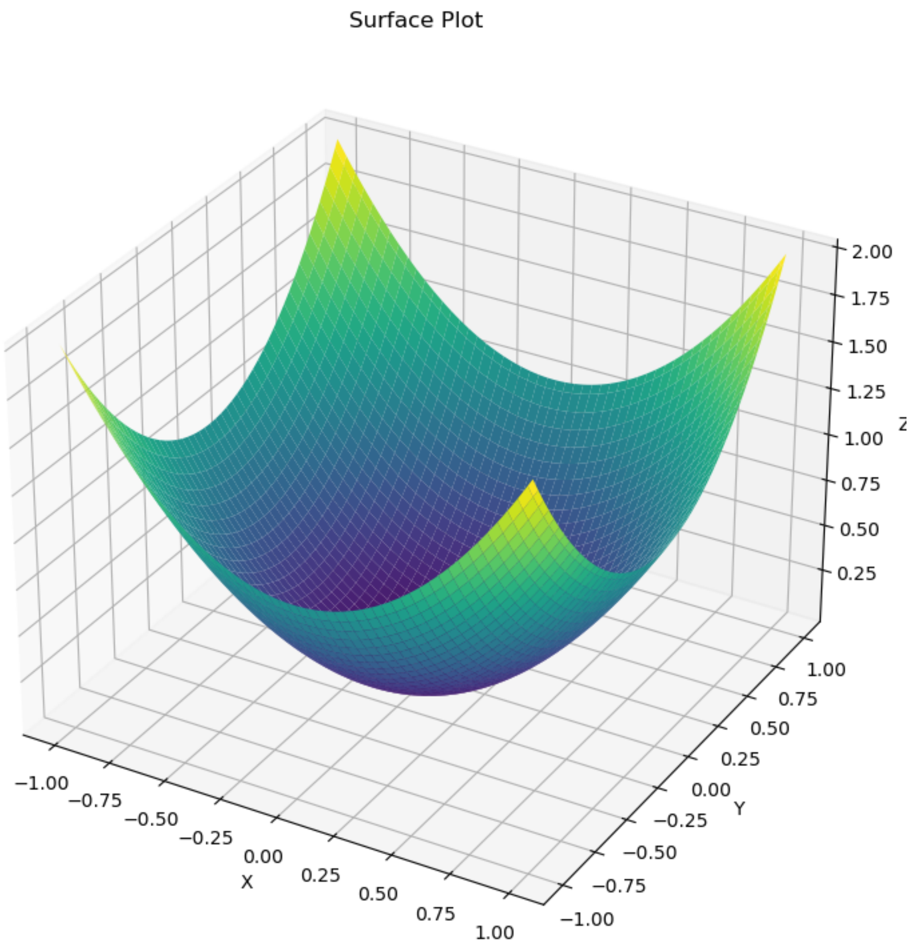
In [80]:

```python
from mpl_toolkits import mplot3d
import numpy as np
import matplotlib.pyplot as plt

# Creating dataset
x = np.outer(np.linspace(-1, 1, 100), np.ones(100))
y = x.copy().T  # Transpose
z = x ** 2 + y ** 2

# print(x,y,z)

# Creating figure
fig = plt.figure(figsize=(14, 9))
ax = plt.axes(projection='3d')

# Creating plot
ax.plot_surface(x, y, z, cmap='viridis')

# Set labels and title
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
ax.set_title('Surface Plot')

# Show plot
plt.show()
```



In [79]:

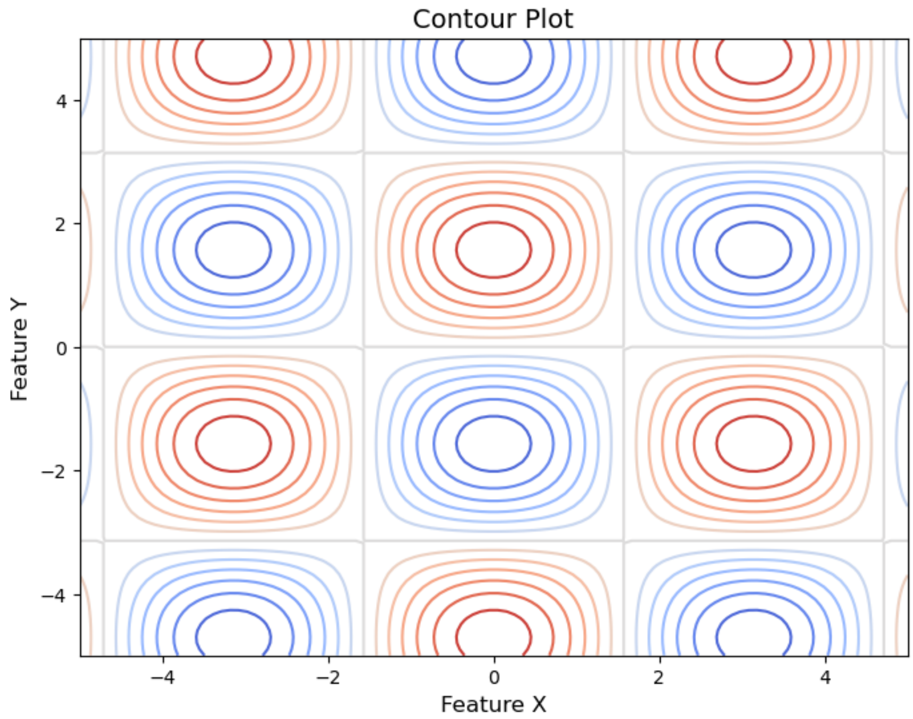
```python
from queue import PriorityQueue
v = 14
graph = [[] for i in range(v)]

# Function For Implementing Best First Search
# Gives output path having lowest cost


def best_first_search(actual_Src, target, n):
    visited = [False] * n
    pq = PriorityQueue()
    pq.put((0, actual_Src))
```

```python
    visited[actual_Src] = True

    while pq.empty() == False:
        u = pq.get()[1]
        # Displaying the path having lowest cost
        print(u, end=" ")
        if u == target:
            break

        for v, c in graph[u]:
            if visited[v] == False:
                visited[v] = True
                pq.put((c, v))
    print()

# Function for adding edges to graph


def addedge(x, y, cost):
    graph[x].append((y, cost))
    graph[y].append((x, cost))


# The nodes shown in above example(by alphabets) are
# implemented using integers addedge(x,y,cost);
addedge(0, 1, 3)
addedge(0, 2, 6)
addedge(0, 3, 5)
addedge(1, 4, 9)
addedge(1, 5, 8)
addedge(2, 6, 12)
addedge(2, 7, 14)
addedge(3, 8, 7)
addedge(8, 9, 5)
addedge(8, 10, 6)
addedge(9, 11, 1)
addedge(9, 12, 10)
addedge(9, 13, 2)

source = 0
target = 9
best_first_search(source, target, v)
```

```
0 1 3 2 8 9
```

In [89]:

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

feature_x = np.linspace(-5, 5, 100)
feature_y = np.linspace(-5, 5, 100)

# Creating 2-D grid of features
X, Y = np.meshgrid(feature_x, feature_y)

# Define the equation for Z
Z = np.cos(X) * np.sin(Y)

fig, ax = plt.subplots(figsize=(8, 6))

