recurrent Neural Networks (RNN) and excel at predicting future patterns from a sequence of data.

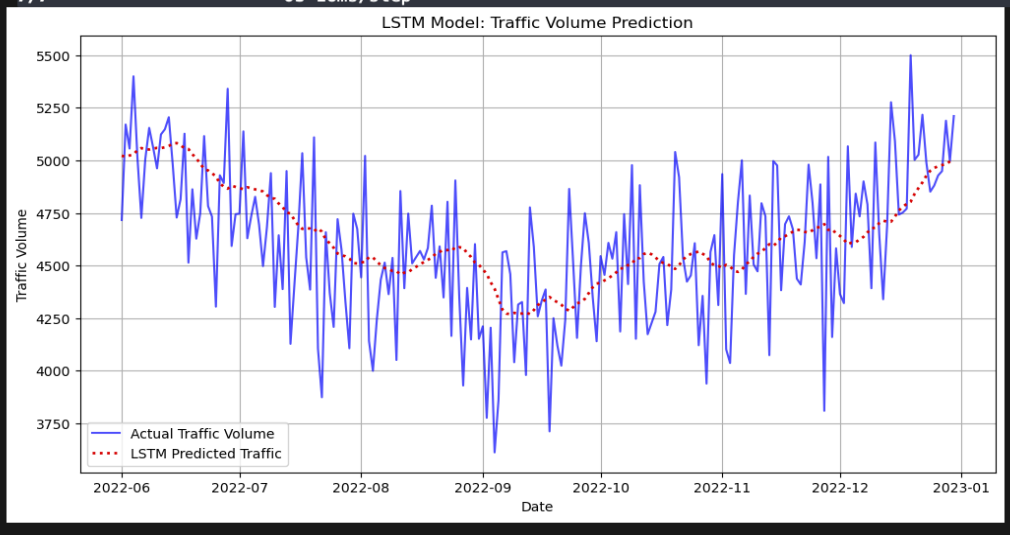
Example: Using LSTM models to forecast traffic patterns by learning long-term dependencies in historical data, such as predicting next week's traffic based on patterns observed over several months.

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split

# Simulate synthetic traffic data (e.g., daily traffic volume for 3 years)
np.random.seed(42)
date_rng = pd.date_range(start="2020-01-01", periods=1095, freq="D")  # 3 years of daily data
seasonal_pattern = 2000 + 500 * np.sin(2 * np.pi * date_rng.dayofyear / 365)  # Yearly seasonality
trend = np.linspace(1000, 3000, 1095)  # Increasing trend
noise = np.random.randn(1095) * 300  # Random fluctuations
traffic_volume = trend + seasonal_pattern + noise  # Combine trend, seasonality & noise

# Create DataFrame
df = pd.DataFrame({"traffic_volume": traffic_volume}, index=date_rng)

# Scale the data using MinMaxScaler (LSTMs work better with normalized input)
scaler = MinMaxScaler(feature_range=(0, 1))
df_scaled = scaler.fit_transform(df)

# Prepare sequences for LSTM (time steps = 30 days)
time_steps = 30
X, y = [], []
for i in range(len(df_scaled) - time_steps):
    X.append(df_scaled[i:i+time_steps])  # Past 30 days
    y.append(df_scaled[i+time_steps])  # Next day prediction

X, y = np.array(X), np.array(y)

# Split data into training and testing sets (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, shuffle=False)

# Build LSTM Model
model = Sequential([
    LSTM(50, return_sequences=True, input_shape=(time_steps, 1)),
    LSTM(50, return_sequences=False),
    Dense(25, activation='relu'),
    Dense(1)
])

model.compile(optimizer="adam", loss="mse")

# Train LSTM Model
model.fit(X_train, y_train, epochs=20, batch_size=32, validation_data=(X_test, y_test))

# Make predictions
predicted_traffic = model.predict(X_test)
predicted_traffic = scaler.inverse_transform(predicted_traffic)  # Convert back to original scale
actual_traffic = scaler.inverse_transform(y_test)  # Convert test labels back
```

```
# **Plot Actual vs. Predicted Traffic Volume**
plt.figure(figsize=(12, 6))
plt.plot(df.index[-len(actual_traffic):], actual_traffic, label="Actual Traffic Volume",
         color="blue", alpha=0.7)
plt.plot(df.index[-len(predicted_traffic):], predicted_traffic, label="LSTM Predicted Traffic"
         , color="red", linestyle="dotted", linewidth=2)

plt.legend()
plt.xlabel("Date")
plt.ylabel("Traffic Volume")
plt.title("LSTM Model: Traffic Volume Prediction")
plt.grid()
plt.show()
```

Output:

```
27/27 ━━━━━━━━━━━━━━━━━━━━ 1s 12ms/step - loss: 0.0767 - val_loss: 0.0067
Epoch 2/20
27/27 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - loss: 0.0100 - val_loss: 0.0063
Epoch 3/20
27/27 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 0.0083 - val_loss: 0.0063
Epoch 4/20
27/27 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - loss: 0.0070 - val_loss: 0.0083
Epoch 5/20
27/27 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - loss: 0.0076 - val_loss: 0.0071
Epoch 6/20
27/27 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - loss: 0.0076 - val_loss: 0.0066
Epoch 7/20
27/27 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - loss: 0.0073 - val_loss: 0.0063
Epoch 8/20
27/27 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - loss: 0.0071 - val_loss: 0.0065
Epoch 9/20
27/27 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - loss: 0.0074 - val_loss: 0.0076
Epoch 10/20
27/27 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - loss: 0.0073 - val_loss: 0.0063
Epoch 11/20
27/27 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - loss: 0.0082 - val_loss: 0.0073
Epoch 12/20
27/27 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - loss: 0.0071 - val_loss: 0.0063
Epoch 13/20
27/27 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - loss: 0.0077 - val_loss: 0.0063
Epoch 14/20
27/27 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - loss: 0.0071 - val_loss: 0.0064
Epoch 15/20
27/27 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - loss: 0.0067 - val_loss: 0.0063
Epoch 16/20
27/27 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - loss: 0.0069 - val_loss: 0.0063
Epoch 17/20
27/27 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - loss: 0.0076 - val_loss: 0.0075
Epoch 18/20
27/27 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - loss: 0.0079 - val_loss: 0.0072
Epoch 19/20
27/27 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - loss: 0.0073 - val_loss: 0.0063
Epoch 20/20
27/27 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - loss: 0.0069 - val_loss: 0.0064
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step
```



9 Big Data & Distributed Computing

In the era of big data, efficiently processing and managing large amounts of data is an important challenge. Distributed computing libraries and frameworks serve to address this challenge by allowing parallel processing and scaling to deal with large datasets. This section presents some influential tools and techniques for working with big data using Python.

9.1 Using Dask and Vaex for Large Datasets

- If working with large data that cannot be stored in memory, one can utilize libraries like Vaex and Dask.
- These support parallel computation in large-scale data processing.
- **Dask**:
  - An adaptive parallelism library that is compatible with current libraries such as Pandas and NumPy.
  - Dask divides large tasks into small ones and spreads them across many machines or cores, so it's perfect for working with large data.
  - **Dask DataFrame**: Similar to Pandas DataFrame but can process data sets that are too big to hold in memory by dividing them into smaller, more manageable chunks.
  - **Dask Arrays**: Like NumPy arrays but distributed across machines or multiple cores.
  - **Example**: Using Dask to process a large dataset of customer transactions that won't fit into memory by breaking the data into smaller chunks and processing them across multiple cores or machines, just like working with a Pandas DataFrame.

```python
import dask.array as da
import numpy as np

# Create a large Dask array (equivalent to a NumPy array but lazy-loaded)
large_array = da.random.random((10000, 10000), chunks=(1000, 1000))  # Divide into 1,000x1,000 chunks

# Compute mean (distributed processing)
mean_value = large_array.mean().compute()
print("Mean Value:", mean_value)

# Compute sum along axis 0
sum_values = large_array.sum(axis=0).compute()
print(sum_values)
```

Output:

```
Mean Value: 0.5000006719502269
[4999.7983915  4953.37795674 4973.61840157 ... 5014.32624496 4983.600012
 5026.63580185]
```

- **Vaex**:
  - A memory-frugal and rapid library for the manipulation of massive tabular datasets (billions of rows).
  - It relies on lazy evaluation, i.e., it does not load data into memory until that's needed, enabling you to process larger-than-memory datasets seamlessly.
  - **Example**: Analyzing a billion-row dataset of user activity logs without loading all data into memory at once, using Vaex's lazy evaluation to only load the data needed for computation.

