CS306: DATA ANALYSIS AND VISUALIZATION

LAB 8: K-Means Clustering

STUDENT ID: 201801407

NAME: PRATVI SHAH

In [114]:
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import random
%matplotlib inline
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler,RobustScaler
import seaborn as sns
from sklearn.metrics import mean_squared_error as mse
from scipy.stats import mode
from sklearn.cluster import KMeans
from sklearn.datasets import load_digits
import warnings
warnings.filterwarnings('ignore')
```

In [ ]:
```
def find_dunn_index(c,d,df):
    numerator=np.inf
    denominator=-1*np.inf

    #minimize the intercluster distance
    for i in range(len(c)):
        for k in range(len(c)):
          if k!=i:
            ans=0
            for j in range(len(c[0])):
                ans+=(c[i][j]-c[k][j])**2
            numerator=min(numerator,ans)

    #maximize the
    return ans
```

In [4]:
```
def find_centroids(k,n):
    random.seed(5)
    idx=[]
    for i in range(k):
        idx.append(random.randint(0,n-1))
    return idx
```

In [5]:
```
def find_clusters(df,c,k,n):
    dist=[[] for _ in range(k)]
    for i in range(n):
        temp=[]
        for j in range(k):
            dist_temp= np.sum((df[i]-c[j])**2)
            temp.append(dist_temp)
        location=temp.index(min(temp))
        dist[location].append(i)
    return dist
```

In [6]:
```
def find_new_centroids(df,dist,c,k,m):
    for i in range(k):
    temp_mean=[0 for _ in range(m)]
    for j in range(len(dist[i])):
        id=dist[i][j]
        temp=0
        for kk in range(m):
            temp_mean[kk]+=df[id][kk]/len(dist[i])
    for j in range(m):
        c[i][j]=temp_mean[j]
    return c
```

In [26]:
```
def plot_2D(dist,df,c,k):
    plt.figure(figsize=[10,6])
    distx=[[] for i in range(k)]
    disty=[[] for i in range(k)]
    for i in range(k):
        for j in range(len(dist[i])):
            id=dist[i][j]
            distx[i].append(df[id][0])
            disty[i].append(df[id][1])
    cx=[];cy=[]
    for i in range(k):
        cx.append(c[i][0])
        cy.append(c[i][1])
        sns.scatterplot(distx[i],disty[i],markers='o',s=200)#,c=next(clr))
    sns.scatterplot(cx,cy,color='.2',marker='*',s=400)
    plt.grid()
    plt.xlabel('X',fontsize=15)
    plt.ylabel('Y',fontsize=15)
```

In [108]:
```
def k_means(df,k,plot):
    # clr= iter(['b', 'r','c', 'm', 'g', 'tab:orange', 'tab:brown', 'tab:pink'])
    n=df.shape[0]
    m=df.shape[1]
    #indices of centroid
    idx=find_centroids(k,n)

    #to store the centroids kxm
    c=[[] for _ in range(k)]
    cprev=[[] for _ in range(k)]

    for i in range(k):
        for j in range(m):
            c[i].append(df[idx[i]][j])
            cprev[i].append(df[idx[i]][j])

    #to get the points in a given cluster kxdim
    #just store the indices in the original array
    dist=[[] for _ in range(k)]
    itr=0
    flag=False
    while flag==False and itr!=1000:

        #k clusters with elements in the cluster
        dist=find_clusters(df,c,k,n)

        #store previous centroids
        cprev=np.copy(c)

        #finding the new centroids
        c=find_new_centroids(df,dist,c,k,m)

        count=0
        for i in range(k):
            for j in range(len(c[0])):
                if c[i][j]==cprev[i][j]:
                    count+=1
        if count==k*len(c[0]):
            break
        itr+=1


    if plot==True:
        plot_2D(dist,df,c,k)

