question1

November 6, 2024

0.1 Importing Libraries

[3]:
```
import numpy as np #for working with arrays
import matplotlib.pyplot as plt #for data visualization
from sklearn.model_selection import train_test_split #for splitting the dataset␣
↪into training and testing sets
from sklearn.metrics import accuracy_score, confusion_matrix #performance␣
↪metrics
import pandas as pd #for data manipulation and analysis
```

##Generating dataset

[4]:
```
# Step 1: Data Generation
np.random.seed(42)  # For reproducibility

# Parameters
n_samples = 3000
n_features = 2
n_classes = 2

# Generate random samples for each class
n1 = np.random.randint(1000, 2000)  # Random number of samples in class 1
n2 = n_samples - n1  # Remaining samples in class 2

# Means and covariances for each class
mu1 = np.random.uniform(0, 1, 2)
mu2 = np.random.uniform(3, 4, 2)
cov1 = np.array([[1, 0.5],[0.5, 1]])
cov2 = np.array([[1, -0.5],[-0.5, 1]])

# Generate data for each class [X = inputs & y = Labels]
X_class1 = np.random.multivariate_normal(mu1, cov1, n1)  # Class 1
X_class2 = np.random.multivariate_normal(mu2, cov2, n2)  # Class 2

# Labels for Class 1 & 2
y_class1 = np.zeros(n1)
y_class2 = np.ones(n2)
```

```
print(f"Class sample sizes: n1 = {n1}, n2 = {n2}")
print("Shape of Input Data:")
print(f"X_class1: {X_class1.shape},\n X_class2: {X_class2.shape}")
print("Shape of Labels:")
print(f"y_class1: {y_class1.shape},\n y_class2: {y_class2.shape}")
print("Mean of Input Data:")
print(f"X_class1 Mean: {np.mean(X_class1, axis=0)},\n X_class2 Mean: {np.
 ↪mean(X_class2, axis=0)}")
print("Covariance of Input Data:")
print(f"X_class1 Covariance:\n {np.cov(X_class1.T)},\n X_class2 Covariance:\n␣
 ↪{np.cov(X_class2.T)}")
```

```
Class sample sizes: n1 = 1102, n2 = 1898
Shape of Input Data:
X_class1: (1102, 2),
 X_class2: (1898, 2)
Shape of Labels:
y_class1: (1102,),
 y_class2: (1898,)
Mean of Input Data:
X_class1 Mean: [0.75288683 0.19327365],
 X_class2 Mean: [3.78989204 3.60738571]
Covariance of Input Data:
X_class1 Covariance:
 [[1.01272444 0.50473865]
 [0.50473865 1.04591071]],
 X_class2 Covariance:
 [[ 0.99290031 -0.49550751]
 [-0.49550751  1.01004166]]
```

0.2 Combining two data classes and their labels to form Data for Classification

[5]:
```
X = np.vstack((X_class1, X_class2))  # Combined data from both classes
y = np.hstack((y_class1, y_class2))  # Combined labels from both classes
print(f"Shape of Input Data: {X.shape}")
print(f"Shape of Labels: {y.shape}")
```

```
Shape of Input Data: (3000, 2)
Shape of Labels: (3000,)
```

0.3 Data Splitting

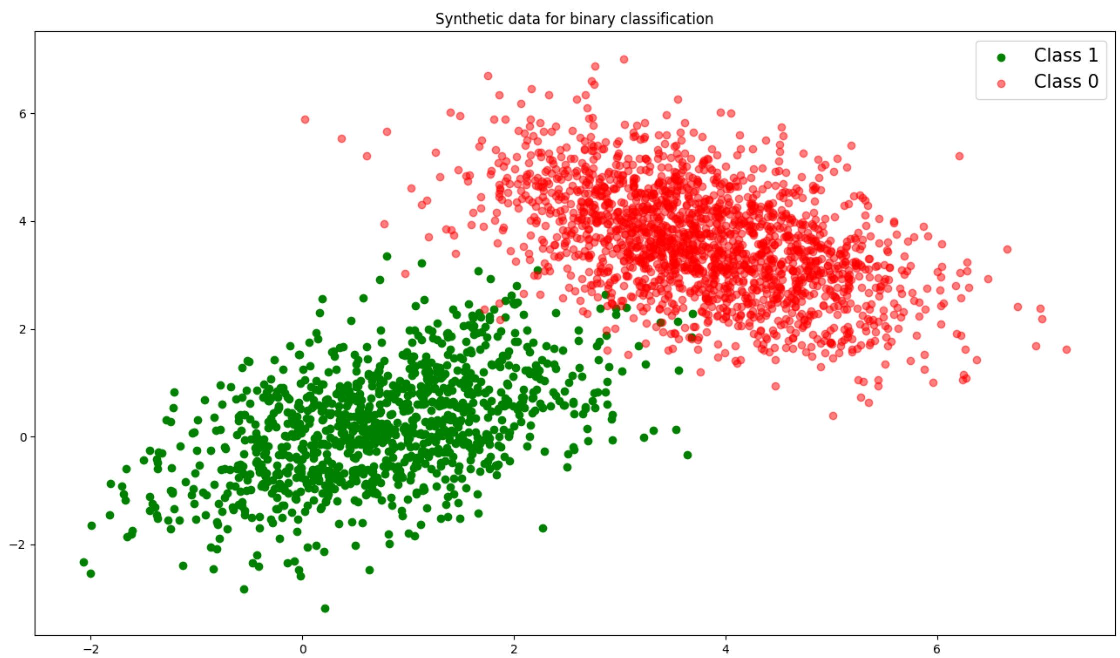
[6]:
```
# Split the dataset into training and testing sets
s = np.random.uniform(0, 0.3)  # Random test set percentage
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=s,␣
 ↪random_state=42)
```

```
# Print the shapes of the resulting sets
print(f"Shape of Training Data: {X_train.shape}")
print(f"Shape of Testing Data: {X_test.shape}")
```

```
Shape of Training Data: (2733, 2)
Shape of Testing Data: (267, 2)
```

0.4 Data Visualization

[7]:
```
plt.figure(figsize = (16, 9))
plt.scatter(X_class1[:, 0], X_class1[:, 1], color = 'green', label = 'Class 1')
plt.scatter(X_class2[:, 0], X_class2[:, 1], color = 'red', alpha = 0.5, label␣
 ↪='Class 0')
plt.title('Synthetic data for binary classification')
plt.legend(fontsize = 15)
plt.show()
```



0.5 LDA and QDA

[8]:
```
# Importing necessary library
import numpy as np

# Class for Linear Discriminant Analysis (LDA)
class LDA:
    # Method to train the LDA model
```

```python
    def fit(self, X, y):
        # Get unique classes from labels and store them
        self.classes = np.unique(y)

        # Dictionaries to hold the mean vector and prior probability for each class
        self.means = {}
        self.priors = {}

        # Calculate mean vectors and prior probabilities for each class
        for cls in self.classes:
            # Extract data points belonging to the current class
            X_cls = X[y == cls]

            # Calculate and store the mean vector of the current class
            self.means[cls] = np.mean(X_cls, axis=0)

            # Calculate and store the prior probability of the current class
            self.priors[cls] = len(X_cls) / len(y)

        # Compute the covariance matrix across the entire dataset
        self.cov_matrix = np.cov(X, rowvar=False)

        # Calculate and store the inverse of the covariance matrix for later use
        self.inv_cov_matrix = np.linalg.inv(self.cov_matrix)

    # Method to make predictions for new data points
    def predict(self, X):
        predictions = []  # List to store predictions for each data point

        # Iterate over each data point to classify
        for x in X:
            scores = []  # List to hold scores for each class

            # Calculate the score for each class
            for cls in self.classes:
                # Difference between the data point and the mean of the current class
                mean_diff = x - self.means[cls]

                # Calculate the score based on the LDA formula
                score = (
                    -0.5 * mean_diff @ self.inv_cov_matrix @ mean_diff.T
                    + np.log(self.priors[cls])
                )

                # Append the calculated score for the current class
```

```
                scores.append(score)

            # Choose the class with the highest score as the prediction
            predictions.append(np.argmax(scores))

        # Return predictions as a NumPy array
        return np.array(predictions)

# Class for Quadratic Discriminant Analysis (QDA)
class QDA:
    # Method to train the QDA model
    def fit(self, X, y):
        # Get unique classes from labels and store them
        self.classes = np.unique(y)

        # Dictionaries to hold the mean vector, covariance matrix, and prior␣
 ↪probability for each class
        self.means = {}
        self.cov_matrices = {}
        self.inv_cov_matrices = {}
        self.priors = {}

        # Calculate mean vectors, covariance matrices, and prior probabilities␣
 ↪for each class
        for cls in self.classes:
            # Extract data points belonging to the current class
            X_cls = X[y == cls]

            # Calculate and store the mean vector of the current class
            self.means[cls] = np.mean(X_cls, axis=0)

            # Calculate and store the covariance matrix for the current class
            self.cov_matrices[cls] = np.cov(X_cls, rowvar=False)

            # Store the inverse of the covariance matrix for the current class
            self.inv_cov_matrices[cls] = np.linalg.inv(self.cov_matrices[cls])

            # Calculate and store the prior probability of the current class
            self.priors[cls] = len(X_cls) / len(y)

    # Method to make predictions for new data points
    def predict(self, X):
        predictions = []  # List to store predictions for each data point

        # Iterate over each data point to classify
        for x in X:
            scores = []  # List to hold scores for each class
```

```
            # Calculate the score for each class
            for cls in self.classes:
                # Difference between the data point and the mean of the current␣
↪class
                mean_diff = x - self.means[cls]

                # Calculate the score based on the QDA formula
                score = (
                    -0.5 * np.log(np.linalg.det(self.cov_matrices[cls]))
                    - 0.5 * mean_diff @ self.inv_cov_matrices[cls] @ mean_diff.T
                    + np.log(self.priors[cls])
                )

                # Append the calculated score for the current class
                scores.append(score)

            # Choose the class with the highest score as the prediction
            predictions.append(np.argmax(scores))

        # Return predictions as a NumPy array
        return np.array(predictions)
```

