Pianalytix

https://pianalytix.com/

DATA SCIENCE MASTERY CHEAT SHEET

: https://instagram.com/pianalytix

: https://www.linkedin.com/company/pianalytix/

(140+ Code Snippets to 10x your Productivity)

1 Introduction

Components

1. Data Collection and Preparation:
   - Gathering relevant data from various sources.
   - Cleaning, preprocessing, and transforming the data for analysis.

2. Statistical Analysis and Modeling:
   - Applying statistical techniques to identify patterns and relationships in the data.
   - Creating mathematical models and algorithms for predictive and prescriptive analysis.

3. Machine Learning and Data Mining:
   - Utilizing machine learning algorithms to automatically extract insights from data.
   - Uncovering hidden patterns and making predictions or classifications.

4. Data Visualization and Communication:
   - Presenting data findings through visualizations, charts, and graphs.
   - Communicating complex concepts and insights to stakeholders effectively.

5. Business Impact and Decision-making:
   - Using data-driven insights to inform strategic decisions and solve problems.
   - Maximizing business value and driving innovation through data-driven approaches.

Libraries

1. NumPy:
   Numerical computation and array manipulation for scientific computing.

2. pandas:
   Data manipulation and analysis for scientific data processing.

3. scikit-learn:
   Machine learning algorithms and tools for scientific modeling.

4. Matplotlib:
   Data visualization library for scientific plotting and graphing.

5. TensorFlow and PyTorch:
   Deep learning frameworks for scientific research and applications.

6. SciPy:
   Scientific computing library providing a wide range of mathematical functions.

7. Seaborn:
   Statistical data visualization library for exploratory data analysis.

2 Data Collection

2.1 Definitions - Web Scraping

Web Scraping

Web Scraping refers to the extraction of data from a website. This information is collected and then exported into a format that is more useful for the user . Be it a spreadsheet or an API.

Web Scraping Techniques:

1. Fetching Webpage Content:
   - Use the requests library to send a GET request and retrieve the HTML content of the webpage.

2. HTML Parsing:
   - Utilize a parsing library like BeautifulSoup or lxml to parse the HTML structure and extract data.

3. Locating Relevant Data:
   - Inspect the HTML structure and identify appropriate HTML tags and attributes to locate the desired data.

4. Data Extraction:
   - Use methods such as find, find_all, or CSS selectors to extract the relevant data from the HTML.

5. Data Processing and Cleaning:
   - Process and clean the extracted data by removing unwanted characters, formatting, or converting data types.

6. Handling Pagination:
   - Implement techniques to handle webpages with multiple pages, such as iterating through page numbers or following next page links.

7. Handling Dynamic Content:
   - Address dynamic content loading by using headless browsers or JavaScript rendering libraries like Selenium or Scrapy-Splash.

8. Error Handling and Retrying:
   - Implement error handling mechanisms to handle exceptions, network errors, and connection timeouts. Include retry logic if necessary.

9. Respecting Website Policies:
   - Follow the website's terms of service, including rate limits, crawling restrictions, and respecting robots.txt guidelines.

10. Data Storage:
    - Store the scraped data in a structured format like CSV, JSON, or a database for further analysis integration with other systems.

2.1.1 Code Snippets - Sequence Padding

Sequence Padding

```
import requests
from bs4 import BeautifulSoup

# Send a GET request to the webpage
response = requests.get(url)

# Parse the HTML content using BeautifulSoup
soup = BeautifulSoup(response.content, 'html.parser')

# Extract relevant data from the webpage
data = soup.find_all('tag_name')
```

Optimization

```
import requests
from bs4 import BeautifulSoup
import time

# Define the URL to scrape
url = 'https://example.com'

# Set headers to mimic a web browser
headers = {
    'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome 91.0.4472.124 Safari/537.36'
}

# Send a GET request to the webpage
response = requests.get(url, headers=headers)

# Check if the request was successful
if response.status_code == 200:
    # Parse the HTML content using BeautifulSoup
    soup = BeautifulSoup(response.content, 'html.parser')

    # Extract relevant data using specific CSS selectors
    data = soup.select('.class_name')

    # Process and print the extracted data
    for item in data:
        print(item.text)
else:
    print('Error:', response.status_code)

# Add a delay between requests to avoid overwhelming the server
time.sleep(1)
```

Strategies

```
1 Pagination:
import requests

base_url = 'https://example.com/page='

for page_num in range(1, 6):
    url = base_url + str(page_num)
    response = requests.get(url)
    # Process the page data
2 Handling Dynamic Content:
from selenium import webdriver

driver = webdriver.Chrome('path_to_chromedriver')
driver.get('https://example.com')

# Wait for dynamic content to load
time.sleep(5)

# Extract data from the loaded page
# Close the driver
driver.quit()
3 Login and Session Handling:
import requests

login_url = 'https://example.com/login'
data = {'username': 'your_username', 'password': 'your_password'}

# Perform login
session = requests.Session()
session.post(login_url, data=data)

# Use the session for subsequent requests
response = session.get('https://example.com/dashboard')
# Process the response
4 Captcha Handling:
import requests

# Fetch the captcha image
response = requests.get('https://example.com/captcha')

# Save the captcha image
with open('captcha.png', 'wb') as f:
    f.write(response.content)

# Manually solve the captcha and get the solution

# Send a POST request with the captcha solution
data = {'captcha': 'your_solution'}
response = requests.post('https://example.com/submit', data=data)
# Process the response
```

2.2 Definitions – API Data Retrieval

API Data Retrieval

An API (i.e., Application Programming Interface) is a standardized and secure interface that allows applications to communicate and work with each other.

The process of API data retrieval involves several steps:

1. Identification and Authentication:
- Identify API and obtain access credentials (API key, tokens, etc.).

2. Making the Request:
- Formulate HTTP request to API endpoint with required parameters.

3. API Server Processing:
- API server processes the request and validates permissions.

4. Receiving the Response:
- API sends structured response (JSON, XML) back to the client.

5. Data Extraction:
- Extract and parse relevant information from the API response.

6. Error Handling:
- Gracefully handle API errors with informative messages and codes.

7. Data Processing:
- Manipulate and transform retrieved data for application requirements.

8. Utilization of Data:
- Use data in the application's logic, display, or other functionalities.

2.2.1 Code Snippets - API Data Retrieval

API Data Retrieval

```
import requests

# Make a GET request to the API endpoint
response = requests.get(url, params=params)

# Get the JSON data from the response
data = response.json()

# Extract relevant information from the JSON data
extracted_data = data['key']
```

Optimization

```
import requests

def retrieve_data_from_api(api_url, params=None, headers=None):
    try:
        # Use connection pooling to improve performance
        session = requests.Session()

        # Set custom headers if needed
        if headers:
            session.headers.update(headers)

        # Make the API request with appropriate parameters
        response = session.get(api_url, params=params)

        # Check for successful response (status code 200)
        if response.status_code == 200:
            # Process the response data
            data = response.json()
            # ... perform additional data processing as needed ...
            return data
        else:
            # Handle API errors with appropriate error handling
            response.raise_for_status()

    except requests.exceptions.RequestException as e:
        # Handle connection errors and exceptions gracefully
        print("Error occurred:", e)

    return None

# Example Usage:
api_url = "https://api.example.com/data"
parameters = {"param1": "value1", "param2": "value2"}
headers = {"Authorization": "Bearer YOUR_ACCESS_TOKEN"}

result = retrieve_data_from_api(api_url, params=parameters, headers=headers)
if result:
    # Process and utilize the retrieved data
    print(result)
```

Strategies

```
1 Caching using requests_cache:
import requests
import requests_cache

requests_cache.install_cache('api_cache', expire_after=3600)

data = requests.get("https://api.example.com/data").json()
print(data)
2 Asynchronous Requests using asyncio and aiohttp:
import aiohttp
import asyncio

async def get_data():
    async with aiohttp.ClientSession() as session:
        async with session.get("https://api.example.com/data") as response:
            return await response.json()

loop = asyncio.get_event_loop()
data = loop.run_until_complete(get_data())
print(data)
3 Rate Limiting and Retry using ratelimit library:
from ratelimit import limits, sleep_and_retry
import requests

@sleep_and_retry
@limits(calls=5, period=1)
def get_data():
    return requests.get('https://api.example.com/data').json()

data = get_data()
print(data)
4 Handling Errors and Retry with retrying:
from retrying import retry
import requests

@retry(wait_exponential_multiplier=1000, wait_exponential_max=10000, stop_max_attempt_number=3)
def get_data():
    return requests.get("https://api.example.com/data").json()

data = get_data()
print(data)
```

Strategies

```
4 Pagination Handling for Large Data:
import requests

data = []
url = 'https://api.example.com/data'
while url:
    response = requests.get(url).json()
    data.extend(response['results'])
    url = response['next']

print(data)
5 Utilizing Webhooks for Real-Time Updates:
from flask import Flask, request, jsonify

app = Flask(__name__)

@app.route('/webhook', methods=['POST'])
def webhook_listener():
    data = request.json
    # Process and store the received data in the application
    # ...

    return jsonify({'message': 'Webhook data received successfully'}), 200

if __name__ == '__main__':
    app.run(port=5000)
```

2.3 Definitions - Database Query

Database Query

A database query for data collection is a command or a set of commands that are used to retrieve specific information or data from a database. In other words, it is a request made to a database management system (DBMS) to fetch records or data that match certain criteria.

A database query allows you to interact with the database and obtain the exact data you need, such as:

Data Retrieval:
Fetching data from one or multiple database tables based on specified conditions.

Data Filtering:
Selecting specific rows or columns that meet certain criteria.

Data Aggregation:
Calculating summary information, like sums or averages, over groups of data.

Data Modification:
Inserting, updating, or deleting records within the database.

Data Joins:
Combining data from multiple tables based on shared columns.

2.3.1 Code Snippets - Database Query

Database Query

```
import sqlite3

# Connect to the database
conn = sqlite3.connect('database.db')
cursor = conn.cursor()

# Execute SQL query to fetch data
cursor.execute('SELECT * FROM table_name')

# Fetch all rows of data
data = cursor.fetchall()

# Close the database connection
conn.close()
```

Optimization

```
1 Use Indexing:
-- Create an index on the "age" column to speed up queries
CREATE INDEX idx_age ON Customers (age);

-- Example query using the index
SELECT * FROM Customers WHERE age > 30;
2 Limit the Number of Rows:
-- Fetch only the top 10 rows from the "Customers" table
SELECT * FROM Customers LIMIT 10;
3 Avoid SELECTing Unnecessary Columns:
-- Fetch only the "name" and "email" columns from the "Customers" table
SELECT name, email FROM Customers WHERE age > 30;
4 Use Proper Joins
-- Prefer INNER JOIN when filtering on both tables
SELECT Customers.name, Orders.order_id
FROM Customers
INNER JOIN Orders ON Customers.customer_id = Orders.customer_id
WHERE Orders.order_amount > 100;
5 Avoid Using SELECT *:
-- Avoid using SELECT *; specify the required columns explicitly
SELECT customer_id, name FROM Customers;
6 Avoid Subqueries and Use JOINs:
-- Avoid subquery for better performance
SELECT name
FROM Customers
WHERE customer_id IN (SELECT customer_id FROM Orders WHERE order_amount > 100);

-- Use JOIN for the same result
SELECT DISTINCT Customers.name
FROM Customers
JOIN Orders ON Customers.customer_id = Orders.customer_id
WHERE Orders.order_amount > 100;
7 Use Aggregate Functions Wisely:
-- Avoid unnecessary aggregation and filtering
SELECT MAX(salary) FROM Employees WHERE department = 'IT';

-- Prefer filtering before aggregation
SELECT MAX(salary) FROM Employees WHERE department = 'IT';
8 Analyze Query Execution Plan:
-- Analyze the execution plan to optimize the query
EXPLAIN SELECT * FROM Customers WHERE age > 30;
9 Optimize Date Comparisons:
-- Use proper date comparisons
SELECT * FROM Orders WHERE order_date BETWEEN '2023-01-01' AND '2023-06-30';
10 Partitioning:
-- Use table partitioning for large datasets
CREATE TABLE Orders (
    order_id INT PRIMARY KEY,
    order_date DATE,
    order_amount DECIMAL
) PARTITION BY RANGE (YEAR(order_date));
```

Strategies

```
-- Create an index on the "age" column in the "Customers" table
CREATE INDEX idx_age ON Customers (age);

-- Select only the necessary columns for the query
SELECT name, email FROM Customers WHERE age > 30;

-- Inner join for matching records between "Customers" and "Orders" tables
SELECT Customers.name, Orders.order_id
FROM Customers
INNER JOIN Orders ON Customers.customer_id = Orders.customer_id
WHERE Orders.order_amount > 100;

-- Use proper aggregation and filtering
SELECT MAX(salary) FROM Employees WHERE department = 'IT';

-- Analyze the execution plan to optimize the query
EXPLAIN SELECT * FROM Customers WHERE age > 30;

-- Use proper date comparisons
SELECT * FROM Orders WHERE order_date BETWEEN '2023-01-01' AND '2023-06-30';
```

2.4 Definitions – File Reading

File Reading

File reading accesses data from files. It's vital in data collection to ingest, preprocess, and integrate information from diverse sources.

Use of File Reading in Data Collection:

Data Ingestion:
File reading enables the transfer of data from files to applications or databases, making it accessible for further processing or analysis.

Data Interoperability:
Different file formats are used for data storage, and file reading provides a standardized way to access data regardless of the file type.

Data Preprocessing:
File reading allows you to parse and preprocess data before using it in applications, ensuring that it is in a usable format.

Data Integration:
In data collection, data often comes from various sources in different formats. File reading helps combine and integrate data into a uniform structure for analysis and reporting.

Automation:
File reading can be incorporated into automated data collection pipelines to streamline the process of extracting and processing data.

File Reading

```
import pandas as pd

# Read data from a CSV file
data = pd.read_csv('file.csv')

# Read data from an Excel file
data = pd.read_excel('file.xlsx')

# Read data from a JSON file
data = pd.read_json('file.json')
```

Optimization

1 Reading Large Files in Chunks:
```
def read_large_file(filename):
    with open(filename, 'r') as file:
        while True:
            chunk = file.read(1024)  # Read in 1KB chunks
            if not chunk:
                break
            # Process the chunk
            # ...
```
2 Using Buffered Reading:
```
def read_file_buffered(filename):
    with open(filename, 'r', buffering=8192) as file:  # Use a larger buffer size
        data = file.read()
        # Process the data
        # ...
```
3 Context Management with Iteration:
```
def read_file_line_by_line(filename):
    with open(filename, 'r') as file:
        for line in file:
            # Process each line
            # ...
```
4 Memory Mapping (for large files):
```
import mmap

def read_large_file_memory_map(filename):
    with open(filename, 'r') as file:
        with mmap.mmap(file.fileno(), length=0, access=mmap.ACCESS_READ) as mmap_obj:
            data = mmap_obj.read()
            # Process the data
            # ...
```
5 Parallel File Reading (using multiprocessing):
```
import multiprocessing

def read_file_parallel(filename):
    with open(filename, 'r') as file:
        num_processes = multiprocessing.cpu_count()
        chunk_size = 1024 * 1024  # 1MB chunk size
        with multiprocessing.Pool(processes=num_processes) as pool:
            results = pool.map(file.read, [chunk_size] * num_processes)
            # Process the results
            # ...
```

Strategies

1 Reading Entire File:
```
def read_entire_file(filename):
    with open(filename, 'r') as file:
        data = file.read()
    # Process the data
    # ...
```
2 Reading Line by Line:
```
def read_file_line_by_line(filename):
    with open(filename, 'r') as file:
        for line in file:
            # Process each line
            # ...
```
3 Reading Binary Files:
```
def read_binary_file(filename):
    with open(filename, 'rb') as file:
        data = file.read()
    # Process the binary data
    # ...
```

2.5 Definitions – Sensor/Device Data Collection

Sensor/Device Data

Sensor/Device Data Collection is the process of gathering information from sensors or devices for analysis, monitoring, or control purposes.
Sensor/Device Data Collection Process:

1. Deployment:
   - Install sensors/devices in the target environment or system.
2. Data Acquisition:
   - Sensors/devices measure physical parameters and generate electrical or digital signals.
3. Data Transmission:
   - Transmit data from sensors/devices to a central collection point.
4. Data Aggregation:
   - Gather data from multiple sensors/devices into a centralized repository.
5. Data Preprocessing:
   - Cleanse and filter raw sensor data to remove noise or outliers.
6. Data Storage:
   - Store collected and preprocessed data in databases or data warehouses.
7. Data Integration:
   - Combine sensor/device data with other relevant sources (e.g., weather data, GIS).
8. Real-time Processing (Optional):
   - Implement real-time data processing for immediate insights and control responses.
9. Data Analysis and Visualization:
   - Analyze data using statistical techniques, machine learning, or domain-specific algorithms.
10. Decision Making and Control:
    - Interpret data insights to make informed decisions or take appropriate actions.
11. Data Archival and Retention:
    - Archive historical sensor/device data for reference or compliance.
12. Data Security and Privacy:
    - Ensure data security during transmission, storage, and access.

