**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:**

*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:**

**A screenshot of a computer program

Description automatically generated**

**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:**

*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:**

**A screenshot of a computer

Description automatically generated**

**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:**

# 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

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:**

**A screenshot of a computer

Description automatically generated**

**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:**

**A computer screen shot of a program

Description automatically generated**

**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:**

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()

# 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:**

**A graph of a function

Description automatically generated**  **A graph of a function

Description automatically generated**

**A graph with green bars

Description automatically generated A graph of a graph

Description automatically generated**

**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:**

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:**

**A screenshot of a computer program

Description automatically generated**

**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:**

# 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:**

A computer screen shot of a program

Description automatically generated

**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:**

# 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:**

**A screenshot of a computer program

Description automatically generated**

**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:**

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")

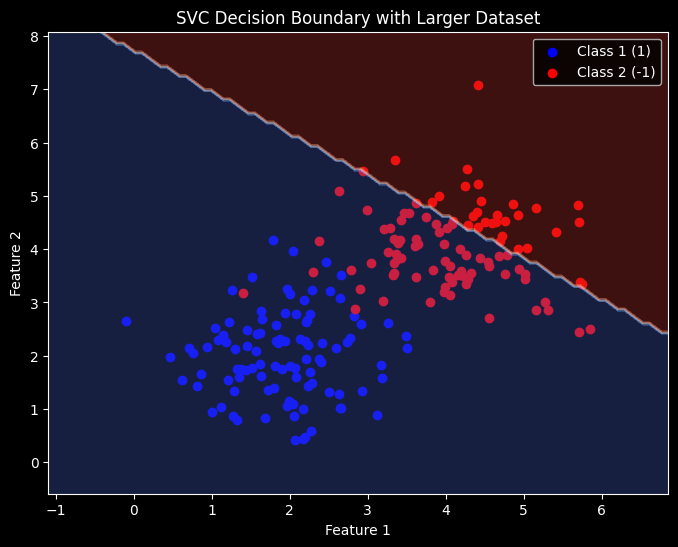
plt.ylabel("Feature 2")

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:**

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

A graph with a red line

Description automatically generated

**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])

*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][feature\_idx]['mean'],

*self*.feature\_stats[cls][feature\_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

A screen shot of a computer screen

Description automatically generated A screenshot of a video game

Description automatically generated

**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

        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

A screenshot of a computer screen

Description automatically generated A red and blue map with black text

Description automatically generated

**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:**

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

A graph with a red line and blue dots

Description automatically generated A graph with a red line

Description automatically generated

**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:**

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

**A screen shot of a computer screen

Description automatically generated A screen shot of a graph

Description automatically generated**

**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:**

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:**

**A screen shot of a data distribution

Description automatically generated A screen shot of a graph

Description automatically generated**

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:**

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:**

**A screen shot of a black background

Description automatically generated A screen shot of a graph

Description automatically generated**

**A screen shot of a graph

Description automatically generated**

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 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.