# Plot contour lines with color map
contour = ax.contour(X, Y, Z, levels=15, cmap='coolwarm')

ax.set_title('Contour Plot', fontsize=14)
ax.set_xlabel('Feature X', fontsize=12)
ax.set_ylabel('Feature Y', fontsize=12)

plt.show()
```



In [93]:

```
def aStarAlgo(start_node, stop_node):
    open_set = set([start_node])
    closed_set = set()
    g = {start_node: 0}
    parents = {start_node: start_node}

    while open_set:
        n = min(open_set, key=lambda x: g[x] + heuristic(x))

        if n == stop_node or n not in Graph_nodes:
            break

        for m, weight in get_neighbors(n):
            if m not in open_set and m not in closed_set:
                open_set.add(m)
                parents[m] = n
                g[m] = g[n] + weight
            else:
                if g[m] > g[n] + weight:
                    g[m] = g[n] + weight
                    parents[m] = n
                    if m in closed_set:
                        closed_set.remove(m)
                        open_set.add(m)

        open_set.remove(n)
        closed_set.add(n)

    if n == stop_node:
        path = []
        while n != start_node:
            path.append(n)
            n = parents[n]
        path.append(start_node)
        path.reverse()
        print('Path found:', path)
        return path

    print('Path does not exist!')
    return None

def get_neighbors(v):
    return Graph_nodes.get(v, None)

def heuristic(n):
    H_dist = {
        'A': 3, 'B': 4, 'C': 2, 'D': 6, 'G': 0, 'S': 5
    }
    return H_dist.get(n, 0)

Graph_nodes = {
```

```
    'S': [('A', 1), ('G', 10)],
    'A': [('B', 2), ('C', 1)],
    'B': [('D', 5)],
    'C': [('D', 3),('G', 4)],
    'D': [('G', 2)]
}

aStarAlgo('S', 'G')
```

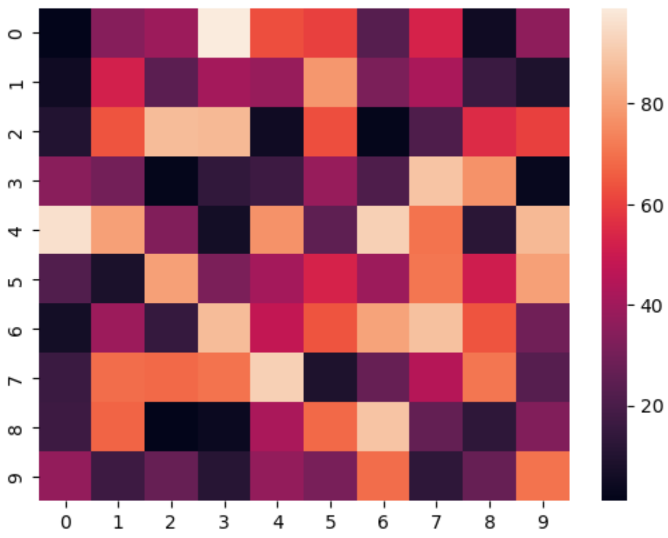
```
Path found: ['S', 'A', 'C', 'G']
```

Out [93]: ['S', 'A', 'C', 'G']

In [37]:

```
import numpy as np
import seaborn as sn
import matplotlib.pyplot as plt

# generating 2-D 10x10 matrix of random numbers
# from 1 to 100
data = np.random.randint(low = 1,
                         high = 100,
                         size = (10, 10))
print("The data to be plotted:\n")
print(data)

# plotting the heatmap
hm = sn.heatmap(data = data)

# displaying the plotted heatmap
plt.show()
```

```
The data to be plotted:

[[ 1 34 39 99 63 60 23 53  5 36]
 [ 5 52 24 41 38 78 32 42 16  9]
 [10 64 87 86  5 63  2 21 55 60]
 [35 30  2 14 17 38 21 89 77  3]
 [96 80 33  6 77 25 92 70 12 86]
 [22  8 80 32 41 53 39 71 51 80]
 [ 6 39 15 87 48 64 81 88 64 29]
 [16 69 68 70 92  9 27 45 71 23]
 [17 67  1  4 42 68 89 26 13 33]
 [37 17 27 11 37 31 69 13 27 70]]
```



In [82]:

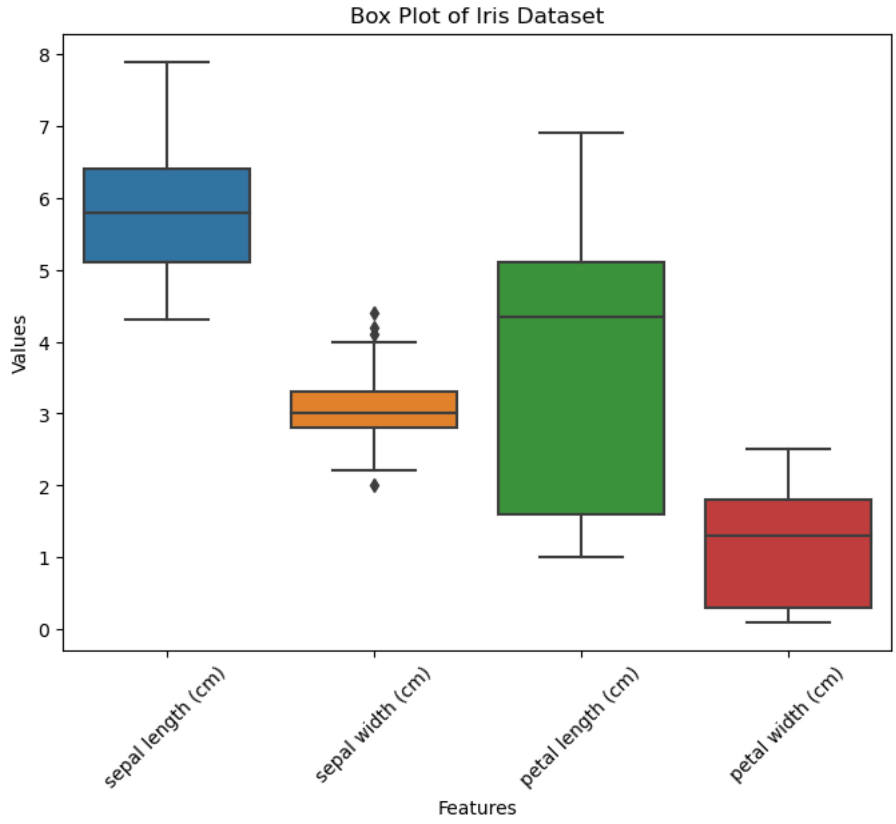
```
import math

def minimax (curDepth, nodeIndex,
             maxTurn, scores,
             targetDepth):

    # base case : targetDepth reached
    if (curDepth == targetDepth):
        return scores[nodeIndex]

    if (maxTurn):
        return max(minimax(curDepth + 1, nodeIndex * 2,
                    False, scores, targetDepth),
                   minimax(curDepth + 1, nodeIndex * 2 + 1,
                    False, scores, targetDepth))

    else:
        return min(minimax(curDepth + 1, nodeIndex * 2,
                     True, scores, targetDepth),
```

```
                   minimax(curDepth + 1, nodeIndex * 2 + 1,
                     True, scores, targetDepth))

