# CS 5783 - Machine Learning - Homework 1

Romit Maulik

September 2018

## 1 Question 1a: Quad-Tree

Code appended as PDF to this document

## 2 Question 1b: kDTree (revised as per meeting)

Code appended as PDF to this document

## 3 Question 2: Dataset generation

Dataset generation embedded in both kDTree and Linear classifier code. Result of dataset generation shown below

## 4 Question 3: Linear Classifier

Result of linear classification:

## 5 Question 4a: Quad-Tree Classifier

Result of Quad-Tree classification:

## 6 Question 4b: KDTree Classifier (revised as per meeting)

Result of Quad-Tree classification:

## 7 Question 5: 10 distribution sampling

Result of linear, Quad and kD tree classifiers for 10 distribution sampling

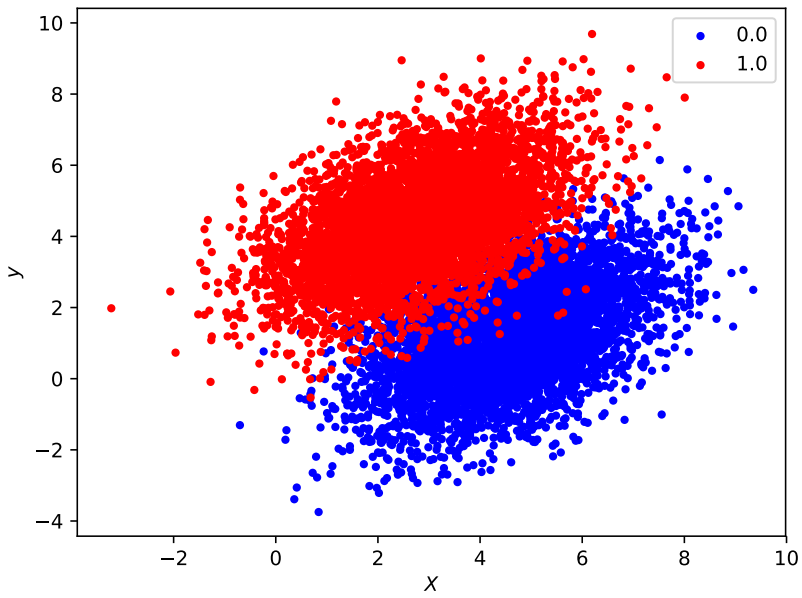


Figure 1: Samples drawn from two multivariate normal distributions

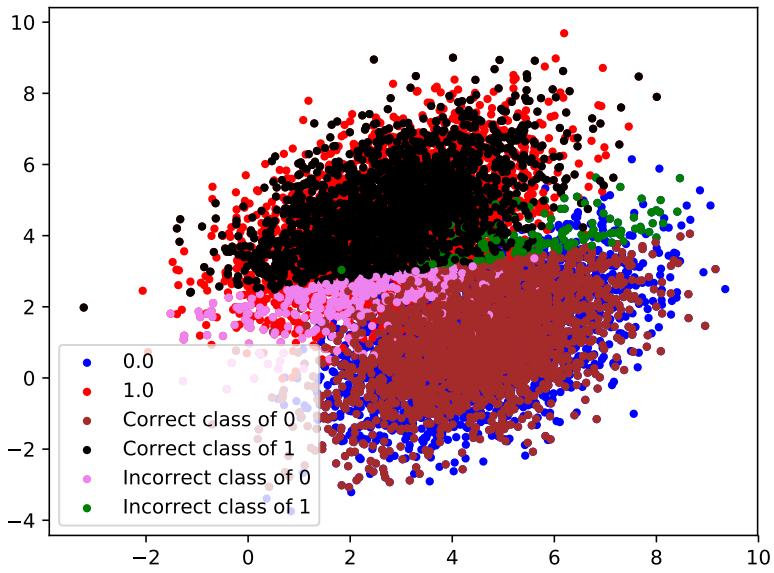


Figure 2: Result of linear classifier (2 distribution sampling). Classification accuracy of 0.912.

