AI-ML Lab

Exercise 1: Familiarization with the Python Programming Language

a. Objective:

To gain familiarity with the Python programming language and its fundamental constructs.

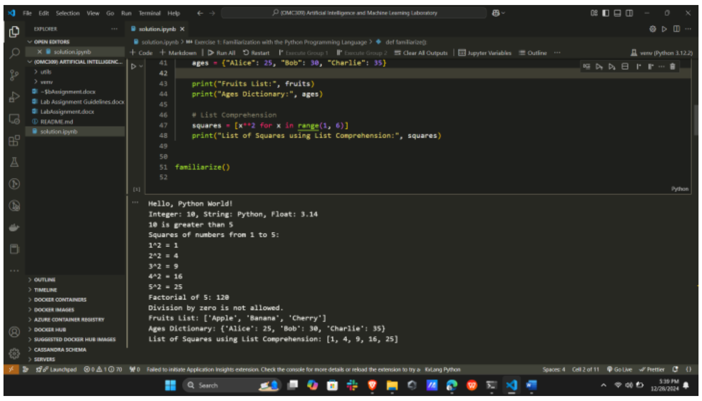
b. Brief Background:

Python is a high-level, interpreted language that emphasizes readability and simplicity. It supports multiple paradigms, including procedural, object-oriented, and functional programming. The exercise focuses on learning the basics of Python, including data types, loops, conditionals, and functions.

c. Python Program Code:

```python
def familiarize():
    # Basic Python Program

    # Print a message
    print("Hello, Python World!")

    # Variables and Data Types
    num = 10
    text = "Python"
    flt = 3.14
    print(f"Integer: {num}, String: {text}, Float: {flt}")

    # Control Structures
    # If-Else
    if num > 5:
        print(f"{num} is greater than 5")
    else:
        print(f"{num} is less than or equal to 5")

    # Loops
    print("Squares of numbers from 1 to 5:")
    for i in range(1, 6):
        print(f"{i}^2 = {i**2}")

    # Functions
    def factorial(n):
        if n == 0 or n == 1:
            return 1
        return n * factorial(n - 1)

    print(f"Factorial of 5: {factorial(5)}")

    # Exception Handling
    try:
        division = num / 0
    except ZeroDivisionError:
        print("Division by zero is not allowed.")

    # Lists and Dictionary
    fruits = ["Apple", "Banana", "Cherry"]
    ages = {"Alice": 25, "Bob": 30, "Charlie": 35}

    print("Fruits List:", fruits)
    print("Ages Dictionary:", ages)

    # List Comprehension
    squares = [x**2 for x in range(1, 6)]
    print("List of Squares using List Comprehension:", squares)


familiarize()
```

d. Sample Results:



```python
    ages = {"Alice": 25, "Bob": 30, "Charlie": 35}

    print("Fruits List:", fruits)
    print("Ages Dictionary:", ages)

    # List Comprehension
    squares = [x**2 for x in range(1, 6)]
    print("List of Squares using List Comprehension:", squares)


familiarize()
```

```
Hello, Python World!
Integer: 10, String: Python, Float: 3.14
10 is greater than 5
Squares of numbers from 1 to 5:
1^2 = 1
2^2 = 4
3^2 = 9
4^2 = 16
5^2 = 25
Factorial of 5: 120
Division by zero is not allowed.
Fruits List: ['Apple', 'Banana', 'Cherry']
Ages Dictionary: {'Alice': 25, 'Bob': 30, 'Charlie': 35}
List of Squares using List Comprehension: [1, 4, 9, 16, 25]
```

e. Conclusion:

This exercise provided a basic understanding of Python programming constructs, including variables, loops, conditionals, functions, and exception handling. It serves as a foundation for further exploration of Python's capabilities.

Exercise 2. Development of a Simple Calculator (Part A)

a. Objective:

To develop a Python program to implement a simple calculator.

b. Brief Background:

Calculators perform basic mathematical operations such as addition, subtraction, multiplication, and division. Implementing a simple calculator helps to understand input/output and basic arithmetic operations in Python.

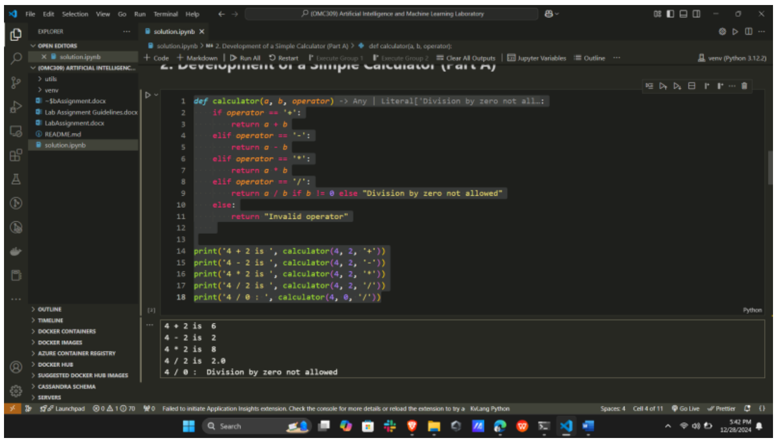
c. Algorithm Steps:

1. Accept user input for two numbers and an operator.
2. Perform the corresponding operation based on the input operator.
3. Display the result.

d. Python Program Code:

```python
def calculator(a, b, operator):
    if operator == '+':
        return a + b
    elif operator == '-':
        return a - b
    elif operator == '*':
        return a * b
    elif operator == '/':
        return a / b if b != 0 else "Division by zero not allowed"
    else:
        return "Invalid operator"


print('4 + 2 is ', calculator(4, 2, '+'))
print('4 - 2 is ', calculator(4, 2, '-'))
print('4 * 2 is ', calculator(4, 2, '*'))
print('4 / 2 is ', calculator(4, 2, '/'))
print('4 / 0 : ', calculator(4, 0, '/'))
```

e. Sample Results:



2. Development of a Simple Calculator (Part A)

```python
def calculator(a, b, operator) -> Any | Literal['Division by zero not all…:
    if operator == '+':
        return a + b
    elif operator == '-':
        return a - b
    elif operator == '*':
        return a * b
    elif operator == '/':
        return a / b if b != 0 else "Division by zero not allowed"
    else:
        return "Invalid operator"


print('4 + 2 is ', calculator(4, 2, '+'))
print('4 - 2 is ', calculator(4, 2, '-'))
print('4 * 2 is ', calculator(4, 2, '*'))
print('4 / 2 is ', calculator(4, 2, '/'))
print('4 / 0 : ', calculator(4, 0, '/'))
```

```
4 + 2 is  6
4 - 2 is  2
4 * 2 is  8
4 / 2 is  2.0
4 / 0 :  Division by zero not allowed
```

f. Conclusion:

The simple calculator successfully performed basic arithmetic operations.

Exercise 2B: Development of a Python Program to Perform Operations on String, Set, Tuple Data Types, and Bitwise Operations

a. Objective:

To understand and implement operations involving strings, sets, tuples, and bitwise operations.

b. Brief Background:

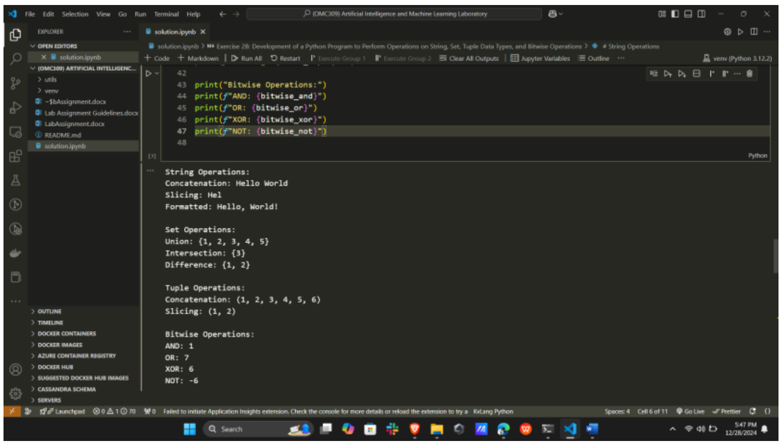
- **Strings**: Immutable sequences of characters with various operations such as concatenation, slicing, and formatting.
- **Sets**: Unordered collections of unique elements with operations like union, intersection, and difference.
- **Tuples**: Immutable ordered collections of elements.
- **Bitwise Operations**: Operations directly on binary representations of numbers, such as AND, OR, XOR, and NOT.

c. Python Program Code:

```python
# String Operations
string1 = "Hello"
string2 = "World"
concat = string1 + " " + string2
slice_str = string1[:3]
formatted = f"{string1}, {string2}!"

# Set Operations
set1 = {1, 2, 3}
set2 = {3, 4, 5}
union = set1 | set2
intersection = set1 & set2
difference = set1 - set2

# Tuple Operations
tuple1 = (1, 2, 3)
tuple2 = (4, 5, 6)
concat_tuple = tuple1 + tuple2
slice_tuple = tuple1[:2]

# Bitwise Operations
a, b = 5, 3  # Binary: 5 = 101, 3 = 011
bitwise_and = a & b  # 101 & 011 = 001
bitwise_or = a | b   # 101 | 011 = 111
bitwise_xor = a ^ b  # 101 ^ 011 = 110
bitwise_not = ~a     # ~101 = -110

# Output Results
```

```python
print("String Operations:")
print(f"Concatenation: {concat}")
print(f"Slicing: {slice_str}")
print(f"Formatted: {formatted}\n")

print("Set Operations:")
print(f"Union: {union}")
print(f"Intersection: {intersection}")
print(f"Difference: {difference}\n")

print("Tuple Operations:")
print(f"Concatenation: {concat_tuple}")
print(f"Slicing: {slice_tuple}\n")

print("Bitwise Operations:")
print(f"AND: {bitwise_and}")
print(f"OR: {bitwise_or}")
print(f"XOR: {bitwise_xor}")
print(f"NOT: {bitwise_not}")
```

e. Sample Results:



```python
print("Bitwise Operations:")
print(f"AND: {bitwise_and}")
print(f"OR: {bitwise_or}")
print(f"XOR: {bitwise_xor}")
print(f"NOT: {bitwise_not}")
```

```
String Operations:
Concatenation: Hello World
Slicing: Hel
Formatted: Hello, World!

Set Operations:
Union: {1, 2, 3, 4, 5}
Intersection: {3}
Difference: {1, 2}

Tuple Operations:
Concatenation: (1, 2, 3, 4, 5, 6)
Slicing: (1, 2)

Bitwise Operations:
AND: 1
OR: 7
XOR: 6
NOT: -6
```

f. Conclusion

Exercise 3. Development of Multidimensional Data Arrays (Part A)

a. Objective:

To create multidimensional arrays and perform operations on them without libraries like NumPy.

b. Brief Background:

Multidimensional arrays are data structures where data is organized in rows, columns, and beyond (e.g., 2D, 3D arrays). They are used for mathematical computations, data storage, and scientific analysis.