# Driver code
scores = [3, 5, 2, 9, 12, 5, 23, 23]

treeDepth = math.log(len(scores), 2)

print("The optimal value is : ", end = "")
print(minimax(0, 0, True, scores, treeDepth))
```

```
The optimal value is : 12
```

In [44]:

```
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.datasets import load_iris

# Load the Iris dataset
iris = load_iris()
data = iris.data
target = iris.target
feature_names = iris.feature_names

# Create a box plot
plt.figure(figsize=(8, 6))
sns.boxplot(data=data)
plt.xticks(ticks=range(len(feature_names)), labels=feature_names, rotation=45)
plt.xlabel('Features')
plt.ylabel('Values')
plt.title('Box Plot of Iris Dataset')
plt.show()
```



In [45]:

```
MAX, MIN = 1000, -1000

# Returns optimal value for current player
#(Initially called for root and maximizer)
def minimax(depth, nodeIndex, maximizingPlayer,
            values, alpha, beta):

    # Terminating condition. i.e
    # leaf node is reached
    if depth == 3:
        return values[nodeIndex]

    if maximizingPlayer:

        best = MIN
```

```
            # Recur for left and right children
            for i in range(0, 2):

                val = minimax(depth + 1, nodeIndex * 2 + i,
                              False, values, alpha, beta)
                best = max(best, val)
                alpha = max(alpha, best)

                # Alpha Beta Pruning
                if beta <= alpha:
                    break

            return best

        else:
            best = MAX

            # Recur for left and
            # right children
            for i in range(0, 2):

                val = minimax(depth + 1, nodeIndex * 2 + i,
                                True, values, alpha, beta)
                best = min(best, val)
                beta = min(beta, best)

                # Alpha Beta Pruning
                if beta <= alpha:
                    break

            return best



values = [3, 5, 6, 9, 1, 2, 0, -1]
print("The optimal value is :", minimax(0, 0, True, values, MIN, MAX))
```

```
The optimal value is : 5
```

In [8]:

```
import pandas as pd
import numpy as np
from sklearn.metrics import confusion_matrix

# Load the Titanic dataset
df = pd.read_csv('titanic.csv')

# Preprocess the dataset
df.drop(['PassengerId', 'Name', 'Ticket', 'Cabin', 'Embarked'], axis=1, inplace=True)
df['Age'].fillna(df['Age'].median(), inplace=True)
df['Fare'].fillna(df['Fare'].median(), inplace=True)
df['Sex'] = df['Sex'].map({'female': 0, 'male': 1})

# Split the dataset into features and target variable
X = df.drop('Survived', axis=1)
y = df['Survived']

print(df)
# Define the Naive Bayes classifier class
class NaiveBayesClassifier:
    def __init__(self):
        self.prior = {}
        self.conditional = {}

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

        # Compute class priors
        for c in self.classes:
            self.prior[c] = np.mean(y == c)

        # Compute conditional probabilities
        for feature in X.columns:
            self.conditional[feature] = {}
            for c in self.classes:
                feature_values = X[feature][y == c]
                self.conditional[feature][c] = {
                    'mean': np.mean(feature_values),
                    'std': np.std(feature_values)
                }
```

```
    def predict(self, X):
        y_pred = []
        for _, sample in X.iterrows():
            probabilities = {}
            for c in self.classes:
                probabilities[c] = self.prior[c]
                for feature in X.columns:
                    mean = self.conditional[feature][c]['mean']
                    std = self.conditional[feature][c]['std']
                    x = sample[feature]
                    probabilities[c] *= self._gaussian_pdf(x, mean, std)
            y_pred.append(max(probabilities, key=probabilities.get))
        return y_pred

    @staticmethod
    def _gaussian_pdf(x, mean, std):
        exponent = np.exp(-((x - mean) ** 2) / (2 * std ** 2))
        return (1 / (np.sqrt(2 * np.pi) * std)) * exponent

# Instantiate and train the Naive Bayes classifier
classifier = NaiveBayesClassifier()
classifier.fit(X, y)

# Predict the target variable
y_pred = classifier.predict(X)

# Print the confusion matrix
cm = confusion_matrix(y, y_pred)
print("Confusion Matrix:")
print(cm)

# Calculate accuracy
accuracy = np.mean(y_pred == y)
print("Accuracy:", accuracy)
```

```
     Survived  Pclass  Sex   Age  SibSp  Parch     Fare
0           0       3    1  22.0      1      0   7.2500
1           1       1    0  38.0      1      0  71.2833
2           1       3    0  26.0      0      0   7.9250
3           1       1    0  35.0      1      0  53.1000
4           0       3    1  35.0      0      0   8.0500
..        ...     ...  ...   ...    ...    ...      ...
886         0       2    1  27.0      0      0  13.0000
887         1       1    0  19.0      0      0  30.0000
888         0       3    0  28.0      1      2  23.4500
889         1       1    1  26.0      0      0  30.0000
890         0       3    1  32.0      0      0   7.7500

[891 rows x 7 columns]
Confusion Matrix:
[[465  84]
 [101 241]]
Accuracy: 0.792368125701459
```

In [2]:

```
import pandas as pd
import numpy as np
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import train_test_split

df = pd.read_csv('titanic.csv')

df = df[['Survived', 'Pclass', 'Age', 'SibSp', 'Parch', 'Fare', 'Embarked']]
df['Age'].fillna(df['Age'].median(), inplace=True)
df['Fare'].fillna(df['Fare'].median(), inplace=True)
df['Embarked'].fillna(df['Embarked'].mode()[0], inplace=True)
df['Embarked'] = df['Embarked'].map({'C': 0, 'Q': 1, 'S': 2})

train1, test1 = train_test_split(df, test_size=0.2, random_state=41)

X_train1 = train1.drop('Survived', axis=1)
y_train1 = train1['Survived']
X_test1 = test1.drop('Survived', axis=1)
y_test1 = test1['Survived']


class NaiveBayesClassifier:
    def __init__(self):
        self.prior = {}
        self.conditional = {}

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

        # Compute class priors
```

```python
        for c in self.classes:
            self.prior[c] = np.mean(y == c)

        # Compute conditional probabilities
        for feature in X.columns:
            self.conditional[feature] = {}
            for c in self.classes:
                feature_values = X[feature][y == c]
                self.conditional[feature][c] = {
                    'mean': np.mean(feature_values),
                    'std': np.std(feature_values)
                }

    def predict(self, X):
        y_pred = []
        for _, sample in X.iterrows():
            probabilities = {}
            for c in self.classes:
                probabilities[c] = self.prior[c]
                for feature in X.columns:
                    mean = self.conditional[feature][c]['mean']
                    std = self.conditional[feature][c]['std']
                    x = sample[feature]
                    probabilities[c] *= self._gaussian_pdf(x, mean, std)
            y_pred.append(max(probabilities, key=probabilities.get))
        return y_pred

    @staticmethod
    def _gaussian_pdf(x, mean, std):
        exponent = np.exp(-((x - mean) ** 2) / (2 * std ** 2))
        return (1 / (np.sqrt(2 * np.pi) * std)) * exponent


classifier = NaiveBayesClassifier()
classifier.fit(X_train1, y_train1)
y_pred = classifier.predict(X_test1)
cm = confusion_matrix(y_test1, y_pred)
print("Confusion Matrix:")
print(cm)
accuracy = np.mean(y_pred == y_test1)
print("Accuracy:", accuracy)
```

