Introduction to Machine Learning Program Assignment #2

Members:
0510002 袁鈺勛
0510020 方鈺豪
0510022 劉孟寰
0510023 李佳任
0086043 陳以嬿

(1)
What environments the members are using?
Ans:
We use python to implement Machine Learning.
We use two packages: Pandas and Matplotlib.
Pandas package is convenient for processing the CSV file.
Matplotlib package is useful for visualizing the outcome.

(2)
K-means code
Ans:
We have two parts to impliment K-means.

```
# (Part1)Compute Clusters
def computeClusters(pair, w, x, y, z, k):
	w_difference = (np.max(w) - np.min(w)) / k
	x_difference = (np.max(x) - np.min(x)) / k
	# w coordinates of random centroids
	w_centroid = [np.random.randint(np.min(w) + w_difference*i, np.min(w) + w_difference*(i + 1)) for i in
range(k)]
	# x coordinates of random centroids
	x_centroid = [np.random.randint(np.min(x) + x_difference*i, np.min(x) + x_difference*(i + 1)) for i in
range(k)]
```

```
        # Check type
        if y is None and z is None: # Two coordinates
                centroid = np.array(list(zip(w_centroid, x_centroid)), dtype = np.float32)
        elif z is None: # Three coordinates
                y_difference = (np.max(y) - np.min(y)) / k
                # y coordinates of random centroids
                y_centroid = [np.random.randint(np.min(y) + y_difference*i, np.min(y) + y_difference*(i + 1)) for i in
range(k)]
                centroid = np.array(list(zip(w_centroid, x_centroid, y_centroid)), dtype = np.float32)
        else: # Four coordinates
                y_difference = (np.max(y) - np.min(y)) / k
                z_difference = (np.max(z) - np.min(z)) / k
                # y coordinates of random centroids
                y_centroid = [np.random.randint(np.min(y) + y_difference*i, np.min(y) + y_difference*(i + 1)) for i in
range(k)]
                # z coordinates of random centroids
                z_centroid = [np.random.randint(np.min(z) + z_difference*i, np.min(z) + z_difference*(i + 1)) for i in
range(k)]
                centroid = np.array(list(zip(w_centroid, x_centroid, y_centroid, z_centroid)), dtype = np.float32)

        clusters = np.zeros(len(pair))
        print(centroid)
        centroid, clusters = getFinalCentroid(centroid, pair, clusters, k)
        print(centroid)

        if y is None and z is None:
                return computeAccuracy(clusters, pitch_type), centroid, clusters
        else:
                return computeAccuracy(clusters, pitch_type)
# (Part2)Get final centroid
def getFinalCentroid(centroid, pair, clusters, k):
        old_centroid = np.zeros(centroid.shape)
        # distance between old centroid and current centroid
        cen_distance = getDistance(centroid, old_centroid, None)

        # Loop will run till the error becomes zero
        while cen_distance != 0:
                # Assigning each value to its closest cluster
                for i in range(len(pair)):
                        distances = getDistance(pair[i], centroid)
                        cluster = np.argmin(distances)
                        clusters[i] = cluster
                # Storing the old centroid values
                old_centroid = deepcopy(centroid)
                # Finding the new centroids by taking the average value
                for i in range(k):
                        points = [pair[j] for j in range(len(pair)) if clusters[j] == i]
                        if len(points) > 0:
                                centroid[i] = np.mean(points, axis = 0)
                cen_distance = getDistance(centroid, old_centroid, None)

        return centroid, clusters
```

In part1(compute clusters), we get K rough centroids by dividing our data into K parts on each axises(x, y, speed, …etc), and the number of dimension of centroids depends on our input data number(from 2~4).

In part2(Get final centroids), we gradually adjust our centroids due to our clustering outcome last time until the centroids are convergent. After this part, we also finish our final clustering, so next step is to check the accuracy.

