भारतीय प्रौद्योगिकी संस्थान हैदराबाद
Indian Institute of Technology Hyderabad

A comprehensive introduction to K-means clustering

Krati Arela
Shreshta Thumati

Indian Institute of Technology, Hyderabad,
IITH Main Road, Near NH-65, Sangareddy, Kandi,
Telangana 502285, India

ee18btech11050@iith.ac.in
ee18btech11041@iith.ac.in

March 28, 2021

Abstract

Amongst the various data analysis techniques under the unsupervised learning domain, clustering is one of the most common exploratory approach used to get an intuition about the structure of the data. It can be defined as the task of identifying homogeneous subgroups/substructure within the data space such that data points in each cluster are as similar as possible according to an application specific similarity measure such as euclidean-based distance, correlation-based distance, etc. Much efforts have been put in by the various communities to further enhance the performance and computational efficiency of the clustering analysis given its prospects and applications. In this work, we present the use of one of the clustering algorithms/methods known as K-means clustering, its applications in the task of image compression and discuss in brief about the evaluation metric for this algorithm and drawbacks of the same.

Contents

1 Introduction

Cluster analysis aims at grouping data objects based only on information found in the data that describes the objects and their relationships. The goal is that after the clustering is done, the objects within a group are similar (or related) to one another and different from (or unrelated to) the objects in other groups. The greater the similarity (or homogeneity) within a group and the greater the difference between groups, the better or more distinct the clustering.
While in some cases, cluster analysis is only a starting point for other purposes such as data summarization, whether for understanding or utility, cluster analysis has long played an important role in a wide variety of fields: psychology and other social sciences, biology, statistics, pattern recognition, information retrieval, machine learning, and data mining.
There are various kinds of clustering methods depending on the need and application at hand. In this work, we present one of the oldest and most used clustering technique - The K-means. The paper is organized as follows:- The next section introduces the k-means algorithm, the idea underlying it and a basic python code snippet to demonstrate the same. The third section discusses the application of the K-means clustering to the problem of image compression. The fourth section formulates the methods to evaluate the performance and drawbacks of the same. In the final section we conclude.

2 K-means algorithm

K-Means Clustering is an Unsupervised Learning algorithm, which groups the unlabeled data set into different clusters. Based on a similarity measure such as euclidean distance between points, it tries to make the inter-cluster data points as similar as possible while also keeping the clusters as different(far) as possible. It assigns data points to a cluster such that the sum of the squared distance between the data points and the cluster's centroid (arithmetic mean of all the data points that belong to that cluster) is at the minimum. The less variation we have within clusters, the more homogeneous (similar) the data points are within the same cluster.
The following is the problem statement that the K-means algorithm addresses:-

Given a set of observations $(\mathbf{x_1}, \mathbf{x_2}, .., \mathbf{x_n})$, where each observation is a d-dimensional real vector, the objective is to partition the *n* observations into *k* $(\leq \mathbf{n})$ sets $\mathbf{S} = \{\mathbf{S_1}, \mathbf{S_2}, ..., \mathbf{S_k}\}$ so as to minimize the within-cluster sum of squares (WCSS) of distance between the data points and their assigned clusters' centroid(i.e variance). Formally,

$$\mathbf{arg}\,\underset{\mathbf{S}}{\mathbf{min}} \sum_{\mathbf{i=1}}^{\mathbf{k}} \sum_{\mathbf{x} \in \mathbf{S_i}} ||\mathbf{x} - \mu_\mathbf{i}||^\mathbf{2} = \mathbf{arg}\,\underset{\mathbf{S}}{\mathbf{min}} \sum_{\mathbf{i=1}}^{\mathbf{k}} |\mathbf{S_i}|\mathbf{Var}\ \mathbf{S_i} \tag{1}$$

where $\mu_\mathbf{i}$ is the mean of points in $\mathbf{S_i}$.

The following steps define the working of the most basic K-means algorithm given the above problem statement :-

```
Step 1: Choose the number of clusters(K)
Step 2: Choose the centroids $c_1, c_2, .....c_k$ randomly from the data set (or by some other optimized method leading to
            better computational performance of the algorithm).
Step 3. Repeat steps 4 and 5 until convergence or until the end of a fixed number of iterations.
Step 4: For each data point $x_i$:
            - find the nearest centroid($c_1, c_2..c_k$)
            - assign the point to that cluster
Step 5: for each cluster j = 1..k
            - new centroid = mean of all points assigned to that cluster
Step 6: End
```

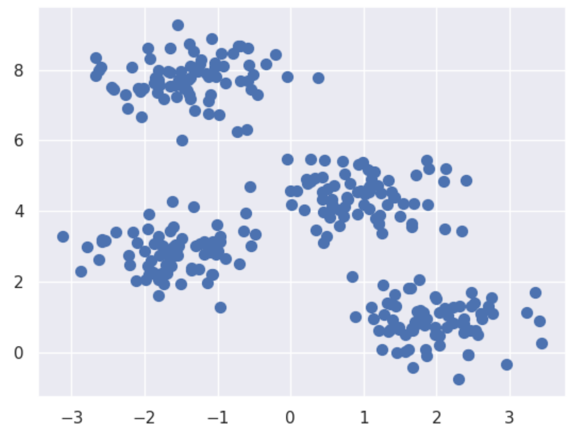
We now present a real life problem statement and apply the k-means algorithm using python to solve the same.