0.6 Training of LDA & QDA Model

[9]:
```
lda_model = LDA()
lda_model.fit(X_train, y_train)

qda_model = QDA()
qda_model.fit(X_train, y_train)
```

0.7 Predictions on Test set and Train set

[10]:
```
lda_pred_test = lda_model.predict(X_test)
lda_pred_train = lda_model.predict(X_train)

qda_pred_test = qda_model.predict(X_test)
qda_pred_train = qda_model.predict(X_train)
```

##Model Evaluation

[11]:
```
#LDA
# Test set
lda_test_cm = confusion_matrix(y_test, lda_pred_test)
lda_test_accuracy = accuracy_score(y_test, lda_pred_test)
#Train set
lda_train_cm = confusion_matrix(y_train, lda_pred_train)
```

```
lda_train_accuracy = accuracy_score(y_train, lda_pred_train)

#QDA
# Test set
qda_test_cm = confusion_matrix(y_test, qda_pred_test)
qda_test_accuracy = accuracy_score(y_test, qda_pred_test)
#Train set
qda_train_cm = confusion_matrix(y_train, qda_pred_train)
qda_train_accuracy = accuracy_score(y_train, qda_pred_train)

metrics_data = {
    "Model": ["LDA", "LDA", "QDA", "QDA"],
    "Dataset": ["Train", "Test", "Train", "Test"],
    "Confusion Matrix": [lda_train_cm, lda_test_cm, qda_train_cm, qda_test_cm],
    "Accuracy": [lda_train_accuracy, lda_test_accuracy, qda_train_accuracy,␣
 ↪qda_test_accuracy]
}

# Create a DataFrame for better visualization
metrics_df = pd.DataFrame(metrics_data)

# Display the metrics table
print("LDA and QDA Metrics Comparison Table")
print(metrics_df)

# Compare LDA and QDA
print(f"LDA vs QDA Accuracy Comparison:")
if lda_test_accuracy > qda_test_accuracy:
    print("LDA has a higher accuracy.")
elif lda_test_accuracy < qda_test_accuracy:
    print("QDA has a higher accuracy.")
else:
    print("Both LDA and QDA have the same accuracy.")
```

```
LDA and QDA Metrics Comparison Table
  Model Dataset         Confusion Matrix  Accuracy
0   LDA   Train  [[888, 108], [0, 1737]]  0.960483
1   LDA    Test     [[96, 10], [0, 161]]  0.962547
2   QDA   Train   [[982, 14], [8, 1729]]  0.991950
3   QDA    Test     [[103, 3], [0, 161]]  0.988764
LDA vs QDA Accuracy Comparison:
QDA has a higher accuracy.
```

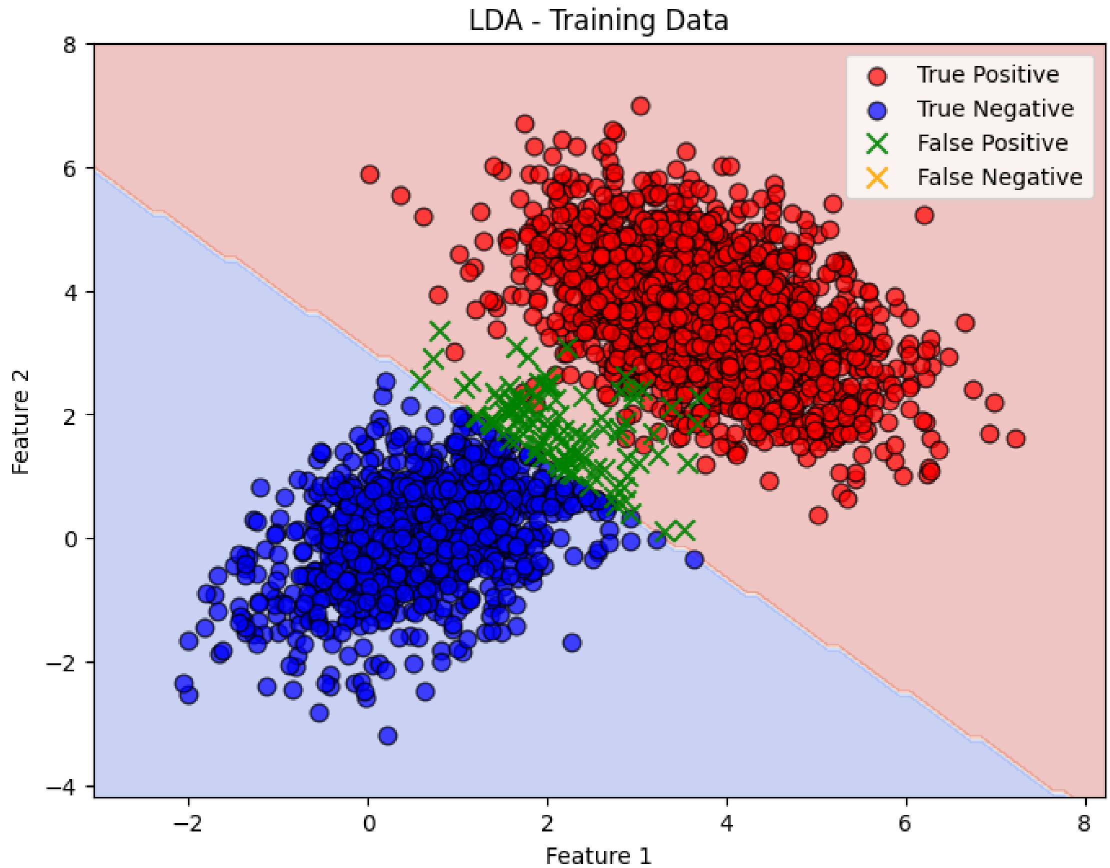
##Plotting the Results

[17]:
```
def plot_results(model, X, y_true, y_pred, title):
    # Define the plot boundaries based on feature ranges
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1  # X-axis range
```