    return c,dist
```

In [9]:
```
def find_inertia(c,d,df):
    ans=0
    for i in range(len(c)):
        indices=d[i]
        for kk in range(len(indices)):
            for j in range(len(c[0])):
                ans+=(c[i][j]-df[indices[kk]][j])**2
    return ans
```

Q1

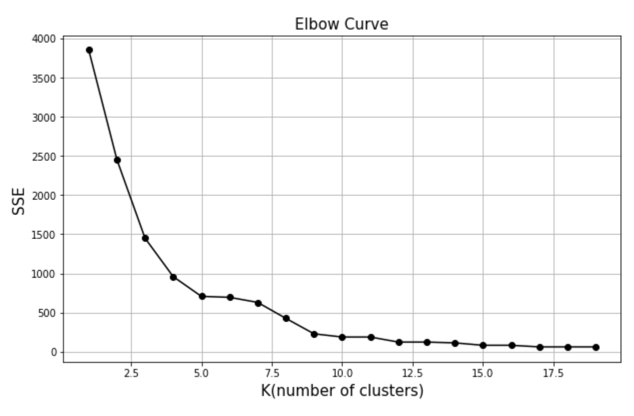
In [28]:
```
x = [10,14,8,12,15,12,15,17,5,18,22,25,35,21,39,27,25,33,30,36]
y = [8,25,10,30,35,12,14,15,22,32,2,21,35,7,15,29,33,23,17,11]
```

In [29]:
```
df=[]
df.append(x)
df.append(y)
df=np.array(df).transpose()
scaler = StandardScaler()
df_scaled = scaler.fit_transform(df)
```

In [30]:
```
SSE_q1=[]
n=20
for i in range(1,n):
    c_q1,dist_q1=k_means(df,i,False)
    SSE_q1.append(find_inertia(c_q1,dist_q1,df))
```

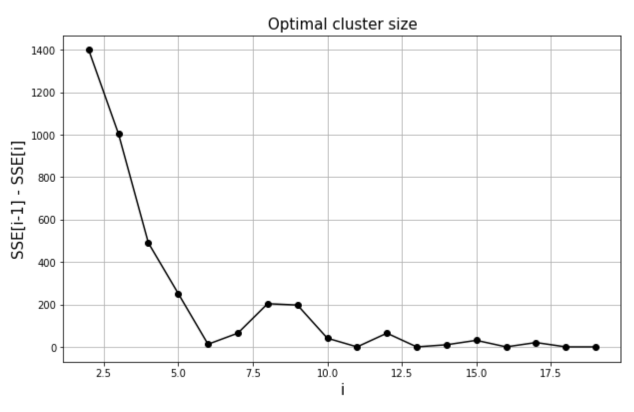
In [31]:
```
x_axis=np.arange(1,20)
plt.figure(figsize=[10,6])
plt.plot(x_axis,SSE_q1,'k-o')
plt.grid()
plt.ylabel('SSE',fontsize=15)
plt.xlabel('K(number of clusters)',fontsize=15)
plt.title('Elbow Curve',fontsize=15)
```

Out[31]: `Text(0.5, 1.0, 'Elbow Curve')`



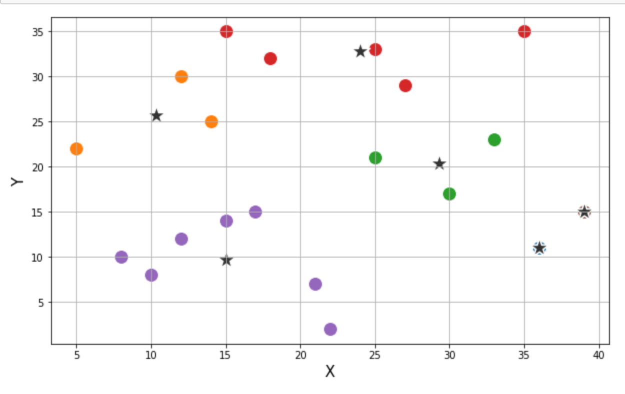
In [32]:
```
diff_x=np.arange(2,20)
diff_sse_q1 = [SSE_q1[i-1]-SSE_q1[i] for i in range(1,len(SSE_q1))]
plt.figure(figsize=[10,6])
plt.plot(diff_x,diff_sse_q1,'k-o')
plt.grid()
plt.ylabel('SSE[i-1] - SSE[i]',fontsize=15)
plt.xlabel('i',fontsize=15)
plt.title('Optimal cluster size',fontsize=15)
```

Out[32]: `Text(0.5, 1.0, 'Optimal cluster size')`



We can see that at i=6 we get almost zero difference and after that we get such a case for i>10 for which the computation time increases significantly hence, K=6 is a good choice for number of clusters.

In [33]:
```
c_q1,dist_q1=k_means(df,6,True)
```



Q2

In [17]:
```
data_digits=load_digits().data
data_labels=load_digits().target
```

In [82]:
```
data_digits.shape
```

Out[82]: `(1797, 64)`

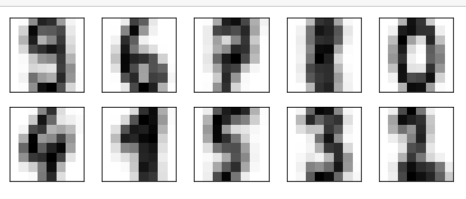
In [18]:
```
scaler = StandardScaler()
data_scaled = scaler.fit_transform(data_digits)
```

In built function

In [19]:
```
kmeans = KMeans(n_clusters = 10, init='k-means++')
kmeans.fit(data_scaled)
SSE_kmeans=kmeans.inertia_
print('SSE using inbuilt function:',SSE_kmeans)
```

```
SSE using inbuilt function: 69426.25372371473
```

In [76]:
```
fig, ax = plt.subplots(2, 5, figsize=(8, 3))
centers = kmeans.cluster_centers_.reshape(10, 8, 8)
for axi, center in zip(ax.flat, centers):
    axi.set(xticks=[], yticks=[])
    axi.imshow(center, interpolation='nearest', cmap=plt.cm.binary)
```

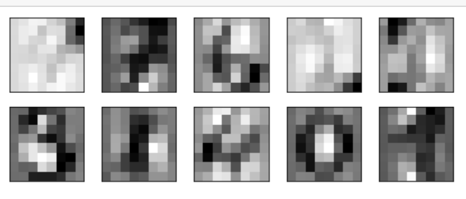


My function

In [109]:
```
c_q2,dist_q2=k_means(data_scaled,10,False)
SSE_q2_myfunc = find_inertia(c_q2,dist_q2,data_scaled)
print('SSE using my function(k_means):',SSE_q2_myfunc)
```