Problem Statement:
Walmart wants to open a chain of stores across a state and wants to find out optimal store locations with respect to customer populat1on density in the state to maximize the sales. Find optimal store locations.

Solution:
The scatter plot for the customer population density is generated using the *samples_generator* module of sklearn library. We then apply K-means algorithm to find the optimum (such that the objective function given in equation (1) is minimized) cluster centers for the data and these centers will be the optimal store locations. *samples_generator* module of sklearn library is used to generate the sampeles for populaiton density

We use a naive implementation of the K-means algorithm as defined by the steps mentioned above and we also compare our results with the inbuilt K-means algorithm module of the sklearn library. (The choice of number of clusters is discussed in the third section of the paper. Right now we assume we know how many clusters to choose (in this case- 4) so as to get the optimal solution.)
The following code plots the scatter plot for customer population density:

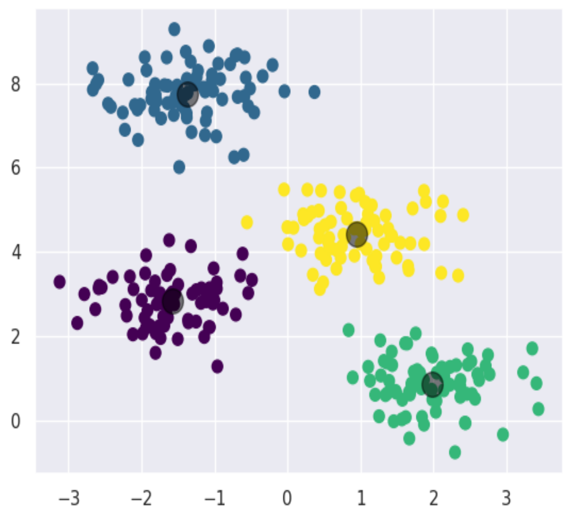
```python
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
from sklearn.datasets.samples_generator import make_blobs

sns.set() #Setting plot style

#Generating the sample data. We generate data distributed in clusters about four centers
#such that the standard deviation for a set of points belonging to a cluster is 0.6. X
    is the
#sample data while y_true are the true centroids of the generated clusters.
X,y_true = make_blobs(n_samples=300, centers =4, cluster_std = 0.60, random_state = 0)

#Plotting the scatter plot for the customer population density.
plt.scatter(X[:,0], X[:,1], s = 50)
plt.show()
```



Scatter plot for customer population density

```
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
from sklearn.datasets.samples_generator import make_blobs
from sklearn.cluster import KMeans
from sklearn.metrics import pairwise_distances_argmin

'''Implementation of K-means algorithm using in-built function'''
kmeans = KMeans(n_clusters=4)
kmeans.fit(X)          #Inbuilt Kmeans algorithm module in sklearn.
y_kmeans = kmeans.cluster_centers_
#y_kmeans are the final centroids output by the inbuilt KMeans algorithm module of
    sklearn library.

'''Our implementation of K-means algorithm'''
#The following is our implementation of the K-means algorithm from scratch using the 6
    steps
#mentioned previously.

def err(y1, y2):   #error function finds distance between new centers and previous
    centers
    return np.sum(np.square(np.linalg.norm(y1-y2)))

def find_clusters(X, n_clusters, error, rseed = 2):
        #X is a (300,2) shaped array containing 300 samples distributed across the x-y
            coordinate system
        #in 4 clusters. The aim is to identify these clusters and corresponding cluster
            centroids.
        #n_clusters is the number of clusters we divide the data into. Here n_clusters =
             4.
        #rseed is the seed value for a pseudo random number generator.

        rng = np.random.RandomState(rseed)
        i = rng.permutation(X.shape[0])[:n_clusters]
        centers = X[i] #We randomly initialize the centroids

        while True: #The loop runs till the centroids converge
            labels = pairwise_distances_argmin(X, centers)
            #labels is the array such that the value of label of i-th data point, labels
                [i] is equal
            # to the number of the cluster that the i-th data point belongs to.
            new_centers = np.array([X[labels==i].mean(0) for i in range(n_clusters)])
            #New centroid for a given cluster is calculated by finding the arithmetic
                mean of the points
            #assigned to that cluster.

            #if the centroids converge then the algorithm ends
            if err(centers, new_centers)<error:
                break
            centers = new_centers

        return centers,labels   #return final coordinates of centroids of the clusters
            and label of each data point

centers,labels = find_clusters(X,4, 0.0001)

print(y_kmeans)  #The centroids for the clusters obtained from the inbuilt module for K-
    means in sklearn.
print(centers)   #The centroids for the clusters obtained from the implementation of the
     k-means algorithm by us.

'''
The following is the output of the above code for the centroid coordinates by inbuilt
K-means module and our implementation of K-means algorithm respectively:-

[[-1.37324398  7.75368871]
 [ 0.94973532  4.41906906]
 [-1.58438467  2.83081263]
 [ 1.98258281  0.86771314]]
```

```
[[ 0.94973532  4.41906906]
 [-1.37324398  7.75368871]
 [ 1.98258281  0.86771314]
 [-1.58438467  2.83081263]]
'''

#Plotting the results obtained
plt.scatter(X[:,0],X[:,1],c=labels, s = 50, cmap = 'viridis')
#Plotting scatter plot for the sample population density with points colored according
    to the different clusters they have been assigned to.
plt.scatter(centers[:,0],centers[:,1],c='black',s=200,alpha=0.5)
#Plotting the centroids of the clusters obtained from the K-means algorithm
plt.show()

'''
 We can observe that the centroids obtained from the inbuilt K-means module by sklearn
 and the centroids obtained by our implementation of the K-means algorithm are the
 same.
 The centroid coordinates indicate the optimal positions for the Walmart stores. In the
 following output scatter plot, the locations of centroids corresponding to the
 different clusters are indicated.
 '''
```



