```
import numpy as np
import pandas as pd

def polynomial_regression(x, y, degree):
    # Construct the design matrix
    X = np.vander(x, degree + 1, increasing=True)
    # Compute coefficients using the normal equation
    coeff = np.linalg.inv(X.T @ X) @ X.T @ y
    return coeff

# Example usage
data = pd.read_csv('/content/Data1.csv')
x = data['X1'].values
y = data['Y'].values
degree = 2

coefficients = polynomial_regression(x, y, degree)
print("Coefficients:", coefficients)
```

```
# 2. Logistic Regression (Pima Indian Diabetes Dataset)
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

def logistic_regression(X, y, learning_rate, iterations):
    n, m = X.shape
    w = np.zeros(m)
    b = 0
    for _ in range(iterations):
        linear_model = X @ w + b
        predictions = sigmoid(linear_model)
        # Gradient descent
        dw = (1 / n) * (X.T @ (predictions - y))
        db = (1 / n) * np.sum(predictions - y)
        w -= learning_rate * dw
        b -= learning_rate * db
    return w, b

data = pd.read_csv('/content/diabetes.csv')
X = data.iloc[:, :-1].values
y = data.iloc[:, -1].values

X = (X - np.mean(X, axis=0)) / np.std(X, axis=0)
learning_rate = 0.01
iterations = 1000

weights, bias = logistic_regression(X, y, learning_rate, iterations)
print("Weights:", weights)
print("Bias:", bias)
```

```
# 3. Logistic Regression (Digits Classification)
# Logistic Regression for Digits Classification
def predict(X, weights, bias):
    z = np.dot(X, weights) + bias
    y_pred = sigmoid(z)
    return (y_pred > 0.5).astype(int)

# Training data (X_train: features, y_train: labels)
X_train = np.array([[1, 2], [2, 1], [2, 3], [3, 4], [4, 3]])
y_train = np.array([0, 0, 1, 1, 1])

# Train the logistic regression model
learning_rate = 0.01
iterations = 1000
weights, bias = logistic_regression(X_train, y_train, learning_rate,
iterations)

# Test data for predictions
X_test = np.array([[1, 1], [4, 4]])
predictions = predict(X_test, weights, bias)

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

```
# 4. K-Means Clustering
def k_means(x, y, k, iterations):
    points = np.array(list(zip(x, y)))
    centroids = points[np.random.choice(len(points), k, replace=False)]

    for _ in range(iterations):
        distances = np.linalg.norm(points[:, None] - centroids, axis=2)
        clusters = np.argmin(distances, axis=1)
        centroids = np.array([points[clusters == i].mean(axis=0) for i in
range(k)])
    return centroids, clusters

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. Hierarchical Clustering
def euclidean_distance(p1, p2):
    return np.linalg.norm(p1 - p2)

def hierarchical_clustering(data):
    clusters = {i: [i] for i in range(len(data))}
    distances = np.array([[euclidean_distance(data[i], data[j]) for j in
range(len(data))] for i in range(len(data))])
    np.fill_diagonal(distances, np.inf)
    merges = []

    while len(clusters) > 1:
        i, j = divmod(np.argmin(distances), distances.shape[1])
        merges.append((i, j, distances[i, j]))
        distances = np.delete(np.delete(distances, j, axis=0), j, axis=1)
        distances = np.vstack((distances, [np.inf] * (len(distances[0]) + 1)))
    return merges

# Dendrogram Plot
def plot_dendrogram(merges):
    from scipy.cluster.hierarchy import dendrogram
    import matplotlib.pyplot as plt
    plt.figure()
    dendrogram(merges)
    plt.show()

data = pd.read_csv('/content/Mall_Customers.csv')
features = data[['Annual Income (k$)', 'Spending Score (1-100)']].values
merges = hierarchical_clustering(features)
plot_dendrogram(merges)
```

```
# 6. SVM (Optimal Hyperplane)
def compute_svm(X, y):
    n_samples, n_features = X.shape
    w = np.zeros(n_features)
    b = 0
    lr = 0.01
    epochs = 1000

    for _ in range(epochs):
        for i in range(n_samples):
            if y[i] * (np.dot(w, X[i]) + b) < 1:
                w += lr * (y[i] * X[i] - 2 * w)
                b += lr * y[i]
            else:
                w -= lr * 2 * w
    return w, b

positive_class = np.array([[3, 1], [3, -1], [6, 1], [6, -1]])
negative_class = np.array([[1, 0], [0, 1], [0, -1], [-1, 0]])
X = np.vstack((positive_class, negative_class))
y = np.hstack((np.ones(len(positive_class)), -
np.ones(len(negative_class))))

w, b = compute_svm(X, y)
print(f"Optimal weight vector: {w}")
print(f"Optimal bias: {b}")
```