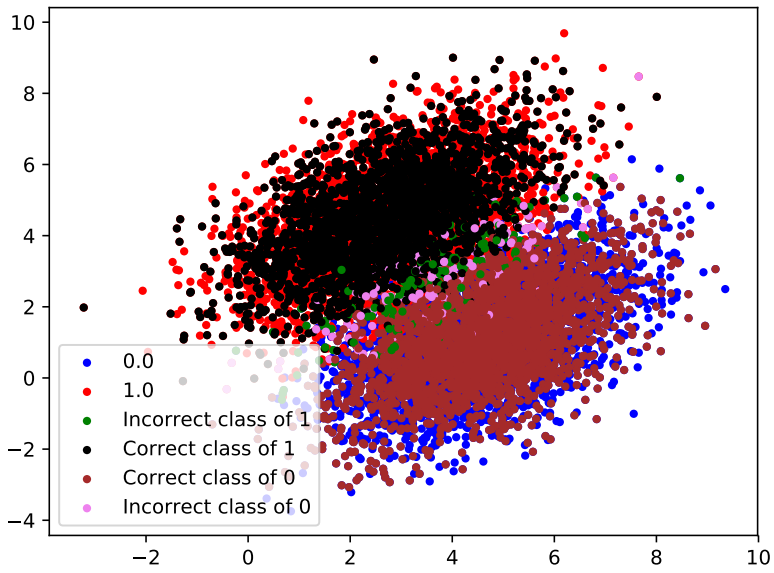


Figure 3: Result of kD Tree classifier (2 distribution sampling). Classification accuracy of 0.948.

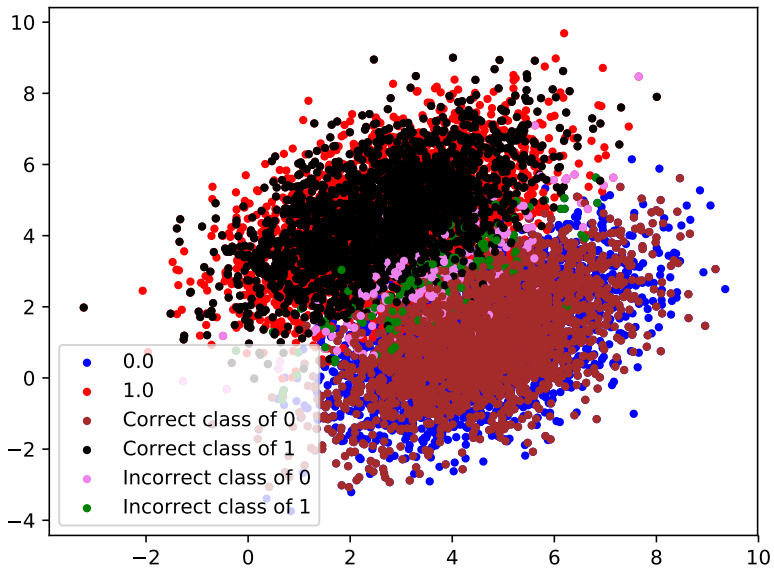


Figure 4: Result of kD Tree classifier (2 distribution sampling). Classification accuracy of 0.947.

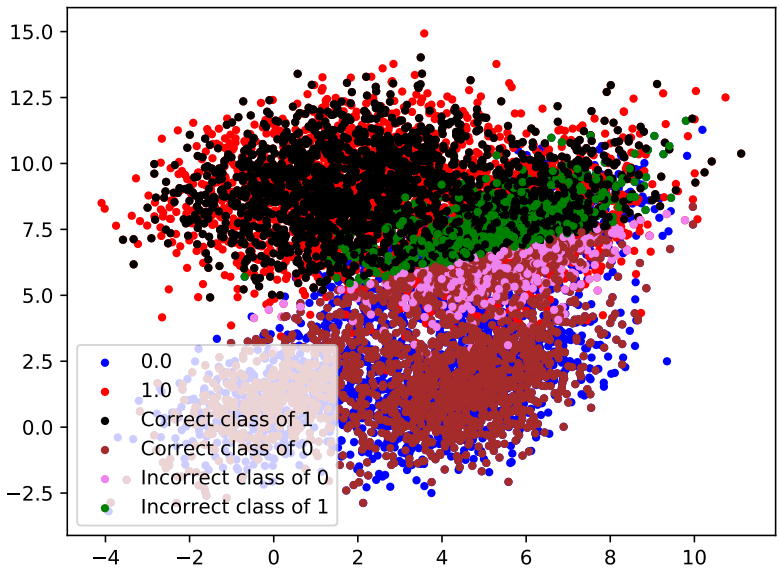


Figure 5: Result of linear classifier (10 distribution sampling). Classification accuracy of 0.832.