Scatter plot for customer population density with points belonging to each cluster and centroids indicated

3 K-Means applied to Image Compression

In this section, we'll implement K-means to compress an image. The image that we'll be working on is of size 427 x 640 x 3. Therefore, for each pixel location we would have 3 8-bit integers that specify the red, green and blue intensity values. Our goal is to reduce the number of colors to 16 from about 16 million (3x8 = 24 bit color representation per pixel which accounts to about 16 million different colors) and represent(compress) the photo using those 16 colors only. To pick which colors to use, we'll use K-means algorithm on the image and treat every pixel as a data point. That means reshape the image from $(heightXwidthXchannels)$ to $(height * width)\ X\ channels$. i.e. we would have 427 x 640 = 273280 data points in 3-dimensional space which are the intensity of RGB values corresponding to the 273280 pixels. Doing so will allow us to represent the image using the 16 centroids for each pixel and would significantly reduce the size of the image.

Our basic idea is to use the RGB values of the centroids of the clusters, in which we divide our data-set into, as proxies for the RGB values of all the data-points belonging to that cluster.

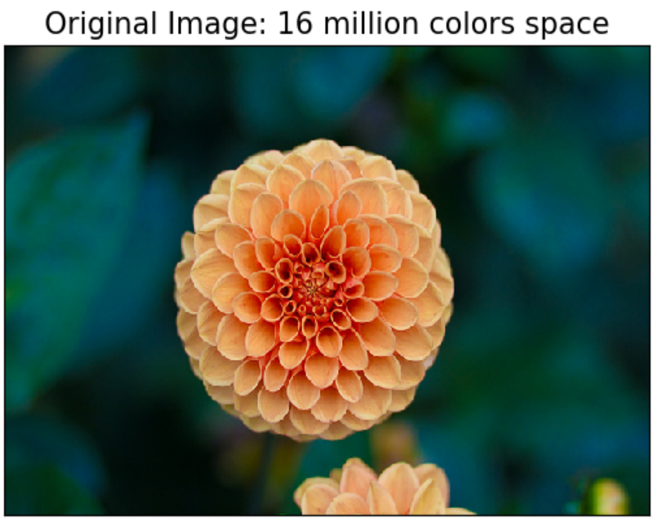
We again implement the basic K-means algorithm defined by the steps in section 1 with a change of the way we initialize the centroids which is explained in the code given below. We also implement a modified version of K-means known as 'Mini batch K-means' inbuilt in the sklearn library.

The MiniBatchKMeans is a variant of the KMeans algorithm which uses mini-batches to reduce the computation time, while still attempting to optimise the same objective function. Mini-batches are subsets of the input data, randomly sampled in each training iteration. These mini-batches drastically reduce the amount of computation required to converge to a local solution. In contrast to other algorithms that reduce the convergence time of k-means, mini-batch k-means produces results that are generally only slightly worse than the standard algorithm.

We then discuss the difference in results of the compressed image by these two approaches.

Following is the python code snippet for the above image compression problem.

