Pramod Sivaram Kaushik(201407538)
Mohd Salman Khan(201305513)
Sajal Sharma(201305526)

Project 3 : Data Clustering

Datasets:

1. *Iris Dataset* :

This is perhaps the best known database to be found in the pattern recognition literature. Fisher's paper is a classic in the field and is referenced frequently to this day. (See Duda & Hart, for example.) The data set contains 3 classes of 50 instances each, where each class refers to a type of iris plant. One class is linearly separable from the other 2; the latter are NOT linearly separable from each other.

Predicted attribute: class of iris plant.

This is an exceedingly simple domain.

This data differs from the data presented in Fishers article (identified by Steve Chadwick, spchadwick '@' espeedaz.net). The 35th sample should be: 4.9,3.1,1.5,0.2,"Iris-setosa" where the error is in the fourth feature. The 38th sample: 4.9,3.6,1.4,0.1,"Iris-setosa" where the errors are in the second and third features.

Attribute Information :
1. sepal length in cm
2. sepal width in cm
3. petal length in cm
4. petal width in cm
5. class:
-- Iris Setosa
-- Iris Versicolour
-- Iris Virginica

2. *Wine Dataset* :

These data are the results of a chemical analysis of wines grown in the same region in Italy but derived from three different cultivars. The analysis determined the quantities of 13 constituents found in each of the three types of wines.

I think that the initial data set had around 30 variables, but for some reason I only have the 13 dimensional version. I had a list of what the 30 or so variables were, but a.) I lost it, and b.), I would not know which 13 variables are included in the set.

The attributes are (dontated by Riccardo Leardi, riclea '@' anchem.unige.it)
1) Alcohol
2) Malic acid
3) Ash
4) Alcalinity of ash
5) Magnesium
6) Total phenols
7) Flavanoids
8) Nonflavanoid phenols
9) Proanthocyanins
10)Color intensity
11)Hue
12)OD280/OD315 of diluted wines
13)Proline

In a classification context, this is a well posed problem with "well behaved" class structures. A good data set for first testing of a new classifier, but not very challenging.

Attribute Information:
All attributes are continuous .No statistics available, but suggest to standardise variables for certain uses (e.g. for us with classifiers which are NOT scale invariant)

Validity Measures

External Measures:

External measures are used when it is known apriori that which data point belong to which cluster.
There are a number of external measures that can be used, two of them are given below:

1. Correlation :

Correlation between two matrices is defines as the similarity between the matrices, i.e. how similar are the matrices to each other. In the assignment we have actual Labels and the predicted labels(given by our algorithm), but it might happen that the labels do not match even when the datapoints are correclty clustered into the same cluster. So, to tackle this issue we used correlation.

We created two 2-D matrices from actual labels and predicted labels. These 2-D matrices are of the shape n * n(where n are the number of data points in the sample), the (i,j)th location in the 2-D matrix will be 1 if the ith and jth datapoint lies in the same cluster and it will be 0 if they do not lie in the same cluster.

By following the above mentioned apporach we got two 2-D matrices and the correlation is found using python's numpy.corrcoef(matrix1, matrix2).

2. Purity :

To compute purity , each cluster is assigned to the class which is most frequent in the cluster, and then the accuracy of this assignment is measured by counting the number of correctly assigned documents and dividing by N.

It is defined as below :
Purity of a cluster = the number of occurrences of the most frequent class / the size of the cluster
and then the purity of all the clusters are added together to get the final purity of the dataset.

Internal Measures:

External measures are used when it is NOT known apriori that which data point belong to which cluster.
There are a number of internal measures that can be used, two of them are given below:

1. Root Mean Squared Error:

It is calculated cluster wise. The mean of each cluster is calculated and then the euclidean distance is calculated of each data point in the same cluster to its mean and is added up. Doing this we get the sum of distances of each data point to its clustered mean, now this is repeated for each cluster and all the distances are added up to get the root mean squared error of the complete dataset.

2. Dunn Index:

The Dunn index (DI) (introduced by J. C. Dunn in 1974) is a metric for evaluating clustering algorithms. This is part of a group of validity indices including the Davies–Bouldin index, in that it is an internal evaluation scheme, where the result is based on the clustered data itself. As do all other such indices, the aim is to identify sets of clusters that are compact, with a small variance between members of the cluster, and well separated, where the means of different clusters are sufficiently far apart, as compared to the within cluster variance. For a given assignment of clusters, a higher Dunn index indicates better clustering. One of the drawbacks of using this, is the computational cost as the number of clusters and dimensionality of the data increase.

Algorithms Used

1. kmeans

Let X = {x1,x2,x3,……..,xn} be the set of data points and V = {v1,v2,…….,vc} be the set of centers.

1) Randomly select 'c' cluster centers.
2) Calculate the distance between each data point and cluster centers.
3) Assign the data point to the cluster center whose distance from the cluster center is minimum of all the cluster centers..
4) Recalculate the new cluster center.
5) Recalculate the distance between each data point and new obtained cluster centers.
6) If no data point was reassigned then stop, otherwise repeat from step 3).

Example:

As a simple illustration of a k-means algorithm, consider the following data set consisting of the scores of two variables on each of seven individuals:

Subject	A	B
1	1.0	1.0
2	1.5	2.0
3	3.0	4.0
4	5.0	7.0
5	3.5	5.0
6	4.5	5.0
7	3.5	4.5

This data set is to be grouped into two clusters. As a first step in finding a sensible initial partition, let the A & B values of the two individuals furthest apart (using the Euclidean distance measure), define the initial cluster means, giving:

	Individual	Mean Vector (centroid)
Group 1	1	(1.0, 1.0)
Group 2	4	(5.0, 7.0)

The remaining individuals are now examined in sequence and allocated to the cluster to which they are closest, in terms of Euclidean distance to the cluster mean. The mean vector is recalculated each time a new member is added. This leads to the following series of steps:

	Cluster 1		Cluster 2	
Step	Individual	Mean Vector (centroid)	Individual	Mean Vector (centroid)
1	1	(1.0, 1.0)	4	(5.0, 7.0)
2	1, 2	(1.2, 1.5)	4	(5.0, 7.0)
3	1, 2, 3	(1.8, 2.3)	4	(5.0, 7.0)
4	1, 2, 3	(1.8, 2.3)	4, 5	(4.2, 6.0)
5	1, 2, 3	(1.8, 2.3)	4, 5, 6	(4.3, 5.7)
6	1, 2, 3	(1.8, 2.3)	4, 5, 6, 7	(4.1, 5.4)

Now the initial partition has changed, and the two clusters at this stage having the following characteristics:

	Individual	Mean Vector (centroid)
Cluster 1	1, 2, 3	(1.8, 2.3)
Cluster 2	4, 5, 6, 7	(4.1, 5.4)

But we cannot yet be sure that each individual has been assigned to the right cluster. So, we compare each

individual's distance to its own cluster mean and to that of the opposite cluster. And we find:

Individual	Distance to mean (centroid) of Cluster 1	Distance to mean (centroid) of Cluster 2
1	1.5	5.4
2	0.4	4.3
3	2.1	1.8
4	5.7	1.8
5	3.2	0.7
6	3.8	0.6
7	2.8	1.1

Only individual 3 is nearer to the mean of the opposite cluster (Cluster 2) than its own (Cluster 1). In other words, each individual's distance to its own cluster mean should be smaller that the distance to the other cluster's mean (which is not the case with individual 3). Thus, individual 3 is relocated to Cluster 2 resulting in the new partition:

	Individual	Mean Vector (centroid)
Cluster 1	1, 2	(1.3, 1.5)
Cluster 2	3, 4, 5, 6, 7	(3.9, 5.1)

The iterative relocation would now continue from this new partition until no more relocations occur. However, in this example each individual is now nearer its own cluster mean than that of the other cluster and the iteration stops, choosing the latest partitioning as the final cluster solution.
Also, it is possible that the k-means algorithm won't find a final solution. In this case it would be a good idea to consider stopping the algorithm after a pre-chosen maximum of iterations.

2. Hierarchical

Let X = {x1, x2, x3, ..., xn} be the set of data points.

1) Begin with the disjoint clustering having level L(0) = 0 and sequence number m = 0.

2) Find the least distance pair of clusters in the current clustering, say pair (r), (s), according to d[(r),(s)] = min d[(i),(j)] where the minimum is over all pairs of clusters in the current clustering.

3) Increment the sequence number: m = m +1.Merge clusters (r) and (s) into a single cluster to form the next clustering m. Set the level of this clustering to L(m) = d[(r),(s)].

4) Update the distance matrix, D, by deleting the rows and columns corresponding to clusters (r) and (s) and adding a row and column corresponding to the newly formed cluster. The distance between the new cluster, denoted (r,s) and old cluster(k) is defined in this way: d[(k), (r,s)] = min (d[(k),(r)], d[(k),(s)]).

5) If all the data points are in one cluster then stop, else repeat from step 2).
Divisive Hierarchical clustering - It is just the reverse of Agglomerative Hierarchical approach.

Source Code

1. kmeans

```
import sys
import csv
import math
import numpy
import random
from sets import Set
import matplotlib.pyplot as plt
import matplotlib.colors

def populateClassMap(allLabels):
        classMap = {}
        j = 0
        for i in allLabels:
                classMap[i] = j
                j = j + 1
        return classMap

'''
Loading data from the input file
into a list named dataset. The
function then returns the list.
'''
def loadData(filePath, fileName):
        fullFilePath = filePath + "/" + fileName
        lines = csv.reader(open(fullFilePath, "rb"))
        dataset = list(lines)
        allLabels = []
        for i in range(0, len(dataset)):
                allLabels.append(dataset[i][len(dataset[0]) - 1])
        allLabels = set(allLabels)
        k = len(allLabels)
        classMap = populateClassMap(allLabels)
        actualLabels = []
        for i in range(0, len(dataset)):
                actualLabels.append(classMap[dataset[i][len(dataset[0]) - 1]])
        i = 0
        for x in dataset:
                del x[len(dataset[i]) - 1]
                i = i + 1
        for i in range(len(dataset)):
                dataset[i] = [float(x) for x in dataset[i]]
        print actualLabels
        return dataset, k, actualLabels

'''
Randomly assigning the initial centroids
for each cluster. K random centroids are
assigned and returned.
'''
def randomClusterCenters(data, k):
        randomCentroids = [[0 for x in range(len(data[0]))] for x in range(k)]
        for i in range(0, k):
                for j in range(0, len(data[0])):
                        tempList = []
                        for l in range(0, len(data)):
                                tempList.append(data[l][j])
                        randomPoint = random.choice(tempList)
                        randomCentroids[i][j] = randomPoint
```

```
        return randomCentroids

def EuclideanDistance(instance1, instance2, length):
        distance = 0
        for x in range(length):
                distance += pow((instance1[x] - instance2[x]), 2)
        return math.sqrt(distance)

def calculateLabels(data, centroids, k):
        labelsAssigned = []
        for i in range(0 , len(data)):
                minDistance = sys.maxint
                label = 100 # random label assigned, going to change eventually
                for j in range(0, k):
                        tempDistance = EuclideanDistance(data[i], centroids[j], len(data[0]))
                        if tempDistance < minDistance:
                                minDistance = tempDistance
                                label = j
                labelsAssigned.append(label)
        return labelsAssigned

def recalculateCentroids(data, labelsAssigned, k):
        noOfSamplesInLabels = [0 for x in range(0, k)]
        centroids = [[0 for x in range(len(data[0]))] for x in range(k)]
        for i in range(0, len(data)):
                noOfSamplesInLabels[labelsAssigned[i]] = noOfSamplesInLabels[labelsAssigned[i]] + 1
                for j in range(0, len(data[0])):
                        centroids[labelsAssigned[i]][j] = centroids[labelsAssigned[i]][j] + data[i][j]
        for i in range(0, k):
                for j in range(0, len(centroids[0])):
                        centroids[i][j] = float(float(centroids[i][j]) / float(noOfSamplesInLabels[i]))
        return centroids

def checkChange(oldCentroids, newCentroids, k):
        sum = 0.0
        for i in range(0, k):
                sum = sum + EuclideanDistance(oldCentroids[i], newCentroids[i], len(oldCentroids[0]))
        if sum == 0.0:
                return False
        else:
                return True

def kmeans(data, k):
        # randomly selecting k cluster centers
        randomCentroids = randomClusterCenters(data, k)
        oldCentroids = randomCentroids
        i = 0
        while True:
                i = i + 1
                labelsAssigned = calculateLabels(data, oldCentroids, k)
                newCentroids = recalculateCentroids(data, labelsAssigned, k)
                returnValue = checkChange(oldCentroids, newCentroids, k)
                if(returnValue == False):
                        break
                oldCentroids = newCentroids
        print "converge in ", i
        print labelsAssigned
        return labelsAssigned, newCentroids

def calculateCorrelation(actualLabels, predictedLabels):
        array1 = []
        array2 = []
        for i in range(0, len(actualLabels)):
```

```
                for j in range(0, len(actualLabels)):
                        if actualLabels[i] == actualLabels[j]:
                                array1.append(1)
                        else:
                                array1.append(0)
        for i in range(0, len(predictedLabels)):
                for j in range(0, len(predictedLabels)):
                        if predictedLabels[i] == predictedLabels[j]:
                                array2.append(1)
                        else:
                                array2.append(0)
        correlationMatrix = numpy.corrcoef(array1, array2)
        return correlationMatrix[0][1]

def generateRandomRGB():
        hexDigits = ['0','1','2','3','4','5','6','7','8','9','a','b','c','d','e','f']
        r1 = random.choice(hexDigits)
        r2 = random.choice(hexDigits)
        g1 = random.choice(hexDigits)
        g2 = random.choice(hexDigits)
        b1 = random.choice(hexDigits)
        b2 = random.choice(hexDigits)
        randomColour = "#" + str(r1) + str(r2) + str(g1) + str(g2) + str(b1) + str(b2)
        return randomColour

def plotResults(actualLabels, predictedLabels, data, k):
        # Plotting actual labels
        colours = []
        for i in range(0, k):
                X = []
                Y = []
                for j in range(0, len(data)):
                        if actualLabels[j] == i:
                                X.append(data[j][0])
                                Y.append(data[j][1])
                randomColour = generateRandomRGB()
                colours.append(randomColour)
                plt.subplot(2,1,1)
                plt.plot(X, Y, randomColour, linestyle=':')
                plt.ylabel('Actual', fontsize=20)
        # Plotting predicted labels
        for i in range(0, k):
                X = []
                Y = []
                for j in range(0, len(data)):
                        if predictedLabels[j] == i:
                                X.append(data[j][0])
                                Y.append(data[j][1])
                plt.subplot(2,1,2)
                plt.plot(X, Y, colours[i], linestyle=':')
                plt.ylabel('Predicted', fontsize=20)
        plt.show()

def calRootMeanSquaredError(centroids, data, predictedLabels):
        rootMeanSquaredError = 0.0
        for i in range(0, len(data)):
                rootMeanSquaredError = rootMeanSquaredError +
EuclideanDistance(centroids[predictedLabels[i]], data[i], len(data[0]))
        return rootMeanSquaredError

def calDunnIndex(data, centroids, predictedLabels, k):
        maxDistancesInCluster = []
        for i in range(0, k):
```

```
                tempData = []
                for j in range(0, len(data)):
                        if predictedLabels[j] == i:
                                tempData.append(data[j])
                maxDistance = -sys.maxint
                for l in range(0, len(tempData)):
                        for m in range(0, len(tempData)):
                                temp = EuclideanDistance(tempData[l], tempData[m], len(tempData[0]))
                                if temp > maxDistance:
                                        maxDistance = temp
                maxDistancesInCluster.append(maxDistance)
        denominator = max(maxDistancesInCluster)
        minDistance = sys.maxint
        for i in range(0, len(centroids)):
                for j in range(0, len(centroids)):
                        if i != j:
                                temp = EuclideanDistance(centroids[i], centroids[j], len(centroids[0]))
                                if temp < minDistance:
                                        minDistance = temp
        numerator = minDistance
        dunnIndex = float(numerator / denominator)
        return dunnIndex

def calPurity(actualLabels, predictedLabels, k):
        purity = 0.0
        for i in range(0, k):
                noOfLabelsOfEachClassInTheCluster = [0 for x in range(0, k)]
                if i == 0:
                        startPoint = 0
                        for j in range(1, len(actualLabels)):
                                if actualLabels[j] != actualLabels[j - 1]:
                                        endPoint = j - 1
                                        break
                        for j in range(startPoint, endPoint + 1):
                                noOfLabelsOfEachClassInTheCluster[predictedLabels[j]]                                =
noOfLabelsOfEachClassInTheCluster[predictedLabels[j]] + 1
                        purity = purity + max(noOfLabelsOfEachClassInTheCluster)
                elif i == k - 1:
                        startPoint = endPoint + 1
                        endPoint = len(actualLabels) - 1
                        for j in range(startPoint, endPoint + 1):
                                noOfLabelsOfEachClassInTheCluster[predictedLabels[j]]                                =
noOfLabelsOfEachClassInTheCluster[predictedLabels[j]] + 1
                        purity = purity + max(noOfLabelsOfEachClassInTheCluster)
                else:
                        startPoint = endPoint + 1
                        for j in range(startPoint + 1, len(actualLabels)):
                                if actualLabels[j] != actualLabels[j - 1]:
                                        endPoint = j - 1
                                        break
                        for j in range(startPoint, endPoint + 1):
                                noOfLabelsOfEachClassInTheCluster[predictedLabels[j]]                                =
noOfLabelsOfEachClassInTheCluster[predictedLabels[j]] + 1
                        purity = purity + max(noOfLabelsOfEachClassInTheCluster)
        purity = float(purity / len(actualLabels))
        return purity

def calMapping(actualLabels, predictedLabels, k):
        labelMap = {}
        for i in range(0, k):
                noOfLabelsOfEachClassInTheCluster = [0 for x in range(0, k)]
                if i == 0:
                        startPoint = 0
```

```
                        for j in range(1, len(actualLabels)):
                                if actualLabels[j] != actualLabels[j - 1]:
                                        endPoint = j - 1
                                        break
                        for j in range(startPoint, endPoint + 1):
                                noOfLabelsOfEachClassInTheCluster[predictedLabels[j]] =
noOfLabelsOfEachClassInTheCluster[predictedLabels[j]] + 1
                elif i == k - 1:
                        startPoint = endPoint + 1
                        endPoint = len(actualLabels) - 1
                        for j in range(startPoint, endPoint + 1):
                                noOfLabelsOfEachClassInTheCluster[predictedLabels[j]] =
noOfLabelsOfEachClassInTheCluster[predictedLabels[j]] + 1
                else:
                        startPoint = endPoint + 1
                        for j in range(startPoint + 1, len(actualLabels)):
                                if actualLabels[j] != actualLabels[j - 1]:
                                        endPoint = j - 1
                                        break
                        for j in range(startPoint, endPoint + 1):
                                noOfLabelsOfEachClassInTheCluster[predictedLabels[j]] =
noOfLabelsOfEachClassInTheCluster[predictedLabels[j]] + 1
                maxVal = -sys.maxint
                for j in range(0, len(noOfLabelsOfEachClassInTheCluster)):
                        temp = noOfLabelsOfEachClassInTheCluster[j]
                        if temp > maxVal:
                                maxVal = temp
                                pos = j
                labelMap[pos] = actualLabels[startPoint]
        return labelMap

def calConfusionMatrix(actualLabels, predictedLabels, k):
        labelMap = calMapping(actualLabels, predictedLabels, k)
        confusionMatrix = [[0 for x in range(k)] for x in range(k)]
        for i in range(0, len(actualLabels)):
                try:
                        confusionMatrix[int(actualLabels[i])][int(labelMap[predictedLabels[i]])] =
confusionMatrix[int(actualLabels[i])][int(labelMap[predictedLabels[i]])] + 1
                except KeyError, e:
                        continue
        return confusionMatrix

def printConfusionMatrix(confusionMatrix):
        print "Confusion Matrix:"
        for i in range(0, len(confusionMatrix)):
                print confusionMatrix[i]

def printResults(correlation, actualLabels, predictedLabels, centroids, data, k):
        print "Actual Labels:", actualLabels
        print "Predicted Labels:", predictedLabels
        print "External Measure(correlation):", correlation
        purity = calPurity(actualLabels, predictedLabels, k)
        print "External Measure(Purity):", purity
        rootMeanSquaredError = calRootMeanSquaredError(centroids, data, predictedLabels)
        print "Internal Measure(Root Mean Squared Error):", rootMeanSquaredError
        dunnIndex = calDunnIndex(data, centroids, predictedLabels, k)
        print "Internal Measure(Dunn Index):", dunnIndex
        confusionMatrix = calConfusionMatrix(actualLabels, predictedLabels, k)
        printConfusionMatrix(confusionMatrix)
        plotResults(actualLabels, predictedLabels, data, k)

def main(arg):
        filePath = str(arg[0])
```

```
        fileName = str(arg[1])
        data, k, actualLabels = loadData(filePath, fileName)
        predictedLabels, centroids = kmeans(data, k)
        correlation = calculateCorrelation(actualLabels, predictedLabels)
        printResults(correlation, actualLabels, predictedLabels, centroids, data, k)

main(sys.argv[1:])
```

2. Hierarichal

```
import csv
import sys
import math
import random
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.colors

def populateClassMap(allLabels):
        classMap = {}
        j = 0
        for i in allLabels:
                classMap[i] = j
                j = j + 1
        return classMap

'''
Loading data from the input file
into a list named dataset. The
function then returns the list.
'''
def loadData(filePath, fileName):
        fullFilePath = filePath + "/" + fileName
        lines = csv.reader(open(fullFilePath, "rb"))
        dataset = list(lines)
        allLabels = []
        for i in range(0, len(dataset)):
                allLabels.append(dataset[i][len(dataset[0]) - 1])
        allLabels = set(allLabels)
        k = len(allLabels)
        classMap = populateClassMap(allLabels)
        actualLabels = []
        for i in range(0, len(dataset)):
                actualLabels.append(classMap[dataset[i][len(dataset[0]) - 1]])
        i = 0
        for x in dataset:
                del x[len(dataset[i]) - 1]
                i = i + 1
        for i in range(len(dataset)):
                dataset[i] = [float(x) for x in dataset[i]]
        return dataset, k, actualLabels
def EuclideanDistance(instance1, instance2, length):
        distance = 0
        for x in range(length):
                distance += pow((instance1[x] - instance2[x]), 2)
        return math.sqrt(distance)

def minDistanceClusters(clusters,centres):
        minv=sys.float_info.max
        for i in range(0,len(clusters)):
                for j in range(0,len(clusters)):
                        if i!=j:
```

```
                                        try:
                                                dist=np.subtract(centres[i],centres[j])
                                                norm=np.linalg.norm(dist)
                                                if norm<minv:
                                                        minv=norm
                                                        c1=i
                                                        c2=j
                                        except ValueError:
                                                print "Error"
                                                print centres[i]
                                                print centres[j]
        #print minv,c1,c2
        return c1,c2

def calculateMean(clusters):
        centres=[]
        for i in range(0,len(clusters)):
                centres.append([])
                centres[i] = list(np.mean(clusters[i],axis=0))
        return centres

def Hierarichal(data, actualLabels, k):
        clusters=[]
        for line in data:
                clusters.append([line])
        print len(clusters)
        centres=calculateMean(clusters)
        while len(clusters) != k:
                c1,c2 = minDistanceClusters(clusters,centres)
                clusters[c1] = clusters[c1]+clusters[c2]
                clusters.remove(clusters[c2])
                centres = calculateMean(clusters)
        return clusters, centres

def calPosOfTheDataPointInData(dataPoint, data):
        pos = 0
        for i in range(0, len(data)):
                if dataPoint == data[i]:
                        pos = i
                        break
        return pos

def calPredictedLabels(clusters, data, k):
        predictedLabels = [0 for x in range(len(data))]
        for i in range(0, k):
                for j in range(0, len(clusters[i])):
                        dataPoint = clusters[i][j]
                        pos = calPosOfTheDataPointInData(dataPoint, data)
                        predictedLabels[pos] = i
        return predictedLabels

def calculateCorrelation(actualLabels, predictedLabels):
        array1 = []
        array2 = []
        for i in range(0, len(actualLabels)):
                for j in range(0, len(actualLabels)):
                        if actualLabels[i] == actualLabels[j]:
                                array1.append(1)
                        else:
                                array1.append(0)
        for i in range(0, len(predictedLabels)):
                for j in range(0, len(predictedLabels)):
                        if predictedLabels[i] == predictedLabels[j]:
```

```
                array2.append(1)
            else:
                array2.append(0)
    correlationMatrix = np.corrcoef(array1, array2)
    return correlationMatrix[0][1]

def generateRandomRGB():
    hexDigits = ['0','1','2','3','4','5','6','7','8','9','a','b','c','d','e','f']
    r1 = random.choice(hexDigits)
    r2 = random.choice(hexDigits)
    g1 = random.choice(hexDigits)
    g2 = random.choice(hexDigits)
    b1 = random.choice(hexDigits)
    b2 = random.choice(hexDigits)
    randomColour = "#" + str(r1) + str(r2) + str(g1) + str(g2) + str(b1) + str(b2)
    return randomColour

def plotResults(actualLabels, predictedLabels, data, k):
    # Plotting actual labels
    colours = []
    for i in range(0, k):
        X = []
        Y = []
        for j in range(0, len(data)):
            if actualLabels[j] == i:
                X.append(data[j][0])
                Y.append(data[j][1])
        randomColour = generateRandomRGB()
        colours.append(randomColour)
        plt.subplot(2,1,1)
        plt.plot(X, Y, randomColour, linestyle=':')
        plt.ylabel('Actual', fontsize=20)
    # Plotting predicted labels
    for i in range(0, k):
        X = []
        Y = []
        for j in range(0, len(data)):
            if predictedLabels[j] == i:
                X.append(data[j][0])
                Y.append(data[j][1])
        plt.subplot(2,1,2)
        plt.plot(X, Y, colours[i], linestyle=':')
        plt.ylabel('Predicted', fontsize=20)
    plt.show()

def calRootMeanSquaredError(centroids, data, predictedLabels):
    rootMeanSquaredError = 0.0
    for i in range(0, len(data)):
        rootMeanSquaredError = rootMeanSquaredError +
EuclideanDistance(centroids[predictedLabels[i]], data[i], len(data[0]))
    return rootMeanSquaredError

def calDunnIndex(data, centroids, predictedLabels, k):
    maxDistancesInCluster = []
    for i in range(0, k):
        tempData = []
        for j in range(0, len(data)):
            if predictedLabels[j] == i:
                tempData.append(data[j])
        maxDistance = -sys.maxint
        for l in range(0, len(tempData)):
            for m in range(0, len(tempData)):
                temp = EuclideanDistance(tempData[l], tempData[m], len(tempData[0]))
```

```
                                        if temp > maxDistance:
                                                maxDistance = temp
                        maxDistancesInCluster.append(maxDistance)
        denominator = max(maxDistancesInCluster)
        minDistance = sys.maxint
        for i in range(0, len(centroids)):
                for j in range(0, len(centroids)):
                        if i != j:
                                temp = EuclideanDistance(centroids[i], centroids[j], len(centroids[0]))
                                if temp < minDistance:
                                        minDistance = temp
        numerator = minDistance
        dunnIndex = float(numerator / denominator)
        return dunnIndex

def calPurity(actualLabels, predictedLabels, k):
        purity = 0.0
        for i in range(0, k):
                noOfLabelsOfEachClassInTheCluster = [0 for x in range(0, k)]
                if i == 0:
                        startPoint = 0
                        for j in range(1, len(actualLabels)):
                                if actualLabels[j] != actualLabels[j - 1]:
                                        endPoint = j - 1
                                        break
                        for j in range(startPoint, endPoint + 1):
                                noOfLabelsOfEachClassInTheCluster[predictedLabels[j]]                                =
noOfLabelsOfEachClassInTheCluster[predictedLabels[j]] + 1
                        purity = purity + max(noOfLabelsOfEachClassInTheCluster)
                elif i == k - 1:
                        startPoint = endPoint + 1
                        endPoint = len(actualLabels) - 1
                        for j in range(startPoint, endPoint + 1):
                                noOfLabelsOfEachClassInTheCluster[predictedLabels[j]]                                =
noOfLabelsOfEachClassInTheCluster[predictedLabels[j]] + 1
                        purity = purity + max(noOfLabelsOfEachClassInTheCluster)
                else:
                        startPoint = endPoint + 1
                        for j in range(startPoint + 1, len(actualLabels)):
                                if actualLabels[j] != actualLabels[j - 1]:
                                        endPoint = j - 1
                                        break
                        for j in range(startPoint, endPoint + 1):
                                noOfLabelsOfEachClassInTheCluster[predictedLabels[j]]                                =
noOfLabelsOfEachClassInTheCluster[predictedLabels[j]] + 1
                        purity = purity + max(noOfLabelsOfEachClassInTheCluster)
        purity = float(purity / len(actualLabels))
        return purity

def calMapping(actualLabels, predictedLabels, k):
        labelMap = {}
        for i in range(0, k):
                noOfLabelsOfEachClassInTheCluster = [0 for x in range(0, k)]
                if i == 0:
                        startPoint = 0
                        for j in range(1, len(actualLabels)):
                                if actualLabels[j] != actualLabels[j - 1]:
                                        endPoint = j - 1
                                        break
                        for j in range(startPoint, endPoint + 1):
                                noOfLabelsOfEachClassInTheCluster[predictedLabels[j]]                                =
noOfLabelsOfEachClassInTheCluster[predictedLabels[j]] + 1
                elif i == k - 1:
```

```
                        startPoint = endPoint + 1
                        endPoint = len(actualLabels) - 1
                        for j in range(startPoint, endPoint + 1):
                                noOfLabelsOfEachClassInTheCluster[predictedLabels[j]] =
noOfLabelsOfEachClassInTheCluster[predictedLabels[j]] + 1
                else:
                        startPoint = endPoint + 1
                        for j in range(startPoint + 1, len(actualLabels)):
                                if actualLabels[j] != actualLabels[j - 1]:
                                        endPoint = j - 1
                                        break
                        for j in range(startPoint, endPoint + 1):
                                noOfLabelsOfEachClassInTheCluster[predictedLabels[j]] =
noOfLabelsOfEachClassInTheCluster[predictedLabels[j]] + 1
                maxVal = -sys.maxint
                for j in range(0, len(noOfLabelsOfEachClassInTheCluster)):
                        temp = noOfLabelsOfEachClassInTheCluster[j]
                        if temp > maxVal:
                                maxVal = temp
                                pos = j
                labelMap[pos] = actualLabels[startPoint]
        return labelMap

def calConfusionMatrix(actualLabels, predictedLabels, k):
        labelMap = calMapping(actualLabels, predictedLabels, k)
        confusionMatrix = [[0 for x in range(k)] for x in range(k)]
        for i in range(0, len(actualLabels)):
                try:
                        confusionMatrix[int(actualLabels[i])][int(labelMap[predictedLabels[i]])] =
confusionMatrix[int(actualLabels[i])][int(labelMap[predictedLabels[i]])] + 1
                except KeyError, e:
                        continue
        return confusionMatrix

def printConfusionMatrix(confusionMatrix):
        print "Confusion Matrix:"
        for i in range(0, len(confusionMatrix)):
                print confusionMatrix[i]

def printResults(correlation, actualLabels, predictedLabels, centroids, data, k):
        print "Actual Labels:", actualLabels
        print "Predicted Labels:", predictedLabels
        print "External Measure(correlation):", correlation
        purity = calPurity(actualLabels, predictedLabels, k)
        print "External Measure(Purity):", purity
        rootMeanSquaredError = calRootMeanSquaredError(centroids, data, predictedLabels)
        print "Internal Measure(Root Mean Squared Error):", rootMeanSquaredError
        dunnIndex = calDunnIndex(data, centroids, predictedLabels, k)
        print "Internal Measure(Dunn Index):", dunnIndex
        confusionMatrix = calConfusionMatrix(actualLabels, predictedLabels, k)
        printConfusionMatrix(confusionMatrix)
        plotResults(actualLabels, predictedLabels, data, k)

def main(arg):
        filePath = str(arg[0])
        fileName = str(arg[1])
        data, k, actualLabels = loadData(filePath, fileName)
        clusters, centroids = Hierarichal(data, actualLabels, k)
        predictedLabels = calPredictedLabels(clusters, data, k)
        correlation = calculateCorrelation(actualLabels, predictedLabels)
        printResults(correlation, actualLabels, predictedLabels, centroids, data, k)

main(sys.argv[1:])
```

Results & Print Screens

1. kmeans(with iris-dataset)

converge in 5 iterations
Actual Labels: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 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, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Predicted Labels: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 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, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 2, 2, 2, 2, 0, 2, 2, 2, 2, 2, 2, 0, 0, 2, 2, 2, 2, 0, 2, 0, 2, 0, 2, 2, 0, 0, 2, 2, 2, 2, 2, 0, 2, 2, 2, 2, 0, 2, 2, 2, 0, 2, 2, 2, 0, 2, 2, 0]
External Measure(correlation): 0.734054668986
External Measure(Purity): 0.893333333333
Internal Measure(Root Mean Squared Error): 97.3259242343
Internal Measure(Dunn Index): 0.671169841783
Confusion Matrix:
[36, 0, 14]
[0, 50, 0]
[2, 0, 48]

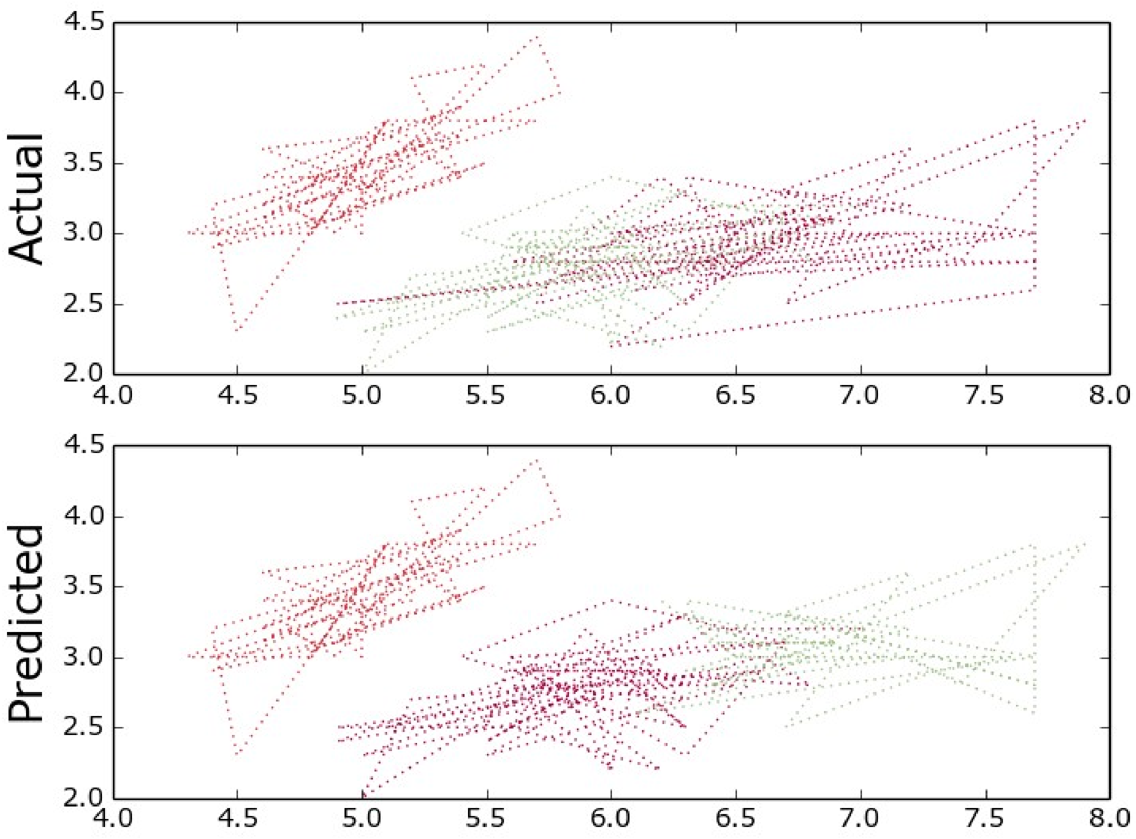

Figure 1 : Clusters so formed

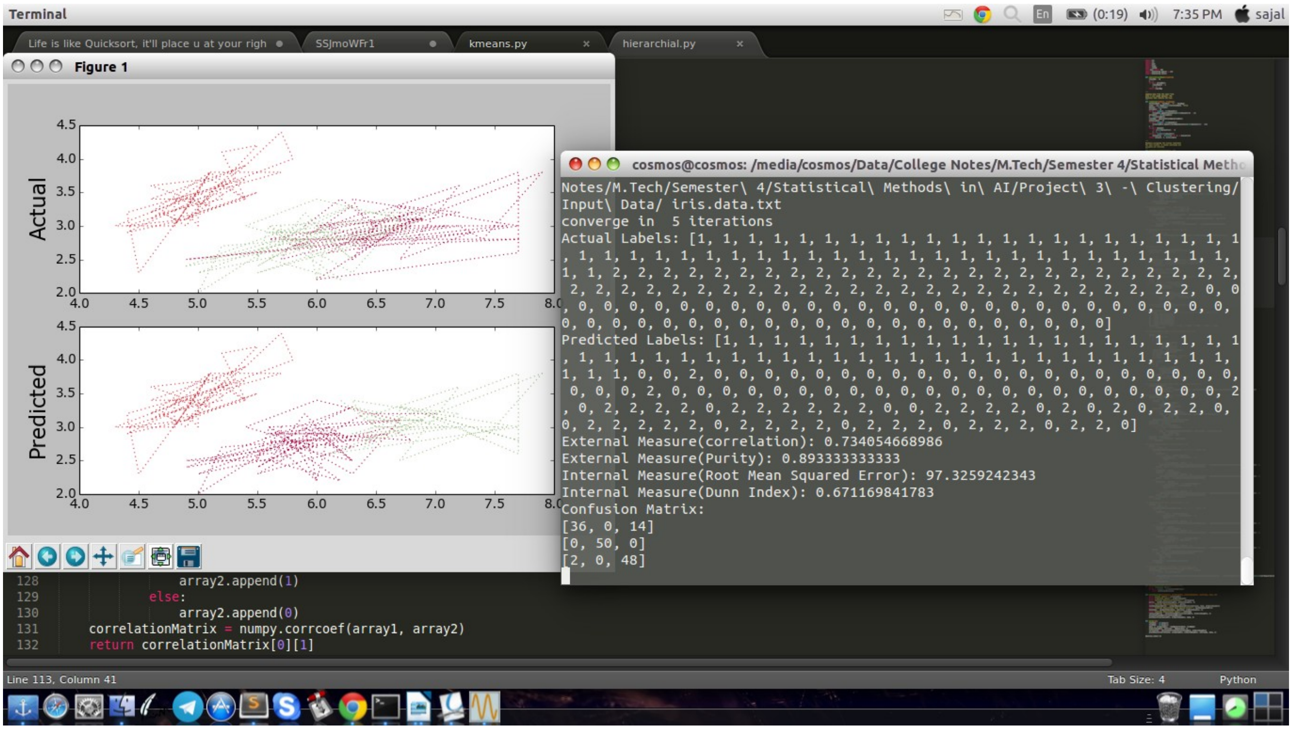

Figure 2 : Print Screen of the result so obtained

2. kmeans(with wine dataset)

converge in 9 iterations
Actual Labels: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 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, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
2, 2, 2, 2, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1]
Predicted Labels: [2, 2, 2, 2, 0, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 0, 0, 0, 2, 2, 0, 0, 2, 2, 0, 2, 2, 2, 2, 2, 2, 0,
0, 2, 2, 0, 0, 2, 2, 0, 0, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 0, 1, 0, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 2, 0, 1, 1, 1, 0, 1,
1, 0, 0, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 0, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1,
1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0,
0, 0, 1, 1, 0, 0, 0, 0, 0, 1]
External Measure(correlation): 0.377965090383
External Measure(Purity): 0.702247191011
Internal Measure(Root Mean Squared Error): 16555.679416
Internal Measure(Dunn Index): 0.380730883516
Confusion Matrix:
[46, 13, 0]
[0, 29, 19]
[1, 20, 50]

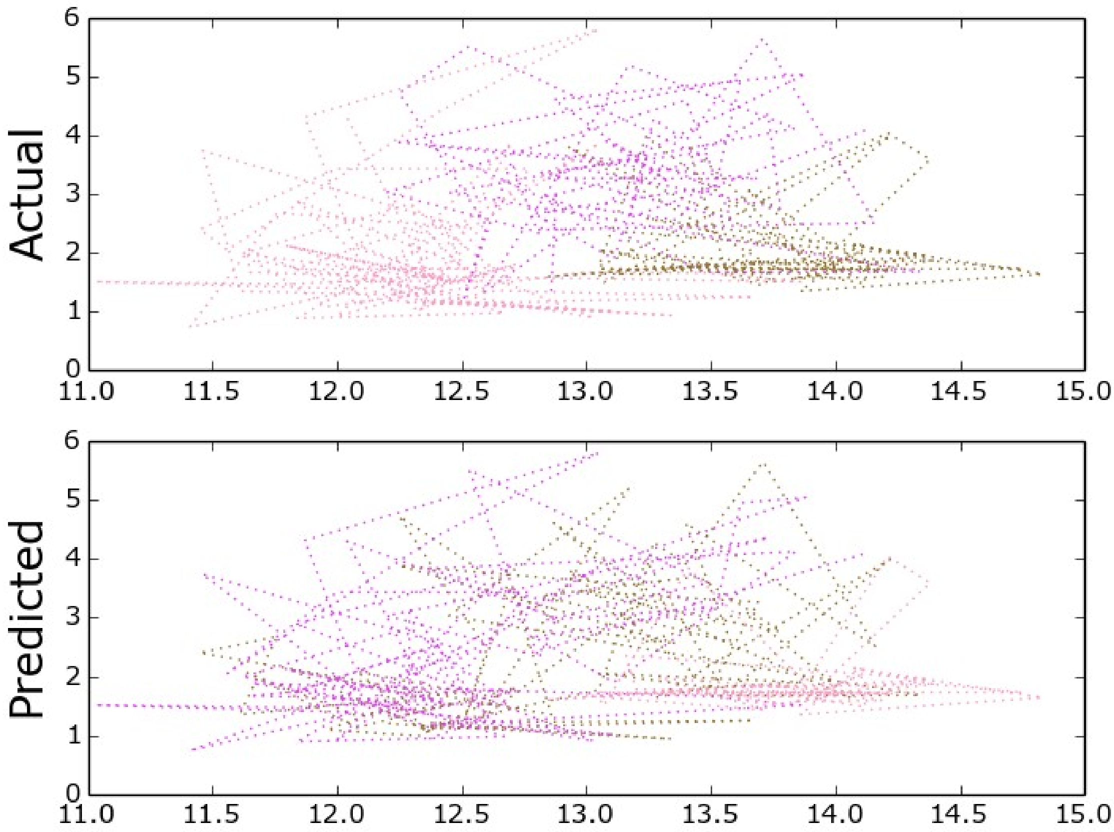

Figure 3 : Clusters so formed

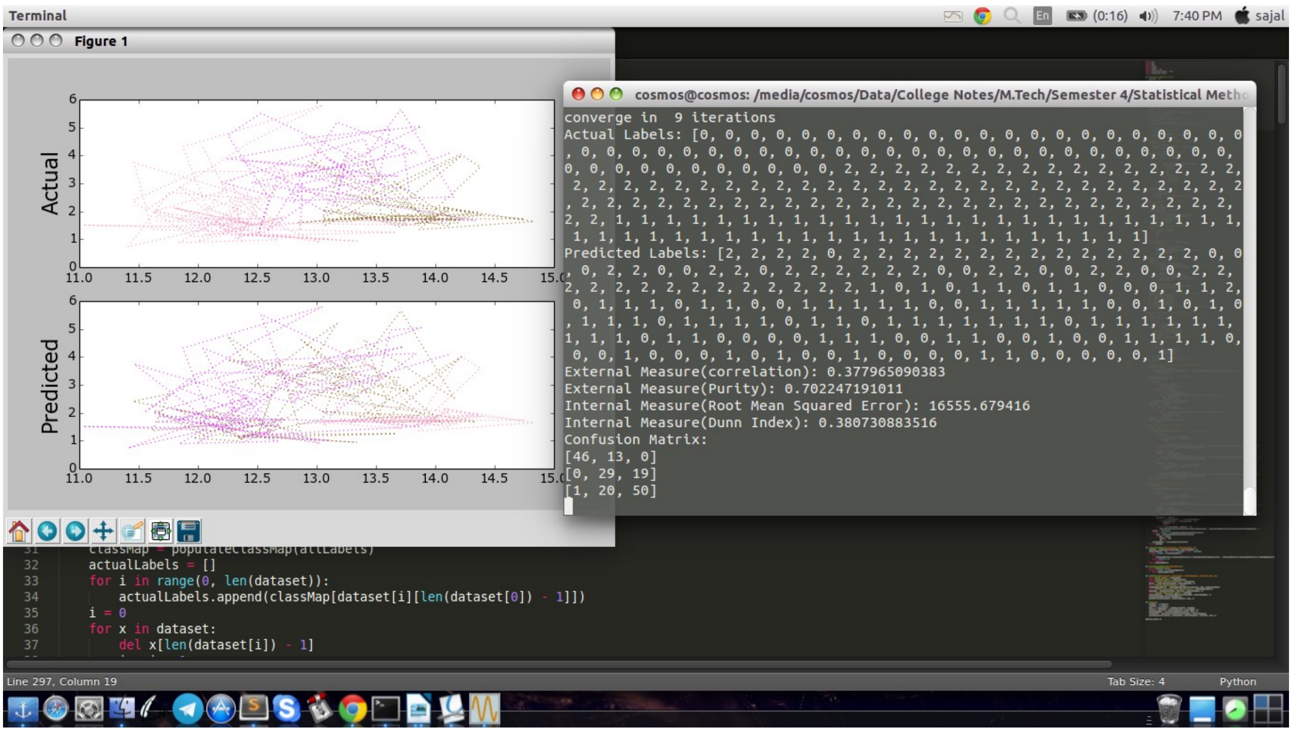

Figure 4 : Print Screen of the result so obtained

3. Hierarchical Clustering(with iris dataset)

Actual Labels: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 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, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Predicted Labels: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 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, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1, 1, 2, 2, 2, 2, 1, 2, 1, 2, 1, 2, 2, 1, 1, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 1, 2, 2, 2, 0, 2, 2, 2, 1, 2, 2, 1]
External Measure(correlation): 0.760030707632
External Measure(Purity): 0.906666666667
Internal Measure(Root Mean Squared Error): 100.952743177
Internal Measure(Dunn Index): 0.386384458752
Confusion Matrix:
[36, 1, 13]
[0, 50, 0]
[0, 0, 50]

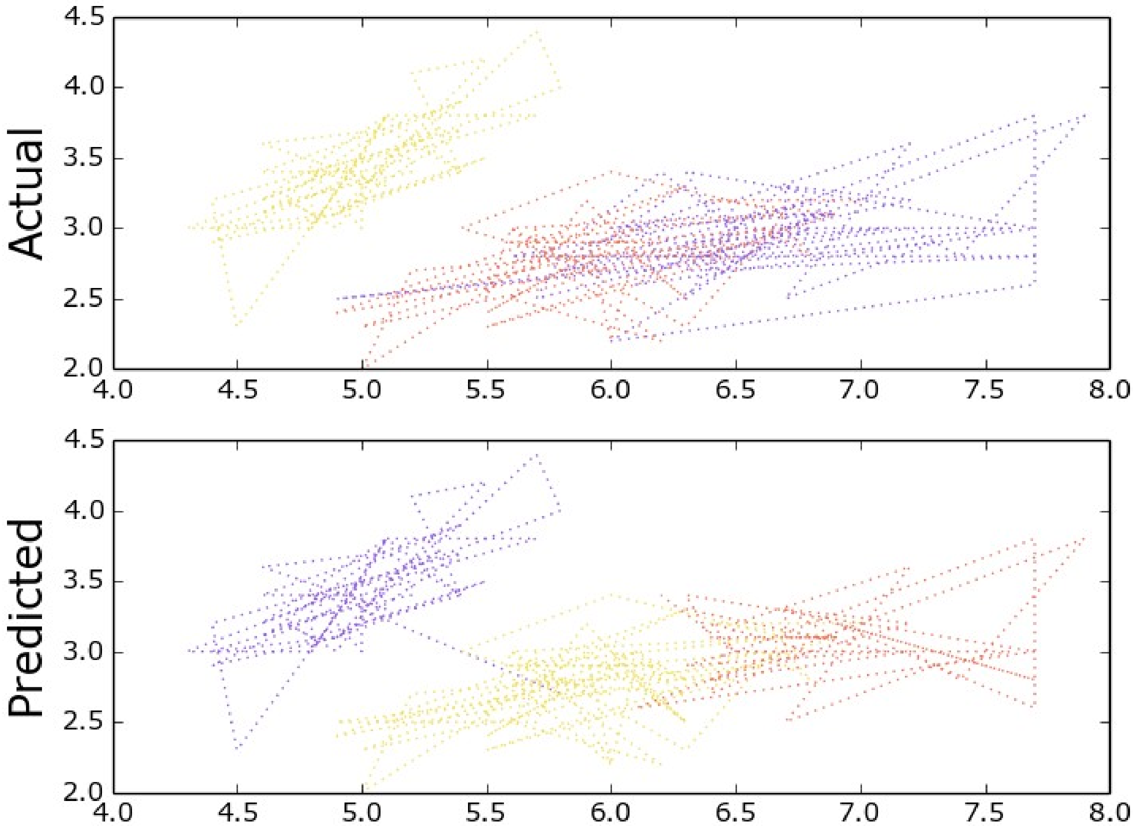

Figure 5 : Clusters so obtained

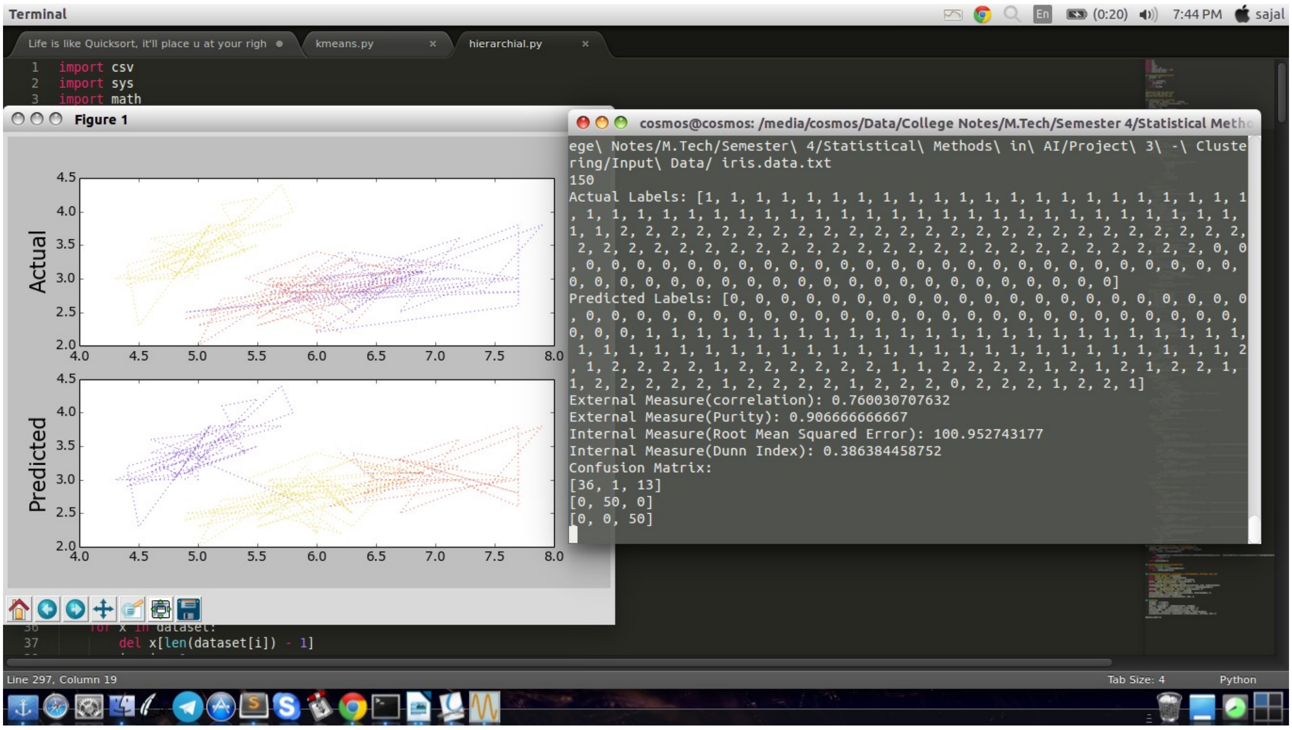

Figure 6 : Print screen of the results so obtained

4. Hierarchical Clustering(with wine data)

Actual Labels: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 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, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
Predicted Labels: [0, 0, 0, 1, 2, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 2, 2, 2, 0, 0, 2, 2, 0, 0, 2, 0, 0, 1, 0, 0, 0, 2, 2, 0, 0, 2, 2, 0, 0, 2, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 0, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 0, 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, 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]
External Measure(correlation): 0.336183607512
External Measure(Purity): 0.88202247191
Internal Measure(Root Mean Squared Error): 22285.5756916
Internal Measure(Dunn Index): 0.60611142776
Confusion Matrix:
[40, 13, 0]
[0, 48, 0]
[2, 69, 0]