```python
import vaex

# Create a small Vaex DataFrame (efficient, does not load everything into memory)
df = vaex.from_arrays(
    user_id=[1, 2, 3, 4, 5],
    event_type=["click", "purchase", "click", "view", "purchase"],
    amount=[10.5, 25.0, 7.8, 0, 30.0]
)

# Perform an efficient groupby operation
grouped = df.groupby("event_type", agg={"total_amount": vaex.agg.sum("amount")})

# Show results
print(grouped)
```

Output:

```
#    event_type    total_amount
0      click            18.3
1    purchase           55.0
2      view             0.0
```

9.2 Introduction to Apache Spark (PySpark)

- Apache Spark is an open-source distributed computing system for quick big data processing.
- It offers APIs for data processing on various nodes of a cluster.
- **PySpark**: Python API for Spark, enabling Python developers to take advantage of the power of Spark for processing large data.

  Example: Using PySpark to process and analyze a massive dataset of user clicks across multiple nodes of a cluster, speeding up computation compared to traditional methods.

```python
from pyspark.sql import SparkSession
from pyspark.sql.functions import col

# Initialize a Spark session
spark = SparkSession.builder.appName("SparkSQLExample").getOrCreate()

# Create a sample DataFrame
data = [
    (1, "Laptop", 1200),
    (2, "Phone", 800),
    (3, "Tablet", 400),
    (4, "Monitor", 150),
    (5, "Keyboard", 50)
]

columns = ["id", "product", "amount"]

df = spark.createDataFrame(data, columns)

# Register DataFrame as a SQL table
df.createOrReplaceTempView("sales")

# Run a SQL query
result = spark.sql("SELECT * FROM sales WHERE amount > 500")

# Show the result
result.show()

# Stop Spark session
spark.stop()
```

Output:

```
+---+---------+------+
| id|product  |amount|
+---+---------+------+
|  1| Laptop  |  1200|
|  2|  Phone  |   800|
+---+---------+------+
```

- **Spark SQL**: Spark module that enables querying of data with SQL-like syntax. Useful for people who are familiar with SQL but wish to work with big data.

  Example: Querying a large dataset of sales transactions with SQL-like syntax (SELECT * FROM sales WHERE amount > 100), but leveraging Spark's distributed power for faster execution on big data.

```python
from pyspark.sql import SparkSession

# Initialize a Spark session
spark = SparkSession.builder.appName("SparkSQLExample").getOrCreate()

# Sample sales data (id, product, amount)
data = [
    (1, "Laptop", 1200),
    (2, "Phone", 800),
    (3, "Tablet", 400),
    (4, "Monitor", 150),
    (5, "Keyboard", 50),
    (6, "Mouse", 40),
    (7, "Printer", 200),
]

columns = ["id", "product", "amount"]

# Create DataFrame
df = spark.createDataFrame(data, columns)

# Register DataFrame as a temporary SQL table
df.createOrReplaceTempView("sales")

# Query sales where amount > 100 using SQL
result = spark.sql("SELECT * FROM sales WHERE amount > 100")

# Show results
result.show()

# Stop Spark session
spark.stop()
```

Output:

```
+---+--------+------+
| id|product |amount|
+---+--------+------+
|  1| Laptop |  1200|
|  2|  Phone |   800|
|  3| Tablet |   400|
|  4|Monitor |   150|
|  7|Printer |   200|
+---+--------+------+
```

- **Resilient Distributed Datasets (RDDs)**: The underlying data structure for Spark, where data can be distributed and processed in parallel.

  Example: Storing and processing distributed data across a cluster, like splitting a large log file into chunks and processing them in parallel to find error patterns.