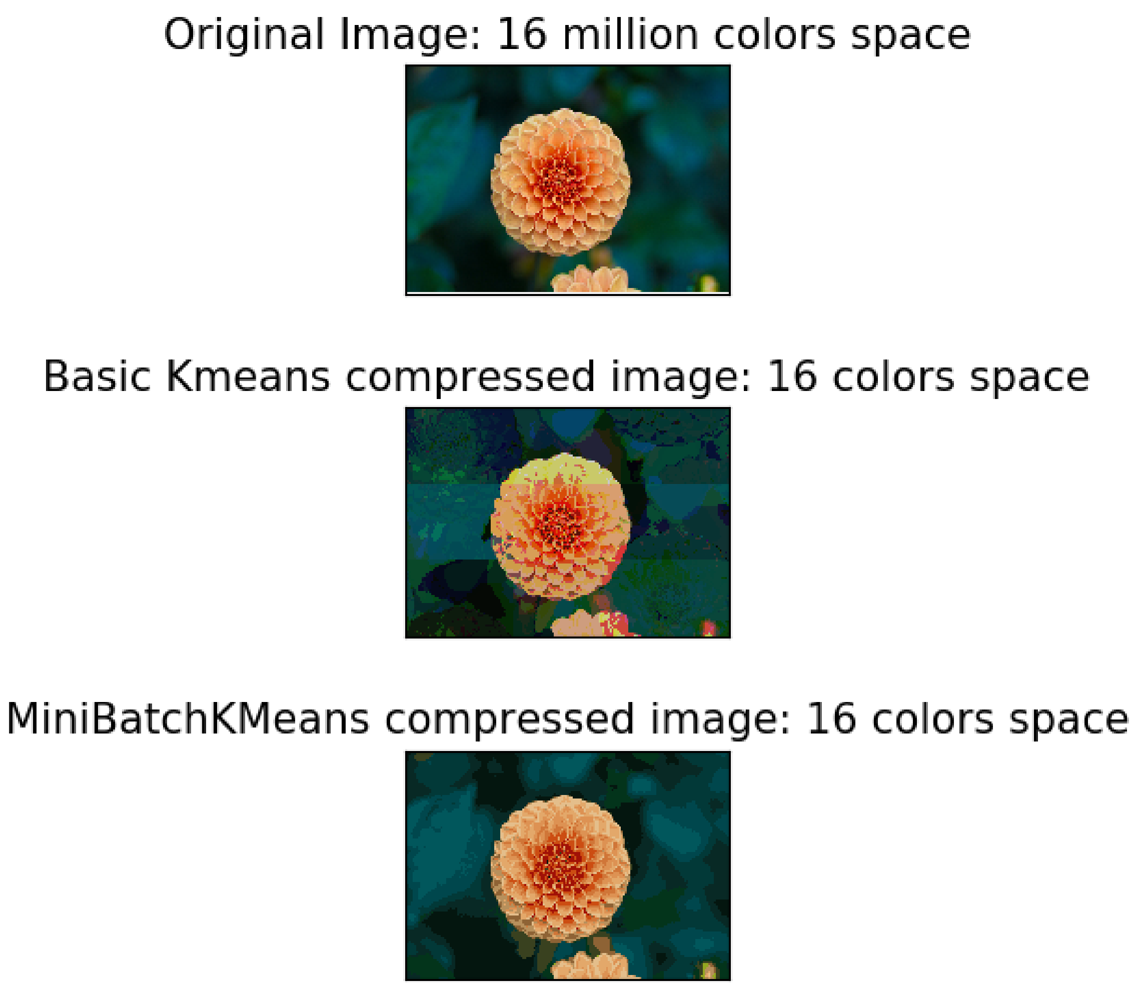
```python
import numpy as np
import random
import matplotlib.pyplot as plt
from sklearn.datasets import load_sample_image
import warnings; warnings.simplefilter('ignore')
from sklearn.cluster import MiniBatchKMeans
'''
Following is the function for K-Means algorithm
y is the data-set of the 273280 pixels, each pixel having 3 values(corresponding to RGB)
in the range (0,1) associated with it. Thus the shape of y is (3,273280).
cn is the the matrix corresponding to the RGB values of the intial K-centroids. The
shape of cn is (3,K). For this case K = 16.
'''
def kmean(y,cn):
    f, r, p, sumt =[], [], [], 0
    #Taking null array p.
    #p[i] will indicate the number of points associated with the i-th centroid.

    p = [0 for j in range(0,k)]

    #c_new will contain the updated centroids.
    c_new=np.zeros((y.shape[0], k))

    #storing nearest centroid for each point in array r
    for i in range(0,y.shape[1]):
        f = [np.square(np.linalg.norm(y[:,i]-cn[:,j])) for j in range(0,k)]
        r.append(np.argmin(f))

    #changing each centroid to centroid of nearest points
    for i in range(0,y.shape[1]):
        c_new[:,r[i]]=c_new[:,r[i]] + y[:,i]
        p[r[i]]=p[r[i]]+1
    for i in range(0,k):
        if(p[i]!=0):
            c_new[:,i]=(c_new[:,i]/p[i])  #centroid is arithmetic mean of all points in
                cluster

    #checking for convergence condition
    for i in range(0,k):
        sumt=sumt+np.square(np.linalg.norm(c_new[:,i]-cn[:,i]))

    print(sumt-eps, '\n') #eps is the user set tolerance for convergence.

    if(sumt < eps):
        print("enter\n")
        print(c_new) #The final centroids
        ans = y
        for i in range(0,y.shape[1]):
            ans[:,i] = c_new[:,r[i]]
        #ans is the dataset of the 272380 pixels with their RGB values replaced by
        #those of the centroid of the cluster they belong to.
        return ans
    else:
        return kmean(y,c_new)

#Reading Input image
x = load_sample_image('flower.jpg')
ax = plt.axes(xticks=[],yticks=[])
ax.imshow(x);
ax.set_title('Original Image: 16 million colors space', size = 16)
plt.show()
```