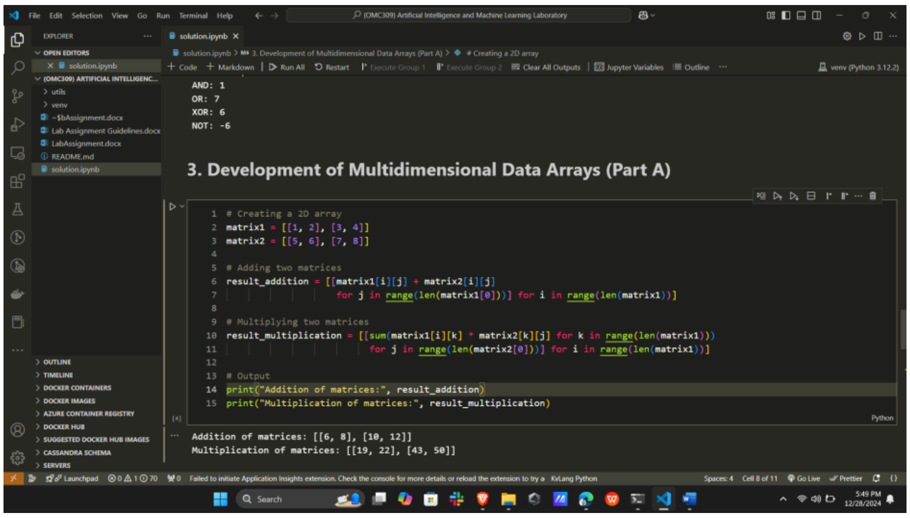
c. Algorithm Steps:

1. Create a 2D array manually using lists of lists.
2. Implement operations like addition and multiplication manually.

d. Python Program Code:

```
# Creating a 2D array
matrix1 = [[1, 2], [3, 4]]
matrix2 = [[5, 6], [7, 8]]

# Adding two matrices
result_addition = [[matrix1[i][j] + matrix2[i][j]
                    for j in range(len(matrix1[0]))] for i in
range(len(matrix1))]

# Multiplying two matrices
result_multiplication = [[sum(matrix1[i][k] * matrix2[k][j] for k in
range(len(matrix1)))
                          for j in range(len(matrix2[0]))] for i in
range(len(matrix1))]

# Output
print("Addition of matrices:", result_addition)
print("Multiplication of matrices:", result_multiplication)
```

e. Sample Results:



```
AND: 1
OR: 7
XOR: 6
NOT: -6
```

3. Development of Multidimensional Data Arrays (Part A)

```python
# Creating a 2D array
matrix1 = [[1, 2], [3, 4]]
matrix2 = [[5, 6], [7, 8]]

# Adding two matrices
result_addition = [[matrix1[i][j] + matrix2[i][j]
                    for j in range(len(matrix1[0]))] for i in range(len(matrix1))]

# Multiplying two matrices
result_multiplication = [[sum(matrix1[i][k] * matrix2[k][j] for k in range(len(matrix1)))
                          for j in range(len(matrix2[0]))] for i in range(len(matrix1))]

# Output
print("Addition of matrices:", result_addition)
print("Multiplication of matrices:", result_multiplication)
```

```
Addition of matrices: [[6, 8], [10, 12]]
Multiplication of matrices: [[19, 22], [43, 50]]
```

f. Conclusion:

The exercise demonstrated manual creation and manipulation of multidimensional arrays.

Exercise 3B: Development of a Python Program to Generate Data Visualizations

a. Objective:

To create data visualizations such as line plots, scatter plots, bar graphs, and histograms.

b. Brief Background:

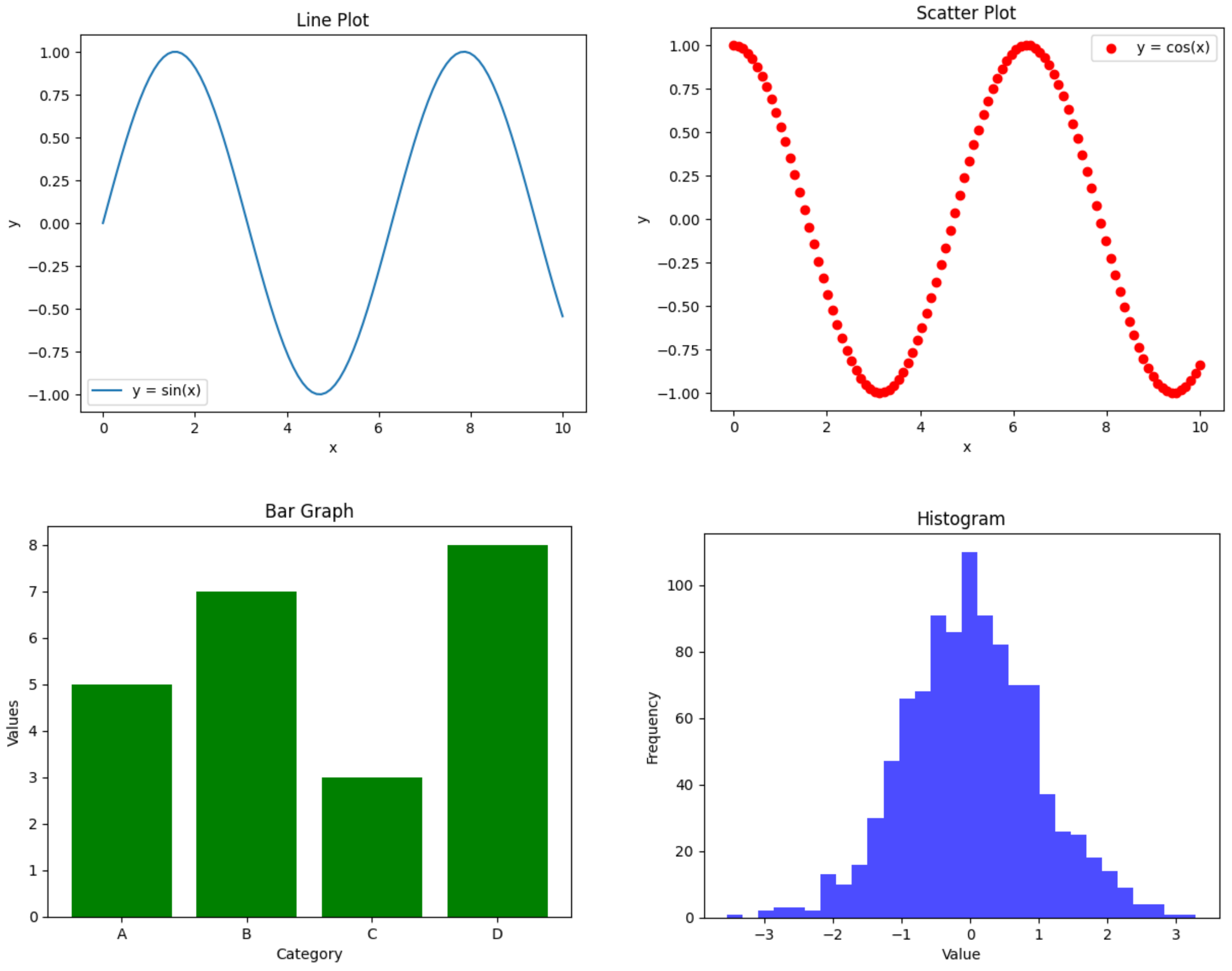
Data visualization helps interpret and analyze data effectively. Libraries like Matplotlib and Seaborn allow creating various plots to understand data distribution, trends, and patterns.

c. Python Program Code:

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

# Generate Data
x = np.linspace(0, 10, 100)
y = np.sin(x)
categories = ['A', 'B', 'C', 'D']
values = [5, 7, 3, 8]
data = np.random.randn(1000)

# Line Plot
plt.figure()
plt.plot(x, y, label="y = sin(x)")
plt.title("Line Plot")
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
plt.show()

# Scatter Plot
plt.figure()
plt.scatter(x, np.cos(x), c='r', label="y = cos(x)")
plt.title("Scatter Plot")
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
plt.show()

# Bar Graph
plt.figure()
plt.bar(categories, values, color='g')
plt.title("Bar Graph")
plt.xlabel("Category")
plt.ylabel("Values")
plt.show()
```

