Data Classification

Implementing k-means and k-medians clustering algorithms

(COMP527)

Assignment 2

Name: Daniel Fox

Student ID: 201278002

Student Email: d.fox4@liverpool.ac.uk

Date:07/05/2021

Contents

Question 1

(25 marks) Implement the *k*-means clustering algorithm to cluster the instances into *k* clusters.

Step 1: How to sorting Data

Read each data file (animals,countries,fruits,veggies) and convert the data into numpy arrays.The data was read using the csv module to help parse the data.

```
In [9]: def parseData(lists):
            """
            Main function to read data and sort
            lists=['animals','countries','fruits','veggies']
            conversion is a function called for each file name in list.
            concatenate the dataset and combine all data and return it.

            """
            animals=conversion(lists[0])
            countries = conversion(lists[1])
            fruits = conversion(lists[2])
            veggies = conversion(lists[3])

            #combining
            dataset = np.concatenate((animals,countries), axis=0)
            dataset = np.concatenate((dataset,fruits), axis=0)
            dataset = np.concatenate((dataset,veggies), axis=0)

            return dataset

        def conversion(list_name):
            """
            reads data using csv import
            convert the data into an array.
            add a new column caled category to track all the data
            combines the new_data (category) to the dataset

            """
            data = open(list_name, 'rt')
            data = csv.reader(data, delimiter=' ')
            data = list(data)
            data = np.array(data)

            # #add new column for a category
            category = np.array([list_name])
            new_data = np.tile(category[np.newaxis,:], (data.shape[0],1))
            dataset = np.concatenate((data,new_data), axis=1)

            return dataset
```

Figure 1:ParseData function

Above is the functions used to read each text file and combined the data into one numpy array. Which will then be used with the K clustering algorithm. Below shows the final dataset of the shape of size (329,302) . The function also adds a new category column to the data to help identify what data belongs to what file to help determined the B-Cubed calculations by taking the max amount of expected values for each data file.

```
In [22]: fNames=['animals','countries','fruits','veggies']
         dataset=parseData(fNames)
         print("Dataset Shape:",dataset.shape)
         print("Dataset:",dataset)

         Dataset Shape: (329, 302)
         Dataset: [['elephant' '-0.015926' '-0.079864' ... '0.0039075' '-0.035713'
           'animals']
          ['leopard' '0.47727' '-0.91587' ... '-0.24957' '0.02851' 'animals']
          ['dog' '-0.33575' '0.38897' ... '-0.23516' '0.039194' 'animals']
          ...
          ['quandong' '-0.36216' '-0.5386' ... '0.14061' '-0.38506' 'veggies']
          ['sunchokes' '-0.14323' '-0.31758' ... '-0.58964' '0.25269' 'veggies']
          ['zucchini' '-0.58577' '-0.37071' ... '-0.84361' '0.087304' 'veggies']]
```

Figure 2:Printing Dataset

```
In [24]: def maxCategoryIndex(dataset,category):
             """
             Used for B-CUBED to find the total amount in index catergory
             """
             maxIndex= np.where(dataset[:,-1]==category)[0][-1]
             return maxIndex
```

```
In [28]: a=maxCategoryIndex(dataset,fNames[0]) #used for B-CUBED find max index values.
         c=maxCategoryIndex(dataset,fNames[1])
         f=maxCategoryIndex(dataset,fNames[2])#find max index by using the category column added to identify data
         v=maxCategoryIndex(dataset,fNames[3])
         print(a)
         print(c)
         print(f)
         print(v)

         49
         210
         268
         328
```

Figure 3:Finding max index for each category.

Above is the function used to find the max index of each category which was found by using the category column added in the last function to identify the data. After this is stored the column is deleted to pass the data in the K-clustering algorithm to be randomised.

```
In [36]: dataset=np.delete(dataset,0,axis=1)#delete the names, first column
         data=np.delete(dataset,-1,axis=1)#delete the categories added earlier to label data (animals,countries..)
         data=np.array(data).astype(np.float) #convert array values from strings to floats

         print(data)

         [[-0.33218    0.034246   0.11045   ... -0.023002   0.0039075 -0.035713 ]
          [-0.2977    -0.22489    0.55337   ... -0.44699   -0.24957    0.02851  ]
          [-0.41929   -0.33219    0.5317    ... -0.14929   -0.23516    0.039194 ]
          ...
          [-0.66052    0.35979    0.22559   ...  0.026223   0.14061   -0.38506  ]
          [-0.39174   -0.63746   -0.12007   ... -0.56286   -0.58964    0.25269  ]
          [-0.12452   -0.60234    0.70299   ... -0.0054729 -0.84361    0.087304 ]]
```

Figure 4:Removing characters and strings

Deleting the string names and category names in the dataset and then convert the data into a numpy array of floats. This is then used to be passed into the algorithm.

Step 2: K-Means algorithm:

```
class kClusteringAlgorithm():
    def __init__(self,data,k,method,norm):
        """
        K Clustering algorithm which can operate as mean or medians
        args:
        data=full dataset for k clustering
        k=k the amount of centroids, and then allocates every data point to the nearest cluster, while keeping the centroids as
        small as possible.
        method= choose between means or medians for k-method clustering algorithm.
        randomise=call randomise function which shuffles the dataset
        norm=default (false) is normalisation l2. If True perform l2 normalisation for the data.Normalise each row so
        that the sum of each row is equal 1.

        kmeans:
            k-means unsupervised  learning  algorithm  that  solve  the clustering problem. The procedure follows a simple way
            to classify a given data set  through a certain number of  clusters (k-clusters). It defines k number of centers,
            one for each cluster. Take each point belonging  to a  given data set and associate it to the nearest center.
            Assigns data points to a cluster such that the sum of the squared distance between the data points and the cluster's

            K means methos involves using the Euclidean Distance to find the distance between two datapoints

        kmedians:
            Instead of calculating the mean for each cluster to determine its centroid, one instead calculates the median.
            This has the effect of minimizing error over all clusters with respect to the 1-norm distance metric,
            as opposed to the squared 2-norm distance metric (which k-means does.)

            Kmedians method involves using the Manhattan Distance to solve the l1 normalisation

        """
        self.data=data
        self.method=method
        self.norm=norm
        self.k=k
```

```
    def randomiseDataset(self):
        """
        randomise the data before passing into kclustering algorithm
        """
        randomise=np.random.randint(self.data.shape[0],size=self.k)#randomise each row
        return randomise

    def Euclidean(self,dist):
            """
                The mean is a least squares estimator of location. It is appropriate to use with squared deviations
                find distance between two points/rows of data based on pythagoream theorem
                dist=distance to each cluster values
            """
            for i in range(len(self.centroids)):
                    dist[:,i]=np.linalg.norm(self.data-self.centroids[i],axis=1)

            return dist
```

The Euclidean distance method was used to calculate the k-means distances for each cluster. For KMedians it was best to use the Manhattan method to find each cluster.

```python
def run(self):
    """
    Runs the algorithm
    first: Append randmoised dataet to centroids
    oldCentroids=makes a copy of centroids to work out the error

    make centroids an array

    Work out error = Centroids - old centroids
    numError=0 counts errors
    While error not equal 0 "Train"

    dist=create an array to hold distanes, set as zeros of datset shape [0] (rows), with k amount (columns)

    The run will then decide on mean or median , depending on the method

    if method == mean:
        Pass the dist variable into the Euclidean to normalise data, np.linalg.norm(self.data-self.centroids[i]) and
        update the dist varible
        clusters= find the argmin of the new dist of columns(axis=1)
        oldcentroids=update the varibale to the new centroids
        work out the mean for each centroid (at k)
        np.mean(data[clusters==i],axis=0)row
        clusters is found from minimum distances and for each clusters where they are equal k and work out the mean
        and then sort the values into centroid groups.

    if method==median:
        Pass the dist variable into the Manhattan to normalise data, np.sum(np.abs(self.data-self.centroids[i])
        and update the dist varible
        clusters= find the argmin of the new dist of columns(axis=1)
        oldcentroids=update the varibale to the new centroids (clustered group)
        work out the median for each centroid (at k).np.median(data[clusters==i],axis=0)row
        clusters is found from minimum distances and for each clusters where they are equal k and work out the
        median and then sort the values into centroid groups.

    error is updated once clusters have been calcualted when error=0 break out of while loop
    return clusters for B-CUBED calcs
    """
    l2=""
    if norm==True: #perform L2 normalisation
        #gives better results from
        #https://macnux.medium.com/normalization-using-numpy-norm-simple-examples-like-geeks-b079bc4ea06b
        l2="with L2 Normalization"
        for i in range(len(self.data)):
            x_norm_col = np.linalg.norm(self.data[i], axis=0)
            self.data[i]=self.data[i]/x_norm_col
            #PROOF that data is normalised
            #L=np.linalg.norm(self.data[i])
            ##print(L) #PRINTS 1 as its normalised each row
        #ANOTHER METHOD
            # length = len(self.data)
            # for i in range(length):
            #     norm = np.sqrt(np.sum(self.data[i] * self.data[i]))
            #     self.data[i] = self.data[i] / norm
```

```python
    np.random.seed(10)#2,6,7,8 #10 works best
    self.centroids=[]#acts as the label/group of clusters
    self.randomise=self.randomiseDataset()#ranmdomise dataset
    for i in self.randomise:
        self.centroids.append(data[i])

    oldCentroids=np.zeros(np.shape(self.centroids))  #used for error
    self.centroids=np.array(self.centroids)#make nparray

    error=np.linalg.norm(self.centroids-oldCentroids)#determine starting error taking
    #self.centroids vs old self.centroids which is currently set at zeros.
    numError=0#counter for errors

    while error != 0:
        dist=np.zeros([self.data.shape[0],self.k])#determine distances for working out
        #clusters all are equal zero at first , taking the row of the dataset by the amount of k self.centroids.
        numError+=1
        if self.method=="mean":
            method_name="Euclidean Distance"
            dist=self.Euclidean(dist)

            clusters=np.argmin(dist, axis = 1)#clusters determined from distance variable,
            #take the min values for finding closest points to each other.
            oldCentroids=np.array(self.centroids)#reupdate old self.centroids after checking error
            for i in range(self.k):
                self.centroids[i] = np.mean(self.data[clusters==i],axis=0)#finding mean range in k

        elif self.method == "median":
            method_name="Manhattan Distance"
            dist=self.Manhattan(dist)

            clusters=np.argmin(dist,axis=1)
            oldCentroids=np.array(self.centroids)#reupdate old self.centroids after checking error
            for i in range(self.k):
                self.centroids[i] = np.median(self.data[clusters == i],axis=0)#calc median
        #update error until error=0 then break out of while loop
        error=np.linalg.norm(self.centroids-oldCentroids)

    predicted_clusters = clusters
    print("-----------------------------------------------------------------")
    print("Final Results of K-%s Clustering with %s while k=%s %s"%(self.method,method_name,self.k,l2))

    return predicted_clusters
```

Figure 5:K-Clustering Algorithm

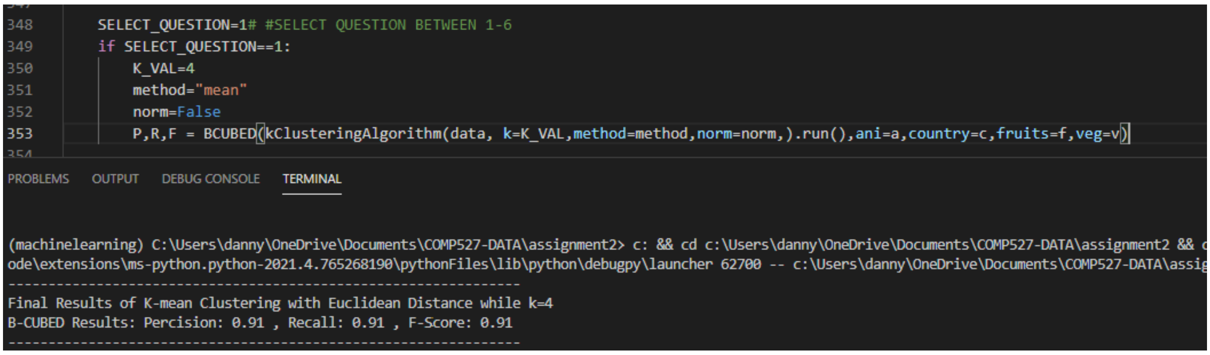

Figure 6:K-Means Results

Using random seed 10 to replicate results for all data.

Question 2

(25 marks) Implement the *k*-medians clustering algorithm to cluster the instances into *k* clusters.