(3)
Cost function and accuracy
Ans:

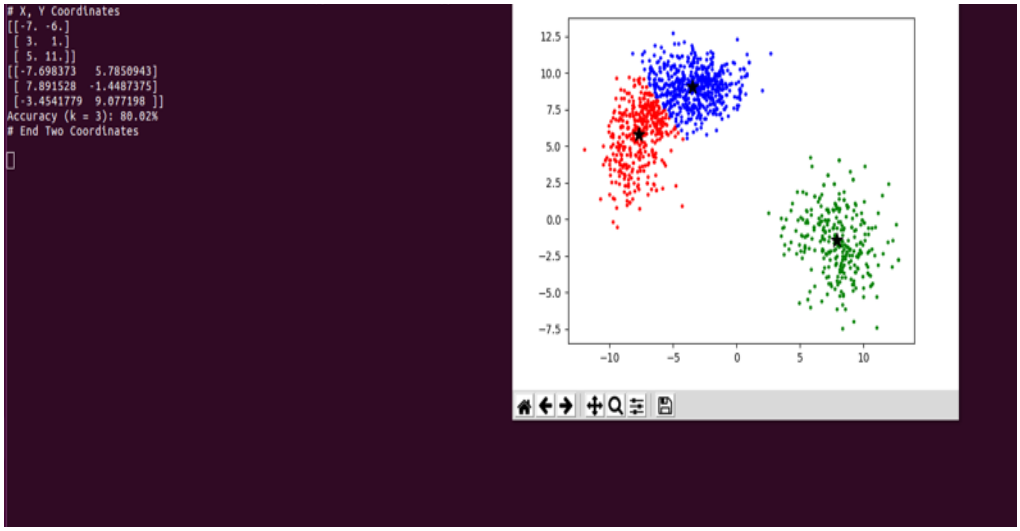
```
# Compute accuracy
def computeAccuracy(clusters, pitch_type):
	CH = [int(clusters[i]) for i in range(len(pitch_type)) if pitch_type[i] == 'CH']
	CU = [int(clusters[i]) for i in range(len(pitch_type)) if pitch_type[i] == 'CU']
	FF = [int(clusters[i]) for i in range(len(pitch_type)) if pitch_type[i] == 'FF']

	count = CH.count(mostCommon(list(CH)))
	count += CU.count(mostCommon(list(CU)))
	count += FF.count(mostCommon(list(FF)))

	return count / len(pitch_type)
```

After clustering, we generate three lists, each list contains the cluster index of the points which are actually belongs to one of {CH, CU, FF}, so each list should have only one number(0, 1, or 2) ideally, but that's not always possible, so we find the most common integer(0, 1, 2) in the list and check the number of the most common integer and sum up of it in each three lists to check the accuracy.

(4)
The result of K-Means clustering
Ans:
We can see the picture that the accuracy is about 80% when we use x, y as input.
(the matrix is the starting centroid and the final centroid).
The clustering output and final centroids are shown in the picture.



(5)
Use another two or more attributes to partition and the reason of k = 3
Ans:
We use another three sets of input([x ,y, speed], [AX, AY, AZ], [Pfx_x, Pfx_z, spin]).
We can see that the accuracy is 100% when we use(x, y, speed) as input, which is pretty ideal, and the others is about 80%, close to the original one.

The reason that k = 3 is the best is that if K is less than 3, we can just get 1 or 2 clusters, but we have 3 kinds of pitch_type, so we can't even get all kinds of them, that doesn't make sense.
On the other hand, when K is larger than 3(take k=4 as example), there will be about 1/4 points be categorized in an unknown group because we have 3 kinds of pitch_type exactly. Also, the accuracy of other three group will be

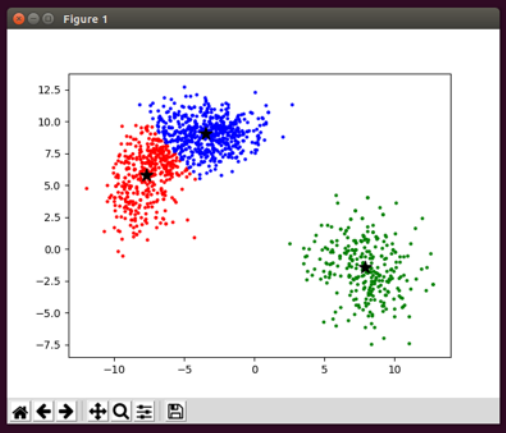
decreased because of excess centroid.
By above reasons, choosing K=3 is the best method to do clustering.

```
# X, Y, Speed Coordinates
[[-11.  -4.  83.]
 [ -1.   1.  89.]
 [  7.   6.  98.]]
[[-8.337901   3.9144444 88.6787  ]
 [ 7.891528  -1.4487375 80.91249 ]
 [-4.629441   8.425117  97.52534 ]]
Accuracy (k = 3): 100.00%
# End Three Coordinates

# AX, AY, AZ Coordinates
[[-21.  23. -40.]
 [ -8.  29. -25.]
 [  3.  40.  -9.]]
[[-14.705055   30.53659   -20.597816 ]
 [ 10.692957   23.240072  -33.63146  ]
 [ -7.3752384  34.08312   -13.319185 ]]
Accuracy (k = 3): 80.62%
# End XYZ Coordinates

# Pfx_x, Pfx_z, Spin, Speed Coordinates
[[ -6.  -4.  80.  78.]
 [ -3.   3. 147.  86.]
 [ 10.   9. 231.  97.]]
[[  7.6994514  -1.0776027  80.756      80.91249  ]
 [ -3.601397    9.333304  200.48952    97.5344   ]
 [ -7.871304    6.0451126 233.56487    94.10012  ]]
Accuracy (k = 3): 80.32%
# End Four Coordinates

# X, Y Coordinates
[[-6. -1.]
 [ 2.  0.]
 [ 4. 11.]]
[[-7.698373   5.7850943]
 [ 7.891528  -1.4487375]
 [-3.4541779  9.077198 ]]
Accuracy (k = 3): 80.02%
# End Two Coordinates
```