```python
# Histogram
plt.figure()
plt.hist(data, bins=30, alpha=0.7, color='b')
plt.title("Histogram")
plt.xlabel("Value")
plt.ylabel("Frequency")
plt.show()
```

e. Sample Results:



f. Conclusion:

Exercise 3C: Development of a Python Program to Perform Basic Scientific Operations Using the SciPy Library

a. Objective:

To perform scientific operations such as optimization, integration, and solving equations using the SciPy library.

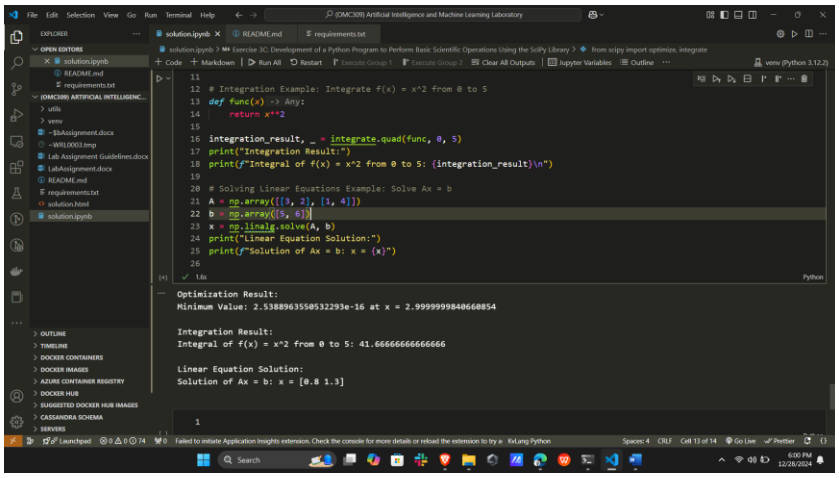
b. Brief Background:

SciPy is a Python library for scientific and technical computing. It builds on NumPy and provides modules for optimization, integration, linear algebra, and more.

c. Python Program Code:

```python
from scipy import optimize, integrate
import numpy as np

# Optimization Example: Minimize the quadratic function f(x) = (x-3)^2
def quadratic(x):
    return (x - 3)**2

result = optimize.minimize(quadratic, x0=0)
print("Optimization Result:")
print(f"Minimum Value: {result.fun} at x = {result.x[0]}\n")

# Integration Example: Integrate f(x) = x^2 from 0 to 5
def func(x):
    return x**2

integration_result, _ = integrate.quad(func, 0, 5)
print("Integration Result:")
print(f"Integral of f(x) = x^2 from 0 to 5: {integration_result}\n")

# Solving Linear Equations Example: Solve Ax = b
A = np.array([[3, 2], [1, 4]])
b = np.array([5, 6])
x = np.linalg.solve(A, b)
print("Linear Equation Solution:")
print(f"Solution of Ax = b: x = {x}")
```

e. Sample Results:



```python

# Integration Example: Integrate f(x) = x^2 from 0 to 5
def func(x) -> Any:
    return x**2

integration_result, _ = integrate.quad(func, 0, 5)
print("Integration Result:")
print(f"Integral of f(x) = x^2 from 0 to 5: {integration_result}\n")

# Solving Linear Equations Example: Solve Ax = b
A = np.array([[3, 2], [1, 4]])
b = np.array([5, 6])
x = np.linalg.solve(A, b)
print("Linear Equation Solution:")
print(f"Solution of Ax = b: x = {x}")

```

```
Optimization Result:
Minimum Value: 2.5388963550532293e-16 at x = 2.9999999840660854

Integration Result:
Integral of f(x) = x^2 from 0 to 5: 41.66666666666666

Linear Equation Solution:
Solution of Ax = b: x = [0.8 1.3]
```

f. Conclusion:

4A. Python Implementation of the DFS Algorithm

a. Objective:

To implement the Depth-First Search (DFS) algorithm for graph traversal.

b. Brief Background:

Depth-First Search (DFS) is an algorithm used to explore all vertices and edges in a graph systematically. It starts from a given source node and explores as far as possible along each branch before backtracking. DFS can be implemented using recursion or an explicit stack.

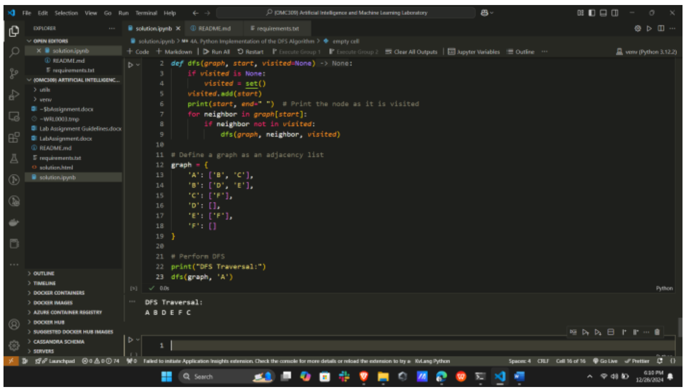
c. Algorithm Steps:

1. Represent the graph using an adjacency list.
2. Create a function for the DFS traversal.
3. Use a visited list to keep track of visited nodes.
4. Start from the given node and explore each unvisited neighbor recursively.
5. Print the visited nodes in the order of traversal.

d. Python Program Code:

```python
# DFS Implementation
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()
    visited.add(start)
    print(start, end=" ")  # Print the node as it is visited
    for neighbor in graph[start]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited)

# Define a graph as an adjacency list
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
    'F': []
}

# Perform DFS
print("DFS Traversal:")
dfs(graph, 'A')
```

e. Sample Results:



```python
def dfs(graph, start, visited=None) -> None:
    if visited is None:
        visited = set()
    visited.add(start)
    print(start, end=" ")  # Print the node as it is visited
    for neighbor in graph[start]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited)

# Define a graph as an adjacency list
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
    'F': []
}

# Perform DFS
print("DFS Traversal:")
dfs(graph, 'A')
```

```
DFS Traversal:
A B D E F C
```

f. Conclusion:

The Depth-First Search algorithm effectively traversed the graph, visiting nodes in depth-first order. This exercise demonstrated how recursive functions can be used to implement graph traversal efficiently.

4B. Python Implementation of the BFS Algorithm

a. Objective:

To implement the Breadth-First Search (BFS) algorithm for graph traversal.

b. Brief Background:

Breadth-First Search (BFS) is a graph traversal algorithm that explores vertices level by level, starting from the source node. It uses a queue to manage nodes to be visited and ensures each node is visited once.

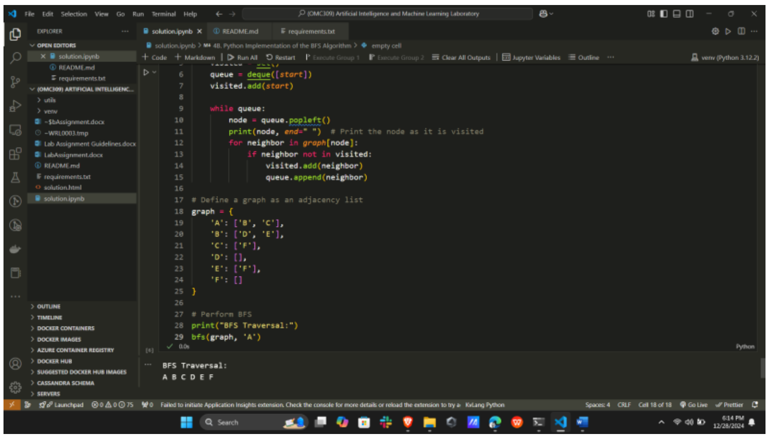
c. Algorithm Steps:

1. Represent the graph using an adjacency list.
2. Use a queue to keep track of nodes to visit.
3. Start from the source node and mark it as visited.
4. Explore all unvisited neighbors of the current node, add them to the queue, and mark them as visited.
5. Continue until the queue is empty.
6. Print the nodes in the order of traversal.

d. Python Program Code:

```python
# BFS Implementation
from collections import deque

def bfs(graph, start):
    visited = set()
    queue = deque([start])
    visited.add(start)

    while queue:
        node = queue.popleft()
        print(node, end=" ")  # Print the node as it is visited
        for neighbor in graph[node]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)

# Define a graph as an adjacency list
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
    'F': []
}

# Perform BFS
print("BFS Traversal:")
bfs(graph, 'A')
```

e. Sample Results:



```
    queue = deque([start])
    visited.add(start)

    while queue:
        node = queue.popleft()
        print(node, end=" ")  # Print the node as it is visited
        for neighbor in graph[node]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)