```
Confusion Matrix:
[[88 17]
 [42 32]]
Accuracy: 0.6703910614525139
```

In [37]:

```python
import numpy as np
from collections import Counter
from sklearn.model_selection import train_test_split

def euclidean_distance(x1, x2):
    distance = np.sqrt(np.sum((x1 - x2) ** 2))
    return distance

class KNN:
    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):
        predictions = [self._predict(x) for x in X]
        return predictions

    def _predict(self, x):
        distances = [euclidean_distance(x, x_train) for x_train in self.X_train]
        k_indices = np.argsort(distances)[:self.k]
        k_nearest_labels = [self.y_train[i] for i in k_indices]
        most_common = Counter(k_nearest_labels).most_common()
        return most_common[0][0]

df = pd.read_csv('glass.csv')
X = df.drop('Type', axis=1).values
y = df['Type'].values
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=45)

clf = KNN(k=3)
clf.fit(X_train, y_train)
predictions = clf.predict(X_test)
```

```
print(predictions)

accuracy = np.sum(predictions == y_test) / len(y_test)
print("Accuracy:", accuracy)
```

```
[2, 1, 2, 2, 2, 2, 1, 1, 6, 1, 7, 1, 2, 3, 1, 2, 1, 6, 3, 1, 5, 1, 7, 2, 1, 5, 1, 7, 5, 2, 2, 7, 3, 1, 2, 5, 7, 1, 2, 1, 2, 1, 1, 7, 1, 1, 1, 1, 7, 2, 1,
[2 1 2 2 7 2 1 3 7 2 7 1 2 1 1 2 1 6 3 1 7 2 7 2 1 2 1 7 2 2 2 7 1 1 2 5 7
 1 2 2 2 2 1 7 1 2 1 1 7 2 2 2 1 3 7 2 6 7 1 2 1 1 2 7 2]
Accuracy: 0.7230769230769231
```

In [10]:

```
import numpy as np
from collections import Counter
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder

def manhattan_distance(x1, x2):
    distance = np.sum(np.abs(x1 - x2))
    return distance

class KNN:
    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):
        predictions = [self._predict(x) for x in X]
        return predictions

    def _predict(self, x):
        distances = [manhattan_distance(x, x_train) for x_train in self.X_train]
        k_indices = np.argsort(distances)[:self.k]
        k_nearest_labels = [self.y_train[i] for i in k_indices]
        most_common = Counter(k_nearest_labels).most_common()
        return most_common[0][0]

df = pd.read_csv('fruit.csv')

# Convert 'fruit_name' column to numerical labels
label_encoder = LabelEncoder()
df['fruit_name'] = label_encoder.fit_transform(df['fruit_name'])
df['fruit_subtype'] = df['fruit_subtype'].factorize()[0]

X = df.drop('fruit_name', axis=1).values
y = df['fruit_name'].values

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=1234)

clf = KNN(k=5)
clf.fit(X_train, y_train)
predictions = clf.predict(X_test)
print(predictions)

accuracy = np.sum(predictions == y_test) / len(y_test)
print("Accuracy:", accuracy)
```

```
[6, 3, 2, 2, 4, 3, 6, 2, 2, 3, 2, 3]
Accuracy: 0.6666666666666666
```

In [30]:

```
import numpy as np
from collections import Counter
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder

def manhattan_distance(x1, x2):
    distance = np.sum(np.abs(x1 - x2))
    return distance

class KNN:
    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):
        predictions = [self._predict(x) for x in X]
```

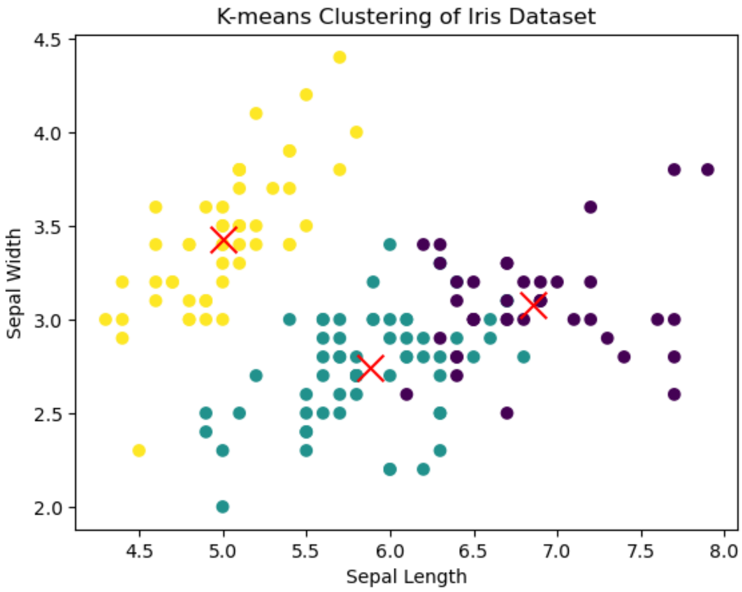
```python
        return predictions

    def _predict(self, x):
        distances = [manhattan_distance(x, x_train) for x_train in self.X_train]
        k_indices = np.argsort(distances)[:self.k]
        k_nearest_labels = [self.y_train[i] for i in k_indices]
        most_common = Counter(k_nearest_labels).most_common()
        return most_common[0][0]

df = pd.read_csv('fruit.csv')

df.drop(['fruit_subtype', 'fruit_name'], axis=1, inplace=True)
X = df.drop('fruit_label', axis=1).values
y = df['fruit_label'].values
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=35)

clf = KNN(k=5)
clf.fit(X_train, y_train)
predictions = clf.predict(X_test)
print(predictions)

accuracy = np.sum(predictions == y_test) / len(y_test)
print("Accuracy:", accuracy)
```

```
[2, 1, 1, 3, 1, 1, 4, 3, 1, 3, 1, 2]
Accuracy: 0.75
```

In [36]:

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

def kmeans(X, K, max_iters=100):
    # Use the first K data points as the initial centroids
    centroids = X[:K]

    for _ in range(max_iters):
        # Assign each data point to the nearest centroid
        labels = np.argmin(np.linalg.norm(X[:, np.newaxis] - centroids, axis=2), axis=1)

        # Update the centroids based on the assigned points
        new_centroids = np.array([X[labels == k].mean(axis=0) for k in range(K)])

        # If the centroids did not change, stop iterating
        if np.all(centroids == new_centroids):
            break

        centroids = new_centroids

    return labels, centroids

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

# Perform K-means clustering
K = 3
labels, centroids = kmeans(X, K)

# Print the resulting labels and centroids
print("Labels:", labels)
print("Centroids:", centroids)

# Plot the clusters
plt.scatter(X[:, 0], X[:, 1], c=labels)
plt.scatter(centroids[:, 0], centroids[:, 1], marker='x', color='red', s=200)
plt.xlabel('Sepal Length')
plt.ylabel('Sepal Width')
plt.title('K-means Clustering of Iris Dataset')
plt.show()
```

```
Labels: [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 2 2 2 2 2 0 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 0 0 0 0 1 0 0 0 0
 0 0 1 1 0 0 0 0 1 0 1 0 1 0 0 1 1 0 0 0 0 0 1 0 0 0 0 1 0 0 0 1 0 0 0 1 0
 0 1]
Centroids: [[6.85384615 3.07692308 5.71538462 2.05384615]
 [5.88360656 2.74098361 4.38852459 1.43442623]
 [5.006      3.428      1.462      0.246     ]]
```