Original image

```
#Tolerance for convergence
eps = 9*1e-5

#Taking number of clusters as 16 for the given problem as we aim to generate 16 clusters
k = 16
y = x.reshape(x.shape[2],x.shape[0]*x.shape[1])

#Initializing cn
cn = np.zeros((y.shape[0], k))
o = 0

#Dividing By 255 for standard input should be in between 0 to 1
y = np.array(y) / 255.0

#Initializing first Centroid as centroid of first N/k points , second
#Centroid as centroid of second interval of N/K points and so on
for i in range(0,k):
    for j in range(o,o-1+int(y.shape[1]/k)):
        cn[:,i] = cn[:,i] + y[:,j]
    cn[:,i] = cn[:,i]/(int(y.shape[1]/k))
    o=o+int(y.shape[1]/k)

#calling Function for K-means
final = kmean(y,cn)
x_recolored = final.reshape(x.shape)
#x_recolored is the final modified array of shape
#(427 x 640 x 3) which we plot as the output image.
```

```python
'''
Following is the implementation of a modified version of K-means known as Mini batch
K-means. We implement this using an inbuilt module of sklearn library.
'''
y_2 = x/255.0
y_2 = y_2.reshape(x.shape[0]*x.shape[1],x.shape[2])
kmeans = MifunctionniBatchKMeans(16)
kmeans.fit(y_2)
new_colors = kmeans.cluster_centers_[kmeans.predict(y_2)]
china_recolored_= new_colors.reshape(x.shape)
'''
We now plot the three images - Original image, the basic K-means compressed image and
the minibatch K-means compressed image.
'''
fig, ax = plt.subplots(3,1, figsize = (16,6), subplot_kw=dict(xticks=[],yticks=[]))
fig.subplots_adjust(wspace=0.5)
ax[0].imshow(x)
ax[0].set_title('Original Image: 16 million colors space', size = 16)
ax[1].imshow(x_recolored)
ax[1].set_title('Basic Kmeans compressed image: 16 colors space', size = 16)
ax[2].imshow(china_recolored_)
ax[2].set_title('MiniBatchKMeans compressed image: 16 colors space',size=16)
plt.show()
```


Original Image: 16 million colors space
Basic Kmeans compressed image: 16 colors space
MiniBatchKMeans compressed image: 16 colors space

Resulting image after applying K-means

We can see that we reduced the color-space from 16 million to 16 and still get a good proxy for the original image. As can be observed, the naive basic implementation of K-means algorithm suffers as compared to the inbuilt module for Mini Batch K-means. The reason for this is the better initialized centroids, inbuilt functions for faster and better convergence to the optimum of the objective function.

4 Evaluation Methods and Drawbacks

4.1 Evaluation Methods

Contrary to supervised learning where we have the ground truth to evaluate the model's performance, clustering analysis, an unsupervised learning method, doesn't have a solid evaluation metric that we can use to evaluate the outcome of different clustering algorithms. Moreover, since K-means requires k as an input and doesn't learn it from the data, there is no right answer in terms of the numbers of clusters that we should have in any problem. Sometimes domain knowledge and intuition may help but usually that is not the case.

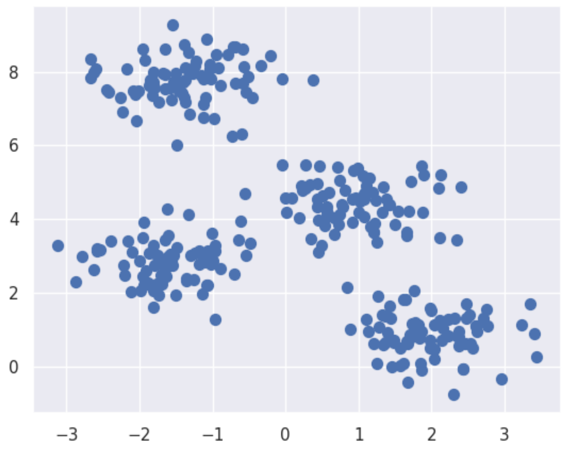
We now discuss a metric that gives some intuition about the choice of k:

Elbow Method

Elbow method gives us an idea on what could be a good k number of clusters, based on the within-cluster sum of squares (WCSS) of distance between the data points and their assigned clusters' centroids. We plot a graph between WCSS and k(i.e. the number of clusters) and pick the optimal value of k at the spot where the curve starts to flatten out and forming an elbow.