2.5.1 Code Snippets - Sensor/Device Data Collection

Sensor/Device Data
```
import serial

# Connect to the serial port of the device
ser = serial.Serial('port', baudrate=9600)
# Read data from the device
data = ser.readline()
# Close the serial connection
ser.close()
```

Optimization

1 Reading Large Data from Sensors in Chunks:
```
# Read data from sensors in chunks (e.g., 1KB at a time)
def read_sensor_data_in_chunks(sensor):
    chunk_size = 1024
    while True:
        data_chunk = sensor.read(chunk_size)
        if not data_chunk:
            break
        # Process data_chunk
        # ...
```
2 Asynchronous Data Acquisition:
```
import asyncio
# Asynchronous data acquisition from multiple sensors
async def read_sensor_data(sensor):
    data = await sensor.read()
    # Process data
    # ...
async def main():
    sensors = [Sensor1(), Sensor2(), Sensor3()]
    tasks = [read_sensor_data(sensor) for sensor in sensors]
    await asyncio.gather(*tasks)
asyncio.run(main())
```
3 Data Preprocessing with Pandas (for CSV data):
```
import pandas as pd
# Read data from a CSV file
data = pd.read_csv('sensor_data.csv')
# Preprocess data (e.g., remove outliers)
data_cleaned = data[(data['value'] >= 0) & (data['value'] <= 100)]
```
4 Efficient Data Storage with SQLite:
```
import sqlite3
# Create a SQLite database and table
conn = sqlite3.connect('sensor_data.db')
cursor = conn.cursor()
cursor.execute('''CREATE TABLE IF NOT EXISTS sensor_data (
                  timestamp TEXT,
                  value REAL
                  )''')
# Insert data into the table
sensor_data = [('2023-07-01 12:00:00', 25.5), ('2023-07-01 12:01:00', 26.8), ...]
cursor.executemany('INSERT INTO sensor_data VALUES (?, ?)', sensor_data)
# Commit changes and close connection
conn.commit()
conn.close()
```
5 Using a Time-Series Database:
```
from influxdb import InfluxDBClient
client = InfluxDBClient(host='localhost', port=8086)
client.switch_database('sensors')
data = [
    {"measurement": "temperature",
        "tags": {"sensor_id": "sensor1", "location": "room1"},
        "time": "2023-07-01T12:00:00Z",
        "fields": {"value": 25.5}
    },
    # More data points...
]
client.write_points(data)
result = client.query('SELECT * FROM temperature WHERE location=\'room1\'')
```

3 Data Preparation

3.1 Definitions – Data Preparation

Data preparation is the process of preparing raw data so that it is suitable for further processing and analysis. Key steps include collecting, cleaning, and labeling raw data into a form suitable for machine learning (ML) algorithms and then exploring and visualizing the data.

3.2 Definitions – Reading Data from CSV

Reading Data from CSV

Reading data from CSV (Comma-Separated Values) is the process of loading structured tabular data stored in a CSV file format into a program or data analysis environment for further processing, analysis, or modeling.
Process of Reading Data from CSV:

1. Importing Required Library:
   - Import the necessary library (e.g., Pandas) to read CSV data in Python.
2. Specifying File Path:
   - Provide the path or filename of the CSV file you want to read.
3. Reading CSV Data:
   - Use the library function (e.g., Pandas `read_csv()`) to load the CSV data.
4. Data Exploration:
   - Examine the loaded data to understand its structure and basic information.
5. Data Cleaning and Preprocessing:
   - Handle missing values, outliers, and perform initial data cleaning if needed.
6. Data Analysis or Modeling (Optional):
   - Proceed with data analysis or modeling tasks if applicable to the project.

3.2.1 Code Snippets - Reading Data from CSV

Reading Data (CSV)
```
import pandas as pd

# Step 1: Specify the file path
file_path = 'path_to_your_csv_file.csv'

# Step 2: Read the CSV file
data = pd.read_csv(file_path)

# Step 3: Explore the data
print(data.head())  # Display the first few rows of the data
print(data.shape)   # Get the number of rows and columns in the data
print(data.info())  # Get information about data types and missing values
```

Optimization

1 Using Chunking for Large Files:
```
import pandas as pd

# Read data in chunks (e.g., 1000 rows at a time)
chunk_size = 1000
chunks = pd.read_csv('data.csv', chunksize=chunk_size)

for chunk in chunks:
    # Process each chunk
    # ...
```
2 Specifying Data Types:
```
# Explicitly set data types while reading CSV
data = pd.read_csv('data.csv', dtype={'column1': int, 'column2': float, 'column3': str})
```
3 Using CSV Reader:
```
import csv

# Read data using CSV Reader
with open('data.csv', 'r') as csv_file:
    csv_reader = csv.reader(csv_file)
    headers = next(csv_reader)  # Skip headers if present
    for row in csv_reader:
        # Process each row
        # ...
```
4 Skipping Rows and Columns:
```
# Skip the first row (header) and read specific columns only
data = pd.read_csv('data.csv', skiprows=1, usecols=['column1', 'column2'])
```
5 Using Faster Libraries:
```
import dask.dataframe as dd

# Read data using Dask for parallel processing
data = dd.read_csv('data.csv')
# Perform operations on 'data' (Dask DataFrame)
```

3.3 Definitions – Handling Missing Values

Handling Missing Values

When dealing with missing data, data scientists can use two primary methods to solve the error: imputation or the removal of data.
Process of Handling Missing Values:

1. Identifying Missing Values:
   - Check for missing values in the dataset.
2. Understanding the Pattern:
   - Analyze the missing data pattern (MCAR, MAR, MNAR).
3. Deletion or Dropping:
   - Remove rows or columns with missing values.
4. Imputation Techniques:
   - Fill missing values with estimated data.
5. Creating Missing Value Indicators:
   - Generate binary indicators for missing values.
6. Advanced Imputation Techniques:
   - Use multiple imputation or probabilistic methods for complex cases.
7. Consider Domain Knowledge:
   - Apply domain knowledge to handle missing values.
8. Evaluate and Monitor:
   - Assess the impact of handling and monitor for potential issues.

Missing Values
```
import pandas as pd

# Step 1: Read data from CSV (assuming 'data' is a DataFrame)
data = pd.read_csv('data.csv')

# Step 2: Identifying missing values
missing_values = data.isnull().sum()

# Step 3: Deletion/Dropping Missing Values (if applicable)
data_cleaned = data.dropna()

# Step 4: Imputation (fill missing values) using mean (for example)
data_imputed = data.fillna(data.mean())

# Step 5 (Optional): Creating Missing Value Indicators
data['column_name_indicator'] = data['column_name'].isnull().astype(int)
```

Optimization

1 Dropping Rows with Missing Values:
```
# Drop rows with missing values in a specific column
data_cleaned = data.dropna(subset=['column_name'])
```
2 Imputation with Mean/Median/Mode:
```
# Fill missing values in a numerical column with its mean
data['numerical_column'].fillna(data['numerical_column'].mean(), inplace=True)

# Fill missing values in a numerical column with its median
data['numerical_column'].fillna(data['numerical_column'].median(), inplace=True)

# Fill missing values in a categorical column with its mode
data['categorical_column'].fillna(data['categorical_column'].mode()[0], inplace=True)
```
3 Forward-Fill and Backward-Fill:
```
# Forward-fill missing values (use the last valid observation to fill)
data.fillna(method='ffill', inplace=True)

# Backward-fill missing values (use the next valid observation to fill)
data.fillna(method='bfill', inplace=True)
```
4 Advance Imputation with KNN:
```
from sklearn.impute import KNNImputer
# Create KNNImputer object with desired neighbors
knn_imputer = KNNImputer(n_neighbors=5)
# Impute missing values using KNN
data_imputed = knn_imputer.fit_transform(data)
```
5 Using Pandas `fillna()` with Dictionary:
```
# Fill missing values using a dictionary (e.g., different values for different columns)
imputation_dict = {'column1': value1, 'column2': value2, ...}
data.fillna(imputation_dict, inplace=True)
```
6 Impute Categorical Variables with 'Unknown' Category:
```
# Fill missing values in a categorical column with 'Unknown'
data['categorical_column'].fillna('Unknown', inplace=True)
```

Optimization

7 Imputation with Linear Regression:
```
from sklearn.linear_model import LinearRegression
# Separate columns with missing values and complete columns
data_missing = data[data['column_with_missing'].isnull()]
data_complete = data.dropna(subset=['column_with_missing'])
# Train a linear regression model to predict missing values
model = LinearRegression()
model.fit(data_complete[['feature1', 'feature2']], data_complete['column_with_missing'])
imputed_values = model.predict(data_missing[['feature1', 'feature2']])

# Fill missing values with predicted values
data.loc[data['column_with_missing'].isnull(), 'column_with_missing'] = imputed_values
```

3.4 Definitions – Data Transformation

Data Transformation

Data transformation is the process of converting, cleansing, and structuring data into a usable format that can be analyzed to support decision making processes, and to propel the growth of an organization.
Process of Data Transformation:

1. Data Cleaning and Preprocessing:
   - Handle missing values and prepare data for further processing.
2. Feature Scaling:
   - Normalize or standardize numerical features to a common scale.
3. Feature Engineering:
   - Create new features or derive relevant information from existing ones.
4. Encoding Categorical Variables:
   - Convert categorical variables into numerical representations.
5. Data Discretization (Binning):
   - Group continuous values into bins or intervals for simplification.
6. Log Transformation:
   - Apply log transformation to reduce the impact of extreme values.
7. Data Normalization:
   - Scale data to ensure equal weightage for all features.
8. Data Smoothing:
   - Apply techniques to remove noise from time series data.
9. Principal Component Analysis (PCA):
   - Reduce data dimensionality for better visualization or modeling.
10. Data Integration and Aggregation:
    - Merge data from different sources and aggregate it for analysis.
11. Data Reduction (Sampling):
    - Reduce the size of large datasets using sampling techniques.
12. Handling Skewed Data (if applicable):
    - Apply transformations to handle data with a skewed distribution.
13. Handling Seasonality (for time series data):
    - Use techniques like seasonal decomposition to extract seasonal components.

3.4.1 Code Snippets - Data Transformation

Data Transformation
```
# Convert categorical variables to numerical using one-hot encoding
data_encoded = pd.get_dummies(data, columns=['categorical_column'])

# Log transformation of a column (for positive values)
data['log_transformed'] = data['original_column'].apply(lambda x: np.log(x))
```

Handling Missing Values with Imputation
```
# Use SimpleImputer for numerical columns
from sklearn.impute import SimpleImputer
imputer = SimpleImputer(strategy='mean')
data[['numeric_feature1', 'numeric_feature2']] = imputer.fit_transform(data[['numeric_feature1',
'numeric_feature2']])

# Use most frequent value for categorical columns
data['categorical_feature'].fillna(data['categorical_feature'].mode()[0], inplace=True)
```

Feature Scaling with Min-Max Scaling
```
from sklearn.preprocessing import MinMaxScaler

# Use MinMaxScaler for scaling numerical features to the range [0, 1]
scaler = MinMaxScaler()
data[['numeric_feature1', 'numeric_feature2']] = scaler.fit_transform(data[['numeric_feature1',
'numeric_feature2']])
```

One-Hot Encoding with get_dummies()
```
data_encoded = pd.get_dummies(data, columns=['categorical_column'], drop_first=True)
```

Log Transformation (avoiding zero values)
```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
data_['numeric_feature1', 'numeric_feature2']] = scaler.fit_transform(data[['numeric_feature1',
'numeric_feature2']])
```

Data Normalization (Z-score Scaling)
```
from sklearn.preprocessing import StandardScaler

# Use StandardScaler for z-score normalization
scaler  StandardScaler()
data_['numeric_feature1', 'numeric_feature2']] = scaler.fi_transform(data[['numeric_feature1',
'numeric_feature2']])
```

Principal Components Analysis(PCA)
```
from sklearn.decomposition import PCA

# Use PCA to reduce data dimensionality
pca = PCA(n_components=2)
pca_data = pca.fit_transform(data[['feature1', 'feature2']])
```

Data Aggregation with GroupBy
```
grouped_data = data.groupby('group_column')['value_column'].mean()
```

Applying Transformations to Multiple Columns
```
# List of numerical columns to scale
numerical_columns = ['numeric_feature1', 'numeric_feature2']

# Apply Min-Max Scaling to multiple columns
data_numerical_columns] = scaler.fit_transform(data[numerical_columns])
```

Data Reduction (Sampling)
```
# Use sample() to randomly sample a fraction of data (e.g., 30%)
sampled_data = data.sample(frac=0.3)

# Use sample() to randomly sample a fixed number of data points (e.g., 1000)
sampled_data = data.sample(n=1000)
```

Handling Skewed Data with Log Transformation
```
import numpy as np

# Use log transformation to handle skewed data (avoiding zero values)
data_['log_transformed'] = data['original_column'].apply(lambda x: np.log(x) if x > 0 else 0)
```

Handling Outliers
```
# Use percentile-based outlier detection and removal
q_low = data['numeric_feature'].quantile(0.01)
q_high = data['numeric_feature'].quantile(0.99)
data = data[(data['numeric_feature'] >= q_low) & (data['numeric_feature'] <= q_high)]
```

Handling Seasonality (for Time Series Data)

```
# Assuming you have a time series DataFrame named 'ts_data'
# Create a DatetimeIndex for time-based indexing
ts_data.index = pd.to_datetime(ts_data.index)

# Use seasonal decomposition using statsmodels' seasonal_decompose
from statsmodels.tsa.seasonal import seasonal_decompose
result = seasonal_decompose(ts_data['Sales'], model='additive', period=12)

# Extract seasonal, trend, and residual components
seasonal = result.seasonal
trend = result.trend
residual = result.resid
```

Data Smoothing

```
# Use rolling window (e.g., 7 days) for moving average smoothing
data['smoothed_values'] = data['original_values'].rolling(window=7).mean()
```

Feature Engineering

```
# Create new features based on existing ones
data['new_feature'] = data['feature1'] + data['feature2']

# Extract date components from datetime feature
data['year'] = data['date_column'].dt.year
data['month'] = data['date_column'].dt.month
```

3.5 Definitions – Data Scaling

Data Scaling

Data scaling is the process of transforming numerical features to a common scale, ensuring that they have comparable magnitudes. It is a crucial step in data preprocessing, especially when working with algorithms that are sensitive to the scale of features, such as distance-based methods and gradient-based optimization algorithms.

Process of Data Scaling:
The process of data scaling involves the following steps:

1. Select Numerical Features:
   Identify the numerical features that require scaling.
2. Choose Scaling Method:
   Decide on the scaling method based on the nature of the data and the requirements of the machine learning model. Common scaling methods include Min-Max Scaling (Normalization) and Z-score Scaling (Standardization).
3. Apply Scaling:
   Scale the numerical features based on the chosen method to bring them to a common scale.

3.5.1 Code Snippets – Data Scaling

Data Scaling

```
from sklearn.preprocessing import StandardScaler

# Scale numerical features to have mean=0 and variance=1
scaler = StandardScaler()
scaled_data = scaler.fit_transform(data[['numeric_feature1', 'numeric_feature2']])
```

Min-Max Scaling (Normalization) with MinMaxScaler

```
from sklearn.preprocessing import MinMaxScaler

# Create a DataFrame with numerical features
data = pd.DataFrame({'Feature1': [10, 20, 30, 40], 'Feature2': [100, 200, 300, 400]})

# Use MinMaxScaler for scaling numerical features to the range [0, 1]
scaler = MinMaxScaler()
scaled_data = scaler.fit_transform(data[['Feature1', 'Feature2']])
```

Z-score Scaling (Standardization) with StandardScaler

```
from sklearn.preprocessing import StandardScaler

# Create a DataFrame with numerical features
data = pd.DataFrame({'Feature1': [10, 20, 30, 40], 'Feature2': [100, 200, 300, 400]})

# Use StandardScaler for z-score normalization
scaler = StandardScaler()
scaled_data = scaler.fit_transform(data[['Feature1', 'Feature2']])
```

Scaling Specific Features with MinMaxScaler

```
from sklearn.preprocessing import MinMaxScaler

# Create a DataFrame with numerical features
data = pd.DataFrame({'Age': [25, 35, 45], 'Income': [50000, 75000, 60000]})

# Use MinMaxScaler to scale specific features to the range [0, 1]
scaler = MinMaxScaler()
data['Scaled_Age'] = scaler.fit_transform(data[['Age']])
```

Applying Scaling to Test Data Using Saved Scaler

```
from sklearn.preprocessing import MinMaxScaler

# Create a DataFrame with numerical features
train_data = pd.DataFrame({'Feature1': [10, 20, 30, 40], 'Feature2': [100, 200, 300, 400]})

# Use MinMaxScaler for scaling numerical features to the range [0, 1]
scaler = MinMaxScaler()
scaled_train_data = scaler.fit_transform(train_data[['Feature1', 'Feature2']])

# Save the scaler for future use (e.g., scaling test data)
import joblib
joblib.dump(scaler, 'scaler.pkl')

# Later, when scaling test data:
loaded_scaler = joblib.load('scaler.pkl')
scaled_test_data = loaded_scaler.transform(test_data[['Feature1', 'Feature2']])
```

3.6 Definitions – Data Splitting for Machine Learning

Data Splitting for ML

Data splitting is when data is divided into two or more subsets. Typically, with a two-part split, one part is used to evaluate or test the data and the other to train the model. Data splitting is an important aspect of data science, particularly for creating models based on data.

Process of Data Splitting:

1. Data Preparation:
   Preprocess, clean, handle missing values, and engineer features for analysis or modeling.
2. Shuffling (Optional):
   Randomly shuffle data to eliminate order bias and ensure better generalization.
3. Train-Test Split:
   Divide data into training and testing sets for model training and evaluation.
4. Optional Validation Set:
   Further split training data for hyperparameter tuning and model optimization.
5. Model Training:
   Train machine learning models using the training set for learning patterns.
6. Model Evaluation:
   Assess model performance on the testing (or validation) set for generalization testing.
7. Adjustment and Improvement:
   Fine-tune models based on evaluation results to improve accuracy and efficiency.