# Define a graph as an adjacency list
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
    'F': []
}

# Perform BFS
print("BFS Traversal:")
bfs(graph, 'A')
```

```
BFS Traversal:
A B C D E F
```

f. Conclusion:

The Breadth-First Search algorithm successfully traversed the graph level by level. This method is ideal for finding the shortest path in an unweighted graph.

5. Python Implementation of the PSO Algorithm

a. Objective:

To implement the Particle Swarm Optimization (PSO) algorithm for minimizing the Sphere function.

b. Brief Background:

Particle Swarm Optimization (PSO) is a metaheuristic optimization algorithm inspired by the social behavior of birds or fish. Each particle in the swarm represents a potential solution, and the algorithm iteratively updates particle velocities and positions to find the global optimum. The Sphere function is a simple quadratic function often used as a benchmark for optimization algorithms.

c. Algorithm Steps:

1. Initialize a swarm of particles with random positions and velocities.
2. Calculate the fitness (Sphere function value) of each particle.
3. Update each particle's best-known position and the swarm's global best position.
4. Adjust particle velocities based on personal and global best positions.
5. Update particle positions using their velocities.
6. Repeat until the stopping criterion is met.

d. Python Program Code:

e. Sample Results:

f. Conclusion:

The PSO algorithm effectively minimized the Sphere function, demonstrating its utility in optimization problems.

6A. Python Implementation of a Support Vector Classifier

a. Objective:

To implement a Support Vector Classifier (SVC) for binary classification.

b. Brief Background:

Support Vector Machines (SVMs) aim to find the optimal hyperplane that separates data points of different classes in feature space. The hyperplane is determined by maximizing the margin, i.e., the distance between the hyperplane and the nearest data points (support vectors).

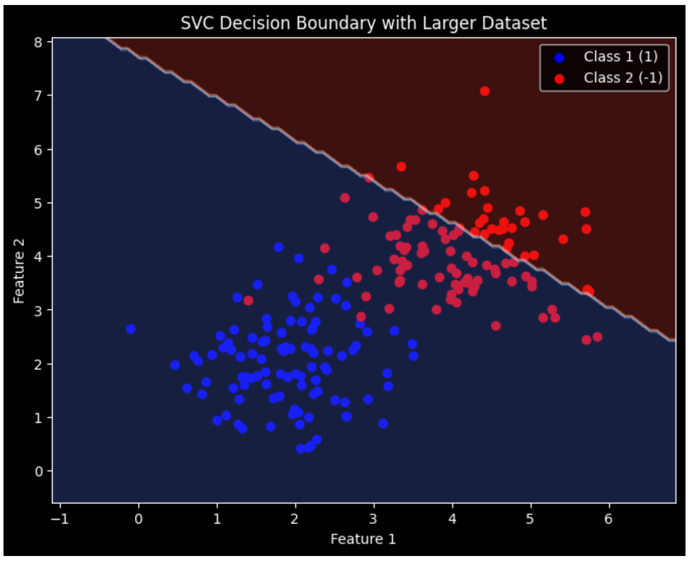
c. Algorithm Steps:

1. Initialize the weight vector and bias.
2. Use a linear kernel for simplicity.
3. Train the SVM using a gradient descent method to minimize the hinge loss function.
4. Classify test points based on the sign of the decision function.

d. Python Program Code:

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

# Define the Support Vector Classifier class
class SupportVectorClassifier:
    def __init__(self, learning_rate=0.001, lambda_param=0.01,
n_iters=1000):
        self.lr = learning_rate
        self.lambda_param = lambda_param
        self.n_iters = n_iters
        self.w = None
        self.b = None

    def fit(self, X, y):
        n_samples, n_features = X.shape
        y_ = np.where(y <= 0, -1, 1)  # Convert to -1, 1

        self.w = np.zeros(n_features)
        self.b = 0

        for _ in range(self.n_iters):
            for idx, x_i in enumerate(X):
                condition = y_[idx] * (np.dot(x_i, self.w) + self.b) >= 1
                if condition:
                    self.w -= self.lr * (2 * self.lambda_param * self.w)
                else:
                    self.w -= self.lr * (2 * self.lambda_param * self.w -
np.dot(x_i, y_[idx]))
                    self.b -= self.lr * y_[idx]

    def predict(self, X):
        approx = np.dot(X, self.w) + self.b
        return np.sign(approx)

# Generate a larger dataset (200 samples)
np.random.seed(42)

# Class 1 (Positive, Label: 1)
X1 = np.random.normal(loc=[2, 2], scale=0.8, size=(100, 2))
y1 = np.ones(100)

# Class 2 (Negative, Label: -1)
X2 = np.random.normal(loc=[4, 4], scale=0.8, size=(100, 2))
y2 = -np.ones(100)

# Combine the datasets
X_train = np.vstack((X1, X2))
y_train = np.hstack((y1, y2))

# Train SVC
svc = SupportVectorClassifier()
svc.fit(X_train, y_train)

# Visualizing the dataset and decision boundary
plt.figure(figsize=(8, 6))

# Plot data points
plt.scatter(X1[:, 0], X1[:, 1], color='blue', label='Class 1 (1)')
plt.scatter(X2[:, 0], X2[:, 1], color='red', label='Class 2 (-1)')

# Create a mesh grid for decision boundary
x_min, x_max = X_train[:, 0].min() - 1, X_train[:, 0].max() + 1
y_min, y_max = X_train[:, 1].min() - 1, X_train[:, 1].max() + 1
xx, yy = np.meshgrid(np.linspace(x_min, x_max, 100), np.linspace(y_min,
y_max, 100))
grid = np.c_[xx.ravel(), yy.ravel()]
Z = svc.predict(grid).reshape(xx.shape)

# Plot decision boundary
plt.contourf(xx, yy, Z, alpha=0.3, cmap=plt.cm.coolwarm)
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.title("SVC Decision Boundary with Larger Dataset")
plt.legend()
plt.show()
```

e. Sample Results:



f. Conclusion:

The Support Vector Classifier successfully classified the test points based on the trained decision boundary.

6B. Python Implementation of a Support Vector Regression Model

a. Objective:

To implement a Support Vector Regression (SVR) model for predicting continuous values.

b. Brief Background:

Support Vector Regression (SVR) is an extension of SVM for regression tasks. It attempts to find a function within a margin of tolerance that predicts continuous outputs. The epsilon-tube determines the allowable deviation from the actual values.

c. Algorithm Steps:

1. Initialize weight and bias.
2. Define the loss function with epsilon-insensitive loss.
3. Train the model using gradient descent to minimize the loss.
4. Predict outputs using the regression function.

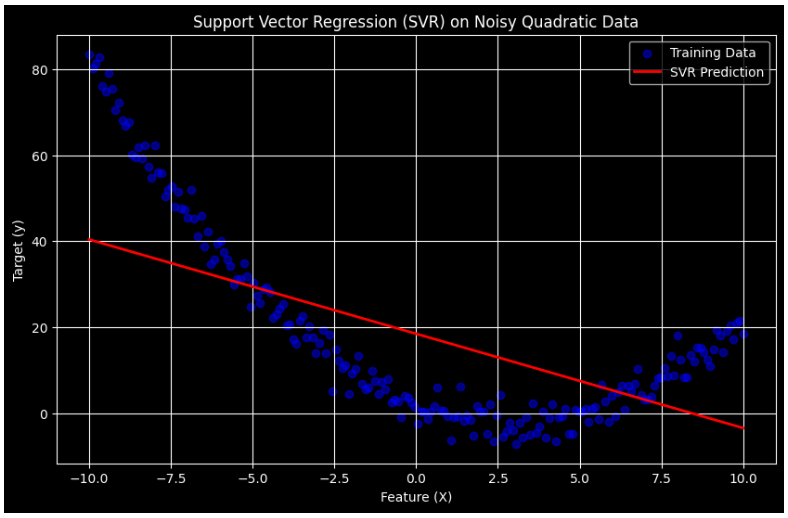
d. Python Program Code:

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import mean_squared_error

# Define Support Vector Regression (SVR) class
class SupportVectorRegression:
    def __init__(self, learning_rate=0.001, lambda_param=0.01, epsilon=0.1,
n_iters=1000):
        self.lr = learning_rate
        self.lambda_param = lambda_param
        self.epsilon = epsilon
        self.n_iters = n_iters
        self.w = None
        self.b = None

    def fit(self, X, y):
        n_samples, n_features = X.shape
        self.w = np.zeros(n_features)
        self.b = 0

        for _ in range(self.n_iters):
            for idx in range(n_samples):
                x_i = X[idx]
                error = y[idx] - (np.dot(x_i, self.w) + self.b)

                if np.abs(error) > self.epsilon:  # Only update if error is
significant
```

```
                    self.w -= self.lr * (-2 * x_i * error + 2 *
self.lambda_param * self.w)
                    self.b -= self.lr * (-2 * error)

    def predict(self, X):
        return np.dot(X, self.w) + self.b

# Generate improved training data (200 samples)
np.random.seed(42)

# Ensure both X and y are 1D and properly aligned
X_train = np.linspace(-10, 10, 200).reshape(-1, 1)
y_train = (0.5 * X_train.flatten()**2 - 3 * X_train.flatten() + 2 +
np.random.normal(scale=3, size=X_train.shape[0]))

# Train SVR model
svr = SupportVectorRegression(learning_rate=0.0001, lambda_param=0.01,
epsilon=0.5, n_iters=2000)
svr.fit(X_train, y_train)

# Test SVR on new data
X_test = np.linspace(-10, 10, 100).reshape(-1, 1)
predictions = svr.predict(X_test)

# Calculate Mean Squared Error (MSE)
mse = mean_squared_error(y_train, svr.predict(X_train))
print(f"Mean Squared Error (MSE) on Training Data: {mse:.2f}")