(6)
K-d tree code
Ans:

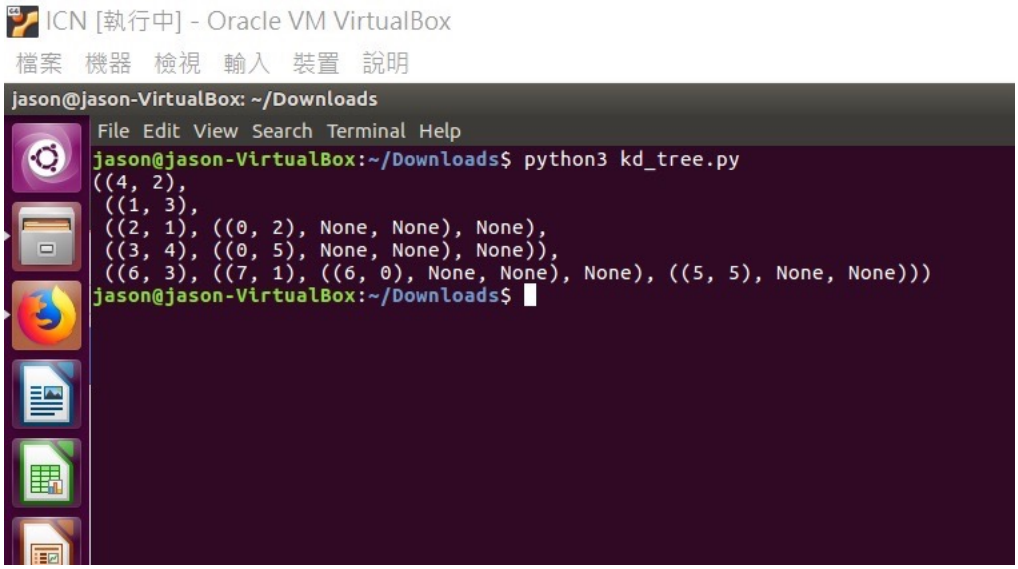
```python
from collections import namedtuple
from operator import itemgetter
from pprint import pformat
import numpy as np
from matplotlib import pyplot as plt
import pandas as pd
def computeVariance(arrayList):
    for ele in arrayList:
        ele = float(ele)
    LEN = float(len(arrayList))
    array = np.array(arrayList)
    sum1 = array.sum()
    array2 = array * array
    sum2 = array2.sum()
    mean = sum1/LEN
    variance = sum2 / LEN - mean**2
    return variance
#node class
class Node(namedtuple('Node', 'location left_child right_child')):
    def __repr__(self):
        return pformat(tuple(self))
def kdtree(x,y,a,b,ax,point_list,depth=0):
    try:
        k=len(point_list[0]) #k dimension
    except IndexError as e: # if not point_list:
        return None
    axis = depth%k# change axis based on depth
    #change axis based on maximum variance
    #max_var=0
    #for i in range(k):
    #    ll=[]
    #    for t in point_list:
    #        ll.append(t[i-1])
    #    var=computeVariance(ll)
    #    if var>max_var:
    #        max_var=var
    #        axis=i-1
    # sort point list according to axis value
    point_list.sort(key=itemgetter(axis))
    # choose median to split to left and right
    median=len(point_list)//2
    #plot line
    if axis==1:
        ax.plot([x,a],[point_list[median][1],point_list[median][1]],color="b")
    else:
        ax.plot([point_list[median][0],point_list[median][0]], [y,b], color="r")
    #find lower left corner and upper right corner of subtree
```

```
    #find lower left corner and upper right corner of subtree
    if axis==0:
        left_x=x
        left_y=y
        right_x=point_list[median][0]
        right_y=y
        left_a=point_list[median][0]
        left_b=b
        right_a=a
        right_b=b
    else:
        left_x=x
        left_y=y
        right_x=x
        right_y=point_list[median][1]
        left_a =a
        left_b=point_list[median][1]
        right_a=a
        right_b=b
    # create node and construct subtrees
    return Node(
        location=point_list[median],
        left_child=kdtree(left_x,left_y,left_a,left_b,ax,point_list[:median],depth+1),
        right_child=kdtree(right_x,right_y,right_a,right_b,ax,point_list[median+1:],depth+1)
    )
def main():
    input_data = pd.read_csv("points.txt", names=['x', 'y'], sep=' ')
    point_list = list(zip(input_data['x'], input_data['y']))
    plt.figure('2d tree', figsize=(max(input_data['x'])+5, max(input_data['y'])+5))
    plt.xlim([0,max(input_data['x'])+5])
    plt.ylim([0, max(input_data['y'])+5])
    ax = plt.gca()
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    x_list = [p[0] for p in point_list]
    y_list = [p[1] for p in point_list]
    ax.scatter(x_list, y_list, c='g', s=40, alpha=0.5)
    tree = kdtree(0,0,max(input_data['x'])+5,max(input_data['y'])+5,ax,point_list)
    print(tree)
    plt.show()

if __name__ == '__main__':
    main()
```