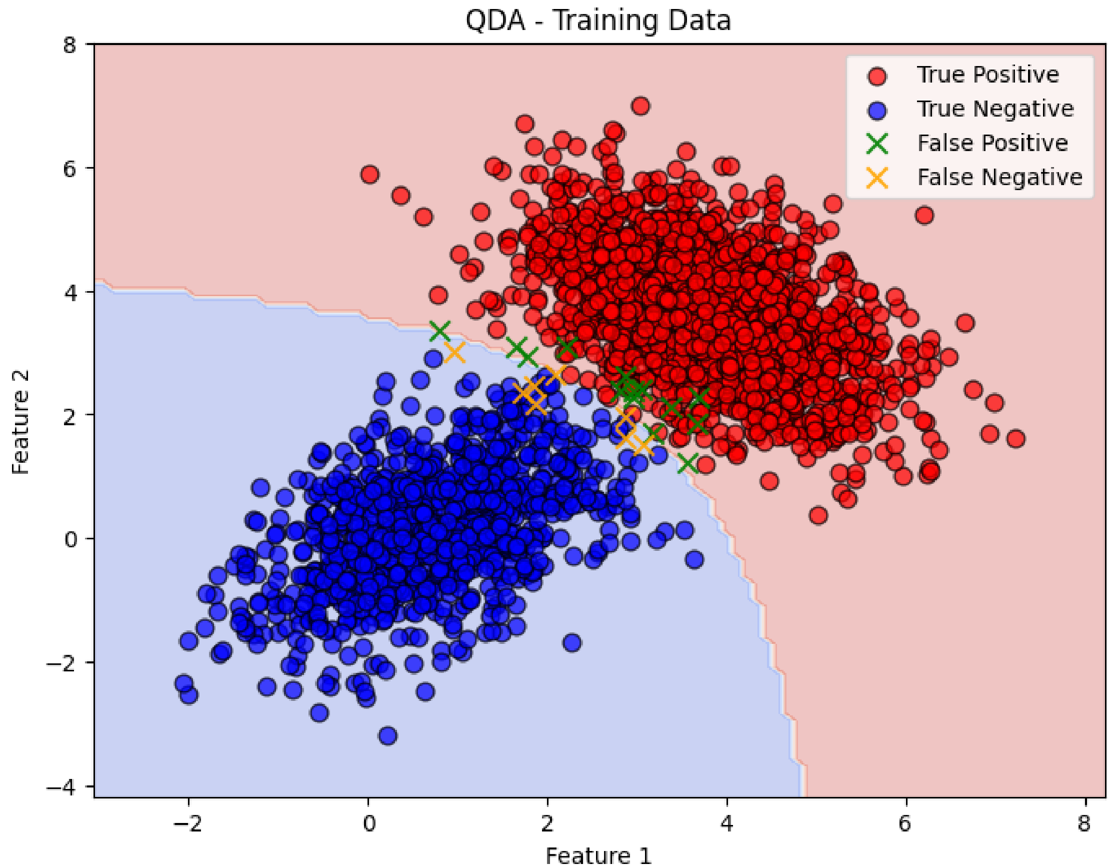
```python
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1  # Y-axis range

    # Create a mesh grid over the feature space to visualize the decision␣
↪boundary
    xx, yy = np.meshgrid(np.linspace(x_min, x_max, 100),
                         np.linspace(y_min, y_max, 100))

    # Predict the class for each point in the grid
    Z = model.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)  # Reshape predictions to match the grid dimensions

    # Begin plotting the decision boundary
    plt.figure(figsize=(8, 6))
    plt.contourf(xx, yy, Z, alpha=0.3, cmap=plt.cm.coolwarm)  # Decision␣
↪boundary in shaded regions

    # Classify data points by comparing true labels and predictions
    tp = (y_true == 1) & (y_pred == 1)  # True Positives
    tn = (y_true == 0) & (y_pred == 0)  # True Negatives
    fp = (y_true == 0) & (y_pred == 1)  # False Positives
    fn = (y_true == 1) & (y_pred == 0)  # False Negatives

    # Plot each type of point with different colors and markers for clear␣
↪distinction
    plt.scatter(X[tp, 0], X[tp, 1], color='red', marker='o', edgecolor='k',␣
↪label='True Positive', s=60, alpha=0.7)
    plt.scatter(X[tn, 0], X[tn, 1], color='blue', marker='o', edgecolor='k',␣
↪label='True Negative', s=60, alpha=0.7)
    plt.scatter(X[fp, 0], X[fp, 1], color='green', marker='x', label='False␣
↪Positive', s=80, alpha=0.9)
    plt.scatter(X[fn, 0], X[fn, 1], color='orange', marker='x', label='False␣
↪Negative', s=80, alpha=0.9)

    # Add labels, title, and legend for clarity
    plt.title(title)
    plt.xlabel("Feature 1")  # Label for X-axis
    plt.ylabel("Feature 2")  # Label for Y-axis
    plt.legend(loc='upper right')  # Legend showing point classifications
    plt.show()  # Display the plot
```

##Training set plot

[18]:
```python
plot_results(lda_model, X_train, y_train, lda_pred_train, "LDA - Training Data")
```

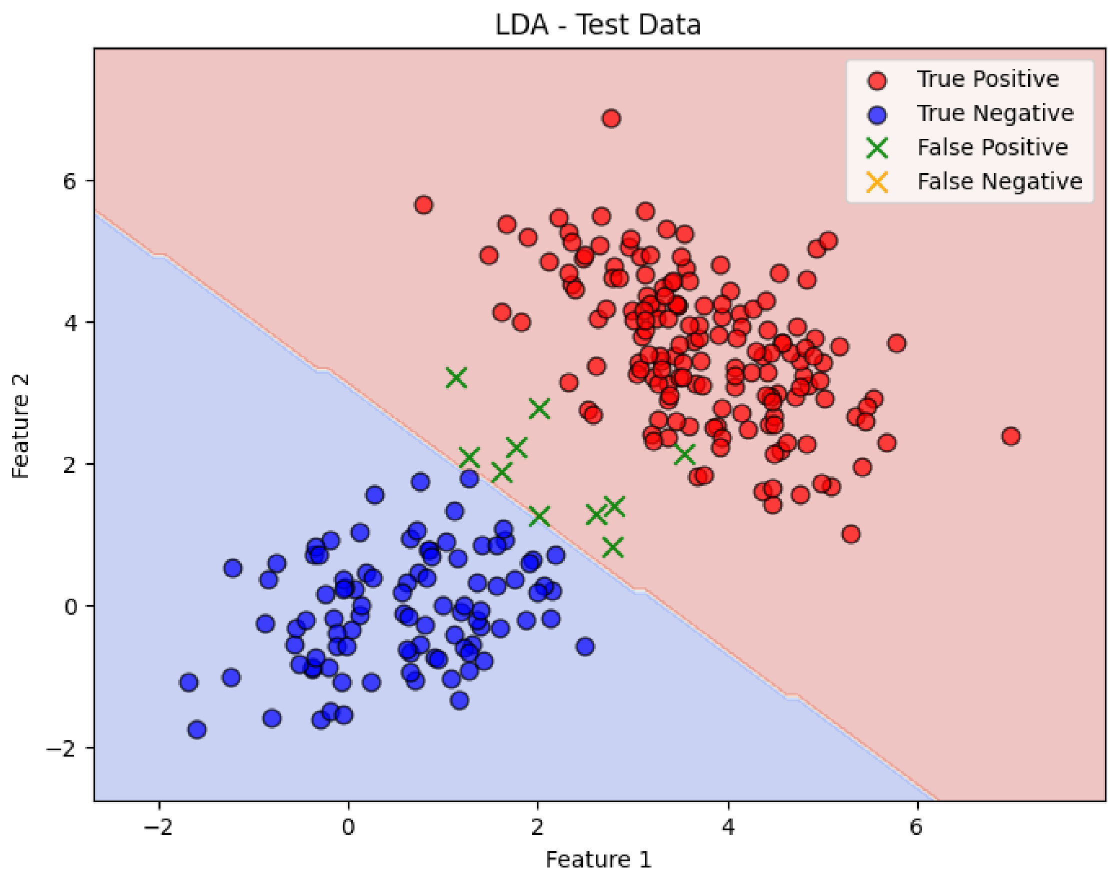


[14]:
```
plot_results(qda_model, X_train, y_train, qda_pred_train, "QDA - Training Data")
```

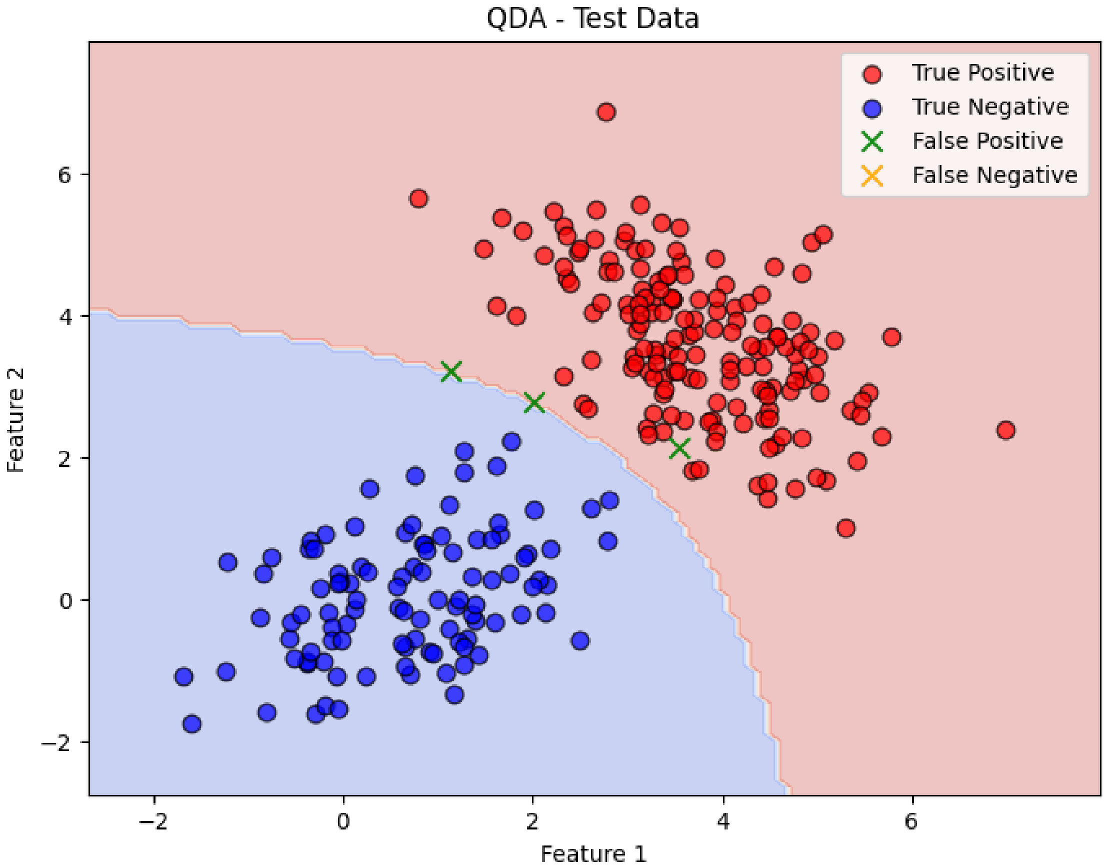


##Test set Plot

[15]:
```
plot_results(lda_model, X_test, y_test, lda_pred_test, "LDA - Test Data")
```