```
SSE using my function(k_means): 69476.70340699745
```

In [110]:
```
fig, ax = plt.subplots(2, 5, figsize=(8, 3))
centers = np.array(c_q2).reshape(10, 8, 8)
for axi, center in zip(ax.flat, centers):
    axi.set(xticks=[], yticks=[])
    axi.imshow(center, interpolation='nearest', cmap=plt.cm.binary)
```



We can see that above 10 centers/centroids identified can be associated to actual digits/labels as:

8/3 7 6 2 9

8/3 5 4 0 1

Although they are not very clear the SSE of the inbuilt and my function is comparable.

To visualize the clustering we can use PCA for 2 components and then apply k means clustering on the obtained data

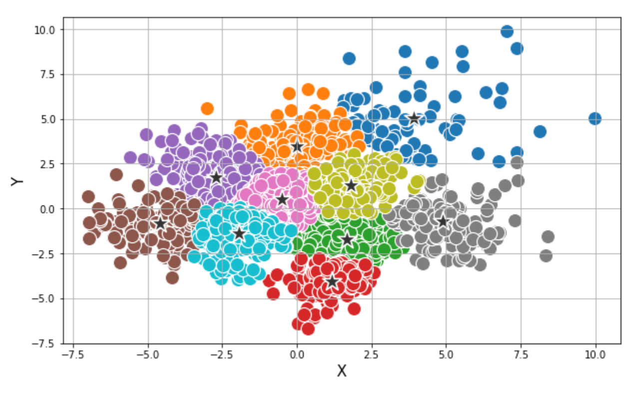
In [111]:
```
estimator = PCA(n_components=2)
x_pca = estimator.fit_transform(data_scaled)
```

In [112]:
```
kmeans = KMeans(n_clusters = 10, init='k-means++')
kmeans.fit(x_pca)
SSE_kmeans_pca=kmeans.inertia_
print('SSE using inbuilt function:',SSE_kmeans_pca)
```

```
SSE using inbuilt function: 2815.3317395769977
```

In [113]:
```
c_pca,dist_pca=k_means(x_pca,10,True)
SSE_pca_myfunc = find_inertia(c_pca,dist_pca,x_pca)
print('SSE using my function(k_means):',SSE_pca_myfunc)
```

```
SSE using my function(k_means): 2818.2310148725037
```



For the above plot X, Y represents the two principal components.

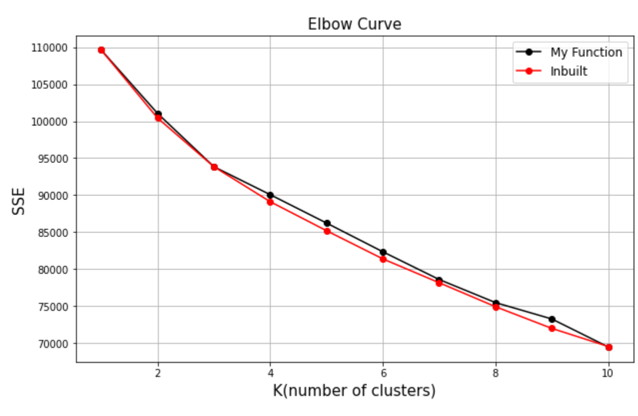
To verify the correctness of my function elbow curve is constructed for 1-10 clusters to show the similarity in SSE between the inbuilt and my function

In [54]:
```
SSE_inbuilt = []
for cluster in range(1,11):
    kmeans = KMeans(n_clusters = cluster, init='k-means++')
    kmeans.fit(data_scaled)
    SSE_inbuilt.append(kmeans.inertia_)
```

In [116]:
```
SSE_q2=[]
n=11
for i in range(1,n):
  c_q2,dist_q2=k_means(data_scaled,i,False)
  SSE_q2.append(find_inertia(c_q2,dist_q2,data_scaled))
```

In [117]:
```
x_axis=[i for i in range(1,11)]
plt.figure(figsize=[10,6])
plt.plot(x_axis,SSE_q2,'k-o',label='My Function')
plt.plot(x_axis,SSE_inbuilt,'r-o',label='Inbuilt')
plt.grid()
plt.ylabel('SSE',fontsize=15)
plt.xlabel('K(number of clusters)',fontsize=15)
plt.title('Elbow Curve',fontsize=15)
plt.legend(fontsize=12)
```

Out[117]: `<matplotlib.legend.Legend at 0x7fdf074c6190>`



SSE shows that the k_means() function and the inbuilt function are comparable as there is very little difference between the sum of square errors of the two implementation.

In [ ]: