1. **Implement Polinomial regression on Data1.csv. Display the coefficients.**

```
import pandas as pd

# Polynomial Regression Function
def polynomial_regression(x, y, degree):
    n = len(x)
    X = [[xi ** d for d in range(degree + 1)] for xi in x]  # Generate X matrix
    XT = list(zip(*X))  # Transpose of X
    XTX = [[sum(XT[i][j] * X[j][k] for j in range(n)) for k in range(degree + 1)] for i in range(degree + 1)]  # X^T * X
    XTY = [sum(XT[i][j] * y[j] for j in range(n)) for i in range(degree + 1)]  # X^T * Y

    # Solve for coefficients using Gaussian elimination
    coeff = [XTY[i] / XTX[i][i] for i in range(degree + 1)]  # Assuming diagonal dominance
    return coeff

# Load data from CSV file
data = pd.read_csv('/content/Data1.csv')

# Extract input (X1) and target (y) variables
x = data['X1'].values  # Input variable
y = data['Y'].values   # Target variable

# Degree of polynomial
degree = 2  # Adjust as needed

# Perform polynomial regression
coefficients = polynomial_regression(x, y, degree)
```

```
# Display results
print("Coefficients:", coefficients)
```

2. Apply Logistic Regression on Pima Indian Diabetes dataset to predict the output.

```
import numpy as np
import pandas as pd

# Sigmoid function
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

# Logistic Regression
def logistic_regression(X, y, learning_rate, iterations):
    n, m = X.shape
    w = np.zeros(m)  # Initialize weights
    b = 0            # Initialize bias

    for _ in range(iterations):
        # Compute linear combination and predictions
        linear_model = np.dot(X, w) + b
        predictions = sigmoid(linear_model)

        # Compute gradients
        dw = (1 / n) * np.dot(X.T, (predictions - y))
        db = (1 / n) * np.sum(predictions - y)

        # Update weights and bias
        w -= learning_rate * dw
        b -= learning_rate * db
```

```
    return w, b

# Load Pima Indian Diabetes Dataset
data = pd.read_csv('/content/diabetes.csv')

# Feature selection and preprocessing
X = data.iloc[:, :-1].values  # All columns except the last one
y = data.iloc[:, -1].values   # Target column

# Normalize features
X = (X - X.mean(axis=0)) / X.std(axis=0)

# Train logistic regression
learning_rate = 0.01
iterations = 1000
weights, bias = logistic_regression(X, y, learning_rate,
iterations)

print("Weights:", weights)
print("Bias:", bias)
```

3.Predict the Digits in Images Using a Logistic Regression Classifier in Python.

```
import numpy as np

# Sigmoid function
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

# Logistic Regression Model
def logistic_regression(X, y, lr=0.01, epochs=1000):
    m, n = X.shape
```

```
    weights = np.zeros(n)  # Initialize weights
    bias = 0

    for _ in range(epochs):
        # Linear model
        z = np.dot(X, weights) + bias
        # Prediction
        y_pred = sigmoid(z)
        # Compute gradients
        dw = (1/m) * np.dot(X.T, (y_pred - y))
        db = (1/m) * np.sum(y_pred - y)
        # Update weights and bias
        weights -= lr * dw
        bias -= lr * db

    return weights, bias

# Prediction function
def predict(X, weights, bias):
    z = np.dot(X, weights) + bias
    y_pred = sigmoid(z)
    return (y_pred > 0.5).astype(int)

# Example Dataset (MNIST digits simplified for binary
classification: digit 0 vs 1)
# Here we use small dummy data for simplicity
X_train = np.array([[1, 2], [2, 1], [2, 3], [3, 4], [4, 3]])  #
Features
y_train = np.array([0, 0, 1, 1, 1])  # Labels (binary
classification)

# Train the model
weights, bias = logistic_regression(X_train, y_train)
```

```
# Test the model
X_test = np.array([[1, 1], [4, 4]])
predictions = predict(X_test, weights, bias)
print("Predictions:", predictions)
```

4.Apply K-means clustering on the following data.
.x = [4, 5, 10, 4, 3, 11, 14 , 6, 10, 12]
y = [21, 19, 24, 17, 16, 25, 24, 22, 21, 21]

```
def k_means(x, y, k, iterations):
    # Initialize centroids as the first `k` points
    centroids = [(x[i], y[i]) for i in range(k)]

    for _ in range(iterations):
        clusters = [[] for _ in range(k)]

        # Assign points to nearest centroid
        for i in range(len(x)):
            distances = [((x[i] - cx) ** 2 + (y[i] - cy) ** 2) for cx,
cy in centroids]
            cluster_index = distances.index(min(distances))
            clusters[cluster_index].append((x[i], y[i]))

        # Update centroids
        centroids = [(sum([p[0] for p in cluster]) / len(cluster),
                      sum([p[1] for p in cluster]) / len(cluster))
                     if cluster else centroids[i]
                     for i, cluster in enumerate(clusters)]

    return centroids, clusters

# Example Usage
```

```
x = [4, 5, 10, 4, 3, 11, 14 , 6, 10, 12]
y = [21, 19, 24, 17, 16, 25, 24, 22, 21, 21]
k = 2
iterations = 10
centroids, clusters = k_means(x, y, k, iterations)
print("Centroids:", centroids)
print("Clusters:", clusters)
```