The k-medians has been integrated into the KClusteringAlgorithm class so the user selects what type of algorithm they want, means or median. The median also used the Manhattan distance to work out the clusters.

```
            elif self.method == "median":
                method_name="Manhattan Distance"
                dist=self.Manhattan(dist)

                clusters=np.argmin(dist,axis=1)
                oldCentroids=np.array(self.centroids)#reupdate old self.centroids after checking error
                for i in range(self.k):
                    self.centroids[i] = np.median(self.data[clusters == i],axis=0)#calc median
            #update error until error=0 then break out of while loop
            error=np.linalg.norm(self.centroids-oldCentroids)

        predicted_clusters = clusters
        print("----------------------------------------------------------------")
        print("Final Results of K-%s Clustering with %s while k=%s %s"%(self.method,method_name,self.k,l2))

        return predicted_clusters
```

Figure 7:K-Clustering algorithm - Median method

```
        #used for median
    def Manhattan(self,dist):
        """
        The main method for median
        The median is the best absolute deviation estimator or location. It is appropriate to use with absolute deviations
        find distance between two points measured along axes at  right angles
        sum of absolute differences between each vector
        When distances is updated its done by taking slice of ith column vector of the matrix where i is the len of
        self.centroids
        """
        for i in range(len(self.centroids)):
                dist[:,i] = np.sum(np.abs(self.data-self.centroids[i]), axis=1)
        return dist
```

Figure 8:Manhattan distance for K-Median

Above shows that if the method variable has the value of median then it will perform the k-median clustering algorithm meanwhile if the method is select to be mean then it will perform the k means.

```
    elif SELECT_QUESTION==2:
        K_VAL=4
        method="median"
        norm=False
        P,R,F=BCUBED(kClusteringAlgorithm(data, k=K_VAL,method=method,norm=norm,).run(),ani=a,country=c,fruits=f,veg=v)

    elif SELECT_QUESTION == 3:
```

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL

----------------------------------------------------------------

(machinelearning) C:\Users\danny\OneDrive\Documents\COMP527-DATA\assignment2> c: && cd c:\Users\danny\OneDrive\Documents\COMP527-DATA\assignment2 && cmd /C "
ode\extensions\ms-python.python-2021.4.765268190\pythonFiles\lib\python\debugpy\launcher 62955 -- c:\Users\danny\OneDrive\Documents\COMP527-DATA\assignment2\
----------------------------------------------------------------
Final Results of K-median Clustering with Manhattan Distance while k=4
B-CUBED Results: Percision: 0.96 , Recall: 0.95 , F-Score: 0.95
----------------------------------------------------------------
```

Figure 9:Printing K-Median results

Question 3

***(10 marks)** Run the k-means clustering algorithm you implemented in part (1) to cluster the given instances. Vary the value of k from 1 to 9 and compute the B-CUBED precision, recall, and F-score for each set of clusters. Plot k in the horizontal axis and the B-CUBED precision, recall and F-score in the vertical axis in the same plot.*

Step1: Introducing B-CUBED function:

```
def BCUBED(predictedClusters,ani,country,fruits,veg):
    """
    Function to work out B-CUBED values: Percision , Recall, F-Score

    Args:
        clusters=predicted clusters from the run function in class Kclustering
        ani=Animals size (49)
        country=countries (210)
        fruits=fruits (268)
        veg=veggies (328)
        k=k for printing to terminal

    """
    #clusters.size == 329
    #creating objects of the index position of the different classes
    a = predictedClusters[:ani+1] #+1 for 0th 0-50 elements
    c = predictedClusters[ani+1:country+1] #50-211 elements
    f = predictedClusters[country+1:fruits+1]#211-269
    v = predictedClusters[fruits+1:veg+1]#269-329
    #print(a)
    TP = 0 #true positives
    TN = 0#true negatives
    FP = 0#false positives
    FN = 0#false negatives

#check animals
    for i,_ in enumerate(a):
        #print(i)
        for j,_ in enumerate(a):#animals
            #print(j)
            if j>i and i!=j:#iterate through and count true positives
                if(a[i]==a[j]): TP+=1
                else: FN+=1 #false negative
        for j,_ in enumerate(c):#through countries
            if(a[i]==c[j]): FP+=1 #check against countries if a[i]==c[j] then false positive
            else: TN+=1 #anything else is a true negative
        for j,_ in enumerate(f): #iterate through fruits
            if(a[i]==f[j]):FP+=1#check against countries if a[i]==f[j] then false positive
            else:TN+=1
        for j,_ in enumerate(v):  #veggies
            if(a[i]==v[j]): FP+=1
            else:TN+=1
```

Continued …

```
#countries
for i,_ in enumerate(c):#start at countries and do the same to check for true positives
    #dont need to do animals since its alread done.
    for j,_ in enumerate(c):
        if j>i and i!=j:
            if(c[i]==c[j]):TP+=1
            else: FN+=1
    for j,_ in enumerate(f):
        if(c[i]==f[j]): FP+=1
        else:TN+=1
    for j,_ in enumerate(v):
        if(c[i]==v[j]): FP+=1
        else:TN+=1

#fruits
for i,_ in enumerate(f):#check fruits
    for j,_ in enumerate(f):
        if j>i and i!=j:
            if(f[i]==f[j]): TP+=1
            else: FN+=1
    for j,_ in enumerate(v):
        if(f[i]==v[j]): FP+=1
        else:TN+=1

#veggie
for i,_ in enumerate(v):#finally check true positives and false negatives for veggies.
    for j,_ in enumerate(v):
        if j>i and i!=j:
            if(v[i]==v[j]): TP+=1
            else: FN+=1

#calcs for B-CUBED
P = round((TP / (TP + FP)),2) #Percision round to 2 decimal places
R = round((TP / (TP + FN)),2)  #Recall round to 2 decimal places
F = round((2 * (P * R) / (P + R)),2) #F-score round to 2 decimal places

print("B-CUBED Results: Percision:", P, ", Recall:", R, ", F-Score:", F)
print("----------------------------------------------------------------")

#for plotting values
return P, R, F
```

Figure 10:BCUBED Function

Step 2: Plotting functions

```
def plot(k,p,r,f,method,l2):
    """
    Plot the B-Cubed results for all of K (1-9)
    k=1-9
    p=percision from B-CUBED
    r=recall from B-CUBED
    f=F-score from B-cubed
    method=Name of method (mean or median)
    Plot results

    """
    plt.plot(k,p,label="Percision")
    plt.plot(k,r,label="Recall")
    plt.plot(k,f,label="F-Score")
    plt.title("K-%s Clustering %s" %(str(method),str(l2)))
    plt.xlabel('Number of Clusters')
    plt.ylabel("Scores")
    plt.legend()
    # Display the plot
    plt.show()

def loopResults(x,norm,method):
    list_k,list_p,list_r,list_f=[],[],[],[] #FOR plotting
    for k in range(1,10):
        list_k.append(k)
        P,R,F=BCUBED(kClusteringAlgorithm(x,k=k,norm=norm,method=method).run(),a,c,f,v)
        list_p.append(P)
        list_r.append(R)
        list_f.append(F)
    if method=="mean":
        l2=""
        if norm==True:
            l2="With L2 Normalization"
        plot(list_k,list_p,list_r,list_f,method="mean",l2=l2)
    if method=="median":
        l2=""
        if norm==True:
            l2="With L2 Normalization"
        plot(list_k,list_p,list_r,list_f,method="median",l2=l2)
    return list_f
```

Figure 11:Plot BCUBED function

Above is showing the plot function which plots the B-CUBED results against the Scores and number of clusters used. The results function is called to be used for questions 3-6.Which creates lists to track the scores and each k used. K will have a value of 1 to 9 as stated in the question.

Step 3: Results of K-Means