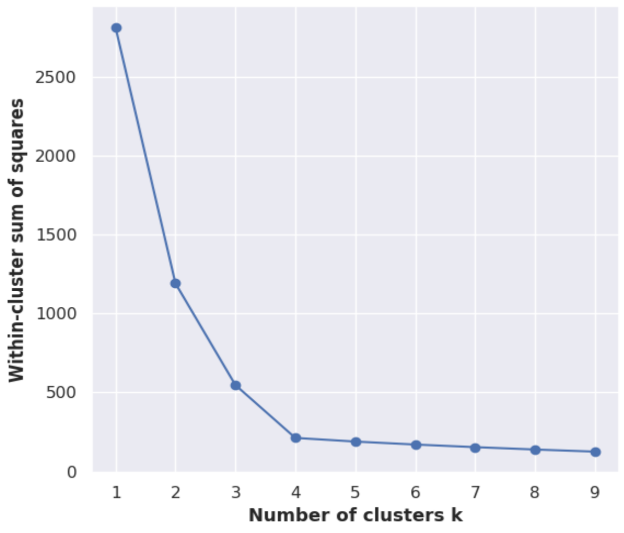
We'll use the data-set we used in section 2 for the problem of optimal store locations and evaluate WCSS for different values of k and see where the curve might form an elbow and flatten out. For this purpose we use the inbuilt K-means module of the sklearn library.

```
import matplotlib . pyplot as plt
import seaborn as sns
import numpy as np
from sklearn.datasets.samples_generator import make_blobs
from sklearn.cluster import KMeans
from sklearn.metrics import pairwise_distances_argmin
sns.set () # Setting plot style
X , y_true = make_blobs ( n_samples =300 , centers =4 , cluster_std = 0.60 ,
    random_state = 0)
'''
Generating the sample data . We generate data distributed in clusters about four centers
such that the std deviation for a set of points belonging to a cluster is 0.6. X is the
sample data while y_true are the true centroids of the generated clusters .
'''

plt.scatter ( X [: ,0] , X [: ,1] , s = 50)
plt.show()
'''
The following is the scatter plot for the customer population density. We know by the
data we generated that the best value for \textit{k} is 4. We now run the K-means
algorithm for different values of k and plot the Elbow curve. Following is the
python code for the same
'''
```



Scatter plot for customer population density

```
# Run the Kmeans algorithm and get the index of data points clusters
wcss = []
list_k = list(range(1, 10))

for k in list_k:
    km = KMeans(n_clusters=k)
    km.fit(X)
    wcss.append(km.inertia_) #km.inertia_ is the wcss value for a given k

# Plot wcss against k
plt.figure(figsize=(6, 6))
plt.plot(list_k, wcss, '-o')
plt.xlabel(r'Number of clusters k', fontweight = 'bold')
plt.ylabel('Within-cluster sum of squares', fontweight = 'bold');
plt.show()
```



Elbow method plot

We can see that the graph flattens out or has an elbow for k = 4. Thus k = 4, as we also know from the data-set we generated, is the optimal value for k. Sometimes it's still hard to figure out a good number of clusters to use because the curve is monotonically decreasing and may not show any elbow or has an obvious point where the curve starts flattening out.

4.2 Drawbacks

Kmeans algorithm is good in capturing structure of the data if clusters have a spherical-like shape. It always try to construct a nice spherical shape around the centroid. That means, the minute the clusters have a complicated geometric shapes, kmeans does a poor job in clustering the data. We'll illustrate three cases where kmeans will not perform well.

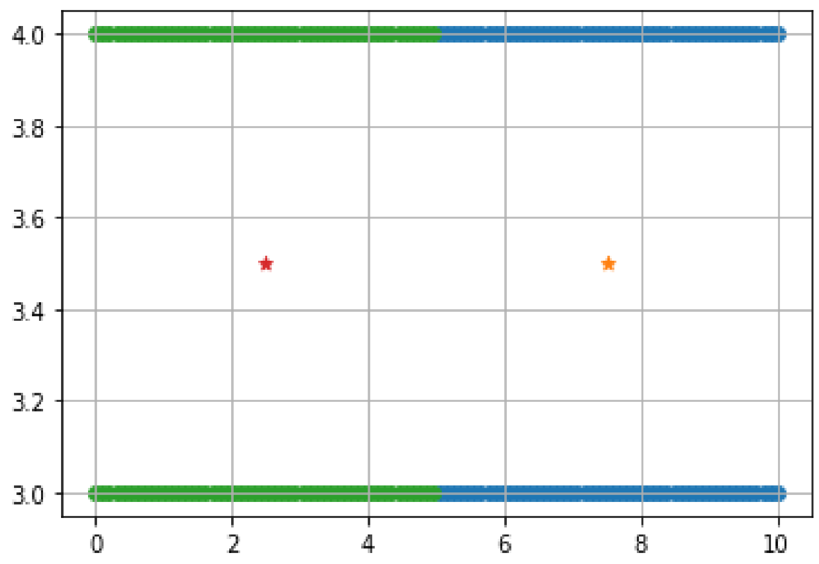
First, kmeans algorithm doesn't let data points that are far-away from each other share the same cluster even though they obviously belong to the same cluster. Below is an example of data points on two different horizontal lines that illustrates how kmeans tries to group half of the data points of each horizontal lines together.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets.samples_generator import (make_blobs,make_circles,make_moons)
from sklearn . cluster import KMeans

X1 = np.linspace(0,10,100)
Y1 = 3*np.ones(100)
X2 = np.linspace(0,10,100)
Y2 = 4*np.ones(100)