[16]:
```
plot_results(qda_model, X_test, y_test, qda_pred_test, "QDA - Test Data")
```



QDA - Test Data
True Positive
True Negative
False Positive
False Negative
Feature 2
Feature 1
−2
0
2
4
6

question2-1

November 6, 2024

##Importing Libraries

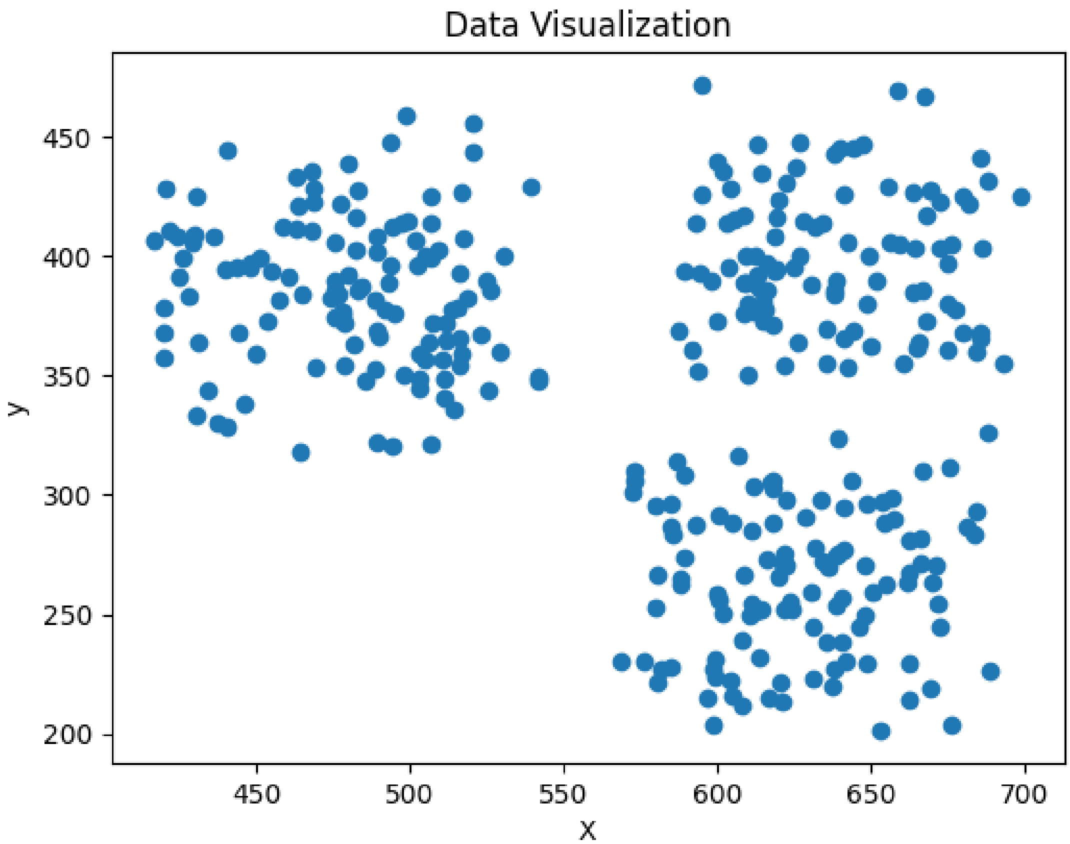
[1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

0.1 Loading Data set

[2]:
```
dataset = pd.read_csv('Dataset 1.csv')
X = dataset[['x', 'y']].values
```

##Data Visualization

[3]:
```
plt.scatter(X[:, 0], X[:, 1])
plt.xlabel('X')
plt.ylabel('y')
plt.title('Data Visualization')
plt.show()
```



##K-Mean Clustering using Eucledian distance

[4]:
```
# Function to calculate the Euclidean distance between two points in space
def euclidean_distance(point1, point2):
    # Sum the squared differences between corresponding coordinates, then take␣
 ↪the square root
    return np.sqrt(np.sum((point1 - point2) ** 2))

# Class definition for the KMeans clustering algorithm
class KMeans:
    # Initialize the KMeans model with specified parameters
    def __init__(self, k, max_iters=100, tolerance=1e-4):
        # k is the number of clusters
        self.k = k
        # max_iters is the maximum number of iterations to run the algorithm
        self.max_iters = max_iters
        # tolerance is the convergence threshold for centroid movement
        self.tolerance = tolerance
        # centroids will store the current centroids of the clusters
        self.centroids = None
```

```python
    # Method to randomly initialize centroids from the data points
    def initialize_centroids(self, X):
        np.random.seed(42)  # Ensure reproducibility
        # Randomly choose k data points to serve as initial centroids
        random_indices = np.random.choice(X.shape[0], self.k, replace=False)
        self.centroids = X[random_indices]

    # Method to assign each data point to the nearest centroid
    def assign_clusters(self, X):
        clusters = []  # List to store the assigned cluster of each point
        for point in X:
            # Calculate the distance of the point to each centroid
            distances = [euclidean_distance(point, centroid) for centroid in self.centroids]
            # Find the index of the closest centroid
            closest_centroid = np.argmin(distances)
            # Assign this point to the closest centroid's cluster
            clusters.append(closest_centroid)
        return np.array(clusters)  # Return clusters as a numpy array

    # Method to update centroids based on the mean of points in each cluster
    def update_centroids(self, X, clusters):
        new_centroids = np.zeros((self.k, X.shape[1]))  # Array to store the updated centroids
        for i in range(self.k):
            # Get all points assigned to the current cluster
            points_in_cluster = X[clusters == i]
            if len(points_in_cluster) > 0:  # Avoid division by zero
                # Update centroid as the mean of points in the cluster
                new_centroids[i] = np.mean(points_in_cluster, axis=0)
        return new_centroids

    # Main method to run the KMeans algorithm
    def fit(self, X):
        self.initialize_centroids(X)  # Start with random centroids
        iteration_count = 0  # Counter to keep track of iterations

        for iteration in range(self.max_iters):
            iteration_count += 1  # Increment iteration count

            # Step 1: Assign each point to the nearest centroid
            clusters = self.assign_clusters(X)

            # Step 2: Update centroids by taking the mean of assigned points
            new_centroids = self.update_centroids(X, clusters)
```

```
            # Step 3: Calculate the error as the total shift of centroids
            error = np.sum([euclidean_distance(new_centroids[i], self.
↪centroids[i]) for i in range(self.k)])

            # Display the current state of centroids and error for this␣
↪iteration
            print(f"Iteration {iteration + 1}, Centroids:
↪\n{new_centroids}\nError: {error}")
            print("******************************")

            # Step 4: Check for convergence based on tolerance
            if np.all(np.abs(new_centroids - self.centroids) < self.tolerance):
                print("Convergence reached.")
                break  # Stop if centroids have stopped moving significantly

            # Update centroids for the next iteration
            self.centroids = new_centroids

        # Print the total number of iterations performed
        print(f"Total iterations performed: {iteration_count}")
        return clusters, self.centroids  # Return the final clusters and␣
↪centroids
```