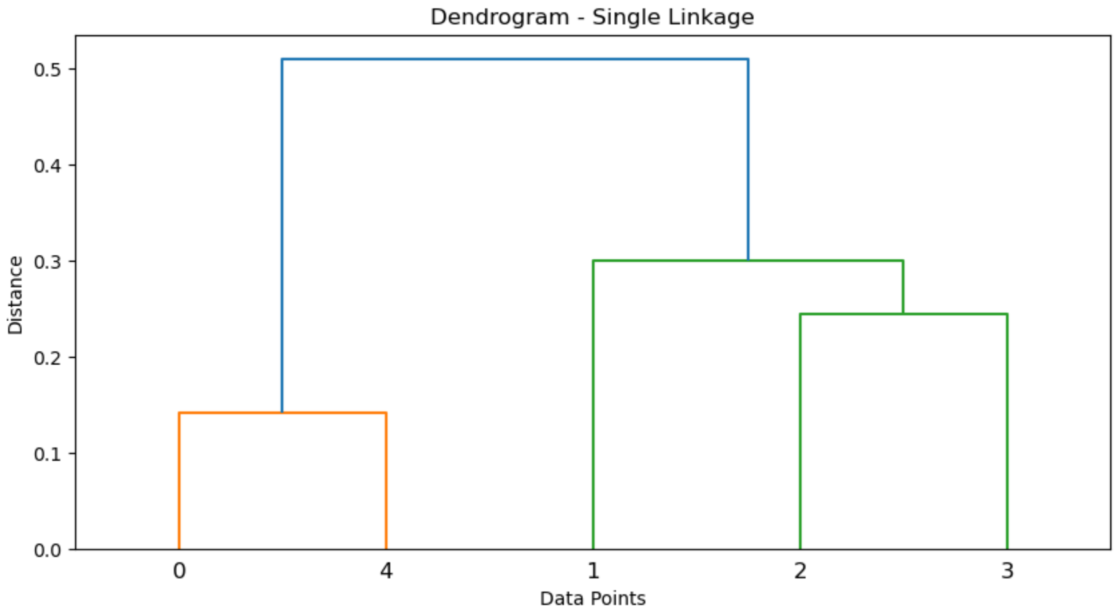


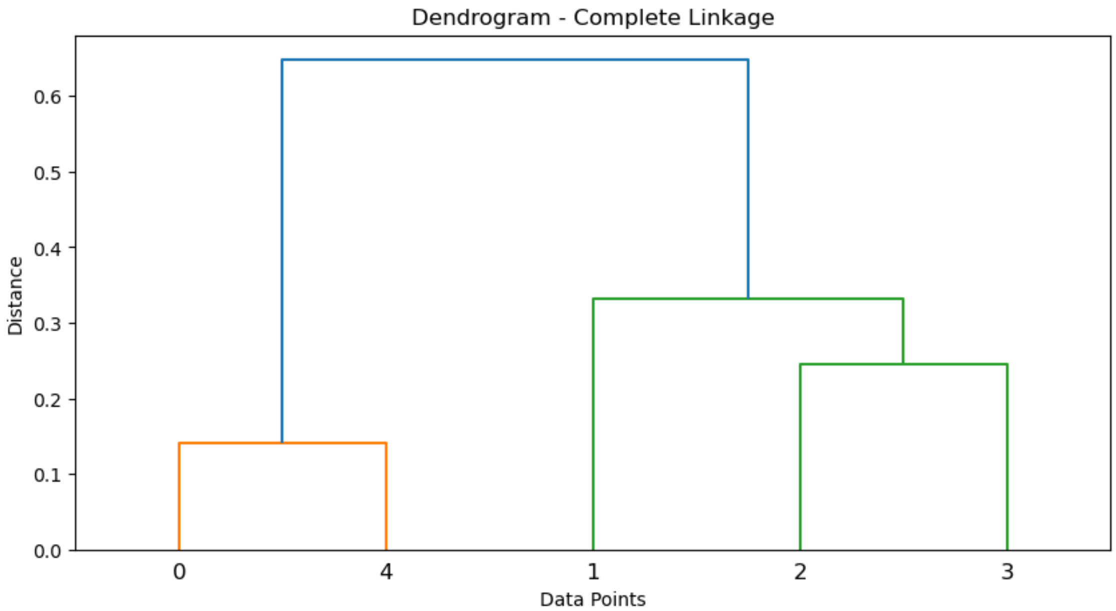
In [83]:

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

iris = load_iris()
data = iris.data[:5]

# Function to calculate the proximity matrix based on single-linkage
def single_linkage(data):
    n = data.shape[0]
    proximity_matrix = np.zeros((n, n))

    for i in range(n):
        for j in range(i+1, n):
            proximity_matrix[i, j] = np.min(np.linalg.norm(data[i] - data[j]))
            proximity_matrix[j, i] = proximity_matrix[i, j]

    return proximity_matrix

# Function to calculate the proximity matrix based on complete-linkage
def complete_linkage(data):
    n = data.shape[0]
    proximity_matrix = np.zeros((n, n))

    for i in range(n):
        for j in range(i+1, n):
            proximity_matrix[i, j] = np.max(np.linalg.norm(data[i] - data[j]))
            proximity_matrix[j, i] = proximity_matrix[i, j]

    return proximity_matrix

# Calculate the proximity matrix using single-linkage
single_linkage_matrix = single_linkage(data)
print("Single-linkage proximity matrix:")
print(single_linkage_matrix)

# Calculate the proximity matrix using complete-linkage
complete_linkage_matrix = complete_linkage(data)
print("\nComplete-linkage proximity matrix:")
print(complete_linkage_matrix)

# Plot the dendrogram using single-linkage
linkage_matrix = linkage(data, method='single')
plt.figure(figsize=(10, 5))
dendrogram(linkage_matrix)
plt.title('Dendrogram - Single Linkage')
plt.xlabel('Data Points')
plt.ylabel('Distance')
plt.show()

# Plot the dendrogram using complete-linkage
linkage_matrix = linkage(data, method='complete')
plt.figure(figsize=(10, 5))
dendrogram(linkage_matrix)
plt.title('Dendrogram - Complete Linkage')
```

```
plt.xlabel('Data Points')
plt.ylabel('Distance')
plt.show()
```

```
Single-linkage proximity matrix:
[[0.         0.53851648 0.50990195 0.64807407 0.14142136]
 [0.53851648 0.         0.3        0.33166248 0.60827625]
 [0.50990195 0.3        0.         0.24494897 0.50990195]
 [0.64807407 0.33166248 0.24494897 0.         0.64807407]
 [0.14142136 0.60827625 0.50990195 0.64807407 0.        ]]