```python
    elif SELECT_QUESTION == 3:
        norm=False
        method="mean"
        loopResults(data,norm=norm,method=method)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
Final Results of K-mean Clustering with Euclidean Distance while k=1
B-CUBED Results: Percision: 0.32 , Recall: 1.0 , F-Score: 0.48
---------------------------------------------------------------
---------------------------------------------------------------
Final Results of K-mean Clustering with Euclidean Distance while k=2
B-CUBED Results: Percision: 0.65 , Recall: 1.0 , F-Score: 0.79
---------------------------------------------------------------
---------------------------------------------------------------
Final Results of K-mean Clustering with Euclidean Distance while k=3
B-CUBED Results: Percision: 0.83 , Recall: 0.99 , F-Score: 0.9
---------------------------------------------------------------
---------------------------------------------------------------
Final Results of K-mean Clustering with Euclidean Distance while k=4
B-CUBED Results: Percision: 0.91 , Recall: 0.91 , F-Score: 0.91
---------------------------------------------------------------
---------------------------------------------------------------
Final Results of K-mean Clustering with Euclidean Distance while k=5
B-CUBED Results: Percision: 0.91 , Recall: 0.58 , F-Score: 0.71
---------------------------------------------------------------
---------------------------------------------------------------
Final Results of K-mean Clustering with Euclidean Distance while k=6
B-CUBED Results: Percision: 0.91 , Recall: 0.49 , F-Score: 0.64
---------------------------------------------------------------
---------------------------------------------------------------
Final Results of K-mean Clustering with Euclidean Distance while k=7
B-CUBED Results: Percision: 0.94 , Recall: 0.45 , F-Score: 0.61
---------------------------------------------------------------
---------------------------------------------------------------
Final Results of K-mean Clustering with Euclidean Distance while k=8
B-CUBED Results: Percision: 0.94 , Recall: 0.42 , F-Score: 0.58
---------------------------------------------------------------
---------------------------------------------------------------
Final Results of K-mean Clustering with Euclidean Distance while k=9
B-CUBED Results: Percision: 0.92 , Recall: 0.34 , F-Score: 0.5
---------------------------------------------------------------
```

Figure 12:Results K-Means ranging k= 1 to 9

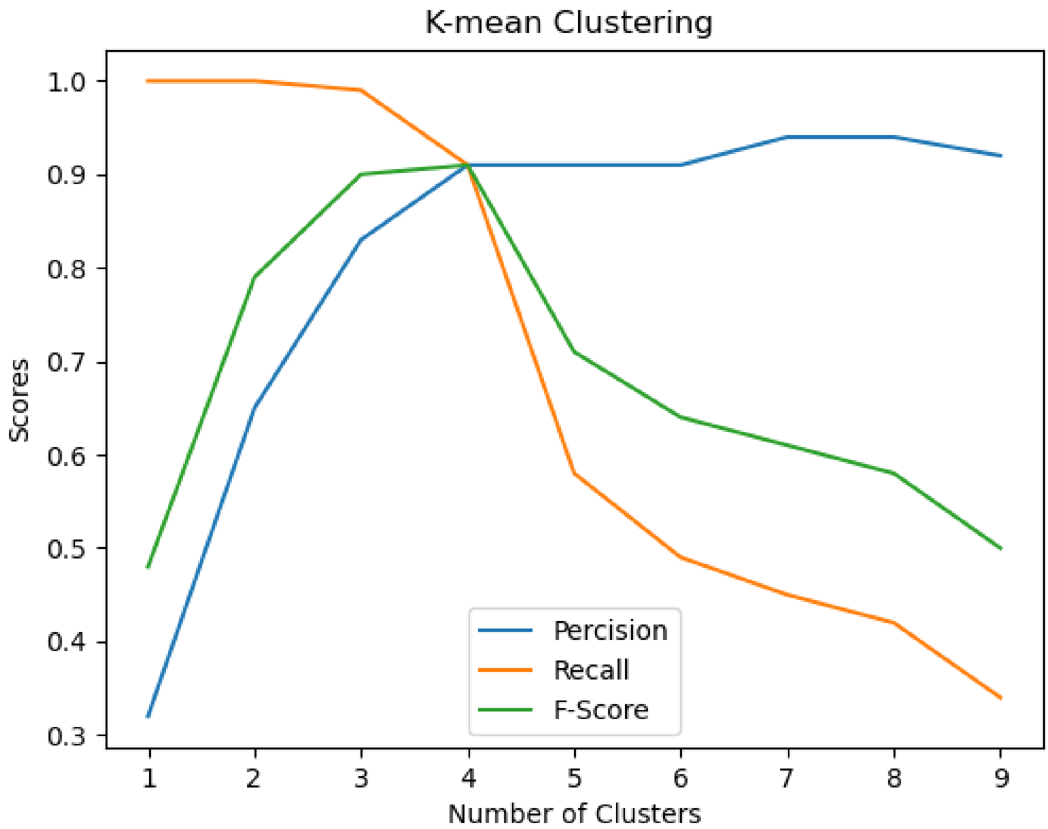

Figure 13:Graph of K-Mean results

Question 4

***10 marks)** Now re-run the k-means clustering algorithm you implemented in part (1) but normalise each object (vector) to unit l_2 length before clustering. Vary the value of k from 1 to 9 and compute the B-CUBED precision, recall, and F-score for each set of clusters. Plot k in the horizontal axis and the B-CUBED precision, recall and F-score in the vertical axis in the same plot.*

Step 1: Introducing L2 Norm

```
    l2=""
    if norm==True: #perform l2 normalisation
        #gives better results from https://macnux.medium.com/
        normalization-using-numpy-norm-simple-examples-like-geeks-b079bc4ea06b
        l2="with L2 Normalization"
        for i in range(len(self.data)):
            x_norm_col = np.linalg.norm(self.data[i], axis=0)
            self.data[i]=self.data[i]/x_norm_col
            #PROOF that data is normalised
            #l=np.linalg.norm(self.data[i])
            ##print(l) #PRINTS 1 as its normalised each row
        #ANOTHER METHOD
            # length = len(self.data)
            # for i in range(length):
            #     norm = np.sqrt(np.sum(self.data[i] * self.data[i]))
            #     self.data[i] = self.data[i] / norm
```

Figure 14:L2 Normalization implementation

The normalisation is performed as shown above which is at the initialisation of the k clustering class. If norm is set to true, then l2 normalisation will be performed before clustering.

Step 2: Results of L2 Norm K-Means:

```
    elif SELECT_QUESTION == 4:
        norm=True
        method="mean"
        loopResults(data,norm=norm,method=method)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
-----------------------------------------------------------------
-----------------------------------------------------------------
Final Results of K-mean Clustering with Euclidean Distance while k=1 with L2 Normalization
B-CUBED Results: Percision: 0.32 , Recall: 1.0 , F-Score: 0.48
-----------------------------------------------------------------
-----------------------------------------------------------------
Final Results of K-mean Clustering with Euclidean Distance while k=2 with L2 Normalization
B-CUBED Results: Percision: 0.65 , Recall: 1.0 , F-Score: 0.79
-----------------------------------------------------------------
-----------------------------------------------------------------
Final Results of K-mean Clustering with Euclidean Distance while k=3 with L2 Normalization
B-CUBED Results: Percision: 0.83 , Recall: 1.0 , F-Score: 0.91
-----------------------------------------------------------------
-----------------------------------------------------------------
Final Results of K-mean Clustering with Euclidean Distance while k=4 with L2 Normalization
B-CUBED Results: Percision: 0.92 , Recall: 0.95 , F-Score: 0.93
-----------------------------------------------------------------
-----------------------------------------------------------------
Final Results of K-mean Clustering with Euclidean Distance while k=5 with L2 Normalization
B-CUBED Results: Percision: 0.87 , Recall: 0.58 , F-Score: 0.7
-----------------------------------------------------------------
-----------------------------------------------------------------
Final Results of K-mean Clustering with Euclidean Distance while k=6 with L2 Normalization
B-CUBED Results: Percision: 0.86 , Recall: 0.52 , F-Score: 0.65
-----------------------------------------------------------------
-----------------------------------------------------------------
Final Results of K-mean Clustering with Euclidean Distance while k=7 with L2 Normalization
B-CUBED Results: Percision: 0.96 , Recall: 0.49 , F-Score: 0.65
-----------------------------------------------------------------
-----------------------------------------------------------------
Final Results of K-mean Clustering with Euclidean Distance while k=8 with L2 Normalization
B-CUBED Results: Percision: 0.95 , Recall: 0.47 , F-Score: 0.63
-----------------------------------------------------------------
-----------------------------------------------------------------
Final Results of K-mean Clustering with Euclidean Distance while k=9 with L2 Normalization
B-CUBED Results: Percision: 0.94 , Recall: 0.36 , F-Score: 0.52
-----------------------------------------------------------------
```

Figure 15:K-Means L2 Normalised Resutls k=1to9

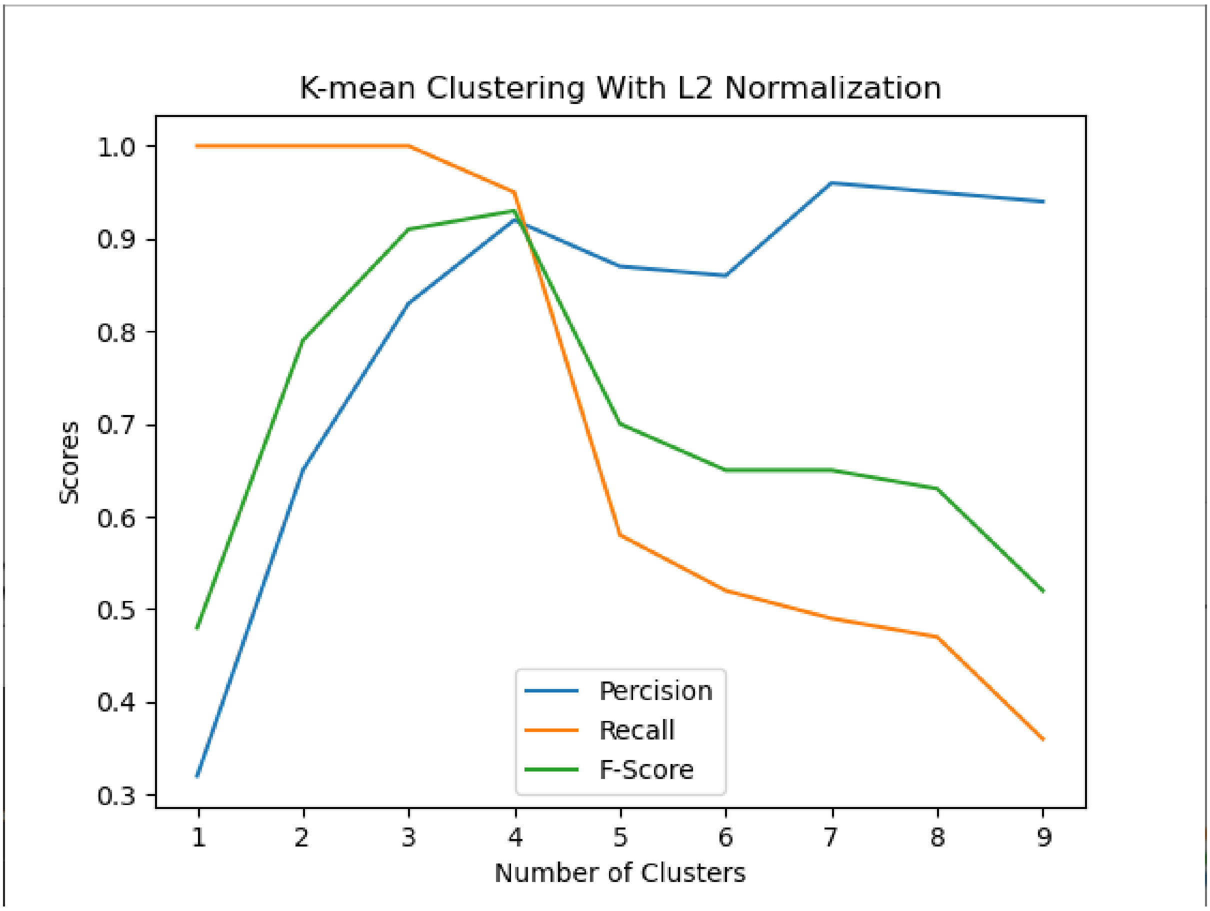

Figure 16:K-Means clustering graph while L2 Normalised

Question 5

***(10 marks)** Run the k-medians clustering algorithm you implemented in part (2) over the unnormalised objects. Vary the value of k from 1 to 9 and compute the B-CUBED precision, recall, and F-score for each set of clusters. Plot k in the horizontal axis and the B-CUBED precision, recall and F-score in the vertical axis in the same plot.*

Step 1: Results of K-Medians