P = np.empty([200,2])
P[0:100,0] = X1
P[0:100,1] = Y1
P[100:,0] = X2
P[100:,1] = Y2

no_of_clusters = 2
kmeans = KMeans(n_clusters=no_of_clusters, random_state=0).fit(P)
lab = kmeans.labels_
centroids = kmeans.cluster_centers_


for j in range(no_of_clusters):
  r1 = np.where(lab == j)[0]
  Xc =P[r1]
  plt.scatter(Xc[:,0] , Xc[:,1])
  plt.scatter(centroids[j, 0], centroids[j, 1], marker='*')
plt.grid()
```

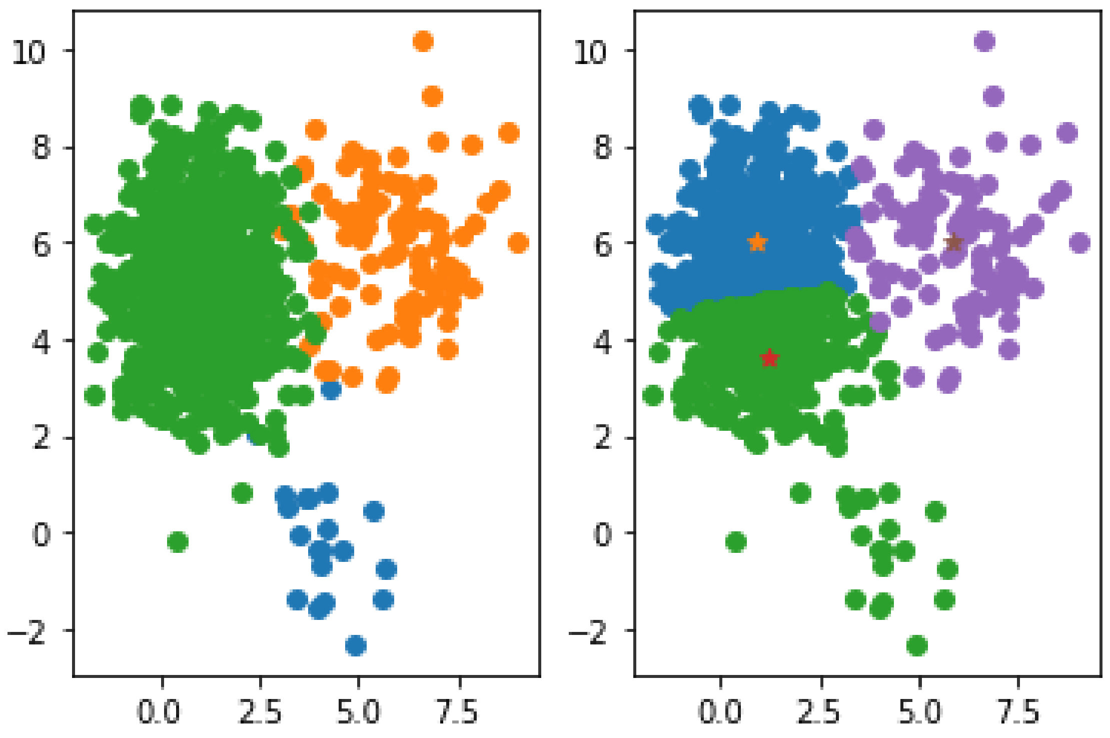
Kmeans considers the point 'B' closer to point 'A' than point 'C' since they have non-spherical shape. Therefore, points 'A' and 'B' will be in the same cluster but point 'C' will be in a different cluster. Note the Single Linkage hierarchical clustering method gets this right because it doesn't separate similar points).

Second, we'll generate data from multivariate normal distributions with different means and standard deviations. So we would have 3 groups of data where each group was generated from different multivariate normal distribution (different mean/standard deviation). One group will have a lot more data points than the other two combined. Next, we'll run kmeans on the data with K=3 and see if it will be able to cluster the data correctly. To make the comparison easier, we are going to plot first the data colored based on the distribution it came from. Then we will plot the same data but now colored based on the clusters they have been assigned to.

```
siz = [20,100,800]
mu = [[4,0],[6,6],[1,5]]
cov = [[[1 , 0] , [0 , 1]],[[2 , 0] , [0 , 2]],[[1 , 0] , [0 , 2]]]

points = np.zeros([np.sum(siz),2])
last = 0

fig, ax = plt.subplots(1, 2)
```



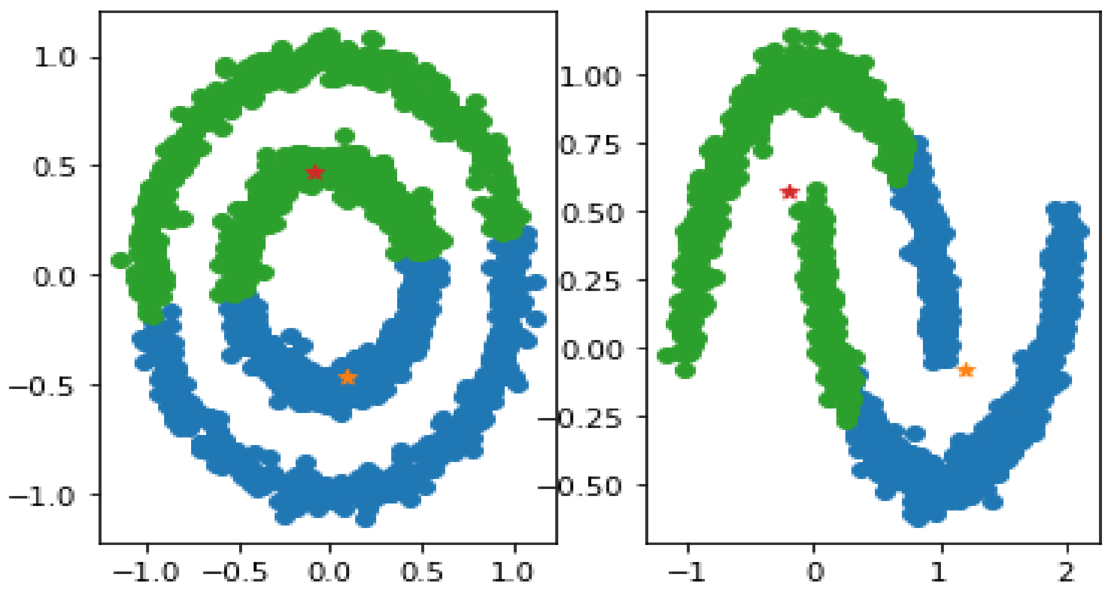
```
for i in range(len(siz)):
  points[last:last+siz[i]] = np.random.multivariate_normal(mu[i],cov[i],size = siz[i])
  ax[0].scatter(points[last:last+siz[i],0],points[last:last+siz[i],1])
  last += siz[i]