Complete-linkage proximity matrix:
[[0.         0.53851648 0.50990195 0.64807407 0.14142136]
 [0.53851648 0.         0.3        0.33166248 0.60827625]
 [0.50990195 0.3        0.         0.24494897 0.50990195]
 [0.64807407 0.33166248 0.24494897 0.         0.64807407]
 [0.14142136 0.60827625 0.50990195 0.64807407 0.        ]]
```



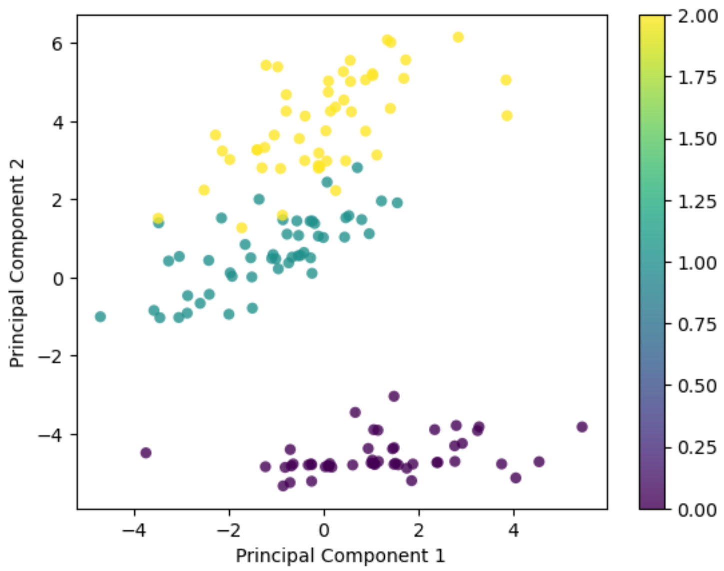


In [76]:

```
import numpy as np


class PCA:

    def __init__(self, n_components):
        self.n_components = n_components
        self.components = None
        self.mean = None

    def fit(self, X):
        # mean centering
        self.mean = np.mean(X, axis=0)
        X = X -  self.mean

        # covariance, functions needs samples as columns
        cov = np.cov(X.T)
```

```
        # eigenvectors, eigenvalues
        eigenvectors, eigenvalues = np.linalg.eig(cov)

        # eigenvectors v = [:, i] column vector, transpose this for easier calculations
        eigenvectors = eigenvectors.T

        # sort eigenvectors
        idxs = np.argsort(eigenvalues)[::-1]
        eigenvalues = eigenvalues[idxs]
        eigenvectors = eigenvectors[idxs]

        self.components = eigenvectors[:self.n_components]

    def transform(self, X):
        # projects data
        X = X - self.mean
        return np.dot(X, self.components.T)


# Testing
if __name__ == "__main__":
    # Imports
    import matplotlib.pyplot as plt
    from sklearn import datasets

    # data = datasets.load_digits()
    data = datasets.load_iris()
    X = data.data
    y = data.target

    # Project the data onto the 2 primary principal components
    pca = PCA(2)
    pca.fit(X)
    X_projected = pca.transform(X)

    print("Shape of X:", X.shape)
    print("Shape of transformed X:", X_projected.shape)

    x1 = X_projected[:, 0]
    x2 = X_projected[:, 1]

    plt.scatter(
        x1, x2, c=y, edgecolor="none", alpha=0.8, cmap="viridis"
    )

    plt.xlabel("Principal Component 1")
    plt.ylabel("Principal Component 2")
    plt.colorbar()
    plt.show()
```

```
Shape of X: (150, 4)
Shape of transformed X: (150, 2)
```

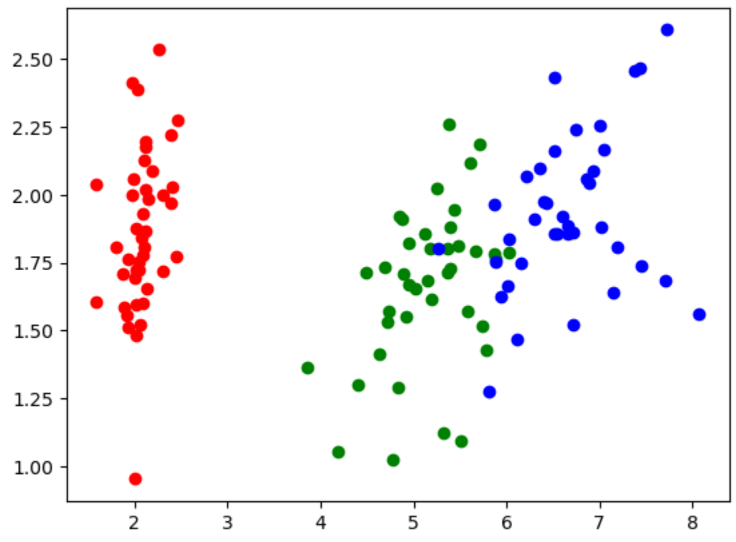


In [77]:

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

```python
class LDA:
    def __init__(self, n_components=None):
        self.n_components = n_components
        self.eig_vectors = None

    def transform(self,X,y):
        height, width = X.shape
        unique_classes = np.unique(y)
        num_classes = len(unique_classes)

        scatter_t = np.cov(X.T)*(height - 1)
        scatter_w = 0
        for i in range(num_classes):
            class_items = np.flatnonzero(y == unique_classes[i])
            scatter_w = scatter_w + np.cov(X[class_items].T) * (len(class_items)-1)

        scatter_b = scatter_t - scatter_w
        _, eig_vectors = np.linalg.eigh(np.linalg.pinv(scatter_w).dot(scatter_b))
        print(eig_vectors.shape)
        pc = X.dot(eig_vectors[:,::-1][:,:self.n_components])
        print(pc.shape)

        if self.n_components == 2:
            if y is None:
                plt.scatter(pc[:,0],pc[:,1])
            else:
                colors = ['r','g','b']
                labels = np.unique(y)
                for color, label in zip(colors, labels):
                    class_data = pc[np.flatnonzero(y==label)]
                    plt.scatter(class_data[:,0],class_data[:,1],c=color)
            plt.show()
        return pc

LDA_obj = LDA(n_components=2)
data = load_iris()
X, y = data.data, data.target
X_train, X_test, Y_train, Y_test = train_test_split(X, y, test_size=0.2)

LDA_object = LDA(n_components=2)
X_train_modified = LDA_object.transform(X_train, Y_train)

print("Original Data Size:",X_train.shape, "\nModified Data Size:", X_train_modified.shape)
```

```
(4, 4)
(120, 2)
```




```
Original Data Size: (120, 4)
Modified Data Size: (120, 2)
```

In [94]:

```python
import pandas as pd
import numpy as np

class NaiveBayesClassifier:
    def __init__(self):
        self.prior = {}
        self.conditional = {}

    def fit(self, X, y):
```

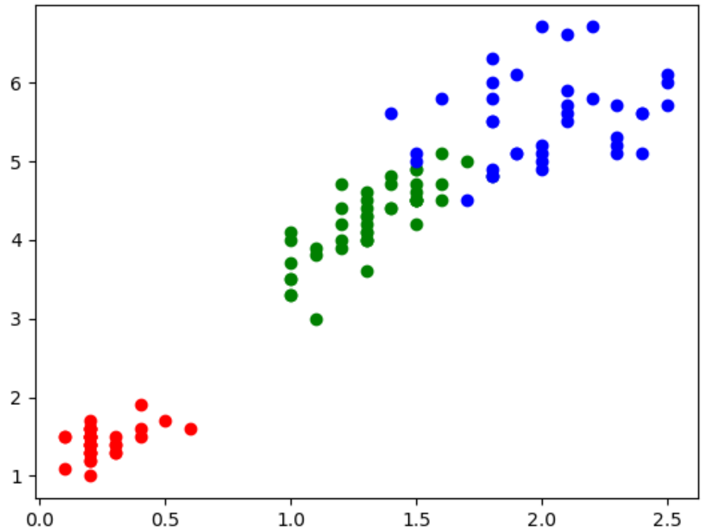
```
        n_samples, n_features = X.shape
        self.classes = np.unique(y)

        for c in self.classes:
            self.prior[c] = np.mean(y == c)

        for feature in X.columns:
            self.conditional[feature] = {}
            for c in self.classes:
                feature_values = X.loc[y == c, feature]
                self.conditional[feature][c] = {
                    'mean': np.mean(feature_values),
                    'std': np.std(feature_values)
                }

    def predict(self, X):
        y_pred = []
        for _, sample in X.iterrows():
            probabilities = {c: self.prior[c] for c in self.classes}
            for feature in X.columns:
                for c in self.classes:
                    mean = self.conditional[feature][c]['mean']
                    std = self.conditional[feature][c]['std']
                    x = sample[feature]
                    probabilities[c] *= self._gaussian_pdf(x, mean, std)
            y_pred.append(max(probabilities, key=probabilities.get))
        return y_pred

    @staticmethod
    def _gaussian_pdf(x, mean, std):
        exponent = np.exp(-((x - mean) ** 2) / (2 * std ** 2))
        return (1 / (np.sqrt(2 * np.pi) * std)) * exponent

df = pd.read_csv('titanic.csv')
df.drop(['PassengerId', 'Name', 'Ticket', 'Cabin', 'Embarked'], axis=1, inplace=True)
df['Age'].fillna(df['Age'].median(), inplace=True)
df['Fare'].fillna(df['Fare'].median(), inplace=True)
df['Sex'] = df['Sex'].map({'female': 0, 'male': 1})

X = df.drop('Survived', axis=1)
y = df['Survived']

classifier = NaiveBayesClassifier()
classifier.fit(X, y)

y_pred = classifier.predict(X)

print(y_pred)
accuracy = np.mean(y_pred == y)
print("Accuracy:", accuracy)
```

```
[0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0, 1, 0, 1, 0,
Accuracy: 0.792368125701459
```

In [1]:

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

class LDA:
    def __init__(self, n_components=None):
        self.n_components = n_components

    def transform(self, X, y):
        unique_classes = np.unique(y)
        scatter_w = np.zeros((X.shape[1], X.shape[1]))
        scatter_b = np.zeros((X.shape[1], X.shape[1]))

        for cls in unique_classes:
            class_items = X[y == cls]
            class_mean = np.mean(class_items, axis=0)
            class_items_centered = class_items - class_mean
            scatter_w += np.cov(class_items_centered.T) * (class_items_centered.shape[0] - 1)

        total_mean = np.mean(X, axis=0)
        X_centered = X - total_mean
        scatter_t = np.cov(X_centered.T) * (X_centered.shape[0] - 1)

        eig_values, eig_vectors = np.linalg.eigh(np.linalg.pinv(scatter_w).dot(scatter_b))
        sorted_indices = np.argsort(eig_values)[::-1]
        eig_vectors = eig_vectors[:, sorted_indices]
```

```
        pc = X.dot(eig_vectors[:, :self.n_components])

        if self.n_components == 2:
            if y is None:
                plt.scatter(pc[:, 0], pc[:, 1])
            else:
                colors = ['r', 'g', 'b']
                for cls, color in zip(unique_classes, colors):
                    class_data = pc[y == cls]
                    plt.scatter(class_data[:, 0], class_data[:, 1], c=color)
            plt.show()

        return pc

data = load_iris()
X, y = data.data, data.target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

lda = LDA(n_components=2)
X_train_modified = lda.transform(X_train, y_train)

print("Original Data Size:", X_train.shape)
print("Modified Data Size:", X_train_modified.shape)
```




```
Original Data Size: (120, 4)
Modified Data Size: (120, 2)
```

In [4]:

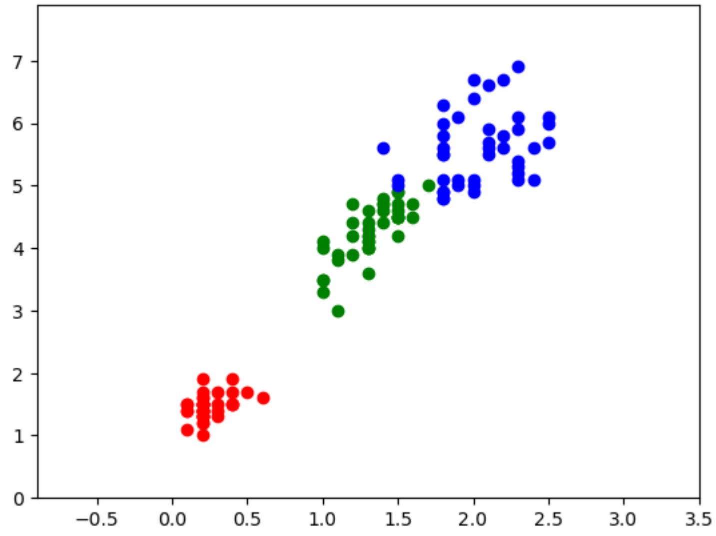
```
import pandas as pd
import numpy as np
from sklearn.metrics import confusion_matrix

# Load the Titanic dataset
df = pd.read_csv('titanic.csv')

# Preprocess the dataset
df.drop(['PassengerId', 'Name', 'Ticket', 'Cabin', 'Embarked'], axis=1, inplace=True)
df['Age'].fillna(df['Age'].median(), inplace=True)
df['Fare'].fillna(df['Fare'].median(), inplace=True)
df['Sex'] = df['Sex'].map({'female': 0, 'male': 1})

print(df)

# Split the dataset into features and target variable
X = df.drop('Survived', axis=1)
y = df['Survived']

# Define the Naive Bayes classifier class
class NaiveBayesClassifier:
    def __init__(self):
        self.prior = {}
        self.conditional = {}

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

        # Compute class priors
        for c in self.classes:
```

```
            self.prior[c] = np.mean(y == c)

        # Compute conditional probabilities
        for feature in X.columns:
            self.conditional[feature] = {}
            for c in self.classes:
                feature_values = X[feature][y == c]
                self.conditional[feature][c] = {
                    'mean': np.mean(feature_values),
                    'std': np.std(feature_values)
                }

    def predict(self, X):
        y_pred = []
        for _, sample in X.iterrows():
            probabilities = {}
            for c in self.classes:
                probabilities[c] = self.prior[c]
                for feature in X.columns:
                    mean = self.conditional[feature][c]['mean']
                    std = self.conditional[feature][c]['std']
                    x = sample[feature]
                    probabilities[c] *= self._gaussian_pdf(x, mean, std)
            y_pred.append(max(probabilities, key=probabilities.get))
        return y_pred

    @staticmethod
    def _gaussian_pdf(x, mean, std):
        exponent = np.exp(-((x - mean) ** 2) / (2 * std ** 2))
        return (1 / (np.sqrt(2 * np.pi) * std)) * exponent

# Instantiate and train the Naive Bayes classifier
classifier = NaiveBayesClassifier()
classifier.fit(X, y)

# Predict the target variable
y_pred = classifier.predict(X)

# Print the confusion matrix
cm = confusion_matrix(y, y_pred)
print("Confusion Matrix:")
print(cm)

# Calculate accuracy
accuracy = np.mean(y_pred == y)
print("Accuracy:", accuracy)
```

```
     Survived  Pclass  Sex   Age  SibSp  Parch     Fare
0           0       3    1  22.0      1      0   7.2500
1           1       1    0  38.0      1      0  71.2833
2           1       3    0  26.0      0      0   7.9250
3           1       1    0  35.0      1      0  53.1000
4           0       3    1  35.0      0      0   8.0500
..        ...     ...  ...   ...    ...    ...      ...
886         0       2    1  27.0      0      0  13.0000
887         1       1    0  19.0      0      0  30.0000
888         0       3    0  28.0      1      2  23.4500
889         1       1    1  26.0      0      0  30.0000
890         0       3    1  32.0      0      0   7.7500

[891 rows x 7 columns]
Confusion Matrix:
[[465  84]
 [101 241]]
Accuracy: 0.792368125701459
```