3.6.1 Code Snippets – Data Splitting for Machine Learning

Data Splitting for ML

```
from sklearn.model_selection import train_test_split

# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(features, target, test_size=0.2)
```

Optimized Data Splitting with Stratified Sampling (for Classification Task)

```
from sklearn.model_selection import train_test_split

# Assuming you have a DataFrame 'data' with features and labels
X = data.drop('target', axis=1)  # Features
y = data['target']  # Labels

# Use stratified sampling to preserve class distribution in training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, stratify=y, random_state=42)
```

Strategies

1 Simple Train-Test Split:

```
from sklearn.model_selection import train_test_split

# Assuming you have a DataFrame 'data' with features and labels
X = data.drop('target', axis=1)  # Features
y = data['target']  # Labels

# Split the data into a training set and a testing set (80-20 split)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

2 Stratified Split for Classification Tasks:

```
from sklearn.model_selection import train_test_split

# Assuming you have a DataFrame 'data' with features and class labels 'target'
X = data.drop('target', axis=1)  # Features
y = data['target']  # Class labels

# Use stratified sampling to preserve class distribution in training and testing sets
X_train, X_test, y_train, y_test  train_test_split(X, y, test_size 0.2, stratify y, random_state 42)
```

3 Time Series Data Splitting:

```
# For time series data, split sequentially based on time
train_size = int(0.8 * len(data))
X_train, X_test = data[:train_size], data[train_size:]
```

4 Cross-Validation Split:

```
from sklearn.model_selection import KFold

# Assuming you have a DataFrame 'data' with features and labels
X = data.drop('target', axis=1)  # Features
y = data['target']  # Labels

# Use K-Fold cross-validation (e.g., K=5)
kf = KFold(n_splits=5, shuffle=True, random_state=42)
for train_index, test_index in kf.split(X):
    X_train, X_test = X.iloc[train_index], X.iloc[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]
```

5 Group-Based Split (for Grouped Data):

```
from sklearn.model_selection import GroupShuffleSplit

# Assuming you have a DataFrame 'data' with features, labels, and groups
X = data.drop(['target', 'group'], axis=1)  # Features
y = data['target']  # Labels
groups = data['group']  # Group labels

# Use GroupShuffleSplit for group-based data splitting
gss = GroupShuffleSplit(n_splits=1, test_size=0.2, random_state=42)
train_index, test_index = next(gss.split(X, y, groups=groups))
X_train, X_test = X.iloc[train_index], X.iloc[test_index]
y_train, y_test = y.iloc[train_index], y.iloc[test_index]
```

3.7 Definitions – Feature Selection

Feature Selection

Feature selection is the process of selecting a subset of relevant and informative features (variables or attributes) from the original set of features in a dataset. The goal of feature selection is to improve the performance of machine learning models by reducing the dimensionality of the data, removing irrelevant or redundant features, and focusing on the most important ones that have a significant impact on the target variable.

Process of Feature Selection:

1. Data Preparation:
   Clean, preprocess, and encode categorical variables if necessary for feature selection.
2. Feature Importance Ranking:
   Rank features based on relevance scores to prioritize their selection.
3. Filter Methods:
   Apply statistical tests to select the most informative features effectively.
4. Wrapper Methods:
   Utilize ML models to evaluate subsets of features iteratively.
5. Embedded Methods:
   Leverage model's built-in feature selection capabilities during training.
6. Validation and Evaluation:
   Assess selected features' impact on model performance through evaluation metrics.
7. Model Adjustment:
   Iterate with different techniques or settings to optimize feature selection outcome.
8. Final Model Training:
   Train ML model with the chosen subset of features for prediction or analysis.

3.7.1 Code Snippets – Feature Selection

Feature Selection

```
from sklearn.feature_selection import SelectKBest, f_regression

# Select top k features based on ANOVA F-value
selector = SelectKBest(score_func=f_regression, k=5)
selected_features = selector.fit_transform(X_train, y_train)
```

Optimization

1 Feature Importing Ranking with Random Forest:

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.feature_selection import SelectFromModel

# Assuming you have X_train and y_train for training
# Create a Random Forest classifier
rf_classifier = RandomForestClassifier(n_estimators=100, random_state=42)

# Fit the classifier to the training data
rf_classifier.fit(X_train, y_train)

# Use feature_importances_ to get feature importance scores
feature_importance  rf_classifier.feature_importances_

# Use SelectFromModel to select features based on a threshold
sfm = SelectFromModel(rf_classifier, threshold=0.1)
X_train_selected = sfm.fit_transform(X_train, y_train)
```

2 Recursive Feature Elimination with Cross-Validation (RFE):

```
from sklearn.feature_selection import RFE
from sklearn.linear_model import LogisticRegression

# Assuming you have X_train and y_train for training
# Create a logistic regression model (or any other model)
logreg_model = LogisticRegression()
# Use RFE to recursively eliminate features with cross-validation
rfe = RFE(estimator=logreg_model, n_features_to_select=5)
X_train_selected  rfe.fit_transform(X_train, y_train)
```

3 Feature Selection with SelectKBest using Chi-Square:

```
from sklearn.feature_selection import SelectKBest, chi2

# Assuming you have X_train and y_train for training
# Use SelectKBest with chi2 to select the top k features
k = 10  # Set the number of top features you want to select
selector = SelectKBest(score_func=chi2, k=k)
X_train_selected = selector.fit_transform(X_train, y_train)
```

4 Feature Selection with L1 Regularization (Lasso):

```
from sklearn.linear_model import LassoCV

# Assuming you have X_train and y_train for training
# Use L1 regularization (Lasso) for feature selection
lasso_model = LassoCV(cv=5)
X_train_selected = lasso_model.fit_transform(X_train, y_train)
```

5 Feature Selection with Mutual Information:

```
from sklearn.feature_selection import SelectKBest, mutual_info_classif

# Assuming you have X_train and y_train for training
# Use SelectKBest with mutual information for feature selection
k = 10  # Set the number of top features you want to select
selector  SelectKBest(score_func mutual_info_classif, k k)
X_train_selected = selector.fit_transform(X_train, y_train)
```

6 Feature Selection with Recursive Feature Elimination and Grid Search:

```
from sklearn.feature_selection import RFE
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV

# Assuming you have X_train and y_train for training
# Create a logistic regression model (or any other model)
logreg_model = LogisticRegression()
# Use RFE to recursively eliminate features with cross-validation
rfe = RFE(estimator=logreg_model)
param_grid = {'n_features_to_select': [5, 10, 15]}  # Set the number of top features you want to select
grid_search = GridSearchCV(estimator=rfe, param_grid=param_grid, cv=5)
grid_search.fit(X_train, y_train)
# Get the best feature subset from grid search
X_train_selected = grid_search.best_estimator_.transform(X_train)
```

Optimization

7 Feature Selection with Variance Threshold:

```
from sklearn.feature_selection import VarianceThreshold

# Assuming you have X_train for training
# Use VarianceThreshold to remove low-variance features
threshold = 0.01  # Set the threshold for variance
vt = VarianceThreshold(threshold=threshold)
X_train_selected = vt.fit_transform(X_train)
```

8 Feature Selection with SelectFromModel and Grid Search:

```
from sklearn.feature_selection import SelectFromModel
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import GridSearchCV

# Assuming you have X_train and y_train for training
# Create a Random Forest classifier
rf_classifier = RandomForestClassifier()

# Use SelectFromModel to select features based on a threshold
sfm = SelectFromModel(rf_classifier)
param_grid = {'threshold': [0.1, 0.2, 0.3]}  # Set the threshold values for feature selection
grid_search = GridSearchCV(estimator=sfm, param_grid=param_grid, cv=5)
grid_search.fit(X_train, y_train)

# Get the best feature subset from grid search
X_train_selected = grid_search.best_estimator_.transform(X_train)
```

3.8 Definitions – Handling Imbalanced Data (Oversampling)

Handling Imbalanced Data

Handling imbalanced data is a critical preprocessing step in machine learning when the distribution of classes in the target variable is highly uneven. Imbalanced data can lead to biased model performance, where the model becomes more accurate in predicting the majority class but performs poorly on the minority class. The goal of handling imbalanced data is to ensure that the model can effectively learn patterns from both classes, providing a more balanced and fair performance.

Process of Handling Imbalanced Data:

1. Data Analysis:
   Examine class distribution to identify imbalance and its impact on model performance.
2. Resampling Techniques:
   Use oversampling, undersampling, or SMOTE to balance the dataset.
3. Class Weights:
   Assign higher weights to the minority class during model training.
4. Ensemble Methods:
   Utilize ensemble techniques that handle imbalanced data effectively.
5. Performance Metrics:
   Use precision, recall, F1-score, or ROC-AUC for evaluation.
6. Fine-Tuning:
   Continuously monitor performance and adjust techniques for better balance and accuracy.

3.8.1 Code Snippets – Handling Imbalanced Data

Handling Imbalanced Data

```
from imblearn.over_sampling import SMOTE

# Apply SMOTE to oversample the minority class
smote = SMOTE()
X_resampled, y_resampled = smote.fit_resample(X_train, y_train)
```

Resampling Techniques

1 Oversampling:

```
from sklearn.utils import resample

# Assuming you have X_train and y_train for training
# Combine X_train and y_train to create a single DataFrame
train_data = pd.concat([X_train, y_train], axis=1)

# Separate majority and minority classes
majority_class = train_data[train_data['target'] == 0]
minority_class = train_data[train_data['target'] == 1]

# Upsample the minority class to match the majority class size
minority_upsampled = resample(minority_class, replace=True, n_samples=len(majority_class), random_state 42)

# Combine majority and upsampled minority classes
balanced_data = pd.concat([majority_class, minority_upsampled])

# Split the balanced data into X_train_balanced and y_train_balanced
X_train_balanced = balanced_data.drop('target', axis=1)
y_train_balanced = balanced_data['target']
```

2 Undersampling:

```
from sklearn.utils import resample

# Assuming you have X_train and y_train for training
# Combine X_train and y_train to create a single DataFrame
train_data = pd.concat([X_train, y_train], axis=1)

# Separate majority and minority classes
majority_class = train_data[train_data['target'] == 0]
minority_class = train_data[train_data['target'] == 1]

# Downsample the majority class to match the minority class size
majority_downsampled = resample(majority_class, replace=False, n_samples=len(minority_class), random_state=42)

# Combine minority and downsampled majority classes
balanced_data = pd.concat([majority_downsampled, minority_class])

# Split the balanced data into X_train_balanced and y_train_balanced
X_train_balanced = balanced_data.drop('target', axis=1)
y_train_balanced = balanced_data['target']
```

Class Weights

```
from sklearn.linear_model import LogisticRegression

# Assuming you have X_train and y_train for training
# Create a logistic regression model with balanced class weights
logreg_model = LogisticRegression(class_weight='balanced', random_state=42)
logreg_model.fit(X_train, y_train)
```

Ensemble Methods

```
from sklearn.ensemble import RandomForestClassifier

# Assuming you have X_train and y_train for training
# Create a Random Forest classifier with balanced class weights
rf_classifier = RandomForestClassifier(class_weight='balanced', n_estimators=100, random_state=42)
rf_classifier.fit(X_train, y_train)
```

SMOTE (Synthetic Minority Over-sampling Technique)

```
from imblearn.over_sampling import SMOTE

# Assuming you have X_train and y_train for training
# Apply SMOTE to upsample the minority class
smote = SMOTE(sampling_strategy='auto', random_state=42)
X_train_balanced, y_train_balanced = smote.fit_resample(X_train, y_train)
```

3.9 Definitions – Data Normalization(Min-Max Scaling)

Data Normalization

Data normalization is the process of reorganizing data within a database so that users can utilize it for further queries and analysis.

Process of Data Normalization:

Selecting Features:
Identify the numerical features (attributes) that need normalization in the dataset.

Choosing Normalization Method:
Decide on the appropriate normalization method based on the data distribution and the requirements of the machine learning algorithm.

Min-Max Normalization:
Scale the features to a predefined range, often [0, 1], using the formula:

Z-Score Normalization (Standardization):
Standardize the features to have a mean of 0 and a standard deviation of 1, using the formula:

Scaling to [-1, 1] Range:
Use a variation of Min-Max normalization to scale features to the range [-1, 1], particularly if the data contains negative values:

3.9.1 Code Snippets – Data Normalization

Data Normalization

```
from sklearn.preprocessing import MinMaxScaler

# Scale features to a specific range (e.g., [0, 1])
scaler = MinMaxScaler()
normalized_data = scaler.fit_transform(data[['numeric_feature1', 'numeric_feature2']])
```

Min-Max Normalization

```
from sklearn.preprocessing import MinMaxScaler

# Assuming you have X_train and X_test for training and testing
# Create a MinMaxScaler object
scaler = MinMaxScaler()
# Fit the scaler on the training data and transform both training and testing data
X_train_normalized = scaler.fit_transform(X_train)
X_test_normalized = scaler.transform(X_test)
```

Z-Score Normalization

```
from sklearn.preprocessing import StandardScaler

# Assuming you have X_train and X_test for training and testing
# Create a StandardScaler object
scaler = StandardScaler()

# Fit the scaler on the training data and transform both training and testing data
X_train_standardized = scaler.fit_transform(X_train)
X_test_standardized = scaler.transform(X_test)
```

Scaling to [-1, 1] Range

```
# Assuming you have X_train and X_test for training and testing
# Calculate the minimum and minimum values of the dataset
max_value = X_train.max().max()
min_value = X_train.min().min()

# Scale the features to the range [-1, 1]
X_train_normalized = 2 * (X_train - min_value) / (max_value - min_value) - 1
X_test_normalized = 2 * (X_test - min_value) / (max_value - min_value) - 1
```

Robust Scaling

```
from sklearn.preprocessing import RobustScaler

# Assuming you have X_train and X_test for training and testing
# Create a RobustScaler object
scaler = RobustScaler()

# Fit the scaler on the training data and transform both training and testing data
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

Log Transformation

```
import numpy as np

# Assuming you have X_train and X_test for training and testing
# Apply log transformation to the features
X_train_log = np.log1p(X_train)
X_test_log = np.log1p(X_test)
```

Power Transformation (Box-Cox or Yeo-Johnson)

```
from sklearn.preprocessing import PowerTransformer

# Assuming you have X_train and X_test for training and testing
# Create a PowerTransformer object with method 'box-cox' or 'yeo-johnson'
scaler = PowerTransformer(method='box-cox')
# Fit the scaler on the training data and transform both training and testing data
X_train_transformed = scaler.fit_transform(X_train)
X_test_transformed = scaler.transform(X_test)
```

3.10 Definitions – Data Binning

Data Scaling

Binning is a way to group a number of more or less continuous values into a smaller number of "bins".

Process of Data Binning:

1. **Data Selection:**
   Choose the numerical feature (attribute) that requires binning.
2. **Determining the Number of Bins:**
   Decide the number of bins or intervals to create. This can be based on domain knowledge, the desired level of granularity, or using statistical methods like Sturges' rule or Scott's rule.
3. **Binning Method:**
   Select a binning method:
   - Equal-Width Binning:
     Divide the data range into equal-sized intervals.
   - Equal-Frequency Binning:
     Divide the data into intervals with an equal number of data points in each bin.
   - Custom Binning:
     Define custom bin boundaries based on specific criteria or requirements.
4. **Binning Process:**
   Assign each data point to its corresponding bin based on its value.
5. **Data Transformation:**
   Replace the original numerical values with the bin labels or bin identifiers.

3.10.1 Code Snippets – Data Binning

Data Scaling

```
from sklearn.preprocessing import StandardScaler

# Scale numerical features to have mean=0 and variance=1
scaler = StandardScaler()
scaled_data = scaler.fit_transform(data[['numeric_feature1', 'numeric_feature2']])
```

Optimization

1 Equal-Width Binning:

```
import pandas as pd

# Assuming you have a DataFrame named 'data' with a numerical column 'age'
# Define the number of bins you want
num_bins = 5

# Calculate the width of each bin
bin_width = (data['age'].max() - data['age'].min()) / num_bins
# Define the bin edges based on the bin width
bin_edges = [data['age'].min() + i * bin_width for i in range(num_bins + 1)]
# Define the bin labels
bin_labels = [f'Bin{i}' for i in range(1, num_bins + 1)]
# Use pd.cut() to create a new 'age_bin' column with the bin labels
data['age_bin'] = pd.cut(data['age'], bins=bin_edges, labels=bin_labels, right=False)
```

2 Equal-Frequency Binning:

```
import pandas as pd

# Assuming you have a DataFrame named 'data' with a numerical column 'score'
# Define the number of bins you want
num_bins = 5
# Use pd.qcut() to create a new 'score_bin' column with the bin labels
data['score_bin'] = pd.qcut(data['score'], q=num_bins, labels=False)
```

3.11 Definitions – Handling DateTime Data

Handling DateTime Data

Handling datetime data is a crucial part of data preparation when working with datasets that contain date and time information. The process involves converting datetime data into a format that is suitable for analysis and machine learning tasks.

Handling datetime data typically includes the following steps:

1. Data Parsing:
   Convert datetime strings to machine-readable format.
2. Feature Extraction:
   Extract relevant components (e.g., year, month) from datetime data.
3. Datetime Indexing:
   Set datetime values as the dataset's index for time-based analysis.
4. Handling Time Zones:
   Address time zone conversions to maintain consistency.
5. Visualizing Temporal Data:
   Create time series plots to identify trends and patterns.

3.11.1 Code Snippets – Handling DateTime Data

Handling DateTime Data

```
# Convert string to datetime
data['date_column'] = pd.to_datetime(data['date_column'])

# Extract features from DateTime (e.g., year, month, day)
data['year'] = data['date_column'].dt.year
data['month'] = data['date_column'].dt.month
data['day'] = data['date_column'].dt.day
```

Optimization

1Data Parsing and Conversion:

```
import pandas as pd

# Assuming you have a DataFrame named 'data' with a column 'date_time' containing datetime strings
# Convert 'date_time' column to datetime objects
data['date_time'] = pd.to_datetime(data['date_time'])
```

2 Feature Extraction :