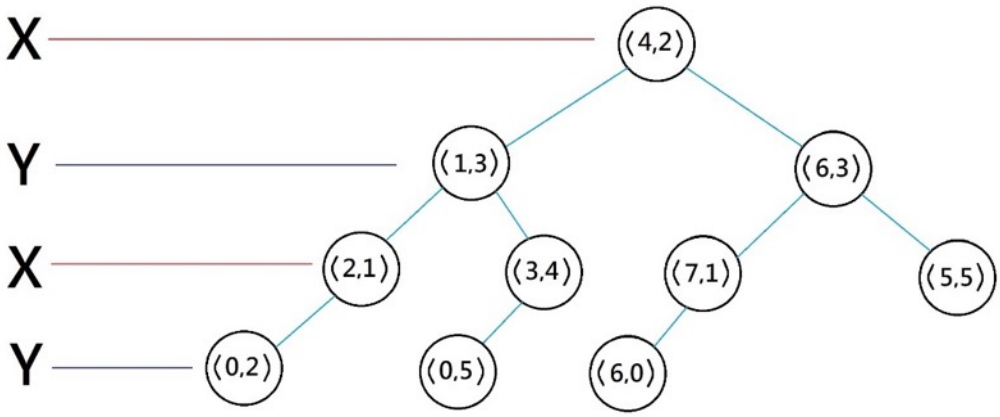
More details please refer to the comment in the code.
For out kdtree function, there are 7 input arguments.First 4 arguments mean that our 2d tree is constructed in the area of rectangle decided by lower left corner (x,y) and upper right corner (a,b),ax is the plot we want to draw on, point_list is the input list and depth is the layer number of the tree starting from 0. To decide the axis we want to use in this layer, we simply start from x-axis and switch to another each step, in the comment there is another way by computing maximum variance. Then we find the median of the data due to axis, draw the line in the rectangle area and split it into left and right. After we find the area of the rectangle for the left and right child tree, we call the function again to obtain the left and right child, by recursively doing this until there is no data left we get the final tree.

(7)
The output of the code:



This output means a kd tree, and the image of this tree, which is made by this output format is presented below.

```
( (4,2) ,                  //top layer node
 (
     (1,3) ,               //second layer node(left)
   ( (2,1) ,               //third layer node(first from left)
     ( (0,2) , None , None ) ,  //bottom layer node(first from left)
     None ) ,
   ( (3,4) ,               //third layer node(second from left)
     ( (0,5) , None , None ) ,  //bottom layer node(second from left)
     None )
  ) ,
 (
     (6,3) ,               //second layer node(right)
   ( (7,1) ,               //third layer node(third from left)
     ( (6,0) , None , None ) ,  //bottom layer node(third from left)
     None) ,
  ( (5,5) ,                //third layer node(fourth from left)
     None , None )
  )
 )
```



K-d tree decomposition for the point set(made from the k-d tree):