In [5]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_iris

class LDA:
    def __init__(self, n_components=None):
        self.n_components = n_components

    def transform(self, X, y):
        unique_classes = np.unique(y)
        scatter_w = np.zeros((X.shape[1], X.shape[1]))
        scatter_b = np.zeros((X.shape[1], X.shape[1]))

        for cls in unique_classes:
            class_items = X[y == cls]
            class_mean = np.mean(class_items, axis=0)
            class_items_centered = class_items - class_mean
            scatter_w += np.cov(class_items_centered.T) * (class_items_centered.shape[0] - 1)
```

```python
        total_mean = np.mean(X, axis=0)
        X_centered = X - total_mean
        scatter_t = np.cov(X_centered.T) * (X_centered.shape[0] - 1)

        eig_values, eig_vectors = np.linalg.eigh(np.linalg.pinv(scatter_w).dot(scatter_b))
        sorted_indices = np.argsort(eig_values)[::-1]
        eig_vectors = eig_vectors[:, sorted_indices]

        pc = X.dot(eig_vectors[:, :self.n_components])

        if self.n_components == 2:
            if y is None:
                plt.scatter(pc[:, 0], pc[:, 1])
            else:
                colors = ['r', 'g', 'b']
                for cls, color in zip(unique_classes, colors):
                    class_data = pc[y == cls]
                    plt.scatter(class_data[:, 0], class_data[:, 1], c=color)

            plt.xlim(pc[:, 0].min() - 1, pc[:, 0].max() + 1)
            plt.ylim(pc[:, 1].min() - 1, pc[:, 1].max() + 1)
            plt.show()

        return pc

data = load_iris()
X, y = data.data, data.target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

lda = LDA(n_components=2)
X_train_modified = lda.transform(X_train, y_train)

print("Original Data Size:", X_train.shape)
print("Modified Data Size:", X_train_modified.shape)
```



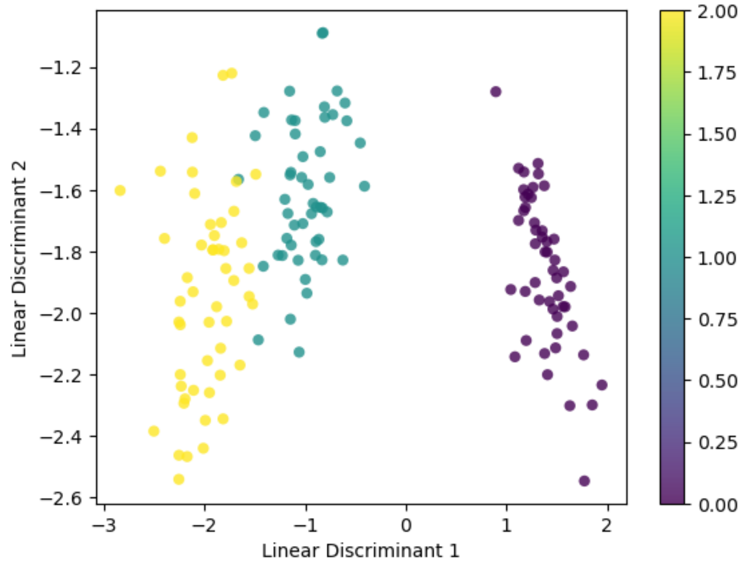

```
Original Data Size: (120, 4)
Modified Data Size: (120, 2)
```

In [7]:

```python
import numpy as np


class LDA:
    def __init__(self, n_components):
        self.n_components = n_components
        self.linear_discriminants = None

    def fit(self, X, y):
        n_features = X.shape[1]
        class_labels = np.unique(y)

        # Within class scatter matrix:
        # SW = sum((X_c - mean_X_c)^2 )

        # Between class scatter:
        # SB = sum( n_c * (mean_X_c - mean_overall)^2 )

        mean_overall = np.mean(X, axis=0)
        SW = np.zeros((n_features, n_features))
        SB = np.zeros((n_features, n_features))
```

```python
        for c in class_labels:
            X_c = X[y == c]
            mean_c = np.mean(X_c, axis=0)
            # (4, n_c) * (n_c, 4) = (4,4) -> transpose
            SW += (X_c - mean_c).T.dot((X_c - mean_c))

            # (4, 1) * (1, 4) = (4,4) -> reshape
            n_c = X_c.shape[0]
            mean_diff = (mean_c - mean_overall).reshape(n_features, 1)
            SB += n_c * (mean_diff).dot(mean_diff.T)

        # Determine SW^-1 * SB
        A = np.linalg.inv(SW).dot(SB)
        # Get eigenvalues and eigenvectors of SW^-1 * SB
        eigenvalues, eigenvectors = np.linalg.eig(A)
        # -> eigenvector v = [:,i] column vector, transpose for easier calculations
        # sort eigenvalues high to low
        eigenvectors = eigenvectors.T
        idxs = np.argsort(abs(eigenvalues))[::-1]
        eigenvalues = eigenvalues[idxs]
        eigenvectors = eigenvectors[idxs]
        # store first n eigenvectors
        self.linear_discriminants = eigenvectors[0 : self.n_components]

    def transform(self, X):
        # project data
        return np.dot(X, self.linear_discriminants.T)


# Testing
if __name__ == "__main__":
    # Imports
    import matplotlib.pyplot as plt
    from sklearn import datasets

    data = datasets.load_iris()
    X, y = data.data, data.target

    # Project the data onto the 2 primary linear discriminants
    lda = LDA(2)
    lda.fit(X, y)
    X_projected = lda.transform(X)

    print("Shape of X:", X.shape)
    print("Shape of transformed X:", X_projected.shape)

    x1, x2 = X_projected[:, 0], X_projected[:, 1]

    plt.scatter(
        x1, x2, c=y, edgecolor="none", alpha=0.8, cmap="viridis"
    )

    plt.xlabel("Linear Discriminant 1")
    plt.ylabel("Linear Discriminant 2")
    plt.colorbar()
    plt.show()
```

```
Shape of X: (150, 4)
Shape of transformed X: (150, 2)
```



Linear Discriminant 1
Linear Discriminant 2
−3
−2
−1
0
1
2
−1.2
−1.4
−1.6
−1.8
−2.0
−2.2
−2.4
−2.6
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00