```
# Assuming you have a DataFrame named 'data' with a column 'date_time' of datetime objects
# Extract year, month, day, hour, and minute as new columns
data['year'] = data['date_time'].dt.year
data['month'] = data['date_time'].dt.month
data['day'] = data['date_time'].dt.day
data['hour'] = data['date_time'].dt.hour
data['minute'] = data['date_time'].dt.minute
```

3 Datetime Indexing:

```
# Assuming you have a DataFrame named 'data' with a 'date_time' column of datetime objects
# Set 'date_time' column as the DataFrame index for time-based analysis
data.set_index('date_time', inplace=True)
```

4 Handling Time Zones:

```
# Assuming you have a DataFrame named 'data' with a 'date_time' column of datetime objects
# Convert datetime to a different time zone (e.g., 'UTC' to 'US/Eastern')
data['date_time'] = data['date_time'].dt.tz_convert('US/Eastern')
```

5 Visualization Temporal Data:

```
import matplotlib.pyplot as plt
# Assuming you have a DataFrame named 'data' with a 'date_time' column of datetime objects
# Create a time series plot for visualization
plt.figure(figsize=(10, 6))
plt.plot(data['date_time'], data['value'])
plt.xlabel('Date Time')
plt.ylabel('Value')
plt.title('Time Series Plot')
plt.show()
```

4 Statistical Analysis

4.1 Definitions – Statistical Analysis

Statistical analysis, or statistics, is the process of collecting and analyzing data to identify patterns and trends, remove bias and inform decision-making. It's an aspect of business intelligence that involves the collection and scrutiny of business data and the reporting of trends.

4.2 Definitions – Descriptive Statistics

Descriptive Statistics

Descriptive statistics is a branch of statistical analysis that involves summarizing and presenting data in a meaningful and interpretable way. It provides an overview of the main features of a dataset, allowing analysts to understand its central tendencies, variability, and distribution.

Process of Descriptive Statistics:

1. **Central Tendency Measures:**
   - Calculate mean, median, and mode to determine central values representing the dataset.
2. **Variability Measures:**
   - Find range, variance, and standard deviation to understand the spread and dispersion of data.
3. **Distribution Shape Analysis:**
   - Use histograms and kernel density plots to visualize data distribution's shape.
   - Assess skewness to understand the symmetry or skewness of the data.
4. **Percentiles and Quartiles:**
   - Calculate percentiles to identify values below specific percentages of the data.
   - Determine quartiles to divide data into four equal parts.
5. **Data Visualization and Interpretation:**
   - Utilize box plots, scatter plots, and bar charts to visualize the data.
   - Interpret descriptive statistics to gain insights into data characteristics and trends.

4.2.1 Code Snippets - Descriptive Statistics

Descriptive Statistics

```
import numpy as np

# Assuming you have a NumPy array or a Pandas Series 'data'
# Calculate mean, median, and standard deviation
mean_value = np.mean(data)
median_value = np.median(data)
std_deviation = np.std(data)
# Count occurrences of each unique value in the data
value_counts = np.unique(data, return_counts=True)
```

Optimization

1 Calculating Mean, Median, and Mode with Pandas:

```
import pandas as pd

# Assuming you have a DataFrame named 'data' with a column 'values'
# Calculate mean, median, and mode using Pandas
mean_value = data['values'].mean()
median_value = data['values'].median()
mode_value = data['values'].mode().iloc[0]
```

2 Variability Measures with NumPy:

```
import numpy as np

# Assuming you have a NumPy array named 'data'
# Calculate range, variance, and standard deviation with NumPy
data_range = np.ptp(data)  # Range: Difference between max and min values
data_variance = np.var(data)  # Variance
data_std_dev = np.std(data)  # Standard deviation
```

3 Quartiles and Percentiles with Pandas:

```
import pandas as pd

# Assuming you have a DataFrame named 'data' with a column 'values'
# Calculate quartiles and percentiles using Pandas
quartiles = data['values'].quantile([0.25, 0.5, 0.75])
percentile_75 = data['values'].quantile(0.75)  # 75th percentile
```

4 Data Visualization with Matplotlib :

```
import matplotlib.pyplot as plt

# Assuming you have a Pandas Series named 'data'
# Create a histogram to visualize data distribution
plt.hist(data, bins=10, edgecolor='black')
plt.xlabel('Values')
plt.ylabel('Frequency')
plt.title('Histogram')
plt.show()
```

4.3 Definitions – Inferential Statistics (Hypothesis Testing)

Inferential statistics is a branch of statistical analysis that focuses on drawing conclusions and making predictions about a population based on a sample of data. It involves using sample data to make inferences or generalizations about a larger population from which the sample was drawn. The primary goal of inferential statistics is to make statistically sound and reliable predictions and to test hypotheses.

4.3.1 Code Snippets – Inferential Statistics

Inferential Statistics

```
from scipy.stats import ttest_ind

# Assuming you have two sets of data: 'data1' and 'data2'
# Perform a two-sample t-test to compare means
t_statistic, p_value = ttest_ind(data1, data2)
if p_value < 0.05:
    print("Reject null hypothesis: Means are significantly different.")
else:
    print("Fail to reject null hypothesis: Means are not significantly different.")
```

Hypothesis Testing (T-Test)

```
from scipy.stats import ttest_ind

# Assuming you have two sets of data: 'data1' and 'data2'
# Perform an independent two-sample t-test
t_statistic, p_value = ttest_ind(data1, data2)
if p_value < 0.05:
    print("Reject null hypothesis: Means are significantly different.")
else:
    print("Fail to reject null hypothesis: Means are not significantly different.")
```

Confidence Intervals

```
from scipy.stats import t

# Assuming you have a sample data 'data' and sample mean 'sample_mean'
# Calculate the confidence interval for a given confidence level (e.g., 95%)
confidence_level = 0.95
n = len(data)
standard_error = data.std() / (n ** 0.5)
margin_of_error = t.ppf((1 + confidence_level) / 2, df=n - 1) * standard_error
lower_bound = sample_mean - margin_of_error
upper_bound = sample_mean + margin_of_error
```

Regression Analysis (Linear Regression)

```
import statsmodels.api as sm

# Assuming you have a DataFrame named 'data' with columns 'X' and 'Y'
# Perform linear regression
X = data['X']
X = sm.add_constant(X)  # Add a constant for the intercept
Y = data['Y']
model = sm.OLS(Y, X).fit()
intercept, slope = model.params
```

One-Way ANOVA

```
from scipy.stats import f_oneway

# Assuming you have multiple sets of data in a list 'data_groups'
# Perform one-way ANOVA to compare means across groups
f_statistic, p_value = f_oneway(*data_groups)
if p_value < 0.05:
    print("Reject null hypothesis: Means are significantly different.")
else:
    print("Fail to reject null hypothesis: Means are not significantly different.")
```

Chi-Square Test

```
from scipy.stats import chi2_contingency

# Assuming you have a contingency table named 'contingency_table'
# Perform a chi-square test of independence
chi2_statistic, p_value, dof, expected = chi2_contingency(contingency_table)
if p_value < 0.05:
    print("Reject null hypothesis: Variables are dependent.")
else:
    print("Fail to reject null hypothesis: Variables are independent.")
```

Confidence Interval (Binomal Distribution)

```
from scipy.stats import binom

# Assuming you have a sample of successes 'successes' and total trials 'trials'
# Calculate the confidence interval for a given confidence level (e.g., 95%)
confidence_level = 0.95
sample_proportion = successes / trials
margin_of_error = binom.ppf((1 + confidence_level) / 2, n=trials, p=sample_proportion) / trials
lower_bound = sample_proportion - margin_of_error
upper_bound = sample_proportion + margin_of_error
```

Non-Parametric Test

```
from scipy.stats import ranksums

# Assuming you have two sets of data: 'data1' and 'data2'
# Perform the Wilcoxon rank-sum test (Mann-Whitney U test)
statistic, p_value = ranksums(data1, data2)
if p_value < 0.05:
    print("Reject null hypothesis: Distributions are significantly different.")
else:
    print("Fail to reject null hypothesis: Distributions are not significantly different.")
```

Power Calculation for Sample Size

```
from statsmodels.stats.power import TTestIndPower

# Assuming you want to calculate the required sample size for a t-test
# Set the parameters for power calculation
effect_size = 0.5
alpha = 0.05
power = 0.8
# Calculate the required sample size
power_analysis = TTestIndPower()
required_sample_size = power_analysis.solve_power(effect_size=effect_size, alpha=alpha, power=power)
```

4.4 Definitions – Correlation Analysis

Correlation Analysis

Correlation analysis is a statistical technique used to measure the relationship or association between two or more variables. It helps to understand how changes in one variable are related to changes in another variable. Correlation analysis is particularly useful for identifying patterns, dependencies, and connections between different data points.

The result of a correlation analysis is called the correlation coefficient, which quantifies the strength and direction of the relationship between variables. The correlation coefficient typically ranges from -1 to +1:

A positive correlation (+1) indicates that as one variable increases, the other variable tends to increase as well.

A negative correlation (-1) indicates that as one variable increases, the other variable tends to decrease.

4.4.1 Code Snippets – Correlation Analysis

Correlation Analysis

```
import pandas as pd

# Assuming you have a DataFrame named 'data' with columns 'feature1' and 'feature2'
# Calculate Pearson correlation coefficient
correlation_coefficient = data['feature1'].corr(data['feature2'])
# Calculate correlation matrix for multiple features
correlation_matrix = data.corr()
```

Optimization

1 Pearson Correlation Coefficient with Pandas:

```
import pandas as pd

# Assuming you have a DataFrame named 'data' with columns 'X' and 'Y'
# Calculate Pearson correlation coefficient between 'X' and 'Y'
correlation_coefficient = data['X'].corr(data['Y'])
```

2 Spearman Rank Correlation with Pandas:

```
import pandas as pd

# Assuming you have a DataFrame named 'data' with columns 'X' and 'Y'
# Calculate Spearman rank correlation coefficient between 'X' and 'Y'
spearman_correlation = data['X'].corr(data['Y'], method='spearman')
```

3 Correlation Matrix with Pandas:

```
import pandas as pd

# Assuming you have a DataFrame named 'data' with multiple columns
# Calculate the correlation matrix for all pairs of columns
correlation_matrix = data.corr()
```

4 Heatmap Visualization with Seaborn:

```
import seaborn as sns
import matplotlib.pyplot as plt
# Assuming you have a correlation matrix named 'correlation_matrix'
# Create a heatmap to visualize the correlation matrix
plt.figure(figsize=(8, 6))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt=".2f", linewidths=0.5)
plt.title('Correlation Heatmap')
plt.show()
```

5 Partial Correlation with Pandas and Statsmodels:

```
import pandas as pd
import statsmodels.api as sm

# Assuming you have a DataFrame named 'data' with columns 'X', 'Y', and 'Z'
# Calculate partial correlation between 'X' and 'Y' while controlling for 'Z'
partial_correlation = sm.OLS(data['X'], sm.add_constant(data[['Y', 'Z']])).fit().params['Y']
```

6 Correlation Significance Testing:

```
import pandas as pd
from scipy.stats import pearsonr
# Assuming you have a DataFrame named 'data' with columns 'X' and 'Y'
# Perform correlation significance testing (Pearson correlation) with p-value
correlation_coefficient, p_value = pearsonr(data['X'], data['Y'])
if p_value < 0.05:
    print("Correlation is significant.")
else:
    print("Correlation is not significant.")
```

7 Parallelized Correlation Matrix Calculation:

```
import pandas as pd
from joblib import Parallel, delayed
# Assuming you have a large DataFrame named 'data' with multiple columns
# Calculate the correlation matrix for all pairs of columns in parallel
def parallel_corr(column1, column2):
    return column1.corr(column2)
correlation_matrix = pd.DataFrame(Parallel(n_jobs=-1)(delayed(parallel_corr)(data[col1], data[col2]) for col1 in data.columns for col2 in data.columns), index=data.columns, columns=data.columns)
```

4.5 Definitions – Analysis of Variance (ANOVA)

Analysis of Variance

Analysis of Variance (ANOVA) is a statistical technique used to compare the means of two or more groups to determine if there are statistically significant differences between them. It assesses whether the variations between group means are greater than the variations within the groups.

ANOVA is commonly used when dealing with categorical independent variables and a continuous dependent variable. It helps to answer questions such as:

Are there any significant differences in the means of a dependent variable across different groups?
Which group means are significantly different from each other?
ANOVA compares the variance between the group means (explained variance) with the variance within each group (unexplained variance). If the explained variance is significantly greater than the unexplained variance, it suggests that there are significant differences between the groups.

There are different types of ANOVA, depending on the number of independent variables and the design of the study:

One-Way ANOVA: Compares the means of a single dependent variable across multiple groups (e.g., comparing the test scores of students from different schools).

Two-Way ANOVA: Considers the effects of two independent variables on a single dependent variable (e.g., studying the impact of both gender and age on exam performance).

Multivariate ANOVA (MANOVA): Handles multiple dependent variables simultaneously, allowing the examination of group differences across several outcomes.

4.5.1 Code Snippets – Analysis of Variance (ANOVA)

ANOVA

```
from scipy.stats import f_oneway
# Assuming you have multiple sets of data in a list 'data_groups'
# Perform one-way ANOVA to compare means across groups
f_statistic, p_value = f_oneway(*data_groups)
if p_value < 0.05:
    print("Reject null hypothesis: Means are significantly different.")
else:
    print("Fail to reject null hypothesis: Means are not significantly different.")
```

One-Way ANOVA

```
from scipy.stats import f_oneway
# Assuming you have multiple sets of data in a list 'data_groups'
# Perform one-way ANOVA to compare means across groups
f_statistic, p_value = f_oneway(*data_groups)

if p_value < 0.05:
    print("Reject null hypothesis: Means are significantly different.")
else:
    print("Fail to reject null hypothesis: Means are not significantly different.")
```

Two-Way Anovo

```
import pandas as pd
from statsmodels.formula.api import ols
from statsmodels.stats.anova import anova_lm
# Assuming you have a DataFrame named 'data' with columns 'dependent_var', 'independent_var1', and
'independent_var2'
# Create the ANOVA model using formula notation
formula = 'dependent_var ~ independent_var1 + independent_var2 + independent_var1:independent_var2'
model = ols(formula, data).fit()

# Perform two-way ANOVA and extract results
anova_results = anova_lm(model)

# Print the ANOVA table with F-statistic and p-value
print(anova_results)
```

Multivariate ANOVA

```
import pandas as pd
import statsmodels.api as sm
from statsmodels.multivariate.manova import MANOVA

# Assuming you have a DataFrame named 'data' with columns 'dependent_var1', 'dependent_var2', and 'group_var'
# Create a multivariate design matrix
y = data[['dependent_var1', 'dependent_var2']].values

# Add a constant column for the intercept
X = sm.add_constant(data['group_var'])

# Perform MANOVA
manova = MANOVA(y, X)

# Print the MANOVA results
print(manova.mv_test())
```

Optimization

1 One-Way ANOVA with DataFrames:

```
import pandas as pd
from scipy.stats import f_oneway

# Assuming you have a DataFrame named 'data' with columns 'dependent_var' and 'group_var'
# Group the data by 'group_var' and perform one-way ANOVA
grouped_data = [group['dependent_var'] for _, group in data.groupby('group_var')]
f_statistic, p_value = f_oneway(*grouped_data)
```

2 One-Way ANOVA with GroupBy :

```
import pandas as pd
from scipy.stats import f_oneway

# Assuming you have a DataFrame named 'data' with columns 'dependent_var' and 'group_var'
# Perform one-way ANOVA using the GroupBy function
grouped_data = data.groupby('group_var')['dependent_var']
f_statistic, p_value = f_oneway(*grouped_data)
```

3 Two-Way ANOVA with Statsmodels:

```
import pandas as pd
import statsmodels.api as sm
from statsmodels.formula.api import ols

# Assuming you have a DataFrame named 'data' with columns 'dependent_var', 'independent_var1', and
'independent_var2'
# Create the ANOVA model using formula notation
formula = 'dependent_var ~ independent_var1 + independent_var2 + independent_var1:independent_var2'
model = ols(formula, data).fit()

# Print the ANOVA table with F-statistic and p-value
anova_table = sm.stats.anova_lm(model)
print(anova_table)
```

4 One-Way ANOVA with Post-hoc Test:

```
import pandas as pd
from statsmodels.stats.multicomp import pairwise_tukeyhsd

# Assuming you have a DataFrame named 'data' with columns 'dependent_var' and 'group_var'
# Perform one-way ANOVA
f_statistic, p_value = f_oneway(*[group['dependent_var'] for _, group in data.groupby('group_var')])

# Perform Tukey's HSD post-hoc test for pairwise comparisons
if p_value < 0.05:
    tukey_result = pairwise_tukeyhsd(data['dependent_var'], data['group_var'])
    print(tukey_result)
```

5 Mixed-Design ANOVA with Pingouin:

```
import pandas as pd
from pingouin import mixed_anova
# Assuming you have a DataFrame named 'data' with columns 'dependent_var', 'between_group_var', and
'within_group_var'
# Perform mixed-design ANOVA (Between-Within ANOVA)
anova_results = mixed_anova(data, dv='dependent_var', between='between_group_var', within='within_group_var')
print(anova_results)
```

4.6 Definitions – Probability Distribution

Probability Distribution

In statistical analysis for data science, a probability distribution is a mathematical function that describes the likelihood of various outcomes or events occurring in a random experiment or process. It provides a systematic way to model and understand the uncertainty associated with random variables.
Commonly used probability distributions in data science include:

1. Normal Distribution (Gaussian Distribution): Often used for continuous data and is characterized by its bell-shaped curve.

2. Binomial Distribution: Applicable for discrete data with two possible outcomes, like success/failure.

3. Poisson Distribution: Suitable for discrete data that represents the number of events in a fixed interval.

4. Exponential Distribution: Useful for modeling time-to-event data, such as the time between two events occurring.

5. Uniform Distribution: Represents data where all outcomes have equal probabilities.

6. Chi-Squared Distribution: Used in hypothesis testing and confidence interval calculations.

7. Student's t-Distribution: Similar to the normal distribution but used when the sample size is small or the population variance is unknown.

4.6.1 Code Snippets – Probability Distribution

Probability Distribution