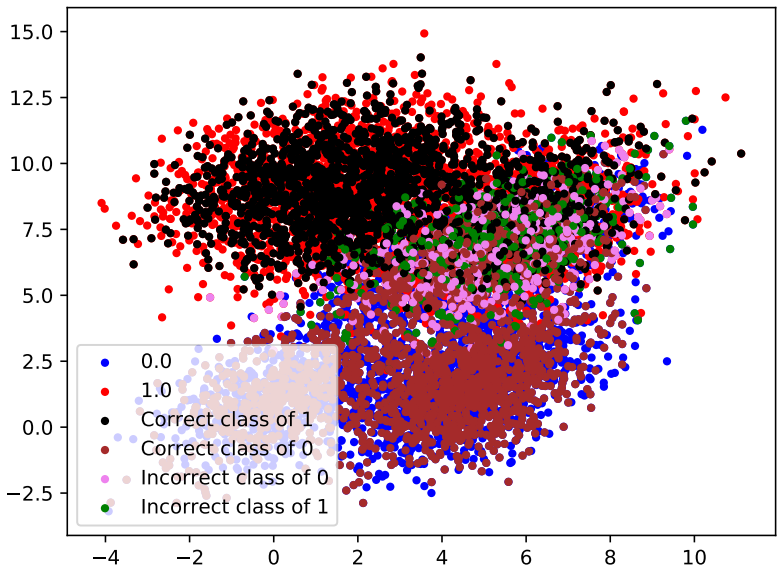


Figure 6: Result of Quad Tree classifier (10 distribution sampling). Classification accuracy of 0.806.