no_of_clusters = len(siz)
kmeans = KMeans ( n_clusters =no_of_clusters).fit(points)
lab = kmeans.labels_
centroids = kmeans.cluster_centers_

for j in range(no_of_clusters):
  r1 = np.where(lab == j)[0]
  Xc =points[r1]
  ax[1].scatter(Xc[:,0] , Xc[:,1])
  ax[1].scatter(centroids[j, 0], centroids[j, 1], marker='*')
```



Looks like kmeans couldn't figure out the clusters correctly. Since it tries to minimize the within-cluster variation, it gives more weight to bigger clusters than smaller ones. In other words, data points in smaller clusters may be left away from the centroid in order to focus more on the larger cluster.

Last, we'll generate data that have complicated geometric shapes such as moons and circles within each other and test kmeans on both of the datasets.

```
X1= make_circles(factor=0.5, noise=0.05, n_samples=1500)
X2= make_moons(n_samples=1500, noise=0.05)

fig, ax = plt.subplots(1, 2)
i =0
for X in ([X1,X2]):
  ax[i].scatter(X [0][: , 0] , X [0][: , 1])
  i+=1


fig1, ax = plt.subplots(1, 2)
i=0
a = np.array([1,2,3,4,5,6])
no_of_clusters = 2

for X in ([X1,X2]):

  kmeans = KMeans(n_clusters=no_of_clusters, random_state=0).fit(X[0])
  lab = kmeans.labels_
  centroids = kmeans.cluster_centers_

  for j in range(no_of_clusters):
    r1 = np.where(lab == j)[0]
    Xc1 =X[0][r1]
    ax[i].scatter(Xc1[:,0] , Xc1[:,1])
    ax[i].scatter(centroids[j, 0], centroids[j, 1], marker='*')

  i+=1
```



As expected, kmeans couldn't figure out the correct clusters for both datasets. However, we can help kmeans perfectly cluster these kind of datasets if we use kernel methods. The idea is we transform to higher dimensional representation that make the data linearly separable (the same idea that we use in SVMs).

5 Conclusion

Kmeans clustering is one of the most popular clustering algorithms and usually the first thing practitioners apply when solving clustering tasks to get an idea of the structure of the dataset. The goal of kmeans is to group data points into distinct non-overlapping subgroups. It does a very good job when the clusters have a kind of spherical shapes. However, it suffers as the geometric shapes of clusters deviates from spherical shapes. Moreover, it also doesn't learn the number of clusters from the data and requires it to be pre-defined. To be a good practitioner, it's good to know the assumptions behind algorithms/methods so that you would have a pretty good idea about the strength and weakness of each method. This will help you decide when to use each method and under what circumstances. In this post, we covered both strength, weaknesses, and some evaluation methods related to kmeans.

Below are the main takeaways:

- Since clustering algorithms including kmeans use distance-based measurements to determine the similarity between data points, it's recommended to standardize the data to have a mean of zero and a standard deviation of one since almost always the features in any dataset would have different units of measurements such as age vs income.

- Given kmeans iterative nature and the random initialization of centroids at the start of the algorithm, different initializations may lead to different clusters since kmeans algorithm may stuck in a local optimum and may not converge to global optimum. Therefore, it's recommended to run the algorithm using different initializations of centroids and pick the results of the run that that yielded the lower sum of squared distance.

- Elbow method in selecting number of clusters doesn't usually work because the error function is monotonically decreasing for all ks.

- Kmeans gives more weight to the bigger clusters.

- Kmeans assumes spherical shapes of clusters (with radius equal to the distance between the centroid and the furthest data point) and doesn't work well when clusters are in different shapes such as elliptical clusters.
- Kmeans may still cluster the data even if it can't be clustered such as data that comes from uniform distributions.

6 GitHub Repository

All above codes and figures are uploaded in our GitHub Repository
Link for the repository :

```
https://github.com/Krati012/DSA-Project
```