```
from scipy.stats import norm

# Assuming you want to calculate probabilities for a normal distribution
# Calculate probability density function (PDF) for a specific value
pdf_value = norm.pdf(x, loc=mean, scale=std_dev)

# Calculate cumulative distribution function (CDF) for a specific value
cdf_value = norm.cdf(x, loc=mean, scale=std_dev)
```

Normal Distribution

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm

# Generate random data from a normal distribution with mean=0 and standard deviation=1
data = np.random.normal(loc=0, scale=1, size=1000)

# Plot the histogram of the generated data
plt.hist(data, bins=30, density=True, alpha=0.6, color='g')

# Plot the probability density function (PDF) for the standard normal distribution
x = np.linspace(-4, 4, 1000)
plt.plot(x, norm.pdf(x), 'r', lw=2)

plt.xlabel('Value')
plt.ylabel('Probability Density')
plt.title('Normal Distribution')
plt.legend(['PDF', 'Data'])
plt.show()
```

Binomial Distribution

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import binom
# Generate random data from a binomial distribution with n=10 and p=0.5
data = np.random.binomial(n=10, p=0.5, size=1000)

# Plot the histogram of the generated data
plt.hist(data, bins=11, range=(-0.5, 10.5), density=True, alpha=0.6, color='b')

# Plot the probability mass function (PMF) for the binomial distribution with n=10 and p=0.5
x = np.arange(0, 11)
plt.plot(x, binom.pmf(x, n=10, p=0.5), 'r', lw=2, marker='o')

plt.xlabel('Number of Successes')
plt.ylabel('Probability Mass')
plt.title('Binomial Distribution')
plt.legend(['PMF', 'Data'])
plt.show()
```

Poisson Distribution

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import poisson

# Generate random data from a Poisson distribution with lambda=3
data = np.random.poisson(lam=3, size=1000)

# Plot the histogram of the generated data
plt.hist(data, bins=15, range=(0, 15), density=True, alpha=0.6, color='m')

# Plot the probability mass function (PMF) for the Poisson distribution with lambda=3
x = np.arange(0, 15)
plt.plot(x, poisson.pmf(x, mu=3), 'r', lw=2, marker='o')

plt.xlabel('Number of Events')
plt.ylabel('Probability Mass')
plt.title('Poisson Distribution')
plt.legend(['PMF', 'Data'])
plt.show()
```

Exponential Distribution

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import expon

# Generate random data from an exponential distribution with scale parameter=2
data = np.random.exponential(scale=2, size=1000)

# Plot the histogram of the generated data
plt.hist(data, bins=30, density=True, alpha=0.6, color='c')

# Plot the probability density function (PDF) for the exponential distribution with scale parameter=2
x = np.linspace(0, 15, 1000)
plt.plot(x, expon.pdf(x, scale=2), 'r', lw=2)

plt.xlabel('Value')
plt.ylabel('Probability Density')
plt.title('Exponential Distribution')
plt.legend(['PDF', 'Data'])
plt.show()
```

Uniform Distribution

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import uniform

# Generate random data from a uniform distribution between 0 and 1
data = np.random.uniform(0, 1, size=1000)

# Plot the histogram of the generated data
plt.hist(data, bins=30, density=True, alpha=0.6, color='y')

# Plot the probability density function (PDF) for the uniform distribution between 0 and 1
x = np.linspace(0, 1, 1000)
plt.plot(x, uniform.pdf(x), 'r', lw=2)

plt.xlabel('Value')
plt.ylabel('Probability Density')
plt.title('Uniform Distribution')
plt.legend(['PDF', 'Data'])
plt.show()
```

Chi-Squared Distribution

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import chi2

# Generate random data from a chi-squared distribution with degrees of freedom=5
data = np.random.chisquare(df=5, size=1000)

# Plot the histogram of the generated data
plt.hist(data, bins=30, density=True, alpha=0.6, color='purple')

# Plot the probability density function (PDF) for the chi-squared distribution with degrees of freedom=5
x   np.linspace(0, 20, 1000)
plt.plot(x, chi2.pdf(x, df=5), 'r', lw=2)

plt.xlabel('Value')
plt.ylabel('Probability Density')
plt.title('Chi-Squared Distribution')
plt.legend(['PDF', 'Data'])
plt.show()
```

Student`s t-Distribution

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import t

# Generate random data from a t-distribution with degrees of freedom=10
data = np.random.standard_t(df=10, size=1000)

# Plot the histogram of the generated data
plt.hist(data, bins=30, density=True, alpha=0.6, color='orange')

# Plot the probability density function (PDF) for the t-distribution with degrees of freedom=10
x   np.linspace(-5, 5, 1000)
plt.plot(x, t.pdf(x, df=10), 'r', lw=2)

plt.xlabel('Value')
plt.ylabel('Probability Density')
plt.title("Student's t-Distribution")
plt.legend(['PDF', 'Data'])
plt.show()
```

Log-Normal Distribution

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import t

# Generate random data from a t-distribution with degrees of freedom=10
data = np.random.standard_t(df=10, size=1000)

# Plot the histogram of the generated data
plt.hist(data, bins=30, density=True, alpha=0.6, color='orange')

# Plot the probability density function (PDF) for the t-distribution with degrees of freedom=10
x = np.linspace(-5, 5, 1000)
plt.plot(x, t.pdf(x, df=10), 'r', lw=2)

plt.xlabel('Value')
plt.ylabel('Probability Density')
plt.title("Student's t-Distribution")
plt.legend(['PDF', 'Data'])
plt.show()
```

Beta Distribution

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import beta

# Generate random data from a beta distribution with shape parameters alpha=2 and beta=5
data = np.random.beta(a=2, b=5, size=1000)

# Plot the histogram of the generated data
plt.hist(data, bins=30, density=True, alpha=0.6, color='gray')

# Plot the probability density function (PDF) for the beta distribution with shape parameters alpha=2 and beta=5
x = np.linspace(0, 1, 1000)
plt.plot(x, beta.pdf(x, a=2, b=5), 'r', lw=2)

plt.xlabel('Value')
plt.ylabel('Probability Density')
plt.title('Beta Distribution')
plt.legend(['PDF', 'Data'])
plt.show()
```

Gamma Distribution

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import gamma

# Generate random data from a gamma distribution with shape=2 and scale=2
data = np.random.gamma(shape=2, scale=2, size=1000)

# Plot the histogram of the generated data
plt.hist(data, bins=30, density=True, alpha=0.6, color='blue')
# Plot the probability density function (PDF) for the gamma distribution with shape 2 and scale 2
x = np.linspace(0, 15, 1000)
plt.plot(x, gamma.pdf(x, a 2, scale 2), 'r', lw 2)

plt.xlabel('Value')
plt.ylabel('Probability Density')
plt.title('Gamma Distribution')
plt.legend(['PDF', 'Data'])
plt.show()
```

Weibull Distribution

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import weibull_min

# Generate random data from a Weibull distribution with shape=2 and scale=2
data = np.random.weibull(a=2, size=1000)

# Plot the histogram of the generated data
plt.hist(data, bins=30, density=True, alpha=0.6, color='green')

# Plot the probability density function (PDF) for the Weibull distribution with shape=2 and scale=2
x = np.linspace(0, 5, 1000)
plt.plot(x, weibull_min.pdf(x, c=2, scale=2), 'r', lw=2)

plt.xlabel('Value')
plt.ylabel('Probability Density')
plt.title('Weibull Distribution')
plt.legend(['PDF', 'Data'])
plt.show()
```

F - Distribution

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import f

# Generate random data from an F-distribution with degrees of freedom 2 and 10
data = np.random.f(dfnum=2, dfden=10, size=1000)

# Plot the histogram of the generated data
plt.hist(data, bins=30, density=True, alpha=0.6, color='purple')

# Plot the probability density function (PDF) for the F-distribution with degrees of freedom 2 and 10
x = np.linspace(0, 5, 1000)
plt.plot(x, f.pdf(x, dfn=2, dfd=10), 'r', lw=2)

plt.xlabel('Value')
plt.ylabel('Probability Density')
plt.title('F-Distribution')
plt.legend(['PDF', 'Data'])
plt.show()
```

Pareto Distribution

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import pareto

# Generate random data from a Pareto distribution with shape=3
data = np.random.pareto(a=3, size=1000)

# Plot the histogram of the generated data
plt.hist(data, bins=30, density=True, alpha=0.6, color='cyan')

# Plot the probability density function (PDF) for the Pareto distribution with shape=3
x   np.linspace(1, 10, 1000)
plt.plot(x, pareto.pdf(x, b=3), 'r', lw=2)

plt.xlabel('Value')
plt.ylabel('Probability Density')
plt.title('Pareto Distribution')
plt.legend(['PDF', 'Data'])
plt.show()
```

Logistic Distribution

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import logistic

# Generate random data from a logistic distribution with location=0 and scale=1
data   np.random.logistic(loc 0, scale 1, size 1000)

# Plot the histogram of the generated data
plt.hist(data, bins=30, density=True, alpha=0.6, color='pink')

# Plot the probability density function (PDF) for the logistic distribution with location=0 and scale=1
x = np.linspace(-5, 5, 1000)
plt.plot(x, logistic.pdf(x), 'r', lw=2)

plt.xlabel('Value')
plt.ylabel('Probability Density')
plt.title('Logistic Distribution')
plt.legend(['PDF', 'Data'])
plt.show()
```

Laplace Distribution

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import laplace

# Generate random data from a Laplace distribution with location=0 and scale=1
data = np.random.laplace(loc=0, scale=1, size=1000)

# Plot the histogram of the generated data
plt.hist(data, bins=30, density=True, alpha=0.6, color='orange')

# Plot the probability density function (PDF) for the Laplace distribution with location=0 and scale=1
x = np.linspace(-5, 5, 1000)
plt.plot(x, laplace.pdf(x), 'r', lw=2)

plt.xlabel('Value')
plt.ylabel('Probability Density')
plt.title('Laplace Distribution')
plt.legend(['PDF', 'Data'])
plt.show()
```

Optimization

1 Fitting a Distribution to Data: Maximum Likelihood Estimation (MLE):

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm
from scipy.optimize import minimize

# Sample data
data = np.random.normal(loc=10, scale=2, size=100)

# Define the negative log-likelihood function for the normal distribution
def neg_log_likelihood(params):
    loc, scale = params
    return -np.sum(norm.logpdf(data, loc=loc, scale=scale))

# Perform MLE using optimization (minimization) with initial parameter guesses
result = minimize(neg_log_likelihood, [0, 1])
optimal_loc, optimal_scale = result.x

# Plot the histogram of the data
plt.hist(data, bins=20, density=True, alpha=0.6)

# Plot the fitted normal distribution
x = np.linspace(5, 15, 100)
plt.plot(x, norm.pdf(x, loc=optimal_loc, scale=optimal_scale), 'r', lw=2)

plt.xlabel('Value')
plt.ylabel('Probability Density')
plt.title('Fitted Normal Distribution')
plt.legend(['Fitted PDF', 'Data'])
plt.show()
```

2 Generating Random Samples with Specific Parameters:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import expon

# Generate random samples from an exponential distribution with scale parameter=2
data = expon.rvs(scale=2, size=1000)

# Plot the histogram of the generated data
plt.hist(data, bins=30, density=True, alpha=0.6, color='c')

# Plot the probability density function (PDF) for the exponential distribution with scale parameter=2
x = np.linspace(0, 15, 1000)
plt.plot(x, expon.pdf(x, scale=2), 'r', lw=2)

plt.xlabel('Value')
plt.ylabel('Probability Density')
plt.title('Exponential Distribution')
plt.legend(['PDF', 'Data'])
plt.show()
```

3 Efficiently Generating Random Samples with Numpy:

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

# Generate random samples from a normal distribution with mean=5 and standard deviation=1
data = np.random.normal(loc=5, scale=1, size=1000)
# Plot the histogram of the generated data
plt.hist(data, bins=30, density=True, alpha=0.6, color='g')
# Plot the probability density function (PDF) for the normal distribution
x = np.linspace(1, 9, 1000)
plt.plot(x, np.exp(-(x-5)**2/2)    np.sqrt(2*np.pi), 'r', lw=2)

plt.xlabel('Value')
plt.ylabel('Probability Density')
plt.title('Normal Distribution')
plt.legend(['PDF', 'Data'])
plt.show()
```

4 Kernel Density Estimation (KDE) for Smoothing Data:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm
from sklearn.neighbors import KernelDensity

# Sample data
data = np.random.normal(loc=10, scale=2, size=100)
# Perform KDE to estimate the probability density function (PDF) from the data
kde = KernelDensity(bandwidth=0.5, kernel='gaussian')
kde.fit(data[:, None])
x = np.linspace(5, 15, 1000)
log_pdf = kde.score_samples(x[:, None])
# Plot the histogram of the data
plt.hist(data, bins=20, density=True, alpha=0.6)

# Plot the estimated PDF using KDE
plt.plot(x, np.exp(log_pdf), 'r', lw=2)

plt.xlabel('Value')
plt.ylabel('Probability Density')
plt.title('Kernel Density Estimation (KDE)')
plt.legend(['Estimated PDF', 'Data'])
plt.show()
```

5 Random Sampling with Given Probabilities:

```
from sklearn.model_selection import GroupShuffleSplit
import numpy as np
import matplotlib.pyplot as plt

# Define custom probabilities for discrete values
values = [1, 2, 3, 4, 5]
probabilities = [0.2, 0.1, 0.3, 0.2, 0.2]

# Generate random samples according to the defined probabilities
data = np.random.choice(values, size=1000, p=probabilities)

# Plot the histogram of the generated data
plt.hist(data, bins=5, density=True, alpha=0.6, color='orange')

# Plot the probability mass function (PMF) based on the defined probabilities
plt.plot(values, probabilities, 'ro', markersize=8)

plt.xlabel('Value')
plt.ylabel('Probability Mass')
plt.title('Random Sampling with Given Probabilities')
plt.legend(['PMF', 'Data'])
plt.show()
```

6 Efficient Random Sampling with NumPy Broadcasting:

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

# Generate random samples from a uniform distribution between 0 and 1
data   np.random.random(size 1000)

# Plot the histogram of the generated data
plt.hist(data, bins=30, density=True, alpha=0.6, color='pink')

# Plot the probability density function (PDF) for the uniform distribution between 0 and 1
x = np.linspace(0, 1, 1000)
plt.plot(x, np.ones_like(x), 'r', lw=2)

plt.xlabel('Value')
plt.ylabel('Probability Density')
plt.title('Uniform Distribution')
plt.legend(['PDF', 'Data'])
plt.show()
```

Chi-Square Test

```
from scipy.stats import chi2_contingency

# Assuming you have a contingency table named 'contingency_table'
# Perform a chi-square test of independence
chi2_statistic, p_value, dof, expected = chi2_contingency(contingency_table)
if p_value < 0.05:
    print("Reject null hypothesis: Variables are dependent.")
else:
    print("Fail to reject null hypothesis: Variables are independent.")
```

Probability Calculation

```
# Assuming you want to calculate the probability of an event
total_trials = 100
successful_trials = 25
# Calculate the probability of success in one trial
probability_one_trial = successful_trials / total_trials
# Calculate the probability of failure in one trial
probability_failure = 1 - probability_one_trial
```

Time Series Analysis (Moving Average)

```
import pandas as pd

# Assuming you have a DataFrame named 'data' with a column 'value' and 'date_time' as the index
# Calculate the rolling mean (moving average) over a window of 5 periods
data['moving_average'] = data['value'].rolling(window=5).mean()
```

Sampling and Random Numbers

```
import numpy as np
# Assuming you want to generate random samples from a normal distribution
# Generate 100 random samples from a normal distribution with mean=0 and standard deviation=1
random_samples = np.random.normal(loc=0, scale=1, size=100)
# Generate random integers between a given range (e.g., 1 to 10)
random_integers = np.random.randint(low=1, high=11, size=100)
```

5 Data Preparation

5.1 Definitions – Statistical Modeling

Statistical modeling is the process of formulating, estimating, and interpreting mathematical models that represent relationships between variables in data. It is a crucial aspect of statistical analysis and data science, used to gain insights, make predictions, and understand complex systems. Statistical models aim to summarize and describe the underlying patterns, distributions, and dependencies in the data.

5.2 Definitions – Linear Regression

Linear Regression

Linear regression is a statistical modeling technique used to determine the linear relationship between a dependent variable and one or more independent variables. It assumes that there is a linear relationship between the input variables and the target variable. The goal of linear regression is to find the best-fitting line (or hyperplane in higher dimensions) that minimizes the difference between the actual values and the predicted values.

In simple linear regression, there is a single independent variable, while in multiple linear regression, there are multiple independent variables. The linear regression model represents the relationship between the independent variables and the dependent variable using a linear equation:

y = β 0 + β 1x1 + β 2x2 + ... + β n*xn

where:

y is the dependent variable (target variable),
β 0 is the y-intercept (the value of y when all independent variables are zero),
β 1, β 2, ..., β n are the coefficients (or slopes) that represent the change in y for a one-unit change in the corresponding independent variable,
x1, x2, ..., xn are the independent variables.
The goal of linear regression is to estimate the coefficients (β 0, β 1, ..., β n) that minimize the sum of the squared differences between the actual y values and the predicted y values. This is usually done using a method called least squares estimation.

5.2.1 Code Snippets – Linear Regression

Linear Regression

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression

# Sample data
X = np.array([1, 2, 3, 4, 5]).reshape(-1, 1)
y   np.array([2, 3.5, 4.8, 5.5, 7])

# Create a linear regression model
model   LinearRegression()
model.fit(X, y)
# Predict using the model
X_test = np.array([6, 7]).reshape(-1, 1)
y_pred = model.predict(X_test)
# Plot the data and the regression line
plt.scatter(X, y, color='blue')
plt.plot(X_test, y_pred, color='red')
plt.xlabel('X')
plt.ylabel('y')
plt.title('Linear Regression')
plt.show()
```