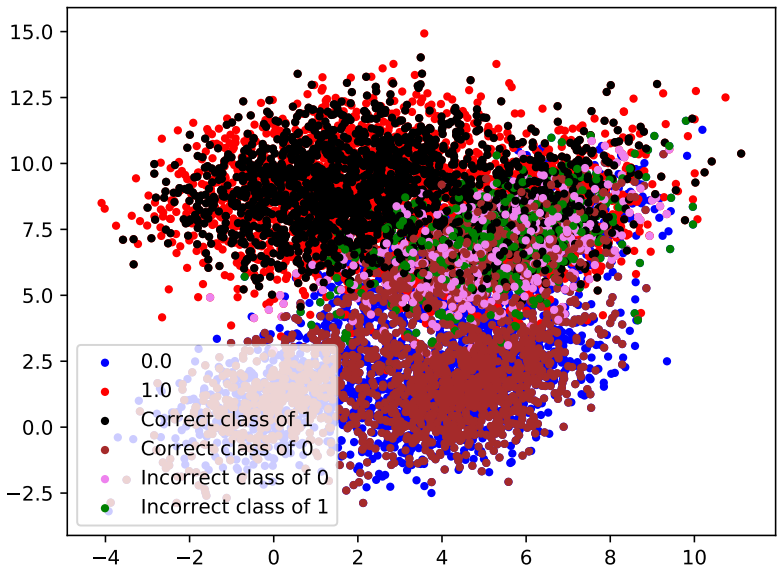


Figure 7: Result of KD Tree classifier (10 distribution sampling). Classification accuracy of 0.803.

```
import numpy as np
import matplotlib.pyplot as plt
from collections import OrderedDict


#Defining class object for KD Tree
class KDTree:
    #Define a base initializer
    def __init__(self, matrix):
        self.max_depth = 0
        self.split_data = {}
        self.split_index = {}

        remainder = np.shape(matrix)[0]
        while remainder > 0 :
            self.max_depth = self.max_depth + 1
            remainder = int(remainder/2)

        print('The maximum depth of this tree is:', self.max_depth)

        #Split data - first for root at depth = 0
        self.split_data[1] = matrix
        self.split_index[1] = [0,0]#[idx,dim]

        dim = 0
        for depth in range(1,self.max_depth):#Starting at depth = 1

            istart = int(2**depth)
            iend = int(2**(depth + 1))

            for i in range(istart,iend):

                split_idx = np.argsort(self.split_data[i // 2][:, dim])[len(self.split_data[i
// 2][:, dim]) // 2]
                self.split_index[i] = [split_idx,dim]#Splitting index of previous level, dimen
sion used to split previous level

                if self.split_data[i//2].size > 3:
                    median_val = np.median(self.split_data[i//2][:,dim])

                    if (i % 2 == 0):#Left
                        self.split_data[i] = self.split_data[i // 2][np.where(self.split_data[
i // 2][:, dim] < median_val)]
                    else:#Right
                        self.split_data[i] = self.split_data[i // 2][np.where(self.split_data[
i // 2][:, dim] >= median_val)]


            if dim == 0:
                dim = 1
            elif dim == 1:
                dim = 0

    def traverse_tree(self,vector):
        self.vector = vector

        dim = 0
        itrack = 1
        for depth in range(1, self.max_depth):  # Starting at depth = 1
            median_val = np.median(self.split_data[itrack][:, dim])

            if vector[dim] < median_val:
                itrack = 2*itrack
```

```
            else:
                itrack = 2*itrack + 1

            if dim == 0:
                dim = 1
            elif dim == 1:
                dim = 0

        return itrack

    def nearest_neighbor(self,vector):

        itrack = self.traverse_tree(vector)
        #Query leaf node
        best_dist = np.infty
        for i in range(np.shape(self.split_data[itrack])[0]):

            new_dist = np.sum(np.square(np.abs(vector-self.split_data[itrack][i,0:2])))

            if new_dist < best_dist:
                best_point = self.split_data[itrack][i,:]
                best_dist = new_dist

        for depth in range(self.max_depth-1,0,-1):
            split_idx = self.split_index[itrack][0]#For the current node - what index in the p
arent was used to split the feature
            itrack = itrack//2

            #Check distance
            new_dist = np.sum(np.square(np.abs(self.vector-self.split_data[itrack][split_idx,0
:2])))

            if new_dist < best_dist:#Hyperplane intersection - needs checking
                best_dist = new_dist

                #Search all of this sub-tree for lowest distance (you're in trouble if this ha
ppens late in your upward traversal)
                best_idx = np.argmin(np.sum(np.square(self.split_data[itrack][:,0:2]-vector)))
                best_point = self.split_data[itrack][best_idx, :]

        return best_dist, best_point



if __name__ == "__main__":
    #Set seed for reproducibility
    np.random.seed(10)

    #Generate some clustered data
    num_clusters = 2
    total_data = 10000
    covariance = [[2, 1], [1, 2]]

    #Labeling zeroes
    for i in range(num_clusters//2):
        mean = [np.random.uniform(low=0,high=6), np.random.uniform(low=1,high=10)]
        cluster_data_temp = np.random.multivariate_normal(mean, covariance, total_data // num_
clusters)
        labels = np.zeros(shape=(total_data//num_clusters, 1))

        if i == 0:
            cluster_data_1 = np.hstack((cluster_data_temp, labels))
        else:
```

```
            cluster_data_temp = np.hstack((cluster_data_temp, labels))
            cluster_data_1 = np.concatenate((cluster_data_1,cluster_data_temp),axis=0)

    # Labeling ones