```python
from pyspark.sql import SparkSession

# Initialize Spark session
spark = SparkSession.builder.appName("RDDExample").getOrCreate()

# Create an RDD from a sample log file (simulating a distributed dataset)
log_data = [
    "INFO: System is running",
    "ERROR: Failed to load config",
    "WARNING: Low disk space",
    "ERROR: Unable to connect to database",
    "INFO: User logged in",
]

# Parallelize the data into an RDD
rdd = spark.sparkContext.parallelize(log_data)

# Filter out only the lines containing "ERROR"
error_logs = rdd.filter(lambda line: "ERROR" in line)

# Show results
print("Error Logs:")
for log in error_logs.collect():
    print(log)

# Stop Spark session
spark.stop()
```

  Output:

```
Error Logs:
ERROR: Failed to load config
ERROR: Unable to connect to database
```

- Spark finds extensive usage for big data operations such as ETL (Extract, Transform, Load), machine learning, and real-time processing.

9.3 Parallel Processing and Scalability

- Parallel processing refers to the method of breaking down work into small sub-tasks that are executed in parallel between multiple systems or processors.
- Parallel processing plays a crucial role in big data since it allows for faster computation through efficient utilization of available resources.
- **Multithreading**: Running multiple threads in a single process to perform tasks concurrently.

  Example: Running multiple threads within a single Python process to handle different parts of a data cleaning pipeline concurrently, improving efficiency on tasks like file reading or string processing.

```python
import threading
import time

def read_file(file_name):
    print(f"Reading {file_name}...")
    time.sleep(2)  # Simulate file reading delay
    print(f"Finished reading {file_name}")

# Creating threads
threads = []
file_names = ["file1.txt", "file2.txt", "file3.txt"]
for file in file_names:
    t = threading.Thread(target=read_file, args=(file,))
    threads.append(t)
    t.start()

# Wait for all threads to complete
for t in threads:
    t.join()

print("All files read successfully.")
```

Output:

```
Reading file1.txt...
Reading file2.txt...
Reading file3.txt...
Finished reading file2.txtFinished reading file1.txt
Finished reading file3.txt

All files read successfully.
```

- **Multiprocessing**: Spawning multiple processes to run on separate CPUs or cores, which is ideal for CPU-bound tasks.

  Example: Using Python's multiprocessing library to process large CSV files in parallel, dividing the workload across multiple CPU cores to speed up data analysis.

```python
import multiprocessing
import math

def compute_square(num):
    return num ** 2

if __name__ == "__main__":
    numbers = list(range(1, 11))  # List of numbers
    with multiprocessing.Pool(processes=4) as pool:
        results = pool.map(compute_square, numbers)
    print("Squared Numbers:", results)
```

Output:

```
Squared Numbers: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

- **Distributed Computing**: Execution of tasks on several machines (distributed system) through tools such as Dask, Spark, or Hadoop.

  Example: Leveraging Apache Spark to split a large dataset of customer transactions across several machines and process them in parallel, significantly reducing processing time compared to a single machine.

```python
from pyspark.sql import SparkSession

# Initialize Spark session
spark = SparkSession.builder.appName("DistributedExample").getOrCreate()

# Sample dataset
data = [("Alice", 2000), ("Bob", 2500), ("Charlie", 1800)]
columns = ["Name", "Salary"]

# Create Spark DataFrame
df = spark.createDataFrame(data, columns)

# Perform distributed computation
df_filtered = df.filter(df["Salary"] > 2000)
df_filtered.show()
```

Output:

```
+-----+------+
|Name|Salary|
+-----+------+
|  Bob|  2500|
+-----+------+
```

- **Scalability**:
  - Means the capacity of a system to cope with increasing data effectively.
  - Big data software and platforms such as Dask and Spark are built to be highly scalable, as they can process much bigger datasets than the memory or storage capacity of an individual machine.
  - **Example**: Using Spark or Dask to process a growing dataset, ensuring the system can handle increased data volume by distributing the workload across more resources (machines, CPUs).

```python
import dask.dataframe as dd

# Read large CSV file in parallel
df = dd.read_csv("large_dataset.csv")

# Compute mean of a column
mean_value = df["price"].mean().compute()
print("Mean Price:", mean_value)
```

Output:

```
Mean Price: 200.0
```

10 Data Science Best Practices & Deployment

In data science, it's essential not only to develop successful models but also ones that are understandable, deployable, and scalable. This topic addresses best practice for model explainability, deployment, version management, and ethical use when employing AI and machine learning models.