##Fitting the Model with K=2

[5]:
```
kmeans_2 = KMeans(k=2)
clusters_2, centroids_2 = kmeans_2.fit(X)
print(f'Centroids: {centroids_2}')
print('No of Centroids:', len(centroids_2))
```

```
Iteration 1, Centroids:
[[436.85009836 397.06266565]
 [590.35384506 344.47242345]]
Error: 137.1438861887892
******************************
Iteration 2, Centroids:
[[473.29706955 389.32879568]
 [622.99435675 331.18877832]]
Error: 72.49849058595521
******************************
Iteration 3, Centroids:
[[480.60429976 385.44506433]
 [631.4147117  328.50696986]]
Error: 17.112310659806475
******************************
Iteration 4, Centroids:
[[481.53950939 386.15485318]
```

```
 [631.58627636 327.83625802]]
Error: 1.8663670327804245
******************************
Iteration 5, Centroids:
[[481.53950939 386.15485318]
 [631.58627636 327.83625802]]
Error: 0.0
******************************
Convergence reached.
Total iterations performed: 5
Centroids: [[481.53950939 386.15485318]
 [631.58627636 327.83625802]]
No of Centroids: 2
```

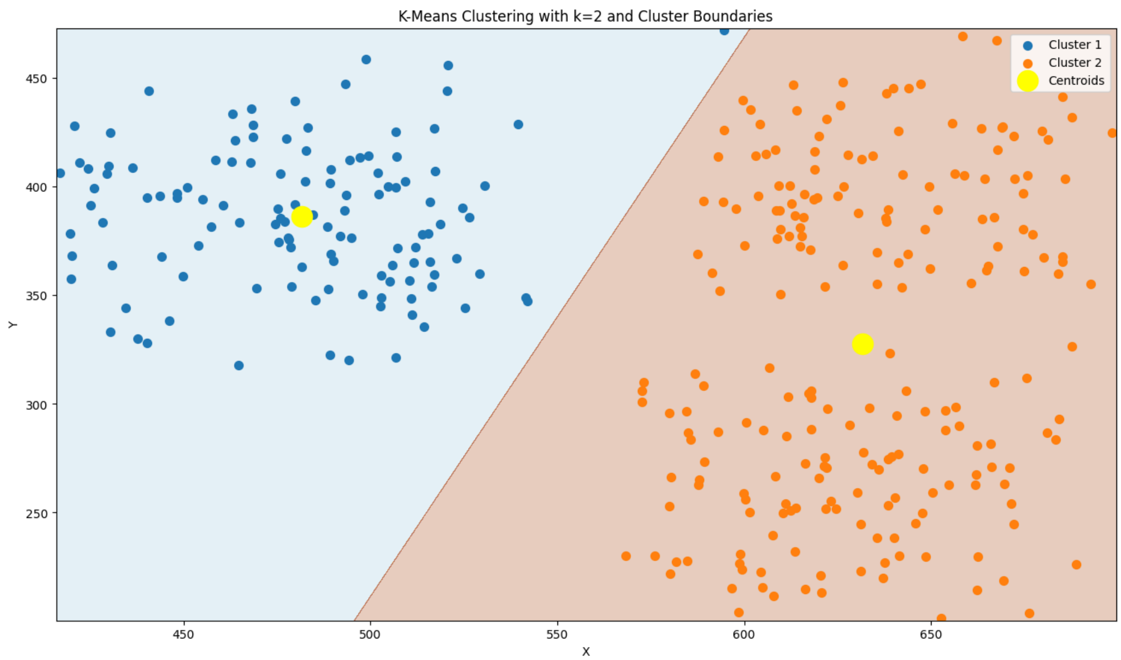
##Function for Plotting Clusters and their Boundaries

[6]:
```python
def plot_clusters_with_boundaries(X, clusters, centroids, k, model):
    plt.figure(figsize=(16, 9))

    # Defining the x and y range for the plot by expanding slightly beyond min␣
↪and max values of X
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1

    # Creating a mesh grid over the defined x and y ranges with a step size of␣
↪0.1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),     # This will help in␣
↪plotting decision boundaries
                         np.arange(y_min, y_max, 0.1))

    # For each point in the mesh grid,we predict the cluster it belongs to␣
↪using the model
    # This will allow us to visualize the boundaries of each cluster
    Z = np.array([model.assign_clusters(np.array([[x, y]])) for x, y in zip(xx.
↪ravel(), yy.ravel())])
    Z = Z.reshape(xx.shape)  # Reshape Z to match the grid shape

    # Use a filled contour plot to shade the regions based on cluster assignment
    # The color of each region corresponds to a cluster
    plt.contourf(xx, yy, Z, alpha=0.3, cmap=plt.cm.Paired)

    # Plotting each cluster's data points with a unique color and add a label␣
↪for each
    for i in range(k):
        plt.scatter(X[clusters == i, 0], X[clusters == i, 1], s=50,␣
↪label=f'Cluster {i+1}')
```

```
    plt.scatter(centroids[:, 0], centroids[:, 1], s=300, c='yellow',␣
 ↪label='Centroids')
    plt.title(f'K-Means Clustering with k={k} and Cluster Boundaries')
    plt.xlabel('X')  # Label for x-axis
    plt.ylabel('Y')  # Label for y-axis
    plt.legend()
    plt.show()
```

##Cluster Visualization for K=2

[7]:
```
plot_clusters_with_boundaries(X, clusters_2, centroids_2, 2, kmeans_2)
```



##Model Fitting with K=3

[8]:
```
kmeans_3 = KMeans(k=3)
clusters_3, centroids_3 = kmeans_3.fit(X)
print(f'Centroids: {centroids_3}')
print('No of Centroids:', len(centroids_3))
```

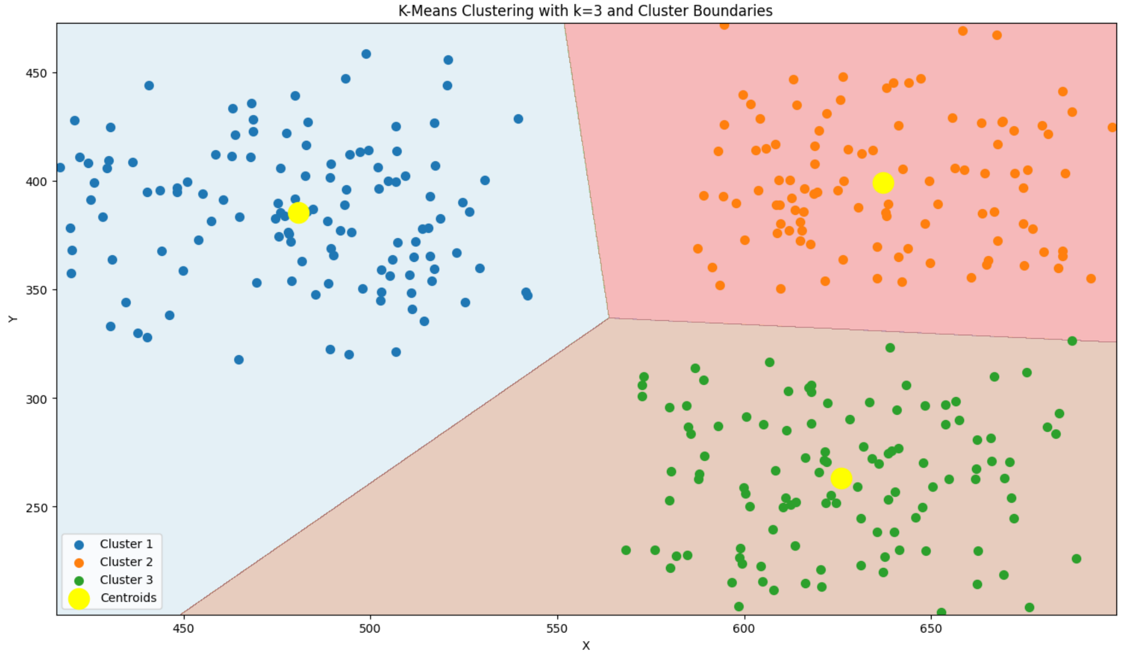
```
Iteration 1, Centroids:
[[436.85009836 397.06266565]
 [495.47491713 382.75349231]
 [631.58627636 327.83625802]]
Error: 82.80811926383538
******************************
Iteration 2, Centroids:
```

```
[[442.04914063 392.26027523]
 [500.8031039  383.17659852]
 [631.58627636 327.83625802]]
Error: 12.422600050682714
******************************
Iteration 3, Centroids:
[[442.60792294 392.05299378]
 [503.55657298 384.36866053]
 [631.9111457  326.84471573]]
Error: 4.639829315249871
******************************
Iteration 4, Centroids:
[[442.60792294 392.05299378]
 [505.76388003 385.31919545]
 [632.24013639 325.91212831]]
Error: 3.3921886370292067
******************************
Iteration 5, Centroids:
[[445.2040844  390.52420341]
 [507.44038081 385.83599018]
 [632.24013639 325.91212831]]
Error: 4.767194706875108
******************************
Iteration 6, Centroids:
[[447.74496981 391.25330835]
 [511.57686975 386.4628234 ]
 [632.46734161 324.83594301]]
Error: 7.927045898419569
******************************
Iteration 7, Centroids:
[[450.51246483 390.26840113]
 [514.87511518 386.94711727]
 [632.6762152  324.50397994]]
Error: 6.6633478921412745
******************************
Iteration 8, Centroids:
[[453.1064424  390.63211022]
 [519.82110008 387.46944887]
 [632.91162917 323.52564138]]
Error: 8.599104638926445
******************************
Iteration 9, Centroids:
[[454.55252426 390.05663656]
 [526.20776669 389.51880664]
 [633.39233738 321.63913906]]
Error: 10.210578311000257
******************************
Iteration 10, Centroids:
```

```
[[458.44859253 387.61317794]
 [537.4782681  393.4084655 ]
 [634.12781935 318.98937332]]
Error: 19.271659816792983
******************************
Iteration 11, Centroids:
[[462.10807122 387.20788824]
 [559.45317716 399.84525281]
 [634.58614396 310.43170168]]
Error: 35.150017781833114
******************************
Iteration 12, Centroids:
[[471.75359602 384.9473471 ]
 [600.25858561 405.81922898]
 [634.29145461 291.72378304]]
Error: 69.85750540592497
******************************
Iteration 13, Centroids:
[[480.11326784 385.0826049 ]
 [633.21196371 402.91339965]
 [629.21118858 269.81876958]]
Error: 63.928425869256216
******************************
Iteration 14, Centroids:
[[480.60429976 385.44506433]
 [637.26607797 399.3318376 ]
 [626.0335445  263.37338614]]
Error: 13.206012013245694
******************************
Iteration 15, Centroids:
[[480.60429976 385.44506433]
 [637.26607797 399.3318376 ]
 [626.0335445  263.37338614]]
Error: 0.0
******************************
Convergence reached.
Total iterations performed: 15
Centroids: [[480.60429976 385.44506433]
 [637.26607797 399.3318376 ]
 [626.0335445  263.37338614]]
No of Centroids: 3
```