    for i in range(num_clusters // 2):
        mean = [np.random.uniform(low=0, high=6), np.random.uniform(low=1, high=10)]
        cluster_data_temp = np.random.multivariate_normal(mean, covariance, total_data // num_
clusters)
        labels = np.ones(shape=(total_data // num_clusters, 1))

        if i == 0:
            cluster_data_2 = np.hstack((cluster_data_temp, labels))
        else:
            cluster_data_temp = np.hstack((cluster_data_temp, labels))
            cluster_data_2 = np.concatenate((cluster_data_2, cluster_data_temp), axis=0)

    #Concatenate matrix and randomize rows
    matrix_data = np.concatenate((cluster_data_1,cluster_data_2),axis=0)
    np.random.shuffle(matrix_data)

    training_data = matrix_data[0:total_data//2, :]
    testing_data = matrix_data[total_data//2:, :]

    #Checking performance of classifier on testing data
    # Plotting
    scatter_x = matrix_data[:, 0]
    scatter_y = matrix_data[:, 1]
    group = matrix_data[:, 2]
    cdict = {0: 'blue', 1: 'red'}
    #
    fig, ax = plt.subplots()
    for g in np.unique(group):
        ix = np.where(group == g)
        ax.scatter(scatter_x[ix], scatter_y[ix], c=cdict[g], label=g, s=10)
    #

    # Checking performance of classifier on testing data
    tree_object = KDTree(training_data)
    incorrect = 0
    for i in range(np.shape(testing_data)[0]):
        vector = testing_data[i, 0:2]
        classification = int(tree_object.nearest_neighbor(vector)[1][2])

        if classification == 1 and int(testing_data[i, 2]) == 1:
            ax.scatter(vector[0], vector[1], c='black', marker='o', s=10, label='Correct class
 of 1')
        elif classification == 1 and int(testing_data[i, 2]) == 0:
            ax.scatter(vector[0], vector[1], c='green', marker='o', s=10, label='Incorrect cla
ss of 1')
        elif classification == 0 and int(testing_data[i, 2]) == 0:
            ax.scatter(vector[0], vector[1], c='brown', marker='o', s=10, label='Correct class
 of 0')
        elif classification == 0 and int(testing_data[i, 2]) == 1:
            ax.scatter(vector[0], vector[1], c='violet', marker='o', s=10, label='Incorrect cl
ass of 0')

        incorrect = incorrect + np.abs(classification - testing_data[i, 2])

    print('Classification accuracy:', float(1.0 - incorrect / 5000))

    # Complete plotting
    handles, labels = plt.gca().get_legend_handles_labels()
    by_label = OrderedDict(zip(labels, handles))
```

```
    ax.legend(by_label.values(), by_label.keys(), loc='lower left')
    plt.show()
```

```
import numpy as np
import matplotlib.pyplot as plt
from collections import OrderedDict

def linear_classifier_function(matrix):
    temp = np.dot(np.transpose(matrix[:,0:2]),matrix[:,0:2])
    temp = np.linalg.inv(temp)
    temp = np.dot(temp,np.transpose(matrix[:,0:2]))
    beta = np.dot(temp,matrix[:,2])

    return beta

def test_classifier(vector,beta):
    val = np.dot(vector,beta)

    if val > 0.5:
        return 1
    else:
        return 0

if __name__ == "__main__":
    #Set seed for reproducibility
    np.random.seed(10)

    #Generate some clustered data
    num_clusters = 10
    total_data = 10000
    covariance = [[2, 1], [1, 2]]

    #Labeling zeroes
    for i in range(num_clusters//2):
        mean = [np.random.uniform(low=0,high=6), np.random.uniform(low=1,high=10)]
        cluster_data_temp = np.random.multivariate_normal(mean, covariance, total_data // num_
clusters)
        labels = np.zeros(shape=(total_data//num_clusters, 1))

        if i == 0:
            cluster_data_1 = np.hstack((cluster_data_temp, labels))
        else:
            cluster_data_temp = np.hstack((cluster_data_temp, labels))
            cluster_data_1 = np.concatenate((cluster_data_1,cluster_data_temp),axis=0)

    # Labeling ones
    for i in range(num_clusters // 2):
        mean = [np.random.uniform(low=0, high=6), np.random.uniform(low=1, high=10)]
        cluster_data_temp = np.random.multivariate_normal(mean, covariance, total_data // num_
clusters)
        labels = np.ones(shape=(total_data // num_clusters, 1))

        if i == 0:
            cluster_data_2 = np.hstack((cluster_data_temp, labels))
        else:
            cluster_data_temp = np.hstack((cluster_data_temp, labels))
            cluster_data_2 = np.concatenate((cluster_data_2, cluster_data_temp), axis=0)