Simple Linear Regresion

```
import numpy as np
from sklearn.linear_model import LinearRegression

# Assuming you have a NumPy array 'X' as the independent variable and 'y' as the dependent variable
X = np.array([1, 2, 3, 4, 5]).reshape(-1, 1)
y = np.array([2, 3.5, 4.8, 5.5, 7])
# Create a linear regression model
model = LinearRegression()
model.fit(X, y)

# Get the coefficients and intercept
coefficients = model.coef_
intercept = model.intercept_

# Predict using the model
X_test = np.array([6, 7]).reshape(-1, 1)
y_pred = model.predict(X_test)
```

Multiple Linear Regression

```
import numpy as np
from sklearn.linear_model import LinearRegression

# Assuming you have a NumPy array 'X' with multiple independent variables and 'y' as the dependent variable
X = np.array([[1, 2], [2, 4], [3, 6], [4, 8], [5, 10]])
y = np.array([2, 3.5, 4.8, 5.5, 7])

# Create a linear regression model
model = LinearRegression()
model.fit(X, y)

# Get the coefficients and intercept
coefficients = model.coef_
intercept = model.intercept_

# Predict using the model
X_test = np.array([[6, 12], [7, 14]])
y_pred = model.predict(X_test)
```

Polynomial Regression

```
import numpy as np
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures

# Assuming you have a NumPy array 'X' as the independent variable and 'y' as the dependent variable
X = np.array([1, 2, 3, 4, 5]).reshape(-1, 1)
y = np.array([2, 3.5, 4.8, 5.5, 7])
# Create polynomial features
poly_features = PolynomialFeatures(degree=2)
X_poly   poly_features.fit_transform(X)
# Create a linear regression model
model   LinearRegression()
model.fit(X_poly, y)
# Make predictions on the training data
y_pred = model.predict(X_poly)
```

Ridge Regression

```
import numpy as np
from sklearn.linear_model import Ridge

# Assuming you have a NumPy array 'X' as the independent variable and 'y' as the dependent variable
X = np.array([1, 2, 3, 4, 5]).reshape(-1, 1)
y = np.array([2, 3.5, 4.8, 5.5, 7])
# Create a Ridge regression model
alpha = 0.1  # Regularization strength
model   Ridge(alpha alpha)
model.fit(X, y)
# Predict using the model
X_test = np.array([6, 7]).reshape(-1, 1)
y_pred = model.predict(X_test)
```

Evaluating Metrics

```
import numpy as np
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score
# Example data
X = np.array([[1], [2], [3], [4], [5]])  # Input features
y = np.array([2, 3.5, 4.8, 5.5, 7])      # Target variable
# Create a linear regression model
model = LinearRegression()
# Fit the model to the data
model.fit(X, y)
# Predict using the trained model
y_pred = model.predict(X)
# Calculate evaluation metrics
mse = mean_squared_error(y, y_pred)
r2   r2_score(y, y_pred)
# Print the evaluation metrics
print("Mean Squared Error:", mse)
print("R-squared:", r2)
```

Prediction with Linear Regression

```
import numpy as np
from sklearn.linear_model import LinearRegression
# Input features (X) and target variable (y)
X = ...  # Your input feature array or matrix
y = ...  # Your target variable array
# Create a linear regression model
model = LinearRegression()
# Fit the model to the data
model.fit(X, y)
# Predict using the trained model
y_pred = model.predict(X)
```

Optimization

```
import numpy as np
from sklearn.linear_model import LinearRegression

# Example data
X = np.array([[1], [2], [3], [4], [5]])  # Input features
y = np.array([2, 3.5, 4.8, 5.5, 7])      # Target variable

# Add a column of ones for the intercept term
X = np.hstack((np.ones((X.shape[0], 1)), X))

# Initialize parameters
theta = np.zeros(X.shape[1])

# Hyperparameters
learning_rate = 0.01
num_iterations = 1000

# Gradient Descent
for _ in range(num_iterations):
    # Compute predictions
    y_pred = np.dot(X, theta)

    # Compute gradients
    gradients = (1   X.shape[0]) * np.dot(X.T, (y_pred - y))

    # Update parameters
    theta -= learning_rate * gradients

# Print the optimized parameters
print("Optimized Parameters:", theta)
```

5.3 Definitions – Logistic Regression

Logistic Regression

In logistic regression, the goal is to model the probability of an instance belonging to a certain class. It uses the logistic or sigmoid function to map the linear combination of input features to the predicted probability.

The logistic function is defined as:

p(x) = 1 / (1 + exp(-z))

where p(x) is the probability of the positive class, z is the linear combination of input features and corresponding coefficients, and exp(-z) is the exponential function.

The equation for logistic regression can be written as:

log(p(x) / (1 - p(x))) = β ₀ + β ₁x₁ + β ₂x₂ + ... + β ₙxₙ

where log(p(x) / (1 - p(x))) is the logarithm of the odds ratio, β ₀, β ₁, β ₂, ... β ₙ are the coefficients, and x₁, x₂, ... xₙ are the input features.

During training, the model estimates the coefficients β ₀, β ₁, β ₂, ... β ₙ that maximize the likelihood of the observed data. These coefficients represent the impact of each input feature on the predicted probability.

5.3.1 Code Snippets – Logistic Regression

Logistic Regression

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_classification
from sklearn.linear_model import LogisticRegression

# Generate synthetic data for binary classification
X, y = make_classification(n_samples=100, n_features=2, n_informative=2, n_redundant=0, random_state=42)
# Create a logistic regression model
model = LogisticRegression()
model.fit(X, y)
# Plot the data and the decision boundary
plt.scatter(X[:, 0], X[:, 1], c=y, cmap='coolwarm')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')

# Plot the decision boundary
ax = plt.gca()
xlim = ax.get_xlim()
ylim = ax.get_ylim()
xx = np.linspace(xlim[0], xlim[1], 100)
yy = np.linspace(ylim[0], ylim[1], 100)
XX, YY = np.meshgrid(xx, yy)
Z = model.predict(np.c_[XX.ravel(), YY.ravel()]).reshape(XX.shape)
plt.contourf(XX, YY, Z, alpha=0.3, cmap='coolwarm')

plt.title('Logistic Regression')
plt.show()
```

Strategies

1 Logistic Regression Model Training:

```
from sklearn.linear_model import LogisticRegression

# Create a logistic regression model
model = LogisticRegression()
# Fit the model to the training data
model.fit(X_train, y_train)
```

2 Prediction with Logistic Regression:

```
# Make predictions on the test data
y_pred = model.predict(X_test)
```

3 Probability Estimation with Logistic Regression:

```
# Predict probabilities for each class
proba = model.predict_proba(X_test)

# Get the probabilities for the positive class
proba_positive = proba[:, 1]
```

4 Threshold Adjustment for Logistic Regression Predictions:

```
# Adjust the threshold for class predictions
threshold = 0.5
y_pred_adjusted = (proba_positive >= threshold).astype(int)
```

5 Random Sampling with Given Probabilities:

```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

# Calculate accuracy
accuracy   accuracy_score(y_test, y_pred)
# Calculate precision
precision   precision_score(y_test, y_pred)
# Calculate recall
recall = recall_score(y_test, y_pred)
# Calculate f1 score
f1 = f1_score(y_test, y_pred)
```

6 Efficient Random Sampling with NumPy Broadcasting:

```
from sklearn.linear_model import LogisticRegression

# Create a logistic regression model with L1 regularization
model_l1 = logisticRegression(penalty='l1', solver='liblinear')

# Create a logistic regression model with L2 regularization
model_l2 = LogisticRegression(penalty='l2')
```

Optimization

```
import numpy as np
# Define the sigmoid function
def sigmoid(z):
    return 1   (1 + np.exp(-z))

# Define the cost function
def cost_function(X, y, theta):
    m = len(y)
    h = sigmoid(np.dot(X, theta))
    cost = (-1/m) * np.sum(y * np.log(h) + (1-y) * np.log(1-h))
    return cost

# Define the gradient descent function
def gradient_descent(X, y, theta, alpha, num_iterations):
    m = len(y)
    costs = []

    for i in range(num_iterations):
        h = sigmoid(np.dot(X, theta))
        gradient = np.dot(X.T, (h - y)) / m
        theta -= alpha * gradient
        cost   cost_function(X, y, theta)
        costs.append(cost)

    return theta, costs

# Add a column of ones to the feature matrix
X = np.column_stack((np.ones(len(X)), X))

# Initialize the parameters
theta = np.zeros(X.shape[1])
# Set the learning rate and number of iterations
alpha = 0.01
num_iterations = 1000

# Perform gradient descent optimization
theta_optimized, costs = gradient_descent(X, y, theta, alpha, num_iterations)
```

5.4 Definitions – Time Series Modeling (ARIMA)

Handling DateTime Data

ARIMA (AutoRegressive Integrated Moving Average) is a time series modeling technique that uses the following equation:

$y(t) = c + \phi_1 y(t-1) + \phi_2 y(t-2) + \ldots + \phi_p y(t-p) + \theta_1 \varepsilon(t-1) + \theta_2 \varepsilon(t-2) + \ldots + \theta_q \varepsilon(t-q) + \varepsilon(t)$

where:
- y(t) represents the value of the time series at time 't'.
- c is the constant term or the intercept.
- $\phi_1, \phi_2, \ldots, \phi_p$ are the autoregressive (AR) coefficients.
- y(t-1), y(t-2), ..., y(t-p) are the lagged values of the time series.
- $\theta_1, \theta_2, \ldots, \theta_q$ are the moving average (MA) coefficients.
- ε(t-1), ε(t-2), ..., ε(t-q) are the residual errors or the noise terms.
- ε(t) is the current residual error or noise term.

The equation combines the autoregressive (AR) terms, moving average (MA) terms, and the constant term to model the dependencies and patterns in the time series. The parameters ϕ and θ are estimated from the historical data using techniques like maximum likelihood estimation.

5.4.1 Code Snippets – Time Series Modeling

Time Series Modeling

```
import numpy as np
import matplotlib.pyplot as plt
from statsmodels.tsa.arima.model import ARIMA

# Generate synthetic time series data
np.random.seed(42)
data = np.cumsum(np.random.normal(size=100))
# Fit an ARIMA model
model = ARIMA(data, order=(1, 1, 1))
result = model.fit()
# Forecast future values
forecast_steps = 10
forecast, stderr, conf_int = result.forecast(steps=forecast_steps)

# Plot the original data and the forecast
plt.plot(data, label='Original Data', color='blue')
plt.plot(np.arange(len(data), len(data) + forecast_steps), forecast, label='Forecast', color='red')
plt.fill_between(np.arange(len(data), len(data) + forecast_steps), conf_int[:, 0], conf_int[:, 1],
color='pink', alpha=0.3)
plt.xlabel('Time')
plt.ylabel('Value')
plt.title('ARIMA Forecast')
plt.legend()
plt.show()
```

Grid Search for Hyperparameter Tuning

```
import itertools
import pandas as pd
import numpy as np
from statsmodels.tsa.arima.model import ARIMA

# Define the range of values for p, d, and q
p_values = range(0, 3)  # Example: 0, 1, 2
d_values = range(0, 3)
q_values = range(0, 3)

# Create a list of all possible combinations of p, d, and q values
param_combinations = list(itertools.product(p_values, d_values, q_values))

# Load the time series data
data = pd.read_csv('time_series_data.csv')
# Iterate over all parameter combinations
best_aic = float('inf')
best_params = None

for params in param_combinations:
    try:
        # Fit the ARIMA model
        model = ARIMA(data, order=params)
        results = model.fit()

        # Calculate AIC (Akaike Information Criterion)
        aic = results.aic

        # Update the best parameters if AIC is lower
        if aic < best_aic:
            best_aic = aic
            best_params = params

    except:
        continue

# Print the best parameters
print("Best Parameters: ", best_params)
```

Rolling Window Cross-Validation

```
import pandas as pd
from statsmodels.tsa.arima.model import ARIMA
from sklearn.metrics import mean_squared_error

# Load the time series data
data = pd.read_csv('time_series_data.csv')

# Split the data into train and test sets
train_size = int(len(data) * 0.8)
train_data, test_data = data[:train_size], data[train_size:]

# Fit the ARIMA model on the training data
model = ARIMA(train_data, order=(p, d, q))
results = model.fit()

# Perform rolling window cross-validation
window_size = 7
mse_scores = []

for i in range(len(test_data) - window_size + 1):
    window = test_data[i:i+window_size]

    # Make predictions on the window
    predictions = results.predict(start=window.index[0], end=window.index[-1])

    # Calculate the mean squared error
    mse = mean_squared_error(window, predictions)
    mse_scores.append(mse)

# Calculate the average mean squared error
avg_mse = np.mean(mse_scores)
print("Average MSE: ", avg_mse)
```

Model Selection using Information Criteria

```
import pandas as pd
import numpy as np
from statsmodels.tsa.arima.model import ARIMA

# Load the time series data
data = pd.read_csv('time_series_data.csv')

# Split the data into train and validation sets
train_size = int(len(data) * 0.8)
train_data, val_data = data[:train_size], data[train_size:]

# Define the range of values for p, d, and q
p_values = range(0, 3)  # Example: 0, 1, 2
d_values = range(0, 3)
q_values = range(0, 3)

best_aic = float('inf')
best_params = None

# Iterate over all parameter combinations
for p in p_values:
    for d in d_values:
        for q in q_values:
            try:
                # Fit the ARIMA model
                model = ARIMA(train_data, order=(p, d, q))
                results = model.fit()

                # Calculate AIC (Akaike Information Criterion)
                aic = results.aic

                # Update the best parameters if AIC is lower
                if aic < best_aic:
                    best_aic = aic
                    best_params = (p, d, q)

            except:
                continue

# Fit the ARIMA model on the combined train and validation data using the best parameters
model = ARIMA(data, order=best_params)
results = model.fit()

# Forecast future values
forecast_steps = 10
forecast, stderr, conf_int = results.forecast(steps=forecast_steps)

# Print the best parameters
print("Best Parameters: ", best_params)
```

5.5 Definitions – Decision Tree

Decision Tree

The decision tree algorithm uses a flowchart-like structure to make decisions or predict values based on input features. The decision-making process involves splitting the data at internal nodes based on different features, and assigning outcomes or predicted values at the leaf nodes. The algorithm uses specific criteria to determine the best feature and the best split at each node.

For classification tasks, the decision tree algorithm uses criteria such as Gini impurity or entropy to measure the purity of each split and select the feature that provides the most information gain.

For regression tasks, the decision tree algorithm uses criteria such as variance reduction or sum of squared errors to measure the quality of each split and choose the feature that minimizes the variance or the error.

The final decision tree structure is created through a recursive process of selecting the best features and splits until a stopping criterion is met, such as reaching a maximum depth or having a minimum number of instances in each leaf.

5.5.1 Code Snippets – Decision Tree

Decision Tree

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier, plot_tree
# Load the Iris dataset
data = load_iris()
X, y = data.data, data.target
# Create a decision tree classifier
model = DecisionTreeClassifier()
model.fit(X, y)
# Visualize the decision tree
plt.figure(figsize=(10, 8))
plot_tree(model, feature_names=data.feature_names, class_names=data.target_names, filled=True)
plt.show()
```

5.6 Definitions – K-Means Clustering

K-Means Clustering

K-Means Clustering is an Unsupervised Learning algorithm, which groups the unlabeled dataset into different clusters. Here K defines the number of pre-defined clusters that need to be created in the process, as if K=2, there will be two clusters, and for K=3, there will be three clusters, and so on. The goal of K-means clustering is to divide the data points into K clusters in such a way that the within-cluster variation is minimized and the between-cluster variation is maximized.

It allows us to cluster the data into different groups and a convenient way to discover the categories of groups in the unlabeled dataset on its own without the need for any training. It is a centroid-based algorithm, where each cluster is associated with a centroid.
The main aim of this algorithm is to minimize the sum of distances between the data point and their corresponding clusters.

5.6.1 Code Snippets – K-Means Clustering

K-Means Clustering

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
from sklearn.cluster import KMeans

# Generate synthetic data for clustering
X, _ = make_blobs(n_samples=300, centers=4, random_state=42)
# Create a KMeans clustering model
model = KMeans(n_clusters=4)
model.fit(X)
# Visualize the clustered data
plt.scatter(X[:, 0], X[:, 1], c=model.labels_, cmap='viridis', s=50, alpha=0.7)
plt.scatter(model.cluster_centers_[:, 0], model.cluster_centers_[:, 1], c='red', marker='x', s=100)
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.title('K-Means Clustering')
plt.show()
```

5.7 Definitions – Principal Component Analysis(PCA)

PCA

PCA, or Principal Component Analysis, is a technique used to reduce the dimensionality of a dataset while retaining most of the information. It achieves this by transforming the original features into a new set of uncorrelated variables called principal components. These components are ranked based on the amount of variance they explain in the data. By selecting the top components, PCA allows for data visualization, noise reduction, and improved computational efficiency in machine learning algorithms.

PCA (Principal Component Analysis):

- Dimensionality Reduction Technique
- Capturing Data Variance
- Utilizing Eigenvectors for Transformation

5.7.1 Code Snippets – Linear Regression

Linear Regression

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.decomposition import PCA

# Load the Iris dataset
data = load_iris()
X, y = data.data, data.target

# Apply PCA for dimensionality reduction to 2 components
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)

# Visualize the reduced data
plt.scatter(X_pca[:, 0], X_pca[:, 1], c=y, cmap='plasma')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.title('PCA - Dimensionality Reduction')
plt.show()
```

5.8 Definitions – Support Vector Machines(SVM)

Upport Vector Machine

Support Vector Machine (SVM) is a supervised machine learning algorithm used for classification and regression tasks. It is particularly effective in solving complex classification problems with a clear margin of separation between classes.