##Cluster Visualization for K=3

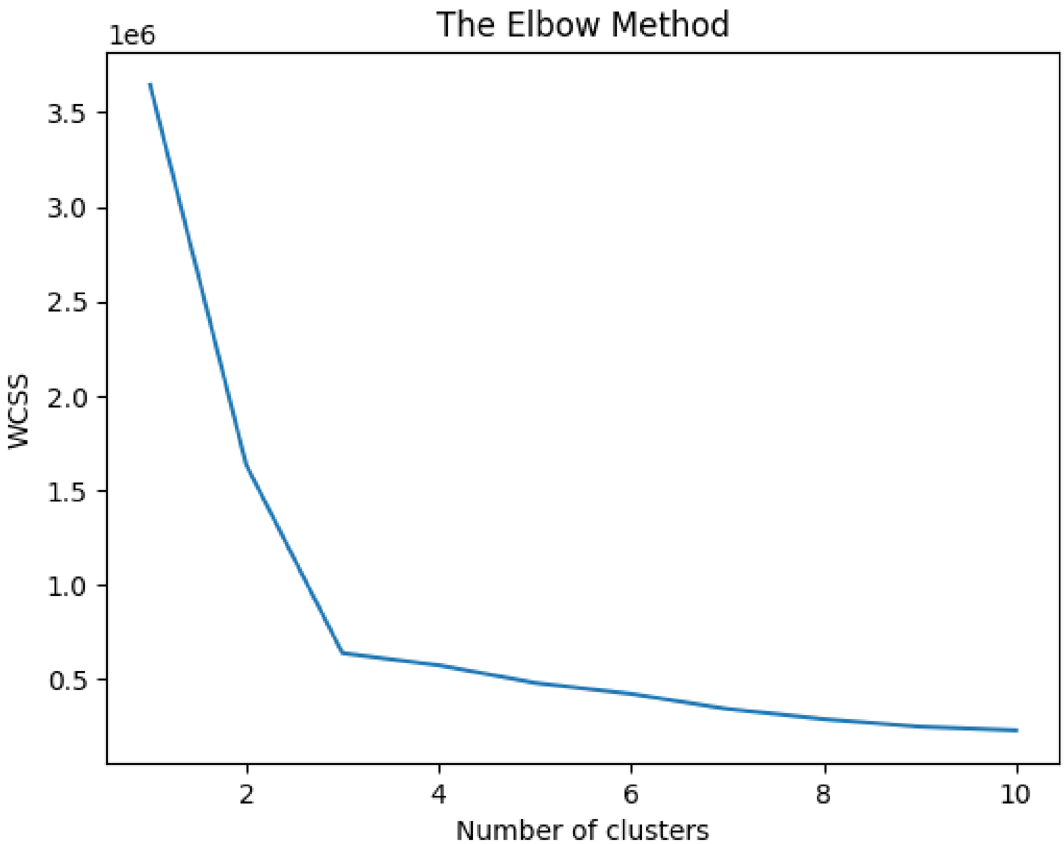
[9]:
```
plot_clusters_with_boundaries(X, clusters_3, centroids_3, 3, kmeans_3)
```



##Finding Optimal No of Clusters for K-means using Elbow Method in comparing k=2 vs k=3

[ ]:
```python
# Import necessary libraries
from sklearn.cluster import KMeans
import numpy as np
import matplotlib.pyplot as plt

# Initialize a list to store the Within-Cluster Sum of Squares (WCSS) for␣
↪different cluster counts
wcss = []

# Iterate over a range of cluster numbers from 1 to 10
for i in range(1, 11):
    # Create a KMeans clustering model with 'i' clusters
    kmeans = KMeans(n_clusters=i, init='k-means++', random_state=42)
    kmeans.fit(X)
    # Append the inertia (WCSS) to the list
    wcss.append(kmeans.inertia_)

# Plot the WCSS against the number of clusters
plt.plot(range(1, 11), wcss)
plt.title('The Elbow Method')
plt.xlabel('Number of clusters')
plt.ylabel('WCSS')
plt.show()
```

```
# Function to find the optimal number of clusters using the elbow method
def find_optimal_clusters(wcss):
    # Define points for the line segment from the first to the last WCSS value
    x1, y1 = 1, wcss[0]
    x2, y2 = len(wcss), wcss[-1]

    # List to store distances from each point to the line segment
    distances = []

    # Calculate the distance from each point in WCSS to the line segment
    for i in range(len(wcss)):
        x0, y0 = i + 1, wcss[i]  # Current point (x0, y0)

        # Calculate the numerator of the distance formula
        numerator = abs((y2 - y1) * x0 - (x2 - x1) * y0 + x2 * y1 - y2 * x1)
        # Calculate the denominator of the distance formula
        denominator = np.sqrt((y2 - y1) ** 2 + (x2 - x1) ** 2)

        # Append the calculated distance to the distances list
        distances.append(numerator / denominator)

    # Return the index of the maximum distance, which indicates the optimal␣
 ↪number of clusters
    return distances.index(max(distances)) + 1

# Find and print the optimal number of clusters
optimal_clusters = find_optimal_clusters(wcss)
print(f"The optimal number of clusters is: {optimal_clusters}")
```




```
The optimal number of clusters is: 3
```

question2-2

November 6, 2024

##Importing Libraries

[24]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

0.1 Loading Data set

[25]:
```
dataset = pd.read_csv('Dataset 2.csv')
X = dataset.iloc[:,:].values
```

##K-Mean Clustering using Eucledian distance

[26]:
```
# Euclidean distance function
def euclidean_distance(point1, point2):
    return np.sqrt(np.sum((point1 - point2) ** 2))

# KMeans algorithm class definition
class KMeans:
    def __init__(self, k, max_iters=100, tolerance=1e-4): #convergence␣
 ↪threshold, which controls when the algorithm should stop if the centroids␣
 ↪move less than this threshold
        self.k = k
        self.max_iters = max_iters
        self.tolerance = tolerance
        self.centroids = None

    def initialize_centroids(self, X):
        np.random.seed(42)
        random_indices = np.random.choice(X.shape[0], self.k, replace=False)
        self.centroids = X[random_indices]

    def assign_clusters(self, X):
        clusters = []
        for point in X:
            distances = [euclidean_distance(point, centroid) for centroid in␣
 ↪self.centroids]
            closest_centroid = np.argmin(distances)
```

```
            clusters.append(closest_centroid)
        return np.array(clusters)

    def update_centroids(self, X, clusters):
        new_centroids = np.zeros((self.k, X.shape[1]))
        for i in range(self.k):
            points_in_cluster = X[clusters == i]
            if len(points_in_cluster) > 0:
                new_centroids[i] = np.mean(points_in_cluster, axis=0)
        return new_centroids

    def fit(self, X):
        self.initialize_centroids(X)
        iteration_count = 0  # Track the number of iterations
        for iteration in range(self.max_iters):
            iteration_count += 1
            # Assign points to the nearest centroid
            clusters = self.assign_clusters(X)
            # Calculate new centroids
            new_centroids = self.update_centroids(X, clusters)
            # Calculate the error as the sum of centroid shifts
            error = np.sum([euclidean_distance(new_centroids[i], self.
↪centroids[i]) for i in range(self.k)])
            # Display iteration details
            print(f"Iteration {iteration + 1}, Centroids:
↪\n{new_centroids}\nError: {error}")
            print("******************************")
            # Check for convergence
            if np.all(np.abs(new_centroids - self.centroids) < self.tolerance):
                print("Convergence reached.")
                break
            self.centroids = new_centroids

        print(f"Total iterations performed: {iteration_count}")
        return clusters, self.centroids
```