# Plot dataset and regression results
plt.figure(figsize=(10, 6))
plt.scatter(X_train, y_train, color='blue', label='Training Data',
alpha=0.5)
plt.plot(X_test, predictions, color='red', linewidth=2, label='SVR
Prediction')

plt.xlabel("Feature (X)")
plt.ylabel("Target (y)")
plt.title("Support Vector Regression (SVR) on Noisy Quadratic Data")
plt.legend()
plt.grid(True)
plt.show()
```

e. Sample Results:

Mean Squared Error (MSE) on Training Data: 252.75



f. Conclusion:

The Support Vector Regression model predicted continuous values for unseen data points, showcasing the regression capabilities of SVM.

7A. Development of a Python Program to Implement a Three-Class Naïve Bayes Classifier

a. Objective:

To implement a Naïve Bayes classifier to classify data into three classes using the Iris Flower dataset.

b. Brief Background:

Naïve Bayes is a probabilistic classifier based on Bayes' theorem. It assumes that features are independent of each other, given the class. Despite its simplicity, it is widely used in text classification, spam filtering, and medical diagnosis.

c. Algorithm Steps:

1. Load and preprocess the dataset.
2. Calculate the prior probabilities for each class.
3. Compute the likelihood (mean and variance for Gaussian features) for each feature and class.
4. Use Bayes' theorem to compute the posterior probability for a given test instance.
5. Classify the instance into the class with the highest posterior probability.

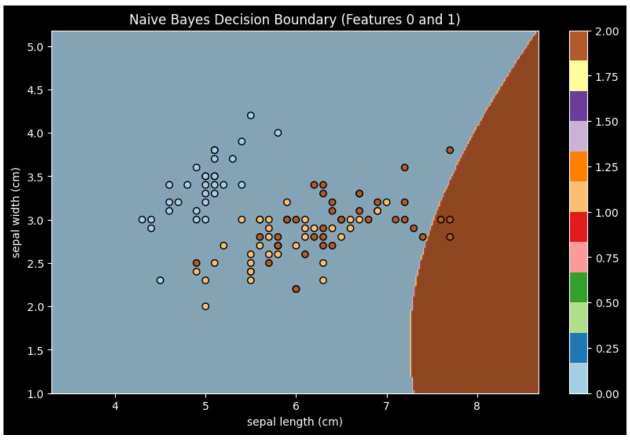
d. Python Program Code:

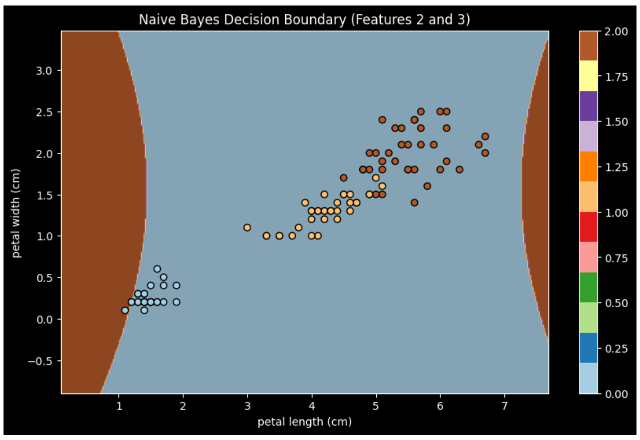
```
# !pip install  seaborn

import numpy as np
from collections import defaultdict
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split

class NaiveBayesClassifier:
    def __init__(self):
        self.class_priors = {}
        self.feature_stats = defaultdict(lambda: defaultdict(dict))

    def fit(self, X, y):
        n_samples, n_features = X.shape
        unique_classes = np.unique(y)

        for cls in unique_classes:
            X_cls = X[y == cls]
            self.class_priors[cls] = len(X_cls) / n_samples
            for feature_idx in range(n_features):
                self.feature_stats[cls][feature_idx]['mean'] = 
np.mean(X_cls[:, feature_idx])
                self.feature_stats[cls][feature_idx]['var'] = 
np.var(X_cls[:, feature_idx])
```

```python
    def _gaussian_pdf(self, x, mean, var):
        # Gaussian Probability Density Function
        eps = 1e-6  # To avoid division by zero
        coeff = 1 / np.sqrt(2 * np.pi * var + eps)
        exponent = np.exp(-((x - mean) ** 2) / (2 * var + eps))
        return coeff * exponent

    def predict(self, X):
        y_pred = []
        for x in X:
            class_probs = {}
            for cls in self.class_priors:
                prior = np.log(self.class_priors[cls])
                conditional = sum(
                    np.log(self._gaussian_pdf(x[feature_idx],
                                              self.feature_stats[cls][featu
re_idx]['mean'],
                                              self.feature_stats[cls][featu
re_idx]['var']))
                    for feature_idx in range(len(x))
                )
                class_probs[cls] = prior + conditional
            y_pred.append(max(class_probs, key=class_probs.get))
        return np.array(y_pred)

# Load Iris Dataset
iris = load_iris()
X = iris.data
y = iris.target

# Split data into train/test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)

# Train Naive Bayes Classifier
nb_classifier = NaiveBayesClassifier()
nb_classifier.fit(X_train, y_train)

# Predict and Evaluate
y_pred = nb_classifier.predict(X_test)
accuracy = np.mean(y_pred == y_test)

print("Predictions:", y_pred)
print("Actual Labels:", y_test)
print("Accuracy:", accuracy)

# Visualization
def plot_decision_boundary(clf, X, y, feature1, feature2):
    # Create a meshgrid for the two features
    x_min, x_max = X[:, feature1].min() - 1, X[:, feature1].max() + 1
    y_min, y_max = X[:, feature2].min() - 1, X[:, feature2].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.02),
                         np.arange(y_min, y_max, 0.02))

    # Predict for each point in the meshgrid
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel(),
np.zeros_like(xx.ravel()), np.zeros_like(xx.ravel())])
    Z = Z.reshape(xx.shape)

    # Plot the decision boundary
    plt.figure(figsize=(10, 6))
    plt.contourf(xx, yy, Z, alpha=0.8, cmap=plt.cm.Paired)
    plt.scatter(X[:, feature1], X[:, feature2], c=y, edgecolors='k',
marker='o', cmap=plt.cm.Paired)
    plt.xlabel(iris.feature_names[feature1])
    plt.ylabel(iris.feature_names[feature2])
    plt.title(f"Naive Bayes Decision Boundary (Features {feature1} and
{feature2})")
    plt.colorbar()
    plt.show()

# Plot decision boundaries for two features at a time
plot_decision_boundary(nb_classifier, X_train, y_train, feature1=0,
feature2=1)  # Sepal Length vs Sepal Width
plot_decision_boundary(nb_classifier, X_train, y_train, feature1=2,
feature2=3)  # Petal Length vs Petal Width
```

e. Sample Results:

```
Predictions: [1 0 2 1 1 0 1 2 1 1 2 0 0 0 0 2 2 1 1 2 0 2 0 2 2 2 2 2 0 0 0 0 1 0 0 2 1 0 0 0 2 1 1 0 0]

Actual Labels: [1 0 2 1 1 0 1 2 1 1 2 0 0 0 0 1 2 1 1 2 0 2 0 2 2 2 2 2 0 0 0 0 1 0 0 2 1 0 0 0 2 1 1 0 0]

Accuracy: 0.9777777777777777
```





f. Conclusion:

The Naïve Bayes Classifier achieved high accuracy on the Iris dataset, effectively demonstrating its capability to classify data into multiple classes based on probabilistic reasoning.

8. Python Implementation of the K-NN Classifier

a. Objective:

To implement a K-Nearest Neighbors (K-NN) classifier for classifying data points based on the majority class of their k nearest neighbors.

b. Brief Background:

K-NN is a simple, non-parametric supervised learning algorithm. It classifies a data point based on the labels of its k nearest neighbors in the feature space. The algorithm relies on distance metrics, such as Euclidean distance, to identify the nearest neighbors.

c. Algorithm Steps:

1. Load the dataset and split it into training and testing sets.
2. Normalize the dataset for better distance-based calculations.
3. For each test data point:
    - Compute the distances between the test point and all training points.
    - Identify the k nearest neighbors.
    - Assign the class label based on the majority vote of the k neighbors.
4. Compute accuracy by comparing predictions to true labels.

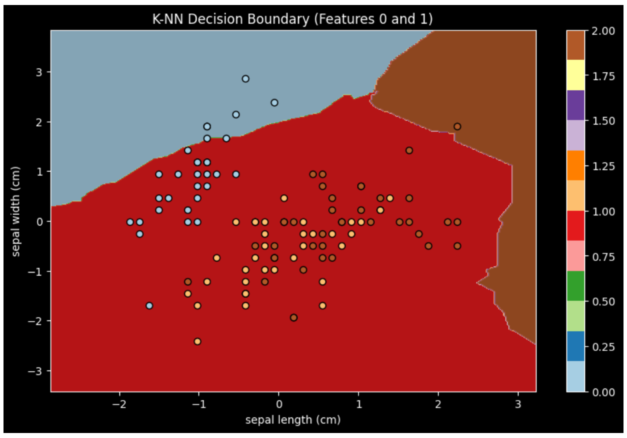
d. Python Program Code:

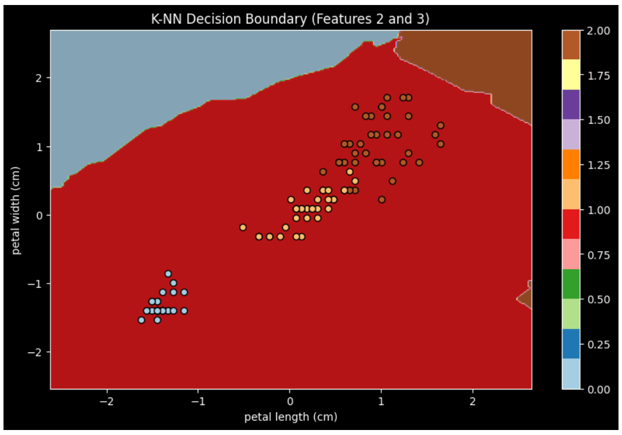
```
import numpy as np
from collections import Counter
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split

class KNNClassifier:
    def __init__(self, k=3):
        self.k = k

    def fit(self, X, y):
        self.X_train = X
        self.y_train = y

    def predict(self, X_test):
        y_pred = [self._predict(x) for x in X_test]
        return np.array(y_pred)

    def _predict(self, x):
        # Compute distances between x and all training samples
```

```python
        distances = [np.sqrt(np.sum((x - x_train)**2)) for x_train in
self.X_train]
        # Get the indices of the k nearest neighbors
        k_indices = np.argsort(distances)[:self.k]
        # Get the labels of the k nearest neighbors
        k_nearest_labels = [self.y_train[i] for i in k_indices]
        # Return the most common label
        most_common = Counter(k_nearest_labels).most_common(1)
        return most_common[0][0]