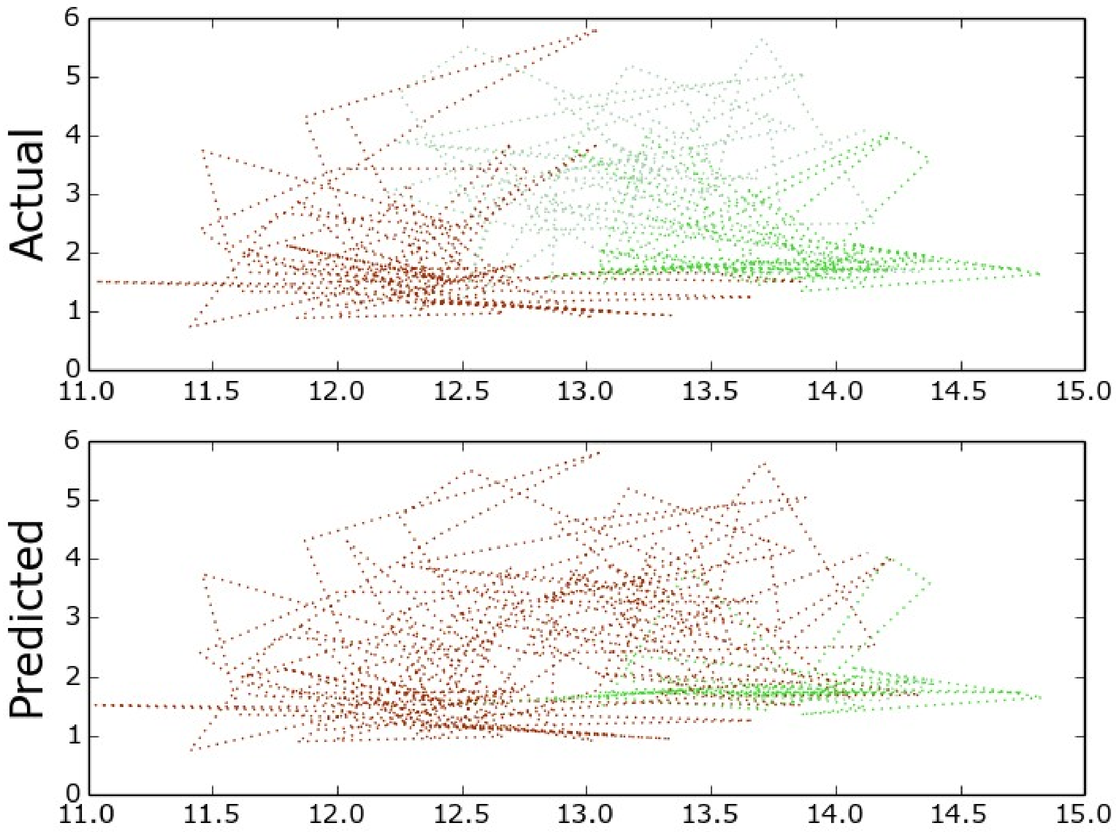

Figure 7 : Clusters so obtained

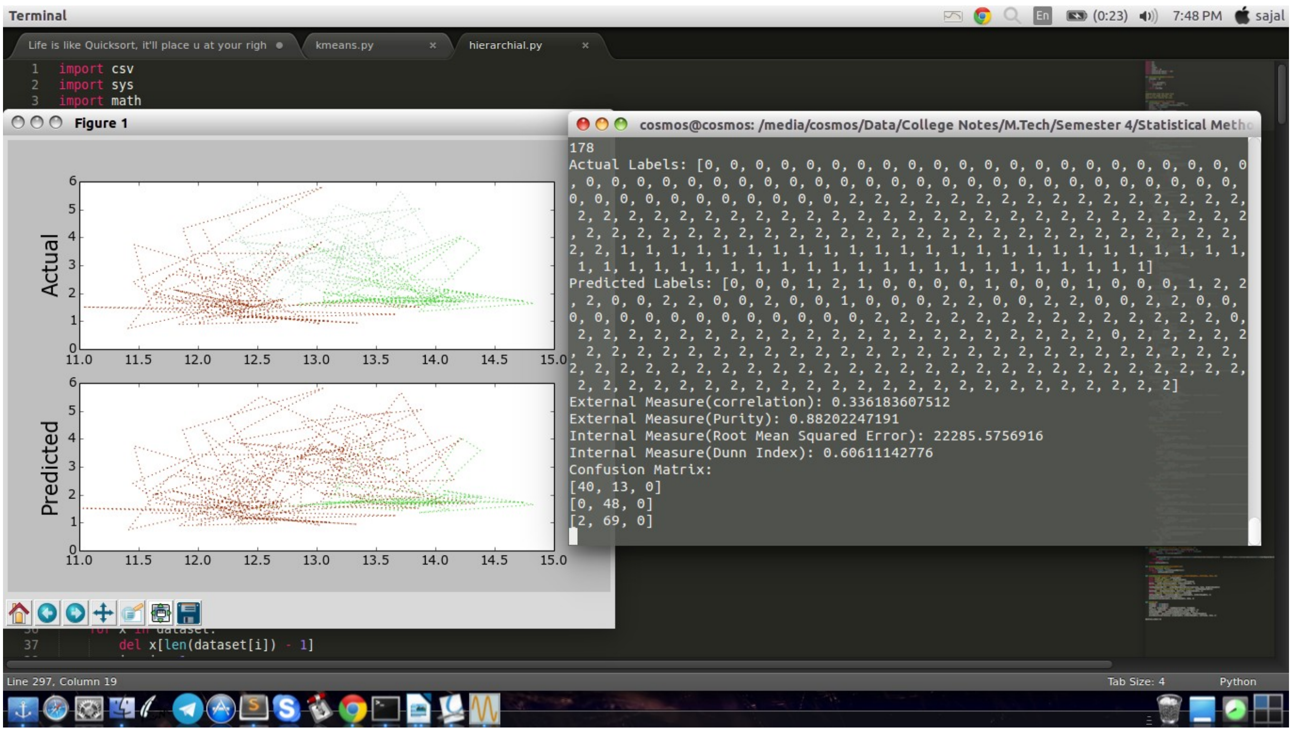

Figure 8 : Print screen of the obtained results

Interesting Observations

The two key features of k-means which make it efficient are often regarded as its biggest drawbacks:

1. Euclidean distance is used as a metric and variance is used as a measure of cluster scatter.
2. The number of clusters k is an input parameter: an inappropriate choice of k may yield poor results. That is why, when performing k-means, it is important to run diagnostic checks for determining the number of clusters in the data set.
3. Convergence to a local minimum may produce counterintuitive ("wrong") results.

A key limitation of k-means is its cluster model. The concept is based on spherical clusters that are separable in a way so that the mean value converges towards the cluster center. The clusters are expected to be of similar size, so that the assignment to the nearest cluster center is the correct assignment. When for example applying k-means with a value of k=3 onto the well-known Iris flower data set, the result often fails to separate the three Iris species contained in the data set. With k=2, the two visible clusters (one containing two species) will be discovered, whereas with k=3 one of the two clusters will be split into two even parts. In fact, k=2 is more appropriate for this data set, despite the data set containing 3 classes. As with any other clustering algorithm, the k-means result relies on the data set to satisfy the assumptions made by the clustering algorithms. It works well on some data sets, while failing on others.