For SVM classification, the decision boundary equation for a linearly separable case can be expressed as:
w^T x + b = 0
where:
- w is the weight vector perpendicular to the decision boundary
- x is the input feature vector
- b is the bias term

The predicted class label for a new sample x can be determined by evaluating the sign of the decision function:
f(x) = sign(w^T x + b)
For non-linear SVM classification, the decision boundary equation is transformed using a kernel function, such as the radial basis function (RBF) kernel:
f(x) = sign(sum(alpha_i * y_i * K(x_i, x) + b))
where:
- alpha_i are the learned Lagrange multipliers (support vectors)
- y_i are the corresponding class labels of the support vectors
- K(x_i, x) is the kernel function that computes the similarity between the support vectors and the new sample x
- b is the bias term

5.8.1 Code Snippets – Support Vector Machines(SVM)

SVM

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_classification
from sklearn.svm import SVC
# Generate synthetic data for binary classification
X, y = make_classification(n_samples=100, n_features=2, n_informative=2, n_redundant=0, random_state=42)
# Create a Support Vector Machine (SVM) classifier
model = SVC(kernel='linear')
model.fit(X, y)
# Plot the data and the decision boundary
plt.scatter(X[:, 0], X[:, 1], c=y, cmap='coolwarm')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
# Plot the decision boundary
ax = plt.gca()
xlim = ax.get_xlim()
ylim = ax.get_ylim()
xx = np.linspace(xlim[0], xlim[1], 100)
yy = np.linspace(ylim[0], ylim[1], 100)
XX, YY = np.meshgrid(xx, yy)
Z = model.decision_function(np.c_[XX.ravel(), YY.ravel()])
Z = Z.reshape(XX.shape)
plt.contour(XX, YY, Z, colors='k', levels=[-1, 0, 1], linestyles=['--', '-', '--'])
plt.title('Support Vector Machine (SVM)')
plt.show()
```

5.9 Definitions – Naive Bayes Classifier

Naive Bayes Classifier

Naive Bayes is a probabilistic machine learning algorithm that is based on Bayes' theorem with the assumption of independence between features. It is commonly used for classification tasks and is particularly suited for text classification and spam filtering.
Bayes theorem is a formula that offers a conditional probability of an event A taking happening given anotheverent B has previously happened. Its mathematical formula is as follows: –
Where,
P(y|A, B) = (P(A, B|y) * P(y)) / P(A, B)

A and B are two events
P(A|B) is the probability of event A provided event B has already happened.
P(B|A) is the probability of event B provided event A has already happened.
P(A) is the independent probability of A
P(B) is the independent probability of B

5.9.1 Code Snippets – Naive Bayes Classifier

Naive Bayes Classifier

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.naive_bayes import GaussianNB
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Load the Iris dataset
data = load_iris()
X, y = data.data, data.target
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Create a Gaussian Naive Bayes classifier
model = GaussianNB()
model.fit(X_train, y_train)

# Make predictions on the test data
y_pred = model.predict(X_test)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
```

5.10 Definitions – Random Forest Classifier

Random Forest Classifier

Random Forest Classifier is an ensemble learning method that combines multiple decision trees to create a powerful classification model. It is a type of supervised learning algorithm that is widely used for both classification and regression tasks.

In Random Forest, a collection of decision trees is created, where each tree is trained on a random subset of the training data and a random subset of input features. During the training process, each tree independently makes predictions, and the final prediction is determined by combining the predictions from all the trees through voting (for classification) or averaging (for regression).

Random Forest works in two-phase first is to create the random forest by combining N decision tree, and second is to make predictions for each tree created in the first phase.

The Working process can be explained in the below steps and diagram:
Step-1: Select random K data points from the training set.
Step-2: Build the decision trees associated with the selected data points (Subsets).
Step-3: Choose the number N for decision trees that you want to build.
Step-4: Repeat Step 1 & 2.
Step-5: For new data points, find the predictions of each decision tree, and assign the new data points to the category that wins the majority votes

5.10.1 Code Snippets – Random Forest Classifier

Random Forest Classifier

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_classification
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Generate synthetic data for binary classification
X, y = make_classification(n_samples=100, n_features=2, n_informative=2, n_redundant=0, random_state=42)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Create a Random Forest classifier
model = RandomForestClassifier(n_estimators=100, random_state=42)
model.fit(X_train, y_train)

# Make predictions on the test data
y_pred = model.predict(X_test)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
```

5.11 Definitions – Poission Regression

Poission Regression

Poisson Regression is a statistical model used for analyzing count data. It models the relationship between predictor variables and the expected counts of an event. It assumes that the response variable follows a Poisson distribution.

The equation for Poisson Regression can be represented as:

$\log(E(Y)) = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \ldots + \beta_p X_p$

Where:
- log(E(Y)) is the natural logarithm of the expected count of the event (response variable Y).
- $\beta_0, \beta_1, \beta_2, \ldots, \beta_p$ are the regression coefficients associated with the predictor variables $X_1, X_2, \ldots, X_p$.
- $X_1, X_2, \ldots, X_p$ are the predictor variables.

The model assumes that the log of the expected count is a linear combination of the predictor variables, which allows for modeling the relationship between the predictors and the event count. The coefficients $\beta_0, \beta_1, \beta_2, \ldots, \beta_p$ represent the effects of the predictor variables on the log of the expected count.

5.11.1 Code Snippets – Poission Regression

Poission Regression

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import poisson
import statsmodels.api as sm

# Generate synthetic count data following a Poisson distribution
np.random.seed(42)
data = np.random.poisson(lam=3, size=100)
# Fit Poisson regression
X = np.arange(1, 101)
X = sm.add_constant(X)
poisson_model = sm.GLM(data, X, family=sm.families.Poisson())
result = poisson_model.fit()

# Print summary of the Poisson regression
print(result.summary())
```

5.12 Definitions – Gradient Boosting

Gradient Boosting is a machine learning technique that combines multiple weak prediction models to create a strong predictive model. It sequentially builds models, focusing on the errors of the previous models to improve predictions.

5.12.1 Code Snippets – Gradient Boosting

Gradient Boosting

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_boston
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

# Load the Boston Housing dataset
data = load_boston()
X, y = data.data, data.target

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Create a Gradient Boosting regression model
model = GradientBoostingRegressor(n_estimators=100, learning_rate=0.1, random_state=42)
model.fit(X_train, y_train)
# Make predictions on the test data
y_pred = model.predict(X_test)
# Calculate Mean Squared Error (MSE)
mse = mean_squared_error(y_test, y_pred)
print("Mean Squared Error:", mse)
```

5.13 Definitions – K-Nearest Neighbors(KNN)

K-Nearest Neighbors

The K-Nearest Neighbors (KNN) algorithm is a non-parametric and supervised machine learning algorithm. It is based on the principle that objects that are close to each other in a multi-dimensional space tend to belong to the same class or category. KNN is a lazy learning algorithm, meaning that it does not explicitly learn a model during the training phase. Instead, it stores the entire training dataset in memory to make predictions

d(x, y) = sqrt((x1 - y1)^2 + (x2 - y2)^2 + ... + (xn - yn)^2)

where d(x, y) is the distance between data points x and y, and (x1, x2, ..., xn) and (y1, y2, ..., yn) are the feature values of x and y, respectively.

In KNN, the equation to compute the distance between two data points x and y can be calculated using various distance metrics such as Euclidean distance, Manhattan distance, or Minkowski distance.

5.13.1 Code Snippets – K-Nearest Neighbors(KNN)

K-Nearest Neighbors

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Load the iris dataset
data = load_iris()
X, y = data.data, data.target

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Create a K-Nearest Neighbors classifier with k=3
model = KNeighborsClassifier(n_neighbors=3)
model.fit(X_train, y_train)

# Make predictions on the test data
y_pred = model.predict(X_test)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
```

5.14 Definitions – Lasso Regression (L1 Regularization)

Lasso Regression

Lasso Regression, also known as L1 regularization, is a linear regression technique that introduces a penalty term to the loss function to promote sparse solutions. It adds the L1 norm of the coefficient vector to the ordinary least squares (OLS) cost function. The L1 norm encourages the model to set some coefficients to exactly zero, effectively performing feature selection.

The equation for Lasso Regression can be expressed as:

Loss Function = OLS Loss + alpha * L1 Norm

where:

OLS Loss: The ordinary least squares loss function, which minimizes the sum of squared differences between the predicted and actual values.
alpha: The regularization parameter that controls the strength of the penalty term. Higher values of alpha result in stronger regularization.
L1 Norm: The sum of the absolute values of the coefficients, multiplied by alpha.

5.14.1 Code Snippets – Lasso Regression (L1 Regularization)

Lasso Regression

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_diabetes
from sklearn.linear_model import Lasso
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

# Load the diabetes dataset
data = load_diabetes()
X, y = data.data, data.target

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Create a Lasso regression model
alpha = 1.0  # Regularization strength (increase for stronger regularization)
model = Lasso(alpha=alpha)
model.fit(X_train, y_train)

# Make predictions on the test data
y_pred = model.predict(X_test)

# Calculate Mean Squared Error (MSE)
mse = mean_squared_error(y_test, y_pred)
print("Mean Squared Error:", mse)
```

5.15 Definitions – Gaussian Mixture Model(GMM)Clustring

Gaussian Mixture Model

A Gaussian Mixture Model (GMM) is a probabilistic model used for clustering and density estimation. It assumes that the data points are generated from a mixture of Gaussian distributions. Each data point is associated with a particular Gaussian component, and the goal is to estimate the parameters of the Gaussian components and assign data points to their most likely components.

The equation for a Gaussian Mixture Model can be expressed as:

P(X) = Σ [π_k * N(X | μ_k, Σ_k)]

where:

P(X): The probability density function (PDF) of the data point X.
Σ: The summation over all Gaussian components in the mixture.
π_k: The mixing coefficient or weight of the k-th Gaussian component, representing the probability of selecting that component.
N(X | μ_k, Σ_k): The Gaussian distribution with mean μ_k and covariance matrix Σ_k, which describes the density of the data points belonging to the k-th component

5.15.1 Code Snippets – Gaussian Mixture Model

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
from sklearn.mixture import GaussianMixture

# Generate synthetic data for clustering
X, _ = make_blobs(n_samples=300, centers=4, random_state=42)

# Create a Gaussian Mixture Model (GMM) clustering model
model = GaussianMixture(n_components=4)
model.fit(X)

# Predict the cluster assignments for each data point
y_pred = model.predict(X)

# Plot the clustered data
plt.scatter(X[:, 0], X[:, 1], c=y_pred, cmap='viridis', s=50, alpha=0.7)
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.title('Gaussian Mixture Model (GMM) Clustering')
plt.show()
```

5.16 Definitions – Factor Analysis

Factor Analysis is a statistical method that identifies underlying latent factors in observed variables, reducing data complexity and revealing patterns of correlation to understand the underlying structure.

5.16.1 Code Snippets – Factor Analysis

Factor Analysis

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_multilabel_classification
from sklearn.decomposition import FactorAnalysis

# Generate synthetic multilabel data
X, _ = make_multilabel_classification(n_samples=100, n_features=5, n_classes=3, random_state=42)
# Apply factor analysis for dimensionality reduction to 2 components
n_components = 2
model = FactorAnalysis(n_components=n_components)
X_fa = model.fit_transform(X)

# Visualize the reduced data
plt.scatter(X_fa[:, 0], X_fa[:, 1])
plt.xlabel('Factor 1')
plt.ylabel('Factor 2')
plt.title('Factor Analysis - Dimensionality Reduction')
plt.show()
```

6 Data Preparation

6.1 Definitions – Machine Learning

Machine learning is a subset of artificial intelligence where algorithms and statistical models enable computers to learn from data, improve performance, and make predictions or decisions without explicit programming.

6.2 Machine Learning Models and Techniques

Models & Techniques

1. **Linear Regression:**
   Linear model for predicting continuous values based on input features.
2. **K-Nearest Neighbors (KNN) Classifier:**
   Non-parametric classification method based on similarity to k-nearest data points.
3. **Decision Tree Classifier:**
   Tree-based classifier that partitions data based on feature thresholds.
4. **Random Forest Classifier:**
   Ensemble of decision trees for improved classification performance.
5. **Support Vector Machine (SVM) Classifier:**
   Linear or nonlinear classifier aiming to find the optimal hyperplane.
6. **Logistic Regression Classifier:**
   Classification model using logistic function for binary outcomes.
7. **Gradient Boosting Classifier:**
   Ensemble learning with sequentially improved weak learners.
8. **Naive Bayes Classifier:**
   Probabilistic classifier based on Bayes' theorem and strong independence assumptions.
9. **Principal Component Analysis (PCA):**
   Dimensionality reduction technique to transform data into orthogonal components.
10. **K-Means Clustering:**
    Unsupervised clustering algorithm to partition data into k clusters.
11. **Decision Tree Regressor:**
    Tree-based model for predicting continuous values based on feature splits.
12. **Random Forest Regressor:**
    Ensemble of decision trees for improved regression performance.
13. **Support Vector Machine (SVM) Regressor:**
    Regression model aiming to find the optimal hyperplane.
14. **radient Boosting Regressor:**
    Ensemble learning with sequentially improved weak regressors.
15. **K-Means Clustering:**
    Unsupervised clustering algorithm to partition data into k clusters.
16. **Latent Dirichlet Allocation (LDA):**
    Unsupervised topic modeling algorithm to identify hidden topics in text data.
17. **Support Vector Machine (SVM) One-Class Classification:**
    Unsupervised learning for detecting anomalies in data.

6.2.1 Code Snippets – Linear Regression

Linear Regression

```
from sklearn.linear_model import LinearRegression

# Create a linear regression model
model = LinearRegression()
```

(KNN) K-Nearest Neighbors

```
from sklearn.neighbors import KNeighborsClassifier

# Create a K-Nearest Neighbors classifier
model = KNeighborsClassifier(n_neighbors=3)
```

Decision Tree Classifier

```
from sklearn.tree import DecisionTreeClassifier

# Create a decision tree classifier
model = DecisionTreeClassifier()
```

Random Forest classifier

```
from sklearn.ensemble import RandomForestClassifier

# Create a Random Forest classifier
model = RandomForestClassifier(n_estimators=100)
```

Support Vector Machine

```
from sklearn.svm import SVC

# Create a Support Vector Machine (SVM) classifier
model = SVC(kernel='linear')
```

Logistic Regression

```
from sklearn.linear_model import LogisticRegression

# Create a Logistic Regression classifier
model = LogisticRegression()
```

Gradient Boosting Classifier

```
from sklearn.ensemble import GradientBoostingClassifier

# Create a Gradient Boosting classifier
model = GradientBoostingClassifier(n_estimators=100)
```

Naïve Bayes

```
from sklearn.naive_bayes import GaussianNB
# Create a Gaussian Naive Bayes classifier
model = GaussianNB()
```

Principal Component Analysis (PCA)

```
from sklearn.decomposition import PCA

# Apply PCA for dimensionality reduction to 2 components
pca = PCA(n_components 2)
```

K-Means Clustering

```
from sklearn.cluster import KMeans

# Create a KMeans clustering model
model = KMeans(n_clusters=4)
```

Decision Tree Regressor

```
from sklearn.tree import DecisionTreeRegressor

# Create a decision tree regressor
model = DecisionTreeRegressor()
```

Random Forest Regressor

```
from sklearn.linear_model import LogisticRegression
# Create a Logistic Regression classifier
model = LogisticRegression()
```

Support Vector Machine (SVM) Regressor

```
from sklearn.svm import SVR
# Create a Support Vector Machine (SVM) regressor
model = SVR(kernel='linear')
```

Gradient Boosting Regressor

```
from sklearn.ensemble import GradientBoostingRegressor
# Create a Gradient Boosting regressor
model = GradientBoostingRegressor(n_estimators 100)
```

Principal Component Analysis (PCA)

```
from sklearn.decomposition import PCA
# Apply PCA for dimensionality reduction to 2 components
pca = PCA(n_components=2)
```

Latent Dirichlet Allocation(LDA)

```
from sklearn.decomposition import LatentDirichletAllocation

# Apply LDA for topic modeling
lda = LatentDirichletAllocation(n_components=5)
```

3 Data Preparation

3.1 Definitions – Data Mining

Data mining is a broad field with various techniques and approaches, and it usually involves implementing complex algorithms. Below are code snippets for some common data mining tasks using Python and popular libraries:

2.7 Data Mining Techniques and Tasks

Techniques & Tasks

1. Clustering:
   Grouping similar data points into clusters based on their similarities.
2. Classification:
   Assigning data points to predefined classes or categories based on their features.
3. Association Rule Mining:
   Discovering interesting relationships between items in large datasets.
4. Anomaly Detection:
   Identifying rare or abnormal instances that deviate significantly from the norm.
5. Regression:
   Predicting continuous numeric values based on input features and their relationships.
6. Dimensionality Reduction:
   Reducing the number of features while preserving essential information.
7. Sentiment Analysis:
   Determining the sentiment or emotion expressed in text data, often for social media or customer feedback.

2.6 Code Snippets – Linear Regression

Clustering with K-Means