```python
    elif SELECT_QUESTION == 5:
        norm=False
        method="median"
        loopResults(data,norm=norm,method=method)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
------------------------------------------------------------
------------------------------------------------------------
Final Results of K-median Clustering with Manhattan Distance while k=1
B-CUBED Results: Percision: 0.32 , Recall: 1.0 , F-Score: 0.48
------------------------------------------------------------
------------------------------------------------------------
Final Results of K-median Clustering with Manhattan Distance while k=2
B-CUBED Results: Percision: 0.65 , Recall: 1.0 , F-Score: 0.79
------------------------------------------------------------
------------------------------------------------------------
Final Results of K-median Clustering with Manhattan Distance while k=3
B-CUBED Results: Percision: 0.83 , Recall: 0.98 , F-Score: 0.9
------------------------------------------------------------
------------------------------------------------------------
Final Results of K-median Clustering with Manhattan Distance while k=4
B-CUBED Results: Percision: 0.97 , Recall: 0.96 , F-Score: 0.96
------------------------------------------------------------
------------------------------------------------------------
Final Results of K-median Clustering with Manhattan Distance while k=5
B-CUBED Results: Percision: 0.95 , Recall: 0.6 , F-Score: 0.74
------------------------------------------------------------
------------------------------------------------------------
Final Results of K-median Clustering with Manhattan Distance while k=6
B-CUBED Results: Percision: 0.95 , Recall: 0.54 , F-Score: 0.69
------------------------------------------------------------
------------------------------------------------------------
Final Results of K-median Clustering with Manhattan Distance while k=7
B-CUBED Results: Percision: 0.93 , Recall: 0.52 , F-Score: 0.67
------------------------------------------------------------
------------------------------------------------------------
Final Results of K-median Clustering with Manhattan Distance while k=8
B-CUBED Results: Percision: 0.93 , Recall: 0.5 , F-Score: 0.65
------------------------------------------------------------
------------------------------------------------------------
Final Results of K-median Clustering with Manhattan Distance while k=9
B-CUBED Results: Percision: 0.91 , Recall: 0.38 , F-Score: 0.54
------------------------------------------------------------
```

Figure 17:Results of K-Medians while k=1to9

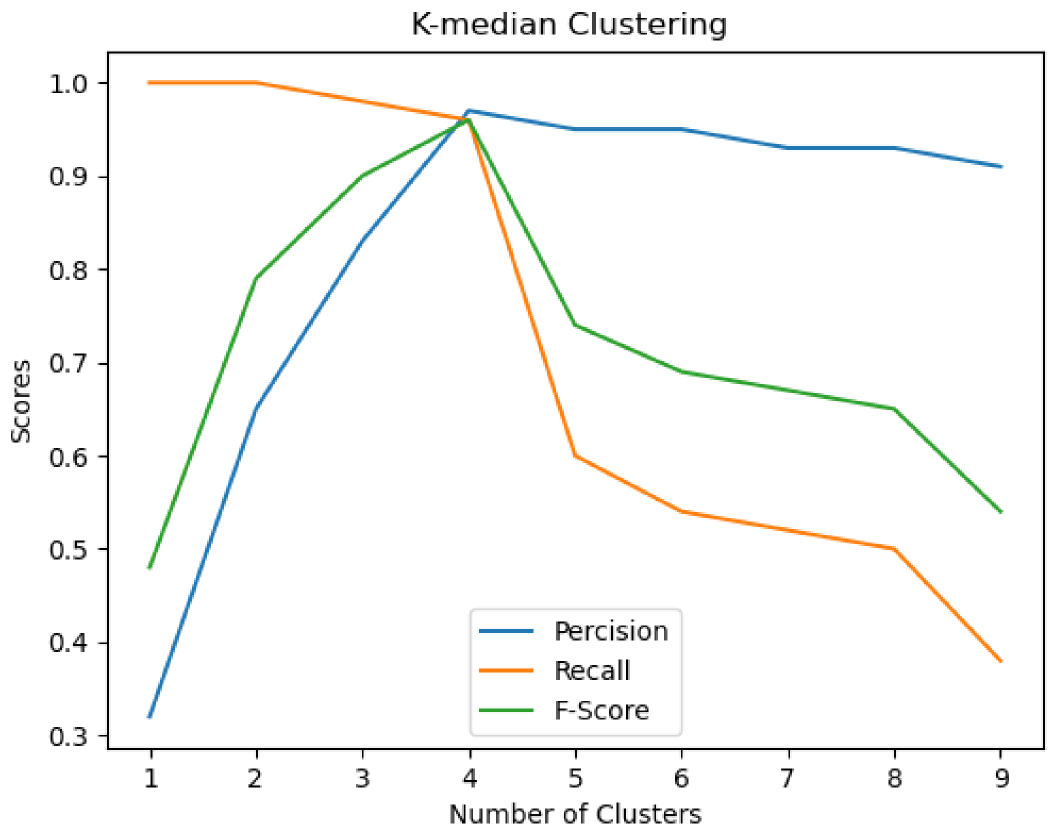

Figure 18:K-Median Clustering graph

Question 6

(10 marks) *Now re-run the k-medians clustering algorithm you implemented in part (2) but normalise each object (vector) to unit l_2 length before clustering. Vary the value of k from 1 to 9 and compute the B-CUBED precision, recall, and F-score for each set of clusters. Plot k in the horizontal axis and the B-CUBED precision, recall and F-score in the vertical axis in the same plot.*

Step 1: Results of L2 Norm K-Medians

