Introduction to Machine Learning Homework 2

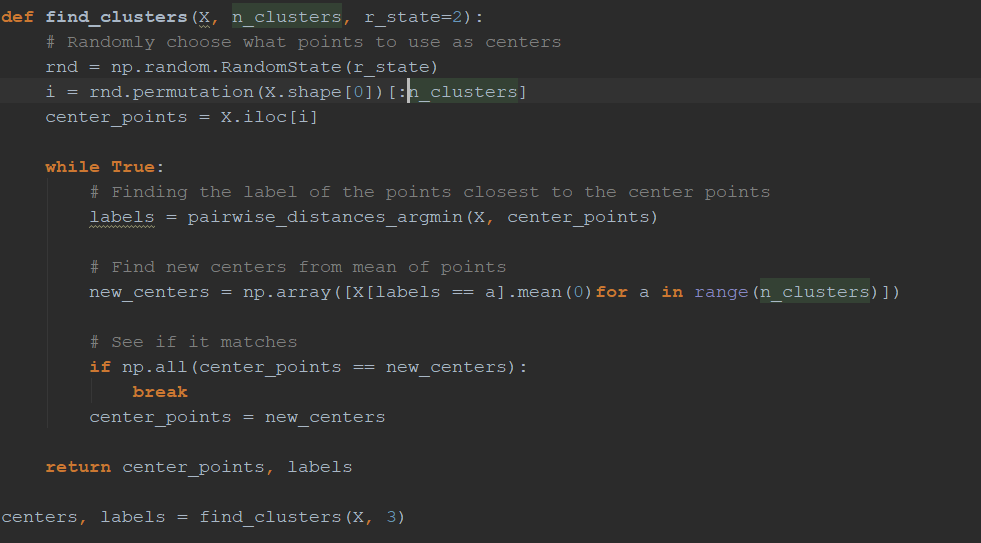
# Environment used

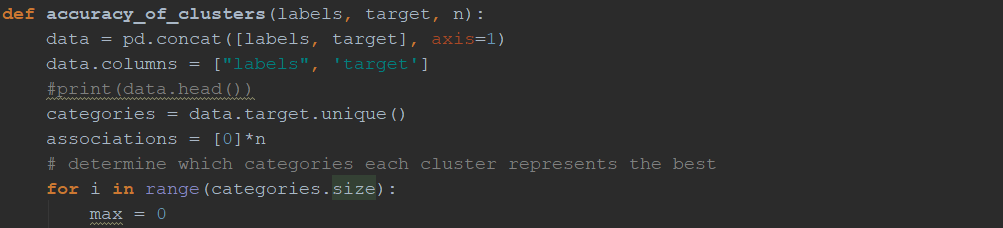
For this homework we used pyCharm IDE and GitHub as a remote repository.

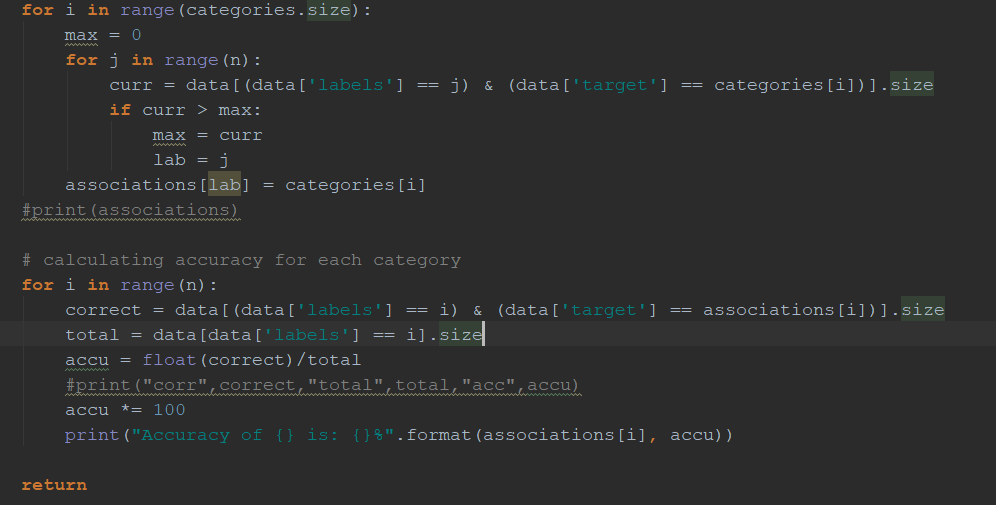
# K-means

K-means Code

The function find\_clusters finds a center point for the number of clusters specified and returns it as center\_points. It also returns a numpy array with the estimated labels:



The function for measuring the clusters accuracy:



K-means result

The results we got from using our K-means clustering function with k=3 and accuracy function was:

Total accuracy: 80.01514004542014%  
Accuracy of CH: 37.971698113207545%  
Accuracy of CU: 100.0%  
Accuracy of FF: 99.83221476510067%