##Fitting the Model with K=2

[27]:
```
kmeans_2 = KMeans(k=2)
clusters_2, centroids_2 = kmeans_2.fit(X)
print(f'Centroids: {centroids_2}')
print('No of Centroids:', len(centroids_2))
```

```
Iteration 1, Centroids:
[[10111.02985075 17966.85074627 23574.56716418  4033.68656716
  10073.10447761  3028.80597015]
 [12339.65683646  3610.12868633  5144.94906166  2899.1769437
```

```
   1589.70241287  1254.72654155]]
Error: 21035.258400029103
******************************
Iteration 2, Centroids:
[[ 9792.32258065 17721.20967742 25442.59677419  3619.64516129
  11066.46774194  3022.0483871 ]
 [12362.45238095  3840.32275132  5082.33068783  2982.0952381
   1538.98412698  1279.3015873 ]]
Error: 2452.8741373052258
******************************
Iteration 3, Centroids:
[[ 9548.8        17782.48333333 25864.85        2714.78333333
  11349.3         2970.83333333]
 [12387.37631579  3903.70526316  5122.81842105  3128.32368421
   1544.47105263  1296.56052632]]
Error: 1236.2914327401036
******************************
Iteration 4, Centroids:
[[ 9548.8        17782.48333333 25864.85        2714.78333333
  11349.3         2970.83333333]
 [12387.37631579  3903.70526316  5122.81842105  3128.32368421
   1544.47105263  1296.56052632]]
Error: 0.0
******************************
Convergence reached.
Total iterations performed: 4
Centroids: [[ 9548.8        17782.48333333 25864.85        2714.78333333
  11349.3         2970.83333333]
 [12387.37631579  3903.70526316  5122.81842105  3128.32368421
   1544.47105263  1296.56052632]]
No of Centroids: 2
```

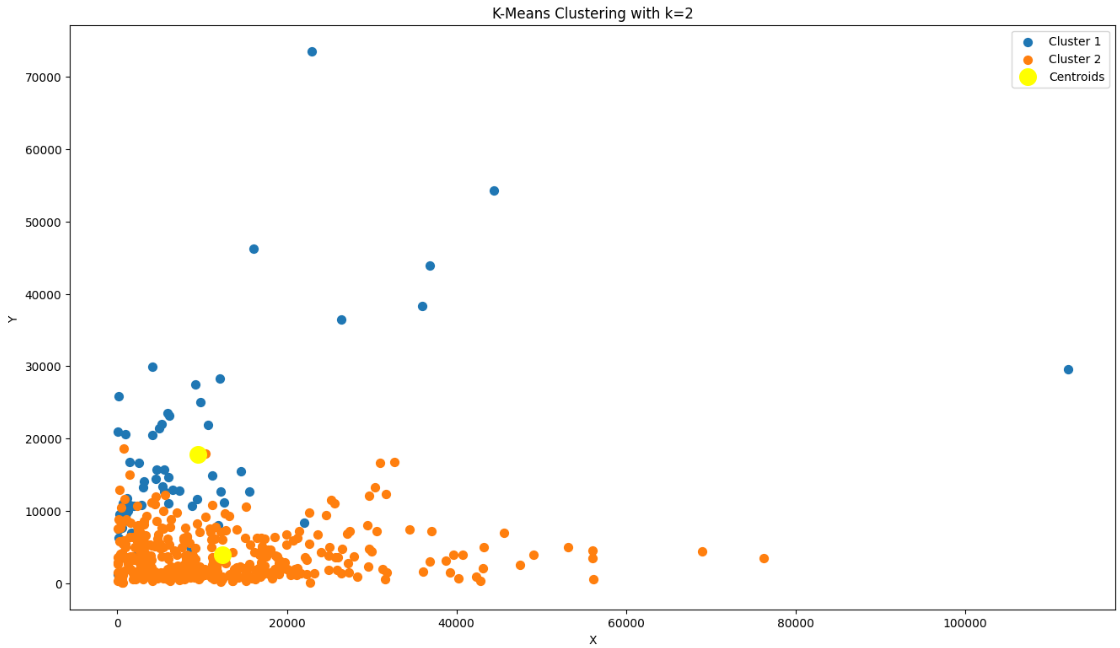
##Function for Plotting Clusters and their Boundaries

[28]:
```python
def plot_clusters(X, clusters, centroids, k):
    plt.figure(figsize=(16, 9))
    for i in range(k):
        plt.scatter(X[clusters == i, 0], X[clusters == i, 1], s=50,␣
↪label=f'Cluster {i+1}')
    plt.scatter(centroids[:, 0], centroids[:, 1], s=200, c='yellow',␣
↪label='Centroids')
    plt.title(f'K-Means Clustering with k={k}')
    plt.xlabel('X')
    plt.ylabel('Y ')
    plt.legend()
    plt.show()
```

##Cluster Visualization for K=2

[29]:

```
plot_clusters(X, clusters_2, centroids_2, 2)
```



##Model Fitting with K=3

[30]:

```
kmeans_3 = KMeans(k=3)
clusters_3, centroids_3 = kmeans_3.fit(X)
print(f'Centroids: {centroids_3}')
print('No of Centroids:', len(centroids_3))
```