```
elif SELECT_QUESTION == 6:
    norm=True
    method="median"
    loopResults(data,norm=norm,method=method)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
Final Results of K-median Clustering with Manhattan Distance while k=1 with L2 Normalization
B-CUBED Results: Percision: 0.32 , Recall: 1.0 , F-Score: 0.48
----------------------------------------------------------------
----------------------------------------------------------------
Final Results of K-median Clustering with Manhattan Distance while k=2 with L2 Normalization
B-CUBED Results: Percision: 0.65 , Recall: 1.0 , F-Score: 0.79
----------------------------------------------------------------
----------------------------------------------------------------
Final Results of K-median Clustering with Manhattan Distance while k=3 with L2 Normalization
B-CUBED Results: Percision: 0.83 , Recall: 0.98 , F-Score: 0.9
----------------------------------------------------------------
----------------------------------------------------------------
Final Results of K-median Clustering with Manhattan Distance while k=4 with L2 Normalization
B-CUBED Results: Percision: 0.97 , Recall: 0.96 , F-Score: 0.96
----------------------------------------------------------------
----------------------------------------------------------------
Final Results of K-median Clustering with Manhattan Distance while k=5 with L2 Normalization
B-CUBED Results: Percision: 0.95 , Recall: 0.6 , F-Score: 0.74
----------------------------------------------------------------
----------------------------------------------------------------
Final Results of K-median Clustering with Manhattan Distance while k=6 with L2 Normalization
B-CUBED Results: Percision: 0.95 , Recall: 0.54 , F-Score: 0.69
----------------------------------------------------------------
----------------------------------------------------------------
Final Results of K-median Clustering with Manhattan Distance while k=7 with L2 Normalization
B-CUBED Results: Percision: 0.93 , Recall: 0.52 , F-Score: 0.67
----------------------------------------------------------------
----------------------------------------------------------------
Final Results of K-median Clustering with Manhattan Distance while k=8 with L2 Normalization
B-CUBED Results: Percision: 0.93 , Recall: 0.5 , F-Score: 0.65
----------------------------------------------------------------
----------------------------------------------------------------
Final Results of K-median Clustering with Manhattan Distance while k=9 with L2 Normalization
B-CUBED Results: Percision: 0.91 , Recall: 0.38 , F-Score: 0.54
----------------------------------------------------------------
```

Figure 19:K-Median scores while L2 Normalised k=1to 9

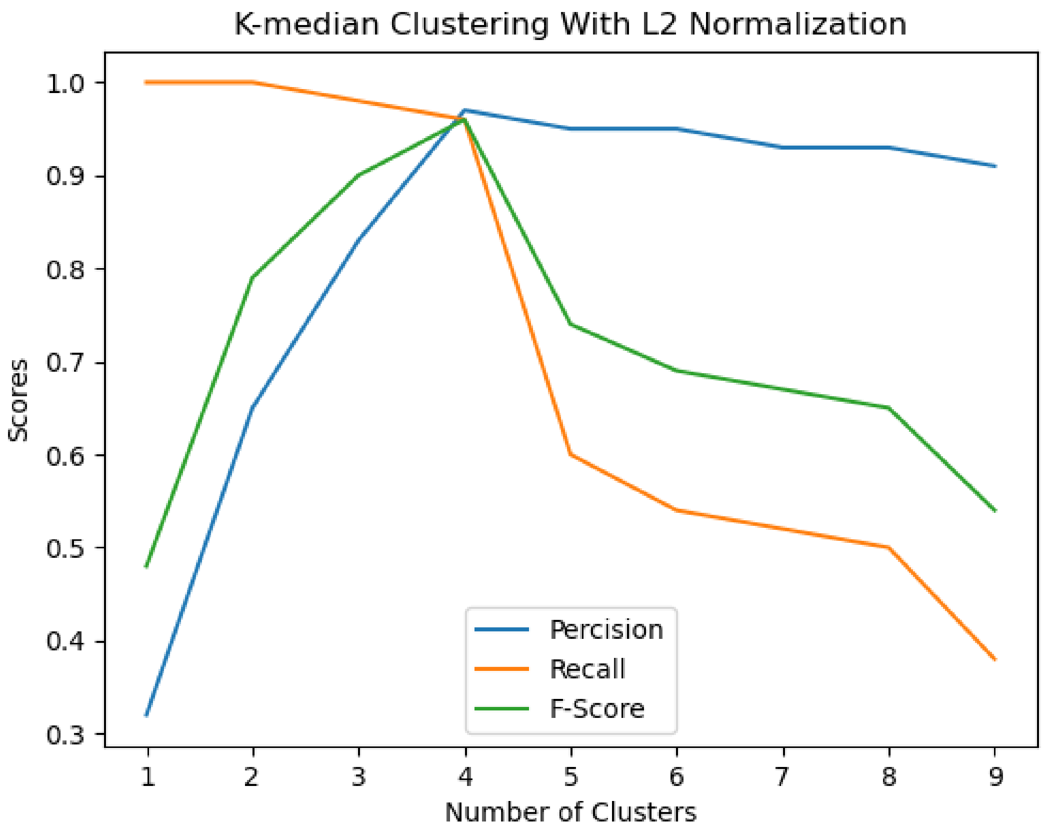

Figure 20:K-Median graph L2 Normalised

Question 7

***(10 marks)** Comparing the different clusterings you obtained in (3)-(6), discuss in which setting you obtained best clustering for this dataset.*

Step 1: Current seed and data comparison for both K-means and K-Medians

```
elif SELECT_QUESTION =="Panda-Compare":
    norm=False
    method="mean"
    list_f=loopResults(data,norm=norm,method=method)

    norm=True
    method="mean"
    list_f_norm=loopResults(data,norm=norm,method=method)

    norm=False
    method="median"
    median_list=loopResults(data,norm=norm,method=method)

    norm=True
    method="median"
    median_list_norm=loopResults(data,norm=norm,method=method)

    counter=[]
    for i in range(1,10):
        counter.append(i)