10.1 Model Interpretability

- Model interpretability involves the capability to comprehend and describe how a model comes to a decision.
- As machine learning models, particularly sophisticated ones such as deep learning networks, grow stronger, they can become "black boxes," such that it is challenging to describe their predictions.
- **There are a number of techniques and methods of model interpretability improvement**:
  - **Feature Importance**: Identifying which features (variables) are most important for the model's predictions.

    Example: Using a Random Forest model to identify which features, such as age or income, most influence the prediction of whether a customer will buy a product.

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

# Generate a sample dataset
X, y = make_classification(n_samples=1000, n_features=5, random_state=42)
feature_names = ['Age', 'Income', 'Education', 'Spending Score', 'Region']
df = pd.DataFrame(X, columns=feature_names)

# Train a Random Forest model
model = RandomForestClassifier()
model.fit(df, y)

# Get feature importances
importance = model.feature_importances_
feature_importance = pd.DataFrame({'Feature': feature_names, 'Importance': importance})
feature_importance = feature_importance.sort_values(by="Importance", ascending=False)

print(feature_importance)
```

Output:

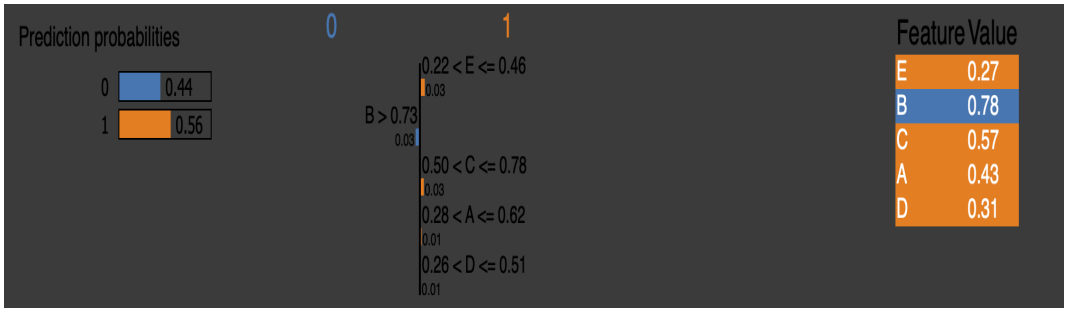
```
           Feature  Importance
0              Age    0.368117
3  Spending Score     0.263615
4           Region    0.174264
1           Income    0.150258
2        Education    0.043745
```

– **LIME (Local Interpretable Model-Agnostic Explanations)**: An approach to explaining a single prediction by locally fitting the model with a simpler, interpretable model.

Example: Using LIME to explain why a machine learning model predicted a specific loan rejection, by creating a simpler model that approximates the complex model's behavior near that specific data point.

```python
import lime
import lime.lime_tabular
import numpy as np
from sklearn.linear_model import LogisticRegression

# Sample dataset
X_train = np.random.rand(100, 5)
y_train = np.random.randint(0, 2, 100)
model = LogisticRegression()
model.fit(X_train, y_train)

# Initialize LIME explainer
explainer = lime.lime_tabular.LimeTabularExplainer(X_train, feature_names=['A', 'B', 'C', 'D', 'E']
                                                   , mode='classification')

# Explain a single prediction
exp = explainer.explain_instance(X_train[0], model.predict_proba)
exp.show_in_notebook()
```

Output:

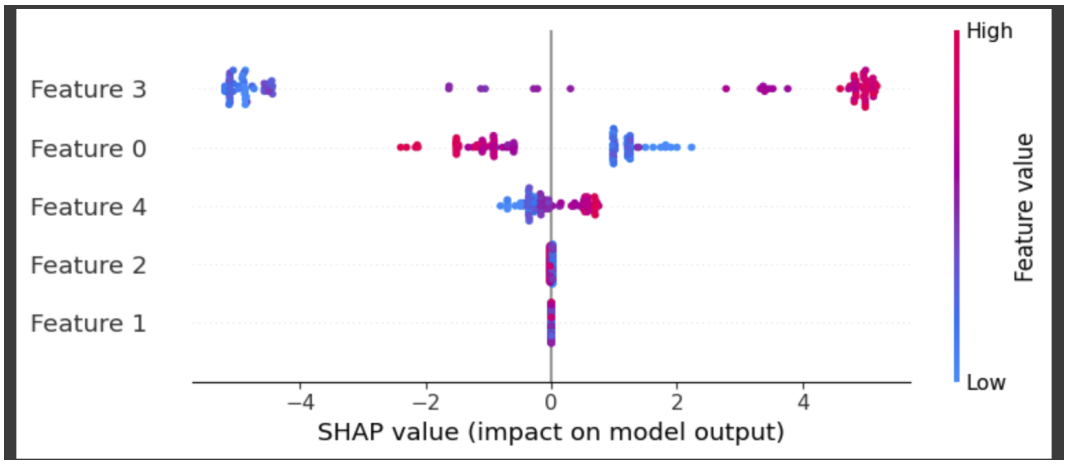


– **SHAP (SHapley Additive exPlanations)**: A game theory-inspired method to explain any machine learning model prediction by distributing the prediction value across features.

Example: Using SHAP to explain a model's prediction on a housing price, showing how each feature (location, size, age) contributed to the predicted price by distributing the prediction value across those features.