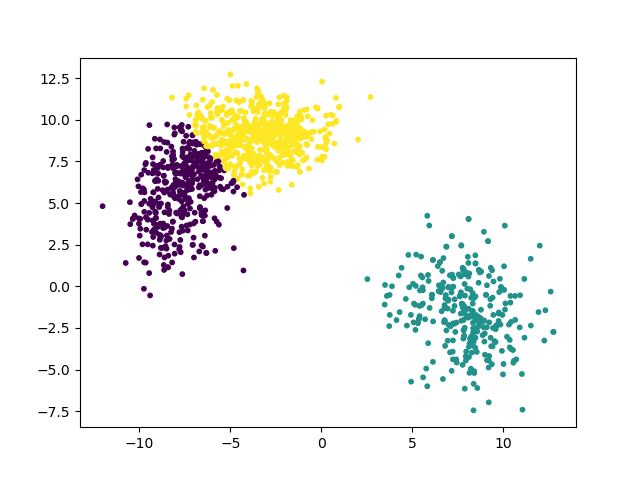
With a cluster that looks like this:

Figure 1, purple is FF, yellow is CH, turquoise is CU

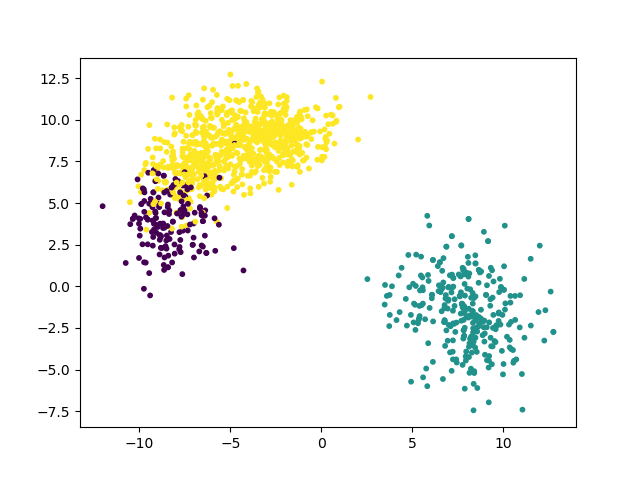
If all the points had its correct label it would look like this:

Figure 2, purple is FF, yellow is CH, turquoise is CU

Results with other parameters

For our second experiment we used ‘speed’ and ‘sz\_bot’ to partition the data with good results:

Total accuracy: 99.47009841029522%  
Accuracy of FF: 100.0%  
Accuracy of CH: 95.85798816568047%  
Accuracy of CU: 100.0%

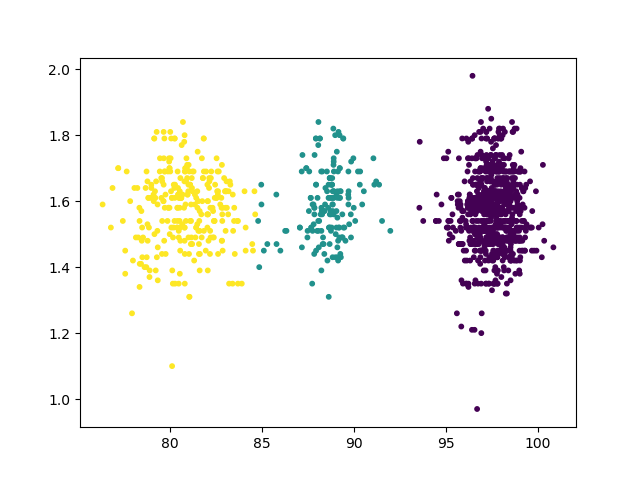
With a cluster that looks like this:

Figure 3, yellow is CU, turquoise is CH, purple is FF

Why does k=3 work best?

For this case, we know that there are three different labels within the target. Therefore, we can hope for that there will be three well defined clusters within the data. With the features ‘x’ and ‘y’ this is kind of true, as we can see in fig. 2. It’s very clear in fig. 3 with the features ‘speed’ and ‘sz\_bot’. If we would’ve used 4 or more different clusters, we also would’ve expected 4 or more different labels within the target, which now is not the case.

# Kd-tree

Kd-tree Code

To construct Kd-tree we used Node class:

class Node:  
 def \_\_init\_\_(self, value, left\_child, right\_child):  
 self.value = value  
 self.left\_child = left\_child  
 self.right\_child = right\_child

Construction algorithm:

def kdtree(points, axis=0):  
  
 if len(points) == 0: #Recursion ending condition  
 return None  
  
 points.sort(key=itemgetter(axis))  
 median = len(points) // 2  
  
 return Node(points[median], kdtree(points[:median], 1 - axis), kdtree(points[median + 1:], 1 - axis))