    data=list(zip(list_f,list_f_norm,median_list,median_list_norm))
    print("---Seed 10-------------")
    print("----Mean F-score--------Median F-Scores-----")
    print(pd.DataFrame(data,index=counter,columns=['Mean','Normalised','Median','Normalised']))
```

Figure 21:Pandas used to print comparison of F-Scores

---Seed 10-------------

----Mean F-score--------Median F-Scores-----

	Mean	Normalised	Median	Normalised
1	0.48	0.48	0.48	0.48
2	0.79	0.79	0.79	0.79
3	0.90	0.91	0.90	0.90
4	0.91	0.93	0.96	0.96
5	0.71	0.70	0.74	0.74
6	0.64	0.65	0.69	0.69
7	0.61	0.65	0.67	0.67
8	0.58	0.63	0.65	0.65
9	0.50	0.52	0.54	0.54

Figure 22:Comparrison of F-Scores for K-means at seed 10 , K-medians and l2 normalisation for k=1-9

After determining the best k clustering values using the BCUBED method for both algorithms while unnormalized and l2 normalised the results were compared by reviewing the overall F-Score of each cluster ranging from k=1 to 9. When analysing the data the best scores relating to the B-CUBED F-Score calculations are typically when k = 4 this makes sense due to knowing the data should be divided into 4 categories(animals,countries,fruits,veggies).

The best F-score can be shown above, using seed(10) and k=4 for both methods have similar high scores above 0.90 for each. The K-means slightly underperforms the K-Medians algorithm in this case above as the K median has a score of 0.96 whereas the K mean scores 0.91 unnormalized and 0.93 l2 normalised. The K-Median has the highest score out of both algorithms while being at 0.96 for both unnormalized and normalised. We noticed that the median tends to have a higher score for k=4 while normalised and unnormalized however when we further vary between different seeds, we notice that the median F-score can largely be affected and more often than not the K-Means with l2 normalisation performs the best with the highest f-score.

Step2: Different seed comparisons

---Seed 10--------------

----Mean F-score--------Median F-Scores-----

	Mean	Normalised	Median	Normalised
1	0.48	0.48	0.48	0.48
2	0.79	0.79	0.79	0.79
3	0.90	0.91	0.90	0.90
4	0.91	0.93	0.96	0.96
5	0.71	0.70	0.74	0.74
6	0.64	0.65	0.69	0.69
7	0.61	0.65	0.67	0.67
8	0.58	0.63	0.65	0.65
9	0.50	0.52	0.54	0.54

---Seed2--------------

----Mean F-score--------Median F-Scores-----

	Mean	Normalised	Median	Normalised
1	0.48	0.48	0.48	0.48
2	0.79	0.79	0.79	0.79
3	0.61	0.64	0.65	0.65
4	0.72	0.73	0.71	0.71
5	0.75	0.70	0.69	0.69
6	0.61	0.57	0.57	0.57
7	0.57	0.61	0.62	0.62
8	0.51	0.55	0.56	0.56
9	0.52	0.53	0.54	0.54

---Seed 1--------------

----Mean F-score--------Median F-Scores-----

	Mean	Normalised	Median	Normalised
1	0.48	0.48	0.48	0.48
2	0.79	0.79	0.79	0.79
3	0.91	0.91	0.90	0.90
4	0.93	0.96	0.96	0.96
5	0.73	0.73	0.74	0.74
6	0.63	0.62	0.62	0.62
7	0.58	0.59	0.59	0.59
8	0.54	0.55	0.55	0.55
9	0.50	0.50	0.52	0.52

---Seed 20--------------

----Mean F-score--------Median F-Scores-----

	Mean	Normalised	Median	Normalised
1	0.48	0.48	0.48	0.48
2	0.79	0.79	0.79	0.79
3	0.79	0.91	0.90	0.90
4	0.90	0.96	0.68	0.68
5	0.88	0.95	0.96	0.96
6	0.65	0.73	0.72	0.72
7	0.56	0.65	0.64	0.64
8	0.64	0.61	0.62	0.62
9	0.63	0.57	0.60	0.60

---Seed 25--------------

----Mean F-score--------Median F-Scores-----

	Mean	Normalised	Median	Normalised
1	0.48	0.48	0.48	0.48
2	0.79	0.79	0.79	0.79
3	0.90	0.91	0.90	0.90
4	0.70	0.95	0.69	0.69
5	0.69	0.72	0.75	0.75
6	0.55	0.64	0.62	0.62
7	0.61	0.61	0.59	0.59
8	0.58	0.58	0.59	0.59
9	0.51	0.51	0.51	0.51

---Seed32--------------

----Mean F-score--------Median F-Scores-----

	Mean	Normalised	Median	Normalised
1	0.48	0.48	0.48	0.48
2	0.79	0.79	0.79	0.79
3	0.88	0.90	0.88	0.88
4	0.95	0.96	0.96	0.96
5	0.93	0.75	0.77	0.77
6	0.69	0.95	0.94	0.94
7	0.66	0.74	0.74	0.74
8	0.66	0.74	0.74	0.74
9	0.66	0.73	0.73	0.73

Figure 23:Seed Comparisons of F-Scores k=1to9

When we analysie multiple variations of different seeds we notice that the results can vary largly depending on the randomness of the seed. We can see our seed 10 which is the main seed used for the questions in this report shows that the K-Medians algorithm is best suited for the value of K being equal to 4 or the K-Means l2 Normalised as they all have the same F-Scores. Since we assume 4 is the most optimal K clustering value due to having 4 data classes animals,countries,fruits an veggies. As the K clusters increases the number of centroids the score drops this seems normal as the algorithm will find it difficuilt to differentiate a higher number of k clusters.

When seeds 1 and 32 are set we notice the same results as seed 10 however even with these seeds the F-score's are similar for all cases apart from K-mean unnormalised we can see that for seed 1 K-means at 4 has value of 0.93 and a normalised value of 0.96, so the best choice would be to normilse the data for k-means. When we look at the K-median for the same seed both of the values are equal

0.96. So in this case either k-means normalised or both versions of K-Medians provides the best F-scores.

Comparing different seeds for K-Means

What can be seen above apart from the k=4 mean result being the highest on average is in fact that as the data is normalised using the l2 method, we can see that while the k increases past 4 and 5 we notice the data typically stays consistent in comparrison to the unnormalised data, even with the randomness of the seeds it shows that the data remains similar when normalised and often has a higher F-score than the unnormalised data. Typically the normalised k-means always has the better highest F-score out of the rest of the data.

We also notice the score at k=3 seems to hold a high score on some seeds , even as the kmeans is normalised more often when k=3 the score is near 0.90 this is most likely due to it being easier to differentiate between the data using 3 clustered centroids for our dataset. When k passes 4 it should become harder too differentiate due to there being 4 datasets , so most likely the data should be worse when increasing above k = 5 since the k clusters centroids group the data making the algorithm difficuilt to group the datasets together. While when k is less than 4 it should be easier to determine errors.

Comparing different seeds for K-Median

The data above shows the F-scores found from using the K-Medians method with the Manhattan distance over time. However in regards to K perfoming best at 4 this is not correct for this case, it can be clearly shown on average that K=4 isnt always the best perofming cluster when it comes to the K-median. This is mainly because of instead of calculating the mean of each cluster to determine the centroid it uses the median for each cluster which minimizes the error using 1 norm distance as opposed to squaring it like L2-norm mean. The K-median can run into problems determining clusters using the 1-norm as it struggles to find centers for the centroids as they are more compact. We can see that for seeds 2,20 and 25 when the seed has a low value for k=4 then the highest F-score shows to be typically when k=2 for seed 2 at 0.79, seed 20 and 25 both have the same highest F-score being at K=3 at 0.90. What we notice is that typically the L2 Normalised K-Means still holds the highest score at k=4 out of all the datasets.

In general its best to use K-Median to help identify outliers in the dataset as Mean can massively effect outliers, as an example if the data points where [1,2,3,5,99] its clear that the outlier is 99. The median of the data if 3 whereas the mean would be 22.

Overall

When analysising seeds 20 and 25 we notice that the value of the F-score has dramitaccly changed apart from the K-mean normlisation value. Which can further show that we can assume the K-Mean l2 normalised holds the best F-score on average as the value is consistently high when varying seeds, at seed 20 and 25 is 0.96 and 0.95 respectively, with this data we can assume that the normalisation has a beneficial effect to the K-means algorithm when the data is randomly choosen. On average using the seeds provided for comparrison the K-Means with L2 Normalisation performs the best. However due to the randomness of choosing our starting points for each centroid there is a large amount of other possibilities where if k is equal 4 centroids out of 329 data points then there is $(\frac{\mathrm{n!}}{\mathrm{r!(n-r)!}}) = (\frac{329!}{4!(329-4)!} = 479k\ ways\ approximatly$ of having different starting positions for the k clustering. So if the algorithm would be used it would most likely work best at K-means l2 Normalised however there is a likelyhood that the randomised data points are not well positioned throughout the entire dataset so there is also a chance that the K-medians could also be another optimal solutioin.

Step 3: Error noted:

```
C:\Users\danny\anaconda3\envs\machinelearning\lib\site-packages\numpy\core\fromnumeric.py:3372: RuntimeWarning: Mean of empty slice.
  return _methods._mean(a, axis=axis, dtype=dtype,
C:\Users\danny\anaconda3\envs\machinelearning\lib\site-packages\numpy\core\_methods.py:162: RuntimeWarning: invalid value encountered in true_divide
  ret = um.true_divide(
```

Figure 24:Notable error found

Note: When changing to other seeds errors occur from the normalised array of data due to the randomness and seed used. Unsure how to fix the error I assume its due to the randomness of the seed. If I ran the data individually most of the time it would compute the results.