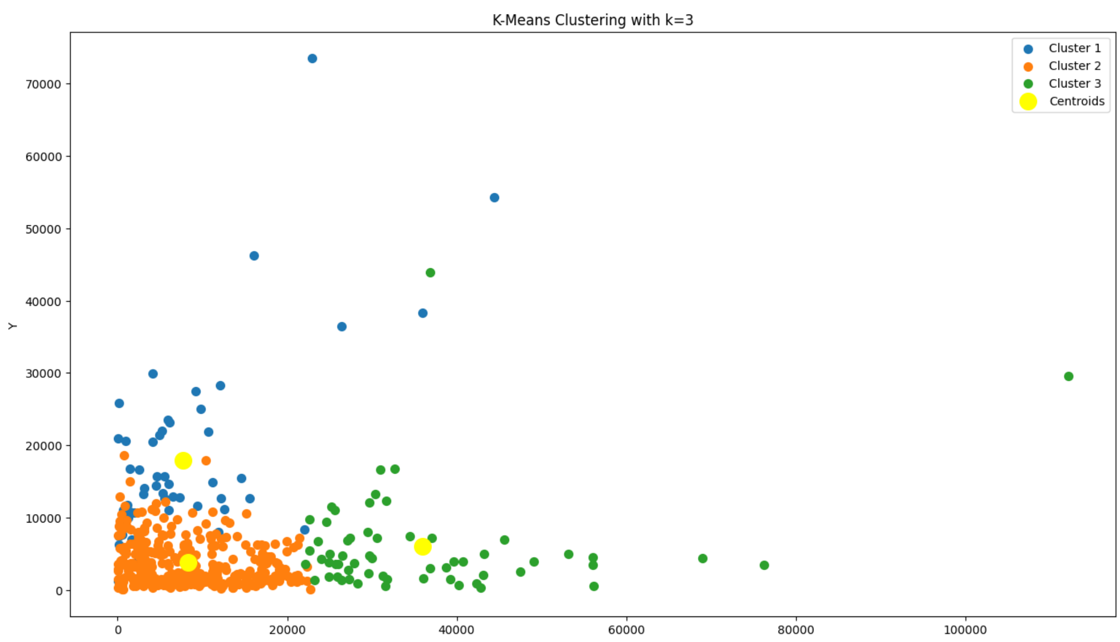
```
Iteration 1, Centroids:
[[ 6721.         18275.94736842 22491.12280702  3777.75438596
   9675.35087719  3097.84210526]
 [ 8056.0295082   3352.99344262  4729.20655738  2586.38688525
   1592.93442623  1039.46557377]
 [31281.34615385  6230.32051282  9925.12820513  4454.74358974
   2955.34615385  2273.44871795]]
Error: 25038.261363070833
******************************
Iteration 2, Centroids:
[[ 7210.3220339  17090.74576271 25855.88135593  1868.98305085
  11519.69491525  2123.13559322]
 [ 7913.61488673  3681.44983819  4852.39805825  2618.74433657
   1620.48867314  1085.72815534]
 [33464.09722222  5617.15277778  6578.80555556  6002.61111111
   1214.77777778  2919.27777778]]
Error: 9671.730250766599
```

```
******************************
Iteration 3, Centroids:
[[ 7547.10714286 17395.58928571 26495.08928571  1951.14285714
  11820.94642857  2148.85714286]
 [ 8082.39937107  3740.94025157  5026.94968553  2598.75157233
   1671.55660377  1129.01572327]
 [34655.90909091  5857.34848485  6307.07575758  6302.77272727
   1126.1969697   2902.72727273]]
Error: 2388.613361852872
******************************
Iteration 4, Centroids:
[[ 7771.33333333 17775.98148148 26812.87037037  1980.2037037
  12046.7962963   2171.88888889]
 [ 8080.90654206  3768.81619938  5120.28660436  2587.21183801
   1699.59190031  1132.81619938]
 [34869.35384615  5856.36923077  6262.38461538  6372.67692308
   1104.01538462  2923.49230769]]
Error: 923.3200297611633
******************************
Iteration 5, Centroids:
[[ 7751.98113208 17910.50943396 27037.90566038  1970.94339623
  12104.86792453  2185.73584906]
 [ 8164.9691358   3808.29320988  5198.04012346  2583.15123457
   1739.60185185  1137.10802469]
 [35298.82539683  5828.77777778  6053.77777778  6511.88888889
    994.73015873  2963.11111111]]
Error: 908.9270041842609
******************************
Iteration 6, Centroids:
[[ 7751.98113208 17910.50943396 27037.90566038  1970.94339623
  12104.86792453  2185.73584906]
 [ 8208.56923077  3800.25538462  5189.44923077  2581.49846154
   1734.56         1135.32615385]
 [35507.91935484  5903.5         6112.61290323  6583.91935484
   1009.14516129  3001.90322581]]
Error: 289.77181306672065
******************************
Iteration 7, Centroids:
[[ 7751.98113208 17910.50943396 27037.90566038  1970.94339623
  12104.86792453  2185.73584906]
 [ 8251.85889571  3798.46319018  5177.96932515  2580.35276074
   1729.78527607  1139.82515337]
 [35724.09836066  5947.55737705  6189.09836066  6655.6557377
   1022.7704918   3008.45901639]]
Error: 290.0569543487896
******************************
Iteration 8, Centroids:
[[ 7751.98113208 17910.50943396 27037.90566038  1970.94339623
```

```
  12104.86792453  2185.73584906]
 [ 8296.          3787.25688073  5162.80122324  2582.11620795
   1724.52293578  1138.01529052]
 [35941.4          6044.45        6288.61666667  6713.96666667
   1039.66666667  3049.46666667]]
Error: 316.4578425559617
******************************
Iteration 9, Centroids:
[[ 7751.98113208 17910.50943396 27037.90566038  1970.94339623
  12104.86792453  2185.73584906]
 [ 8296.          3787.25688073  5162.80122324  2582.11620795
   1724.52293578  1138.01529052]
 [35941.4          6044.45        6288.61666667  6713.96666667
   1039.66666667  3049.46666667]]
Error: 0.0
******************************
Convergence reached.
Total iterations performed: 9
Centroids: [[ 7751.98113208 17910.50943396 27037.90566038  1970.94339623
  12104.86792453  2185.73584906]
 [ 8296.          3787.25688073  5162.80122324  2582.11620795
   1724.52293578  1138.01529052]
 [35941.4          6044.45        6288.61666667  6713.96666667
   1039.66666667  3049.46666667]]
No of Centroids: 3
```

##Cluster Visualization for K=3

[31]:
```
plot_clusters(X, clusters_3, centroids_3, 3)
```



[32]:
```
kmeans_4 = KMeans(k=4)
clusters_4, centroids_4 = kmeans_4.fit(X)
print(f'Centroids: {centroids_4}')
print('No of Centroids:', len(centroids_4))
```

```
Iteration 1, Centroids:
[[ 6721.         18275.94736842 22491.12280702  3777.75438596
   9675.35087719  3097.84210526]
 [ 7816.19333333  3384.75666667  4779.24666667  2600.63
   1614.03333333  1040.89666667]
 [21724.34285714  9137.97142857 16384.85714286  2935.02857143
   5657.48571429  2474.22857143]
 [37329.66666667  3611.91666667  4360.91666667  5279.22916667
    711.25        1989.5625    ]]
Error: 34866.32211662641
******************************
Iteration 2, Centroids:
[[ 6604.87272727 17602.72727273 26062.38181818  1889.61818182
  11766.03636364  2113.43636364]
 [ 7872.19407895  3570.24671053  4621.05592105  2628.05592105
   1518.38157895  1065.35855263]
 [20925.5625      8595.375      13282.25        3784.28125
   3835.03125     3993.3125    ]
 [37838.73469388  4526.53061224  4802.          6687.65306122
    743.18367347  2103.04081633]]
Error: 10899.359507603733
******************************
Iteration 3, Centroids:
[[ 7217.66037736 17779.86792453 26914.16981132  1942.73584906
  12113.8490566   2183.77358491]
 [ 7492.73578595  3659.08361204  4699.35451505  2637.13712375
   1563.39799331  1048.48494983]
 [21158.2173913   6920.10869565  9948.84782609  3657.04347826
   2727.7173913   3283.06521739]
 [40094.97619048  4657.92857143  4984.69047619  6951.35714286
    783.14285714  2159.16666667]]
Error: 7772.003421048425
******************************
Iteration 4, Centroids:
[[ 7680.62264151 17901.67924528 26987.28301887  2002.37735849
  12152.26415094  2184.20754717]
 [ 6253.97426471  3810.15808824  5047.79779412  2414.36764706
   1715.02941176  1042.52941176]
 [21300.32941176  4370.09411765  6118.56470588  3669.71764706
   1522.47058824  1786.38823529]
```

```
 [45381.63333333  6458.23333333  5838.56666667  9229.66666667
    929.63333333  3992.3       ]]
Error: 13164.79602361588
******************************
Iteration 5, Centroids:
[[ 8027.41176471 18375.92156863 27342.54901961  2014.31372549
  12314.60784314  2233.25490196]
 [ 5750.60305344  3956.26717557  5353.64503817  2385.94274809
   1869.61832061  1053.28244275]
 [20973.28        3867.35        5338.95        3603.63
   1246.88        1677.07      ]
 [46916.55555556  7033.62962963  6205.25925926  9757.03703704
    936.44444444  4199.25925926]]
Error: 4136.4554894651765
******************************
Iteration 6, Centroids:
[[ 8174.76       18573.56       27516.96        2051.94
  12426.1         2262.4       ]
 [ 5486.34509804  4073.65882353  5540.93333333  2277.60392157
   1956.00392157  1050.39215686]
 [20642.78181818  3705.74545455  5050.90909091  3873.40909091
   1136.00909091  1657.83636364]
 [48066.76        7010.56        6167.04        9687.56
    912.4         4304.44      ]]
Error: 2406.0917828069573
******************************
Iteration 7, Centroids:
[[ 8149.83673469 18715.85714286 27756.59183673  2034.71428571
  12523.02040816  2282.14285714]
 [ 5442.96850394  4120.07086614  5597.08661417  2258.15748031
   1989.2992126   1053.27165354]
 [20598.38938053  3789.42477876  5027.27433628  3993.53982301
   1120.14159292  1638.39823009]
 [48777.375       6607.375       6197.79166667  9462.79166667
    932.125       4435.33333333]]
Error: 1405.4910036119027
******************************
Iteration 8, Centroids:
[[ 8149.83673469 18715.85714286 27756.59183673  2034.71428571
  12523.02040816  2282.14285714]
 [ 5442.96850394  4120.07086614  5597.08661417  2258.15748031
   1989.2992126   1053.27165354]
 [20598.38938053  3789.42477876  5027.27433628  3993.53982301
   1120.14159292  1638.39823009]
 [48777.375       6607.375       6197.79166667  9462.79166667
    932.125       4435.33333333]]
Error: 0.0
******************************
```

```
Convergence reached.
Total iterations performed: 8
Centroids: [[ 8149.83673469 18715.85714286 27756.59183673  2034.71428571
  12523.02040816  2282.14285714]
 [ 5442.96850394  4120.07086614  5597.08661417  2258.15748031
   1989.2992126   1053.27165354]
 [20598.38938053  3789.42477876  5027.27433628  3993.53982301
   1120.14159292  1638.39823009]
 [48777.375       6607.375       6197.79166667  9462.79166667
    932.125       4435.33333333]]
No of Centroids: 4
```

[33]:

```
plot_clusters(X, clusters_4, centroids_4, 4)
```