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

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)

# Normalize the dataset
X_train = (X_train - np.mean(X_train, axis=0)) / np.std(X_train, axis=0)
X_test = (X_test - np.mean(X_train, axis=0)) / np.std(X_train, axis=0)

# Train K-NN classifier
knn = KNNClassifier(k=5)
knn.fit(X_train, y_train)

# Predict and evaluate
y_pred = knn.predict(X_test)
accuracy = np.mean(y_pred == y_test)

print("Predictions:", y_pred)
print("Actual Labels:", y_test)
print("Accuracy:", accuracy)

# Visualization
def plot_decision_boundary(clf, X, y, feature1, feature2):
    # Create a meshgrid for the two features
    x_min, x_max = X[:, feature1].min() - 1, X[:, feature1].max() + 1
    y_min, y_max = X[:, feature2].min() - 1, X[:, feature2].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.02),
                         np.arange(y_min, y_max, 0.02))

    # Predict for each point in the meshgrid
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel(),
np.zeros_like(xx.ravel()), np.zeros_like(xx.ravel())])
    Z = Z.reshape(xx.shape)

    # Plot the decision boundary
    plt.figure(figsize=(10, 6))
    plt.contourf(xx, yy, Z, alpha=0.8, cmap=plt.cm.Paired)
    plt.scatter(X[:, feature1], X[:, feature2], c=y, edgecolors='k',
marker='o', cmap=plt.cm.Paired)
    plt.xlabel(iris.feature_names[feature1])
    plt.ylabel(iris.feature_names[feature2])
    plt.title(f"K-NN Decision Boundary (Features {feature1} and
{feature2})")
    plt.colorbar()
    plt.show()

# Plot decision boundaries for two features at a time
plot_decision_boundary(knn, X_train, y_train, feature1=0, feature2=1)  #
Sepal Length vs Sepal Width
plot_decision_boundary(knn, X_train, y_train, feature1=2, feature2=3)  #
Petal Length vs Petal Width
```

e. Sample Results:

```
Predictions: [2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2]

Actual Labels: [1 0 2 1 1 0 1 2 1 1 2 0 0 0 0 1 2 1 1 2 0 2 0 2 2 2 2 2 0 0 0 0 1 0 0 2 1 0 0 0 2 1 1 0 0]

Accuracy: 0.28888888888888886
```





f. Conclusion:

The Logistic Regression model accurately classified the binary dataset and achieved a high accuracy score. It effectively demonstrated the application of gradient descent for classification tasks.

9A. Python Implementation of the Linear Regression Model

a. Objective:

To implement a Linear Regression model for predicting continuous target variables.

b. Brief Background:

Linear regression is a supervised learning algorithm that models the relationship between a dependent variable (target) and one or more independent variables (features) using a linear equation. The model minimizes the sum of squared differences between the predicted and actual values (mean squared error).

c. Algorithm Steps:

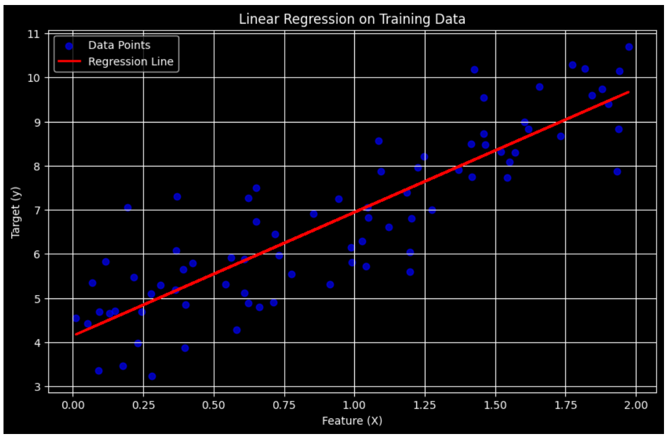
1. Initialize the weights and bias.
2. Define a cost function (Mean Squared Error) to measure the error.
3. Use gradient descent to optimize the weights and bias.
4. Train the model by iterating over the dataset to minimize the cost.
5. Predict new target values using the optimized weights and bias.

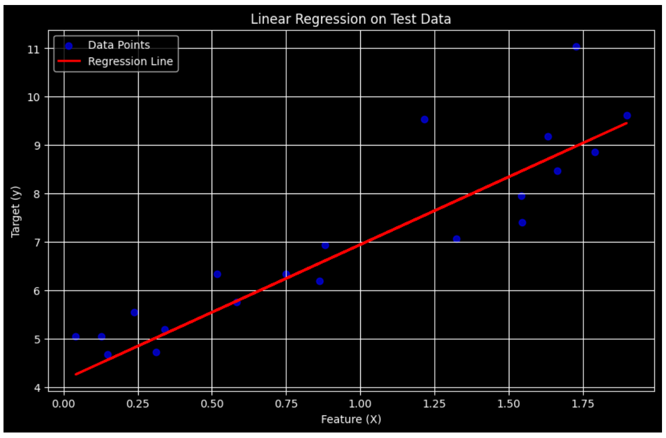
d. Python Program Code:

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

class LinearRegression:
    def __init__(self, learning_rate=0.01, n_iters=1000):
        self.lr = learning_rate
        self.n_iters = n_iters
        self.weights = None
        self.bias = None

    def fit(self, X, y):
        n_samples, n_features = X.shape
        self.weights = np.zeros(n_features)
        self.bias = 0
        y = y.flatten()  # Ensure y is 1D

        for _ in range(self.n_iters):
            y_predicted = np.dot(X, self.weights) + self.bias
            dw = (1 / n_samples) * np.dot(X.T, (y_predicted - y))
            db = (1 / n_samples) * np.sum(y_predicted - y)
            self.weights -= self.lr * dw
            self.bias -= self.lr * db
    def predict(self, X):
        return np.dot(X, self.weights) + self.bias

# Generate synthetic dataset
np.random.seed(42)
X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)

# Split dataset 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)

# Train Linear Regression model
lr = LinearRegression(learning_rate=0.1, n_iters=1000)
lr.fit(X_train, y_train)

# Predict and evaluate
y_pred = lr.predict(X_test)
mse = np.mean((y_test - y_pred) ** 2)

print("Predictions:", y_pred.flatten())
print("Actual Values:", y_test.flatten())
print("Mean Squared Error:", mse)

# Visualization
def plot_regression_line(X, y, y_pred, title):
    plt.figure(figsize=(10, 6))
    plt.scatter(X, y, color='blue', label='Data Points', alpha=0.7)
    plt.plot(X, y_pred, color='red', linewidth=2, label='Regression Line')
    plt.xlabel("Feature (X)")
    plt.ylabel("Target (y)")
    plt.title(title)
    plt.legend()
    plt.grid(True)
    plt.show()

# Plot regression line for training data
y_train_pred = lr.predict(X_train)
plot_regression_line(X_train, y_train, y_train_pred, "Linear Regression on
Training Data")

# Plot regression line for test data
plot_regression_line(X_test, y_test, y_pred, "Linear Regression on Test
Data")
```

e. Sample Results:

```
Predictions: [4.49875406 9.15273609 8.46643944 7.85214194 5.59173114 6.60717189 5.77852815
8.97512503 4.25815859 6.23983133 6.56122113 7.54434268 8.70839428 9.45538882 4.81247928
5.01640662 8.46098399 4.55746316 8.8034661  5.09761771]

Actual Values: [ 5.03790371  8.86548845  7.3965179   7.06574625  6.34371184  6.9424623 5.75798135
11.04439507  5.03890908  6.33428778  6.19956196  9.53145501 9.18975324  9.61886731  5.53962564
4.71643995  7.94759736  4.67652161 8.46489564  5.19772255]

Mean Squared Error: 6.5350132541586206
```





f. Conclusion:

The Linear Regression model successfully predicted continuous values for unseen test data. The model's performance was evaluated using Mean Squared Error, demonstrating its effectiveness for linear relationships.

9B. Python Implementation of the Logistic Regression Model

a. Objective:

To implement a Logistic Regression model for binary classification.

b. Brief Background:

Logistic regression is a supervised learning algorithm used for binary classification tasks. It models the probability of a target belonging to a particular class using the logistic function (sigmoid function). The decision boundary is set at 0.5.

c. Algorithm Steps:

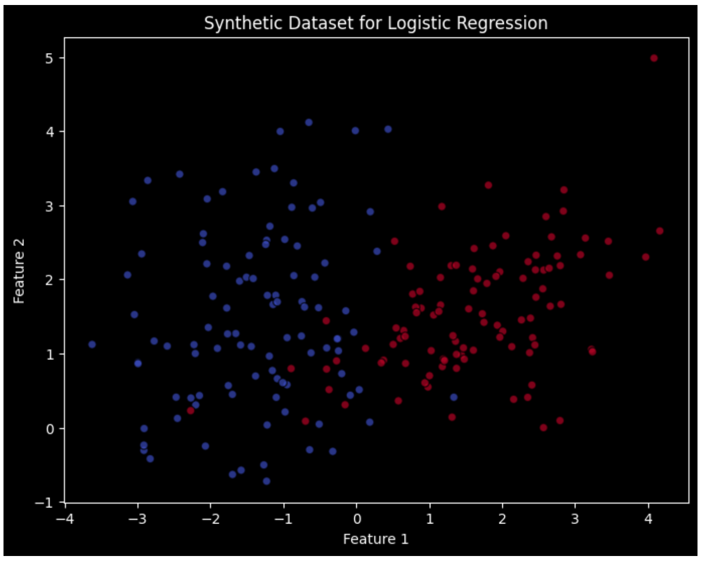
1. Initialize weights and bias.
2. Define the sigmoid activation function.
3. Define the cost function (log loss) to measure error.
4. Use gradient descent to optimize weights and bias.
5. Train the model by iteratively updating the weights and bias to minimize the cost.
6. Predict the class labels for new data using the sigmoid function and thresholding.