    #Concatenate matrix and randomize rows
    matrix_data = np.concatenate((cluster_data_1,cluster_data_2),axis=0)
    np.random.shuffle(matrix_data)

    training_data = matrix_data[0:total_data//2, :]
    testing_data = matrix_data[total_data//2:, :]

    # Plotting
```

```
    scatter_x = matrix_data[:, 0]
    scatter_y = matrix_data[:, 1]
    group = matrix_data[:, 2]
    cdict = {0: 'blue', 1: 'red'}
    #
    fig, ax = plt.subplots()
    for g in np.unique(group):
        ix = np.where(group == g)
        ax.scatter(scatter_x[ix], scatter_y[ix], c=cdict[g], label=g, s=10)
    #
    cdict_class = {0: 'black', 1: 'yellow'}

    # Checking performance of classifier on testing data
    beta = linear_classifier_function(training_data)
    incorrect = 0
    for i in range(np.shape(testing_data)[0]):
        vector = testing_data[i, 0:2]
        classification = test_classifier(vector,beta)
        incorrect = incorrect + np.abs(classification - testing_data[i, 2])

        if classification == 1 and int(testing_data[i,2]) == 1:
            ax.scatter(vector[0], vector[1], c='black', marker='o', s=10,label='Correct classi
fication of 1')
        elif classification == 1 and int(testing_data[i,2]) == 0:
            ax.scatter(vector[0], vector[1], c='green', marker='o', s=10, label='Incorrect cla
ssification of 1')
        elif classification == 0 and int(testing_data[i,2]) == 0:
            ax.scatter(vector[0], vector[1], c='brown', marker='o', s=10, label='Correct class
ification of 0')
        elif classification == 0 and int(testing_data[i,2]) == 1:
            ax.scatter(vector[0], vector[1], c='violet', marker='o', s=10, label='Incorrect cl
assification of 0')

    print('Classification accuracy:', float(1.0 - incorrect / 5000))

    # Complete plotting
    handles, labels = plt.gca().get_legend_handles_labels()
    by_label = OrderedDict(zip(labels, handles))
    ax.legend(by_label.values(), by_label.keys())
    plt.show()
```

```
import numpy as np
import matplotlib.pyplot as plt
from collections import OrderedDict


#Defining class object for KD Tree
class KDTree:
    #Define a base initializer
    def __init__(self, matrix):
        self.feature_means = np.mean(matrix,axis=0) #Rows are samples, columns are features
        self.max_depth = 0
        self.global_matrix = matrix
        self.location = 'root'
        self.depth = 0

        remainder = np.shape(matrix)[0]
        while remainder > 0 :
            self.max_depth = self.max_depth + 1
            remainder = int(remainder/4)

        print('The maximum depth of this tree is:', self.max_depth)

    #Define a traversal
    def traverse_tree(self,vector):

        self.local_matrix = self.global_matrix
        self.temp_matrix = self.global_matrix
        self.vector = vector

        for depth in range(self.max_depth):
            #Classify vector first - changes level
            self.classify()
            #Update feature mean to this particular class
            self.update_local_matrix()
            # Check for early exit
            if self.local_matrix.size == 0 or depth == self.max_depth - 2:
                break
            else:
                self.feature_means = np.mean(self.local_matrix, axis=0)
                self.temp_matrix = self.local_matrix

            #Increase depth
            self.depth = self.depth + 1

        nearest_neighbor = self.nearest_neighbor_classify()

        #print('The nearest neighbor is:',nearest_neighbor[0:2])
        #print('The classification is:', nearest_neighbor[2])

        return nearest_neighbor


    #Find nearest neighbor from array and classify
    def nearest_neighbor_classify(self):

        idx = np.abs(self.temp_matrix[:,0:2] - np.reshape(self.vector,(1,2)))
        idx = np.sqrt(idx[:,0]**2 + idx[:,1]**2).argmin()
        return self.temp_matrix[idx,:]


    #Define a classifier
    def classify(self):
        if self.vector[0] > self.feature_means[0] and self.vector[1] > self.feature_means[1]:
```

```
            self.location = 'top-right'

        elif self.vector[0] > self.feature_means[0] and self.vector[1] < self.feature_means[1]
:
            self.location = 'bottom-right'

        elif self.vector[0] < self.feature_means[0] and self.vector[1] < self.feature_means[1]
:
            self.location = 'bottom-left'