```python
import shap
import xgboost

# Train an XGBoost model
X_train, y_train = make_classification(n_samples=100, n_features=5, random_state=42)
model = xgboost.XGBClassifier()
model.fit(X_train, y_train)

# Explain model predictions using SHAP
explainer = shap.Explainer(model)
shap_values = explainer(X_train)

# Visualize feature importance
shap.summary_plot(shap_values, X_train)
```

Output:



10.2 Model Deployment

- Deploying a machine learning model is deploying it to production wherein the model is utilized by end-users or systems.
- Deployment then comes into play to make the model scalable and useable.
- **Model Serving**:
  - It is hosting the model in a format which is being served via an API so that other apps or people can even make predictions.
  - The model is distributed via APIs of frameworks like Flask or FastAPI.
  - **Example**: Hosting a fraud detection model via a FastAPI service so that other applications can make real-time predictions on transaction data by calling the API.

```python
from fastapi import FastAPI
import pickle
import numpy as np

# Load trained model
with open("model.pkl", "rb") as f:
    model = pickle.load(f)

app = FastAPI()

@app.get("/")
def home():
    return {"message": "Model is running"}

@app.post("/predict/")
def predict(data: list):
    prediction = model.predict(np.array(data).reshape(1, -1))
    return {"prediction": prediction.tolist()}
```

- **Model Monitoring**:
  - Once implemented, monitoring the performance of the model constantly in real time is necessary so that it would always be functioning at its peak.
  - Methods such as A/B testing and detection of drift are used to keep a lookout for problems such as model degradation with time.
  - **Example**: Using A/B testing to compare a newly deployed recommendation model with the previous one to see if it provides better results or detecting drift by monitoring performance over time to catch any degradation in accuracy.

```python
from evidently.dashboard import Dashboard
from evidently.tabs import DataDriftTab

dashboard = Dashboard(tabs=[DataDriftTab()])
dashboard.calculate(reference_data, current_data)
dashboard.show()
```

- **Containerization**:
  - Software such as Docker enables you to bundle your model and its dependencies into a deployable container across various environments.
  - **Example**: Using Docker to package a sentiment analysis model along with its required libraries and dependencies, ensuring that the model can be deployed consistently across different environments (development, staging, production).

```dockerfile
FROM python:3.9
WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY . .
CMD ["uvicorn", "app:app", "--host", "0.0.0.0", "--port", "8000"]
```

10.3 Version Control for ML Models

- Version control is an essential aspect of machine learning to maintain model, dataset, and code versions in sync over time.
- This facilitates reproducibility and model evolution to be handled more easily.
- **Git**:
  - Although Git is widely used for code versioning, it may also be utilized for model versioning.
  - For big files such as datasets and models, however, you might rather have tools like Git LFS (Large File Storage) or DVC (Data Version Control) at your disposal.
  - **Example**: Using Git to manage changes to your machine learning codebase and workflows while employing Git LFS for large files like pre-trained models or datasets.

    Setup:

```bash
git init
git lfs install
git lfs track "*.h5"  # Track large model files (e.g., Keras models)
git add .gitattributes
git add model.h5
git commit -m "Add trained model"
```

- **DVC**:
  - DVC is a version control system that is model and machine learning data-specific.
  - It is used to version and track models, datasets, and experiments without keeping large files in Git itself.

- **Example**: Using DVC to track different versions of your model and datasets during experimentation, so you can easily reproduce results and ensure that the model and its training data are properly versioned.

  Setup:

```
pip install dvc
dvc init
dvc add dataset.csv
git add dataset.csv.dvc .gitignore
git commit -m "Track dataset with DVC"
```

10.4 Ethical Considerations and Bias in AI

- As the usage of machine learning and artificial intelligence algorithms grew, it is crucial to manage the AI bias and ethical issues in such a manner that it becomes fair and not become a catastrophic entity.
- **AI Bias**:
  - AI bias in training data could create models that make biased or discriminatory choices.
  - One must strictly prepare and examine the data for the possibility of bias opportunities.
  - **Example**: If a facial recognition system is trained on predominantly white faces, it may perform poorly or inaccurately on individuals from other racial backgrounds, leading to biased results.
- **Fairness**:
  - AI models must be made equitable to everyone and not prejudicial to a section of the population.
  - Methods such as Fairness Constraints and Bias Mitigation can be utilized to minimize unfair outcomes.
  - **Example**: Ensuring a hiring algorithm does not discriminate against candidates based on gender or ethnicity by applying fairness constraints that remove biases from training data or model predictions.
- **Accountability and Transparency**:
  - AI systems must be transparent to ensure stakeholders are aware of how decisions are being made.
  - Models must be maintained as interpretable and explainable, which is a central aspect of this.
  - **Example**: Providing clear explanations to stakeholders on how a loan approval model makes decisions, ensuring that the process is interpretable and justifiable to avoid discrimination and maintain trust.
- **Privacy**:
  - Artificial intelligence systems must maintain the privacy and security of users, particularly in processing sensitive personal data.
  - **Example**: Implementing data anonymization techniques in healthcare AI systems to protect patient identities while ensuring that sensitive medical data used for training models remains secure and private.
- **Example Tool**:
  - **IBM AI Fairness 360 (AIF360)**: Detecting Bias in a Dataset.

– **Fairlearn**: Mitigating Bias in a Classifier.
– **SHAP**: Explaining Model Decisions.
– **Google What-If Tool**: Interactive Bias Analysis.

11 Conclusion

- **Data Science with Python**: Python is really the sweetheart language in science data because of how beautiful it is, the great culture of libraries, and because bringing intelligent support bases together is simply so strong.
- **Data Acquisition & Processing**: Interleaving data processing with file handling, web scraping, APIs, and data cleansing gives good quality data for analysis.
- **Exploratory Data Analysis (EDA)**: EDA, which stands for Exploratory Data Analysis, which is a whole bunch of cool stuff like cool stats summaries, cool data visualization, and cool methods of dimension reduction that are all great at showing insights and patterns.
- **Python Machine Learning**: Unsupervised and supervised learning activities like regression, classification, clustering, model selection, and hyperparameter tuning improve prediction power.
- **Deep Learning & Neural Networks**: Advanced deep learning systems that are really sophisticated like using CNNs for stuff like vision, RNNs for things with time—like stock quotes or speech records—plus transformers that excel at language processing deliver super high performance in high class models.
- **Natural Language Processing (NLP)**: Word embeddings, TF-IDF, processing of text, sentiment analysis and transformer models all work really well together for working with textual information. They all work very nicely and complement each other really well. Each one does different things but they are used together to get maximal effectiveness. It's like having a team where each member plays a different role but together they are great.
- **Time Series Analysis**: Seasonality detection, moving averages, and time-series forecasting models like ARIMA, SARIMA, and LSTMs can be applied on time-based data.
- **Big Data & Distributed Computing**: Dask, Vaex and Apache Spark (which uses Python Spark called pyspark) are super good at big data processing and doing computing work on super big sets of data at the same time.
- **Model Deployment & Versioning**: Model deployment with Flask, FastAPI, and cloud computing and versioning with Git, DVC, and MLflow provide reproducibility and reliability.
- **Ethical Issues & Model Explainability**: AI fairness management, bias detection, and interpretability through methods like SHAP and LIME provide ethical AI.
- **Final Thoughts**:Python remains king for data science because pragmatic pros want to analyze, process and implement data solutions that scale while playing nice with ethics so that they can make powerful AI adoption happen.