```
from sklearn.cluster import KMeans
# Create a K-Means clustering model
kmeans = KMeans(n_clusters=3)
```

Classification with Decision Tree

```
from sklearn.tree import DecisionTreeClassifier

# Create a Decision Tree classifier
model = DecisionTreeClassifier()
```

Association Rule Mining

```
from mlxtend.frequent_patterns import apriori
from mlxtend.frequent_patterns import association_rules

# Generate frequent itemsets using Apriori
frequent_itemsets = apriori(data, min_support 0.2, use_colnames True)

# Generate association rules
rules = association_rules(frequent_itemsets, metric='confidence', min_threshold=0.6)
```

Anomaly Detection with Isolation Forest

```
from sklearn.ensemble import IsolationForest

# Create an Isolation Forest model for anomaly detection
model = IsolationForest(contamination=0.1)
```

Regression with Linear Regression

```
from sklearn.linear_model import LinearRegression

# Create a Linear Regression model
model = LinearRegression()
```

Dimensionality Reduction with PCA

```
from sklearn.decomposition import PCA

# Apply PCA for dimensionality reduction
pca = PCA(n_components=2)
```

Sentiment Analysis with NPL

```
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.naive_bayes import MultinomialNB

# Create a CountVectorizer for text processing
vectorizer = CountVectorizer()

# Create a Naive Bayes classifier for sentiment analysis
classifier = MultinomialNB()
```

Time Series Forecasting (ARIMA MODEL)

```
from statsmodels.tsa.arima.model import ARIMA

# Create an ARIMA model for time series forecasting
model = ARIMA(data, order=(1, 1, 1))
```

Filtering with Matrix Factorization

```
from sklearn.decomposition import TruncatedSVD
# Apply TruncatedSVD for collaborative filtering
svd = TruncatedSVD(n_components=10)
```

Frequent Items Mining with FP-Growth

```
from mlxtend.frequent_patterns import fpgrowth
# Generate frequent itemsets using FP-Growth
frequent_itemsets = fpgrowth(data, min_support=0.2, use_colnames=True)
```

3 Data Preparation

3.1 Definitions – Data Visualization

Data visualization is the graphical representation of information and data, using charts, graphs, and plots to communicate patterns, trends, and insights for better understanding and decision-making.

2.7 Selecting the Right Visualization for Your Data

Concepts

1. Understand Data Type: Identify whether your data is categorical, numerical, or time-based, as different visualizations suit different data types.

2. Identify Purpose: Define the purpose of the visualization: comparison, distribution, relationship, or trend analysis.

3. Consider Data Size: For small datasets, use detailed plots like scatter plots; for larger datasets, use summary plots like histograms.

4. Chart Complexity: Choose simple charts for basic data representation, and use complex ones for in-depth analysis.

5. Data Dimensionality: Assess the number of variables/features; avoid clutter by using appropriate multi-dimensional plots.

6. Audience Consideration: Select visualizations that resonate with your target audience, ensuring easy understanding.

7. Emphasize Insights: Highlight the main insights by using the right visualization technique to avoid confusion.

8. Accuracy and Clarity: Opt for accurate and clear visualizations that do not distort the data.

9. Avoid Misleading Visuals: Be cautious with 3D charts, pie charts with too many categories, and axis truncation.

2.6 Code Snippets – Data Visualization Code

Basic Outline For Data Visualization

```
# Import required libraries
import matplotlib.pyplot as plt

# Load or generate data
# Choose the appropriate type of plot (line plot, bar chart, scatter plot, etc.)
# Customize the plot (labels, title, colors, etc.)
# Display the plot using plt.show()
```

Line Plot

```
import matplotlib.pyplot as plt

# Sample data
x = [1, 2, 3, 4, 5]
y = [10, 15, 7, 8, 12]

# Line plot
plt.plot(x, y)
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Line Plot')
plt.show()
```

Bar Chart

```
import matplotlib.pyplot as plt

# Sample data
categories = ['Category A', 'Category B', 'Category C']
values = [30, 45, 25]

# Bar chart
plt.bar(categories, values)
plt.xlabel('Categories')
plt.ylabel('Values')
plt.title('Bar Chart')
plt.show()
```

Scatter Plot

```
import matplotlib.pyplot as plt

# Sample data
labels   ['Category A', 'Category B', 'Category C']
sizes   [30, 45, 25]

# Pie chart
plt.pie(sizes, labels=labels, autopct='%1.1f%%')
plt.title('Pie Chart')
plt.show()
```

Pie Chart

```
import matplotlib.pyplot as plt

# Sample data
labels = ['Category A', 'Category B', 'Category C']
sizes = [30, 45, 25]

# Pie chart
plt.pie(sizes, labels labels, autopct '%1.1f%%')
plt.title('Pie Chart')
plt.show()
```

Histogram

```
import matplotlib.pyplot as plt

# Sample data
data = [10, 15, 20, 25, 30, 35, 40, 45]
# Histogram
plt.hist(data, bins=5)
plt.xlabel('Value')
plt.ylabel('Frequency')
plt.title('Histogram')
    plt.show()
```

Area Chart

```
import matplotlib.pyplot as plt

# Sample data
data   [10, 15, 20, 25, 30, 35, 40, 45]
# Box plot
plt.boxplot(data)
plt.ylabel('Value')
plt.title('Box Plot')
plt.show()
```

Box Plot

```
import matplotlib.pyplot as plt

# Sample data
data = [10, 15, 20, 25, 30, 35, 40, 45]
# Box plot
plt.boxplot(data)
plt.ylabel('Value')
plt.title('Box Plot')
plt.show()
```

Heatmap

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

# Sample data
data = np.random.rand(5, 5)
# Heatmap
plt.imshow(data, cmap='viridis', interpolation='nearest')
plt.colorbar()
plt.title('Heatmap')
plt.show()
```

3D Plot

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

# Sample data
x = np.linspace(-5, 5, 100)
y = np.linspace(-5, 5, 100)
X, Y = np.meshgrid(x, y)
Z   np.sin(np.sqrt(X**2 + Y**2))
# 3D Plot
fig   plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(X, Y, Z, cmap='viridis')
ax.set_xlabel('X-axis')
ax.set_ylabel('Y-axis')
ax.set_zlabel('Z-axis')
plt.title('3D Plot')
plt.show()
```

Violin Plot

```
import matplotlib.pyplot as plt
import seaborn as sns

# Sample data
data   sns.load_dataset('iris')
# Violin plot
sns.violinplot(x 'species', y 'sepal_length', data data)
plt.xlabel('Species')
plt.ylabel('Sepal Length')
plt.title('Violin Plot')
plt.show()
```

Stacked Bar Chart

```
import matplotlib.pyplot as plt

# Sample data
categories = ['Category A', 'Category B', 'Category C']
values1 = [30, 45, 25]
values2 = [15, 25, 35]
# Stacked bar chart
plt.bar(categories, values1, label='Value 1')
plt.bar(categories, values2, bottom=values1, label='Value 2')
plt.xlabel('Categories')
plt.ylabel('Values')
plt.title('Stacked Bar Chart')
plt.legend()
plt.show()
```

Joint Plot

```
import seaborn as sns

# Sample data
data = sns.load_dataset('tips')
# Joint plot
sns.jointplot(x='total_bill', y='tip', data=data, kind='scatter')
plt.xlabel('Total Bill')
plt.ylabel('Tip')
plt.title('Joint Plot')
plt.show()
```

Pair Plot

```
import seaborn as sns

# Sample data
data = sns.load_dataset('iris')
# Pair plot
sns.pairplot(data, hue='species')
plt.title('Pair Plot')
plt.show()
```

Word Cloud

```
from wordcloud import WordCloud

# Sample data
text = "Data science is amazing and full of possibilities."
# Word cloud
wordcloud = WordCloud().generate(text)
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis('off')
plt.title('Word Cloud')
plt.show()
```

Geographic Map

```
import geopandas as gpd

# Sample data - Shapefile
world = gpd.read_file(gpd.datasets.get_path('naturalearth_lowres'))
# Geographic map
world.plot()
plt.title('Geographic Map')
plt.show()
```

7 DSQL and DataBase Integration

7.1 Definitions – SQL and DataBase Integration

SQL and Database Integration: Relational databases are built using the structured query language (SQL) to create, store, update, and retrieve data.
Data integration is the process of combining data from multiple source systems to create unified sets of information for both operational and analytical uses.

7.2 Code Snippets – SQL and DataBase Integration

SQL Common Data Science Tasks

1 Connection to a Database:

```
-- Connect to a database
USE your_database_name;
```

2 Selecting Data from a Table:

```
-- Select all rows and columns from a table
SELECT * FROM your_table_name;

-- Select specific columns from a table
SELECT column1, column2 FROM your_table_name;
```

3 Filtering Data with WHERE Clause:

```
-- Select rows that meet a condition
SELECT * FROM your_table_name WHERE column1 = 'value';
-- Using logical operators
SELECT * FROM your_table_name WHERE column1 > 10 AND column2 = 'category';
```

4 Sorting Data with ORDER BY:

```
-- Sort data in ascending order
SELECT * FROM your_table_name ORDER BY column1;

-- Sort data in descending order
SELECT * FROM your_table_name ORDER BY column1 DESC;
```

5 Aggregating Data with GROUP BY:

```
-- Calculate aggregates for each group
SELECT column1, SUM(column2) FROM your_table_name GROUP BY column1;
```

6 Joining Tables:

```
-- Inner Join
SELECT * FROM table1 INNER JOIN table2 ON table1.column = table2.column;

-- Left Join
SELECT * FROM table1 LEFT JOIN table2 ON table1.column = table2.column;
```

7 Subqueries:

```
-- Use subquery in WHERE clause
SELECT * FROM your_table_name WHERE column1 IN (SELECT column2 FROM another_table);
```

8 Conditional Statements with CASE:

```
-- Perform conditional statements
SELECT column1,
    CASE
        WHEN column2 > 10 THEN 'High'
        WHEN column2 > 5 THEN 'Medium'
        ELSE 'Low'
    END AS 'Priority'
FROM your_table_name;
```

9 Data Cleaning:

```
-- Remove duplicates from a table
DELETE FROM your_table_name
WHERE column1 IN (SELECT column1 FROM your_table_name GROUP BY column1 HAVING COUNT(*) > 1);

-- Replace NULL values with a default value
UPDATE your_table_name SET column1 = 'default_value' WHERE column1 IS NULL;
```

10 Creating and Modifying Tables:

```
-- Create a new table
CREATE TABLE new_table (column1 datatype, column2 datatype, ...);

-- Add a new column to an existing table
ALTER TABLE your_table_name ADD new_column datatype;

-- Rename a column in a table
ALTER TABLE your_table_name RENAME COLUMN old_column TO new_column;
```

11 Working with Dates:

```
-- Extract year from a date column
SELECT DATEPART(YEAR, date_column) AS year FROM your_table_name;

-- Calculate the difference between two dates
SELECT DATEDIFF(day, date1, date2) AS days_difference FROM your_table_name;
```

12 Data Transformation:

```
-- Apply mathematical operations on columns
SELECT column1 + column2 AS sum_column FROM your_table_name;
-- Concatenate strings from multiple columns
SELECT CONCAT(column1, ' ', column2) AS full_name FROM your_table_name;
```

13 Conditional Aggregation:

```
-- Calculate aggregates with conditions
SELECT column1,
       SUM(CASE WHEN column2 > 10 THEN 1 ELSE 0 END) AS count_above_10,
       AVG(CASE WHEN column2 < 5 THEN column3 ELSE NULL END) AS avg_below_5
FROM your_table_name
GROUP BY column1;
```

14 Data Analysis Functions:

```
-- Calculate the maximum value of a column
SELECT MAX(column1) AS max_value FROM your_table_name;

-- Calculate the average value of a column
SELECT AVG(column1) AS average_value FROM your_table_name;

-- Calculate the total count of rows
SELECT COUNT(*) AS total_rows FROM your_table_name;
```

15 Window Functions:

```
-- Calculate the row number for each row
SELECT column1, column2, ROW_NUMBER() OVER (ORDER BY column1) AS row_number FROM your_table_name;

-- Calculate the average value of a column over a window of rows
SELECT column1, column2, AVG(column2) OVER (PARTITION BY column1) AS avg_per_column1 FROM your_table_name;
```

16 Conditional Filtering with HAVING:

```
-- Filter groups based on aggregate conditions
SELECT column1, COUNT(*) AS count FROM your_table_name GROUP BY column1 HAVING COUNT(*) > 5;
```

17 Pivoting Data with PIVOT:

```
-- Pivot data from rows to columns
SELECT * FROM your_table_name
PIVOT (SUM(column2) FOR column1 IN ('Value1', 'Value2', 'Value3')) AS pivoted_table;
```

18 Error Handling:

```
try:

# Execute SQL query or other database operation
    ...
except (sqlite3.Error, mysql.connector.Error, psycopg2.Error) as e:
    # Handle database error
    print("Database error:", e)
finally:
    # Close the cursor and connection
    cursor.close()
    conn.close()
```

SQL Common Data Science Tasks

19 Transactions:

```
# Begin a transaction
conn.begin()

# Execute multiple SQL queries within a transaction
query1 = "UPDATE table_name SET column1 = value1 WHERE condition"
query2 = "INSERT INTO table_name (column1, column2) VALUES (?, ?)"
params2 = ('value1', 'value2')
cursor.execute(query1)
cursor.execute(query2, params2)

# Commit the transaction
conn.commit()
```

20 Executing SQL Queries:

```
# Create a cursor object
cursor = conn.cursor()

# Execute a SQL query
query = "SELECT * FROM table_name"
cursor.execute(query)

# Fetch all the rows
rows = cursor.fetchall()

# Fetch a single row
row = cursor.fetchone()

# Commit the changes (for data modification queries)
conn.commit()

# Close the cursor and connection
cursor.close()
conn.close()
```

21 Writing Data ti a SQL DataBase:

```
# Create a DataFrame
df = pd.DataFrame(data)

# Write DataFrame to a SQL database
df.to_sql('table_name', conn, if_exists='replace')
```

22 Parameterized Queries:

```
# Execute a parameterized SQL query
query = "SELECT * FROM table_name WHERE column1 = ? AND column2 = ?"
params = ('value1', 'value2')
cursor.execute(query, params)
```

23 Executing Stored Procedures:

```
# Execute a stored procedure
query = "CALL stored_procedure_name(?, ?)"
params = ('value1', 'value2')
cursor.execute(query, params)
```

8 Data Model Deployment

8.1 Definitions – Model Deployment

Model deployment is the process of making a trained machine learning model operational and accessible for real-time predictions or decision-making in a production environment. It involves preparing the model, setting up the infrastructure, integrating it with the application, scaling and optimizing its performance, monitoring and maintaining its performance, handling versioning and updates, and considering security aspects.

8.2 Model Deployment

Concepts

1. Model Preparation: Finalize the trained machine learning model for deployment, including packaging it with necessary dependencies and libraries.

2. Infrastructure Setup: Configure the necessary infrastructure, such as servers or cloud resources, to host and make the model available for use.

3. Integration with Application: Connect the deployed model with the application or system that will utilize its predictions, ensuring proper input and output data flow.

4. Scaling and Optimization: Scale the deployed model to handle anticipated workload, optimize its performance, and implement techniques like load balancing or parallelization.

5. Monitoring and Maintenance: Continuously monitor the model's performance, track relevant metrics, and address any issues or errors that arise. Ensure the model remains accurate and reliable.

6. Versioning and Updates: Manage updates and improvements to the model, ensuring proper versioning to maintain consistency and avoid disruptions in production systems.

8.2.1 Code Snippets – Model Deployment

Model Deployment

1 Saving and Loading Models using Pickle:

```
import pickle
# Saving a trained model
with open('model.pkl', 'wb') as file:
    pickle.dump(model, file)

# Loading a saved model
with open('model.pkl', 'rb') as file:
    loaded_model = pickle.load(file)
```

2 Deploying a Model with Flask:

```
from flask import Flask, request, jsonify

app = Flask(__name__)
# Define a route for model prediction
@app.route('/predict', methods=['POST'])
def predict():
    # Get input data from the request
    data = request.json['data']

    # Preprocess the data
    processed_data = preprocess_data(data)

    # Make predictions using the loaded model
    predictions = loaded_model.predict(processed_data)

    # Return the predictions as JSON response
    response = {'predictions': predictions.tolist()}
    return jsonify(response)

# Run the Flask app
if __name__ == '__main__':
    app.run()
```

3 Deploying a Model with FastAPI:

```
from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()

# Define a request model
class PredictionRequest(BaseModel):
    data: list
# Define a response model
class PredictionResponse(BaseModel):
    predictions: list

# Define a route for model prediction
@app.post('/predict', response_model=PredictionResponse)
def predict(request: PredictionRequest):
    # Get input data from the request
    data = request.data

    # Preprocess the data
    processed_data = preprocess_data(data)

    # Make predictions using the loaded model
    predictions = loaded_model.predict(processed_data)

    # Return the predictions as a response
    response = PredictionResponse(predictions=predictions.tolist())
    return response
```

4 Containerization with Docker:

```
# Dockerfile
# Use an official Python runtime as the base image
FROM python:3.8-slim
# Set the working directory in the container
WORKDIR /app
# Copy the requirements file into the container
COPY requirements.txt .
# Install the Python dependencies
RUN pip install --no-cache-dir -r requirements.txt

# Copy the model file and code into the container
COPY model.pkl .
COPY app.py .

# Expose the port for the Flask app
EXPOSE 5000
# Set the command to run the Flask app
CMD ["python", "app.py"]
```