        else:
            self.location = 'top-left'


    #Update local matrix (i.e. data in same class as current class of vector)
    def update_local_matrix(self):
        if self.location == 'top-right':

            self.local_matrix = self.local_matrix[np.where(
                (self.local_matrix[:, 0] > self.feature_means[0]) & (self.local_matrix[:, 1] >
 self.feature_means[1]))]

        elif self.location == 'bottom-right':

            self.local_matrix = self.local_matrix[np.where(
                (self.local_matrix[:, 0] > self.feature_means[0]) & (self.local_matrix[:, 1] <
 self.feature_means[1]))]

        elif self.location == 'bottom-left':

            self.local_matrix = self.local_matrix[np.where(
                (self.local_matrix[:, 0] < self.feature_means[0]) & (self.local_matrix[:, 1] <
 self.feature_means[1]))]

        else:

            self.local_matrix = self.local_matrix[np.where(
                (self.local_matrix[:, 0] < self.feature_means[0]) & (self.local_matrix[:, 1] >
 self.feature_means[1]))]



if __name__ == "__main__":
    #Set seed for reproducibility
    np.random.seed(10)

    #Generate some clustered data
    num_clusters = 2
    total_data = 10000
    covariance = [[2, 1], [1, 2]]

    #Labeling zeroes
    for i in range(num_clusters//2):
        mean = [np.random.uniform(low=0,high=6), np.random.uniform(low=1,high=10)]
        cluster_data_temp = np.random.multivariate_normal(mean, covariance, total_data // num_
clusters)
        labels = np.zeros(shape=(total_data//num_clusters, 1))

        if i == 0:
            cluster_data_1 = np.hstack((cluster_data_temp, labels))
        else:
            cluster_data_temp = np.hstack((cluster_data_temp, labels))
            cluster_data_1 = np.concatenate((cluster_data_1,cluster_data_temp),axis=0)
```

```
    # Labeling ones
    for i in range(num_clusters // 2):
        mean = [np.random.uniform(low=0, high=6), np.random.uniform(low=1, high=10)]
        cluster_data_temp = np.random.multivariate_normal(mean, covariance, total_data // num_
clusters)
        labels = np.ones(shape=(total_data // num_clusters, 1))

        if i == 0:
            cluster_data_2 = np.hstack((cluster_data_temp, labels))
        else:
            cluster_data_temp = np.hstack((cluster_data_temp, labels))
            cluster_data_2 = np.concatenate((cluster_data_2, cluster_data_temp), axis=0)

    #Concatenate matrix and randomize rows
    matrix_data = np.concatenate((cluster_data_1,cluster_data_2),axis=0)
    np.random.shuffle(matrix_data)

    training_data = matrix_data[0:total_data//2, :]
    testing_data = matrix_data[total_data//2:, :]

    #Plotting
    scatter_x = matrix_data[:,0]
    scatter_y = matrix_data[:,1]
    group = matrix_data[:,2]
    cdict = {0: 'blue', 1: 'red'}
    #
    fig, ax = plt.subplots()
    for g in np.unique(group):
        ix = np.where(group == g)
        ax.scatter(scatter_x[ix], scatter_y[ix], c=cdict[g], label=g, s=10)
    #

    #Checking performance of classifier on testing data
    tree_object = KDTree(training_data)
    incorrect = 0
    for i in range(np.shape(testing_data)[0]):
        vector = testing_data[i,0:2]
        classification = int(tree_object.traverse_tree(vector)[2])

        if classification == 1 and int(testing_data[i,2]) == 1:
            ax.scatter(vector[0], vector[1], c='black', marker='o', s=10,label='Correct classi
fication of 1')
        elif classification == 1 and int(testing_data[i,2]) == 0:
            ax.scatter(vector[0], vector[1], c='green', marker='o', s=10, label='Incorrect cla
ssification of 1')
        elif classification == 0 and int(testing_data[i,2]) == 0:
            ax.scatter(vector[0], vector[1], c='brown', marker='o', s=10, label='Correct class
ification of 0')
        elif classification == 0 and int(testing_data[i,2]) == 1:
            ax.scatter(vector[0], vector[1], c='violet', marker='o', s=10, label='Incorrect cl
assification of 0')

        incorrect = incorrect + np.abs(classification-testing_data[i,2])

    print ('Classification accuracy:',float(1.0-incorrect/5000))

    #Complete plotting
    handles, labels = plt.gca().get_legend_handles_labels()
    by_label = OrderedDict(zip(labels, handles))
    ax.legend(by_label.values(), by_label.keys())
    plt.show()
```