Axis selection is might be done before construction:

x\_variance = np.var([point[0] for point in points])  
y\_variance = np.var([point[1] for point in points])  
  
axis = 0 if x\_variance >= y\_variance else 1

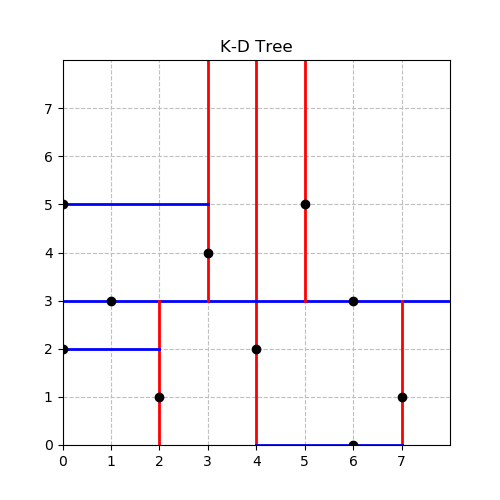
Result of Kd-tree

Figure 4, Result of Kd-tree on points.txt data

Screenshots

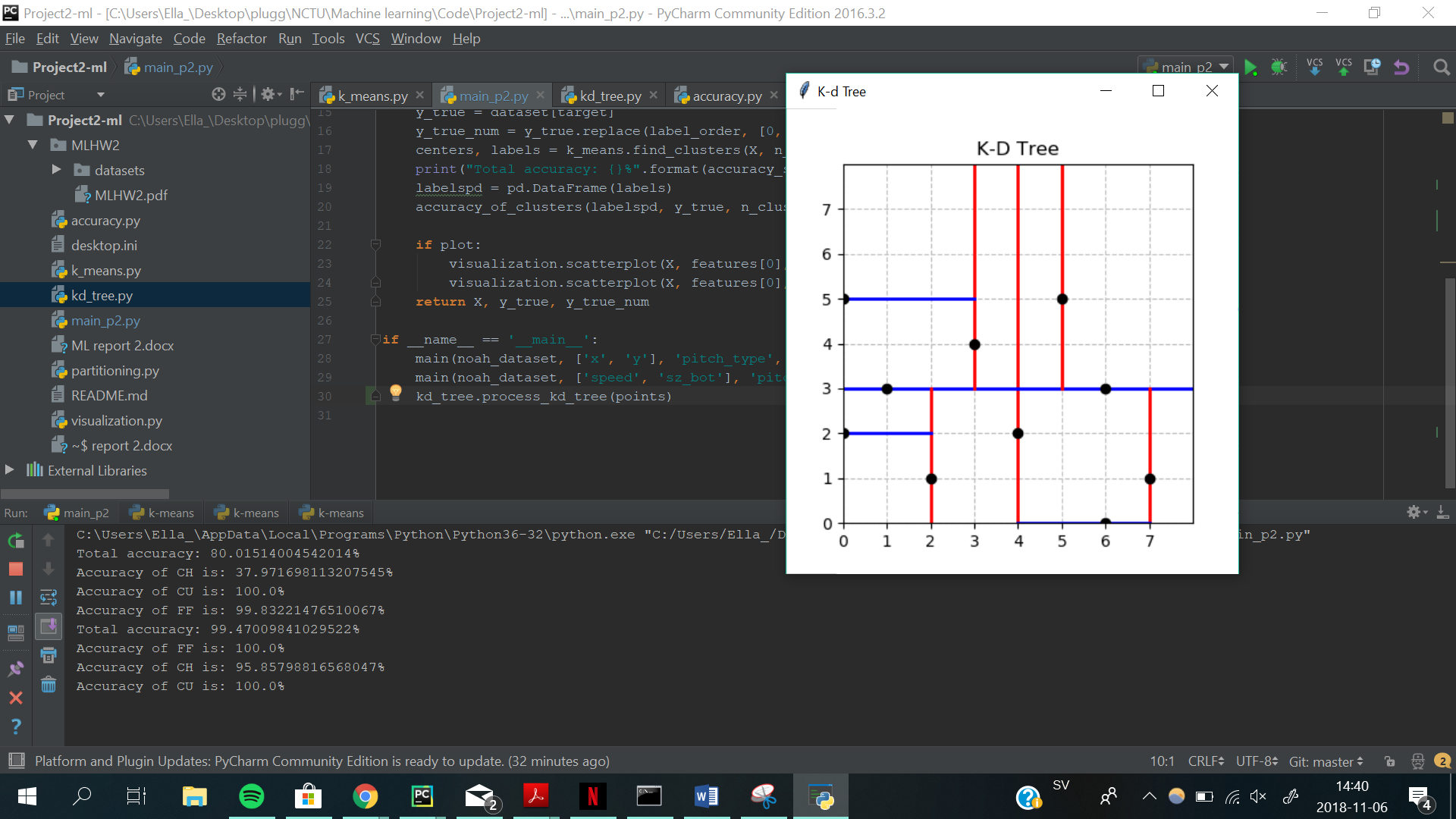


Figure 6, Ella Söderberg

A screenshot of a computer screen

Description automatically generated

Figure 6, Jorge Pineda