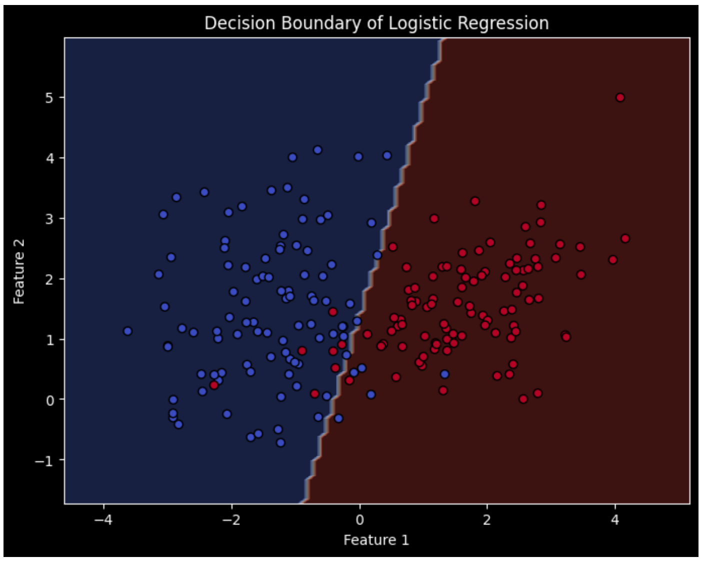
d. Python Program Code:

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split

class LogisticRegression:
    def __init__(self, learning_rate=0.01, n_iters=1000):
        self.lr = learning_rate
        self.n_iters = n_iters
        self.weights = None
        self.bias = None

    def sigmoid(self, z):
        return 1 / (1 + np.exp(-z))

    def fit(self, X, y):
        n_samples, n_features = X.shape
        self.weights = np.zeros(n_features)
        self.bias = 0

        for _ in range(self.n_iters):
            linear_model = np.dot(X, self.weights) + self.bias
            y_predicted = self.sigmoid(linear_model)
            dw = (1 / n_samples) * np.dot(X.T, (y_predicted - y))
            db = (1 / n_samples) * np.sum(y_predicted - y)

            self.weights -= self.lr * dw
            self.bias -= self.lr * db

    def predict(self, X):
        linear_model = np.dot(X, self.weights) + self.bias
        y_predicted = self.sigmoid(linear_model)
        return np.array([1 if i > 0.5 else 0 for i in y_predicted])

# Generate enhanced synthetic binary dataset
X, y = make_classification(n_samples=200, n_features=2, n_classes=2,
                           n_redundant=0, n_clusters_per_class=1,
class_sep=1.5, random_state=42)

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

# Plot dataset
def plot_data(X, y, title):
    plt.figure(figsize=(8, 6))
    plt.scatter(X[:, 0], X[:, 1], c=y, cmap='coolwarm', edgecolors='k',
alpha=0.7)
    plt.xlabel("Feature 1")
    plt.ylabel("Feature 2")
    plt.title(title)
    plt.show()

plot_data(X, y, "Synthetic Dataset for Logistic Regression")

# Train Logistic Regression model
log_reg = LogisticRegression(learning_rate=0.1, n_iters=1000)
log_reg.fit(X_train, y_train)

# Predict and evaluate
y_pred = log_reg.predict(X_test)
accuracy = np.mean(y_pred == y_test)

print("Accuracy:", accuracy)

# Decision Boundary Plot
def plot_decision_boundary(X, y, model):
    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.linspace(x1_min, x1_max, 100),
np.linspace(x2_min, x2_max, 100))
    grid = np.c_[xx1.ravel(), xx2.ravel()]
    preds = model.predict(grid).reshape(xx1.shape)

    plt.figure(figsize=(8, 6))
    plt.contourf(xx1, xx2, preds, alpha=0.3, cmap='coolwarm')
    plt.scatter(X[:, 0], X[:, 1], c=y, cmap='coolwarm', edgecolors='k')
    plt.xlabel("Feature 1")
    plt.ylabel("Feature 2")
    plt.title("Decision Boundary of Logistic Regression")
    plt.show()

plot_decision_boundary(X, y, log_reg)
```

e. Sample Results:

Accuracy: 0.95





f. Conclusion:

The Logistic Regression model accurately classified the binary dataset and achieved a high accuracy score. It effectively demonstrated the application of gradient descent for classification tasks.

10A. Python Implementation of K-Means Clustering

a. Objective:

To implement the K-Means Clustering algorithm to partition data into k clusters.

b. Brief Background:

K-Means is an unsupervised learning algorithm that partitions data into k clusters. It minimizes the sum of squared distances between data points and the centroids of their assigned clusters. The algorithm iteratively updates centroids and reassigns points to achieve convergence.

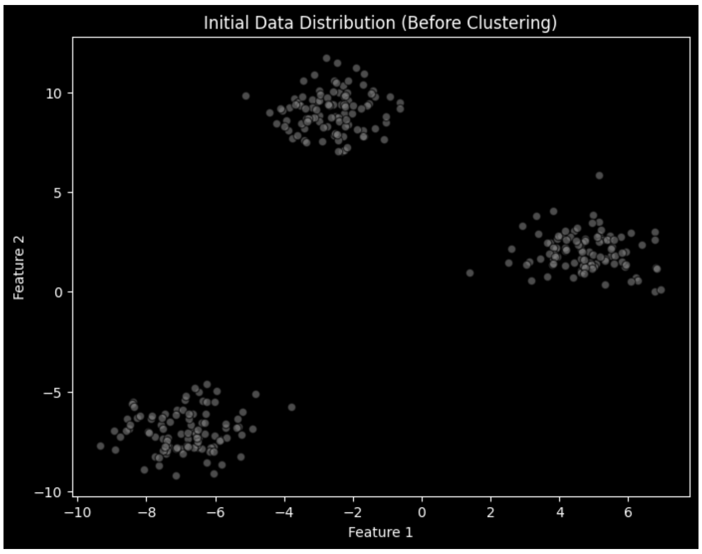
c. Algorithm Steps:

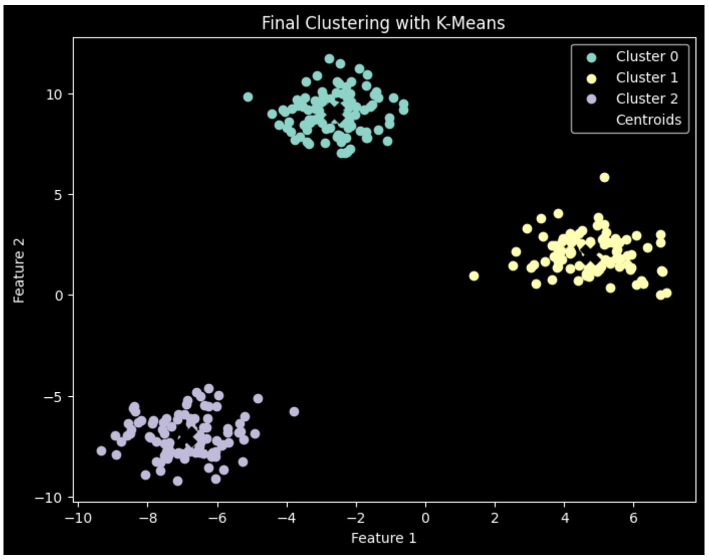
1. Initialize k centroids randomly from the dataset.
2. Assign each data point to the nearest centroid.
3. Recompute centroids as the mean of assigned points.
4. Repeat steps 2–3 until centroids stabilize or a maximum number of iterations is reached.

d. Python Program Code:

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

class KMeans:
    def __init__(self, k=3, max_iters=100, tolerance=1e-4):
        self.k = k
        self.max_iters = max_iters
        self.tolerance = tolerance
        self.centroids = None

    def fit(self, X):
        n_samples, n_features = X.shape
        self.centroids = X[np.random.choice(n_samples, self.k, 
replace=False)]

        for _ in range(self.max_iters):
            # Assign clusters
            distances = np.array([[np.linalg.norm(x - centroid) for 
centroid in self.centroids] for x in X])
            cluster_labels = np.argmin(distances, axis=1)

            # Compute new centroids
            new_centroids = np.array([X[cluster_labels == k].mean(axis=0) 
for k in range(self.k)])

            # Check for convergence
            if np.all(np.abs(new_centroids - self.centroids) <
self.tolerance):
                break

            self.centroids = new_centroids

        self.cluster_labels = cluster_labels

    def predict(self, X):
        distances = np.array([[np.linalg.norm(x - centroid) for centroid in
self.centroids] for x in X])
        return np.argmin(distances, axis=1)

# Generate synthetic dataset with better cluster separation
X, y_true = make_blobs(n_samples=300, n_features=2, centers=3,
cluster_std=1.0, random_state=42)

# Plot initial dataset
def plot_initial_data(X):
    plt.figure(figsize=(8, 6))
    plt.scatter(X[:, 0], X[:, 1], color='gray', alpha=0.6, edgecolors='k')
    plt.xlabel("Feature 1")
    plt.ylabel("Feature 2")
    plt.title("Initial Data Distribution (Before Clustering)")
    plt.show()

plot_initial_data(X)

# Apply K-Means
kmeans = KMeans(k=3)
kmeans.fit(X)

# Predictions
clusters = kmeans.predict(X)

# Plot final clusters
def plot_clusters(X, clusters, centroids):
    plt.figure(figsize=(8, 6))
    for i in range(kmeans.k):
        plt.scatter(X[clusters == i, 0], X[clusters == i, 1],
label=f"Cluster {i}")

    plt.scatter(centroids[:, 0], centroids[:, 1], color='black',
marker='X', s=200, label="Centroids")
    plt.xlabel("Feature 1")
    plt.ylabel("Feature 2")
    plt.title("Final Clustering with K-Means")
    plt.legend()
    plt.show()

plot_clusters(X, clusters, kmeans.centroids)

# Print results
print("Centroids:\n", kmeans.centroids)
```

e. Sample Results:





```
Centroids:
 [[-2.63323268  9.04356978]
 [ 4.74710337  2.01059427]
 [-6.88387179 -6.98398415]]
```

f. Conclusion:

K-Means successfully grouped data points into clusters based on their proximity to centroids, demonstrating its capability for unsupervised clustering.