5. Perform Hierarchical Clusteringon Mall_Customers_data.csv. draw the dendogram.

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

# Load Mall_Customers_data.csv
data = pd.read_csv('/content/Mall_Customers.csv')

# Extract relevant columns (e.g., Annual Income and Spending Score)
# Adjust column names based on your dataset
features = data[['Annual Income (k$)', 'Spending Score (1-100)']].values

# Compute Euclidean distance between two points
def euclidean_distance(p1, p2):
    return np.sqrt(np.sum((p1 - p2) ** 2))

# Perform hierarchical clustering
def hierarchical_clustering(data):
    n = len(data)
    clusters = {i: [i] for i in range(n)}  # Each point is its own cluster
```

```
    distances = [[euclidean_distance(data[i], data[j]) for j in
range(n)] for i in range(n)]
    merges = []

    while len(clusters) > 1:
        # Find closest clusters
        min_dist = float('inf')
        to_merge = None
        for i in clusters:
            for j in clusters:
                if i < j:
                    dist = min(distances[p1][p2] for p1 in clusters[i]
for p2 in clusters[j])
                    if dist < min_dist:
                        min_dist = dist
                        to_merge = (i, j)

        # Merge clusters
        i, j = to_merge
        merges.append((i, j, min_dist))
        clusters[i].extend(clusters[j])
        del clusters[j]

    return merges

# Plot dendrogram
def plot_dendrogram(merges):
    plt.figure(figsize=(8, 5))
    current_positions = {i: i for i in range(len(merges) + 1)}
    for i, (a, b, height) in enumerate(merges):
        x1, x2 = current_positions[a], current_positions[b]
        x_mid = (x1 + x2) / 2
        plt.plot([x1, x1, x2, x2], [0, height, height, 0], 'b')
```

```
        current_positions[a] = x_mid  # Update cluster position
        del current_positions[b]
    plt.xlabel("Data Points")
    plt.ylabel("Distance")
    plt.title("Dendrogram")
    plt.show()

# Run and plot
merges = hierarchical_clustering(features)
plot_dendrogram(merges)
```

7.Find the optimal hyperplane for SVM use the following data set
positive class:(3,1),(3,-1),(6,1),(6,-1)
Negative Class:(1,0),(0,1),(0,-1),(-1,0)

```
import numpy as np

# Dataset: Positive and Negative classes
positive_class = np.array([[3, 1], [3, -1], [6, 1], [6, -1]])
negative_class = np.array([[1, 0], [0, 1], [0, -1], [-1, 0]])

# Combine data and labels
X = np.vstack((positive_class, negative_class))
y = np.hstack((np.ones(len(positive_class)),
-np.ones(len(negative_class))))

# Helper functions for SVM
def compute_svm(X, y):
    n_samples, n_features = X.shape
```

```
    # Initialize weights and bias
    w = np.zeros(n_features)
    b = 0
    lr = 0.01  # Learning rate
    epochs = 1000

    # Gradient Descent for optimization
    for _ in range(epochs):
        for i in range(n_samples):
            if y[i] * (np.dot(w, X[i]) + b) < 1:  # Misclassified
                w += lr * (y[i] * X[i] - 2 * (1 / epochs) * w)
                b += lr * y[i]
            else:  # Correct classification
                w -= lr * (2 * (1 / epochs) * w)

    return w, b

# Solve for the optimal hyperplane
w, b = compute_svm(X, y)

# Decision boundary equation
print(f"Optimal weight vector: {w}")
print(f"Optimal bias: {b}")
print(f"Decision boundary: {w[0]} * x1 + {w[1]} * x2 + {b} =
0")

# Plot the data and decision boundary
import matplotlib.pyplot as plt

# Plot data points
plt.scatter(positive_class[:, 0], positive_class[:, 1],
color='blue', label='Positive Class')
```

```
plt.scatter(negative_class[:, 0], negative_class[:, 1],
color='red', label='Negative Class')

# Plot decision boundary
x1 = np.linspace(-2, 7, 100)
x2 = -(w[0] * x1 + b) / w[1]
plt.plot(x1, x2, color='green', label='Decision Boundary')

plt.xlabel('x1')
plt.ylabel('x2')
plt.title('SVM Decision Boundary')
plt.legend()
plt.grid()
plt.show()
```