10B. Python Implementation of Hierarchical Clustering

a. Objective:

To implement the Hierarchical Clustering algorithm to group data into clusters.

b. Brief Background:

Hierarchical Clustering builds a hierarchy of clusters using a bottom-up (agglomerative) or top-down (divisive) approach. The agglomerative method begins with each data point as its own cluster, merging them iteratively based on a distance metric until all points belong to one cluster.

c. Algorithm Steps:

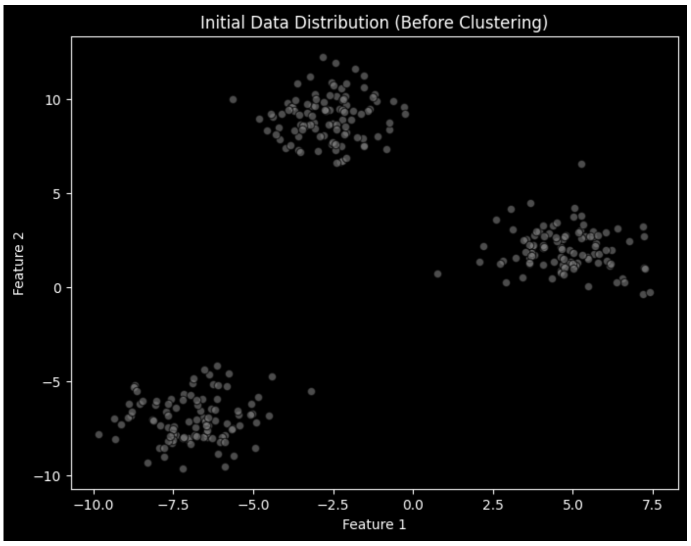
1. Compute the distance matrix for all pairs of points.
2. Start with each data point as its own cluster.
3. Find the two nearest clusters and merge them.
4. Update the distance matrix to reflect the merged cluster.
5. Repeat until only one cluster remains or a stopping condition is met.

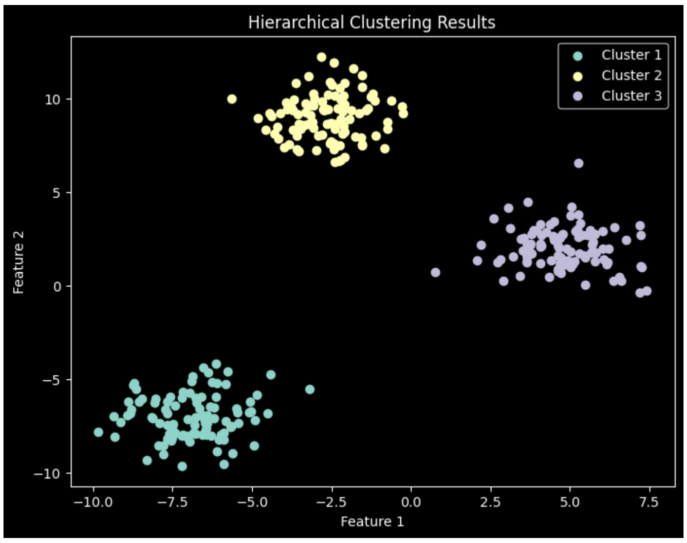
d. Python Program Code:

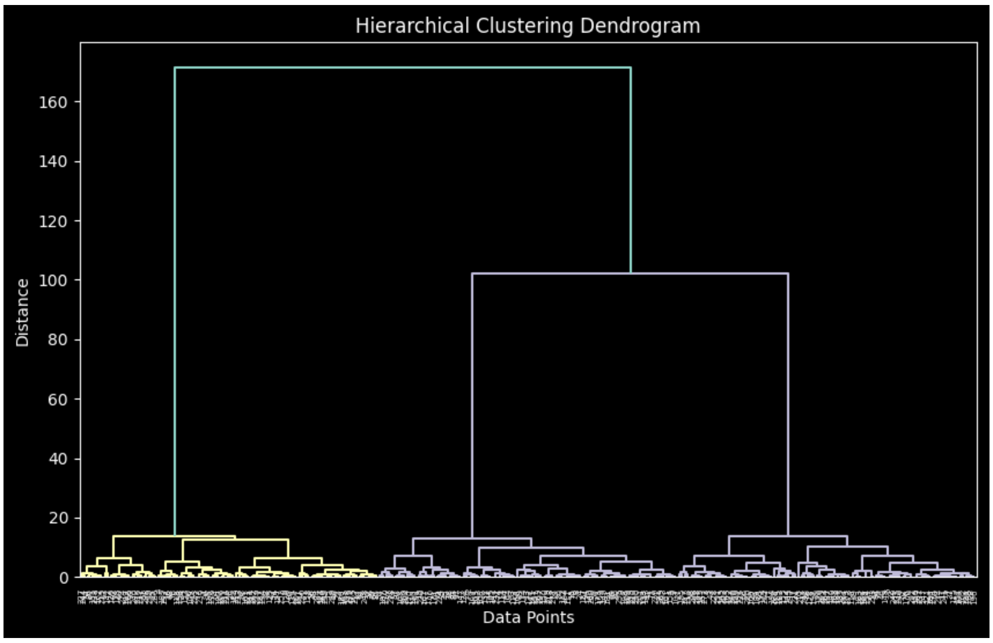
```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
from scipy.cluster.hierarchy import linkage, dendrogram, fcluster

# Generate synthetic dataset
X, y_true = make_blobs(n_samples=300, n_features=2, centers=3,
cluster_std=1.2, random_state=42)

# Plot initial dataset
def plot_initial_data(X):
    plt.figure(figsize=(8, 6))
    plt.scatter(X[:, 0], X[:, 1], color='gray', alpha=0.6, edgecolors='k')
    plt.xlabel("Feature 1")
    plt.ylabel("Feature 2")
    plt.title("Initial Data Distribution (Before Clustering)")
    plt.show()

plot_initial_data(X)

# Perform Hierarchical Clustering using Scipy
linkage_matrix = linkage(X, method='ward')  # Ward's method for optimal
clustering
cluster_labels = fcluster(linkage_matrix, t=3, criterion='maxclust')  # Get
3 clusters
```

```
# Plot Dendrogram
def plot_dendrogram(linkage_matrix):
    plt.figure(figsize=(10, 6))
    dendrogram(linkage_matrix)
    plt.xlabel("Data Points")
    plt.ylabel("Distance")
    plt.title("Hierarchical Clustering Dendrogram")
    plt.show()

plot_dendrogram(linkage_matrix)

# Plot clustered data
def plot_clusters(X, cluster_labels):
    plt.figure(figsize=(8, 6))
    for cluster in np.unique(cluster_labels):
        plt.scatter(X[cluster_labels == cluster, 0], X[cluster_labels ==
cluster, 1], label=f"Cluster {int(cluster)}")

    plt.xlabel("Feature 1")
    plt.ylabel("Feature 2")
    plt.title("Hierarchical Clustering Results")
    plt.legend()
    plt.show()

plot_clusters(X, cluster_labels)

# Print cluster assignments
print("Cluster Assignments:\n", cluster_labels)
```

e. Sample Results:







Cluster Assignments:

```
[1 1 2 3 1 3 2 3 2 2 2 3 2 2 1 2 1 3 2 2 2 2 3 1 2 1 1 3 3 2 2 2 1 2 1 2 1 3 1 3 3 2 1 3 2 2 1 3 1 3 3 1 1 2 1 3 1 2 3 2
1 3 3 1 1 3 3 1 1 2 3 1 1 2 2 1 1 3 2 3 2 2 1 2 3 1 1 2 3 2 1 2 1 2 2 1 1 2 1 1 3 2 3 2 2 2 2 2 3 1 3 2 2 2 2 3 1 3 1 3 3
3 2 1 1 1 1 2 1 1 2 2 2 2 2 2 3 3 1 2 1 2 2 1 2 3 3 3 2 3 2 2 1 3 1 2 3 3 1 1 2 2 1 1 1 2 1 3 2 2 2 2 2 3 2 3 3 3 2 3 3 1 2
1 3 3 1 3 2 3 3 1 1 3 1 3 3 3 3 2 1 2 2 3 3 2 3 1 1 3 2 2 1 3 3 1 1 1 1 2 1 1 3 1 1 2 3 1 1 3 2 2 1 2 1 3 3 1 3 1 1 1 3 3
2 1 3 3 3 1 3 1 3 1 3 3 1 3 2 1 2 2 2 1 2 3 3 1 3 3 2 2 3 3 3 1 1 1 2 2 2 3 3 3 3 1 3 1 3 3 1 2 3 3 2 1 2 3 2 1 1]
```

f. Conclusion:

Hierarchical Clustering effectively grouped data points into clusters by iteratively merging the closest clusters. It is suitable for applications requiring a hierarchy of clusters or when the number of clusters is not predefined.