Deng_Yehong_HW7

March 15, 2020

[29]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.manifold import TSNE
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score
```

1. (5 points) Imitate the k-means random initialization part of the algorithm by assigning each observation to a cluster at random.

[30]:
```
input_1 = np.array([5,8,7,8,3,4,2,3,4,5])
input_2 = np.array([8,6,5,4,3,2,2,8,9,8])
```

[31]:
```
np.random.seed(456123)
df =pd.DataFrame({'input_1': input_1, 'input_2': input_2})
get_labels = np.random.choice(3, 10, replace = True)
df['k'] = get_labels
df
```

[31]:
```
   input_1  input_2  k
0        5        8  1
1        8        6  0
2        7        5  2
3        8        4  2
4        3        3  0
5        4        2  1
6        2        2  0
7        3        8  0
8        4        9  2
9        5        8  1
```

2. (5 points) Compute the cluster centroid and update cluster assignments for each observation iteratively based on spatial similarity.

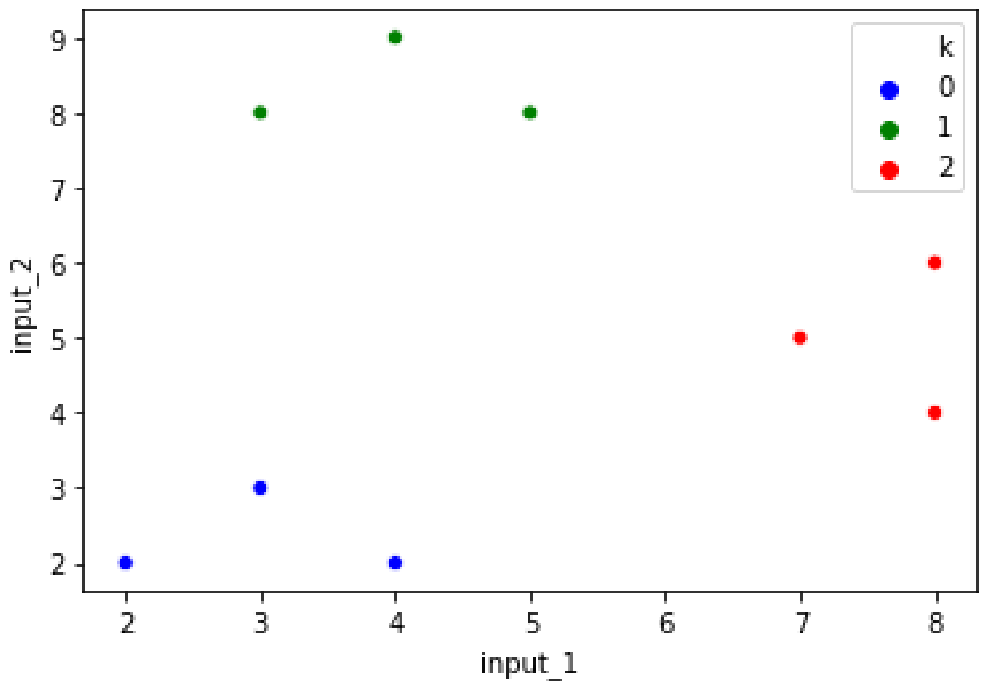
[32]:
```
df_k3 = df.copy()
for i in range(100):
    centroids = {}
    for j in range(3):
        cen_input1 = df_k3[df_k3['k'] == j]['input_1'].mean()
        cen_input2 = df_k3[df_k3['k'] == j]['input_2'].mean()
        centroids[j] = (cen_input1, cen_input2)
    updated_k = []
    for index, row in df_k3.iterrows():
        min_distance = 100
        for key, value in centroids.items():
            distance = ((row['input_1'] - value[0]) ** 2 + (row['input_2'] -␣
 ↪value[1])**2)**0.5
            if distance < min_distance:
                min_distance = distance
                update_k = key
        updated_k.append(update_k)
    df_k3['k'] = updated_k
df_k3
```

[32]:
```
   input_1  input_2  k
0        5        8  1
1        8        6  2
2        7        5  2
3        8        4  2
4        3        3  0
5        4        2  0
6        2        2  0
7        3        8  1
8        4        9  1
9        5        8  1
```

3. (5 points) Present a visual description of the final, converged (stopped) cluster assignments.

[33]:
```
sns.scatterplot(x = 'input_1', y = 'input_2', hue = 'k', palette = ['blue',␣
 ↪'green','red'], data = df_k3)
```

[33]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1f2ba6ca648>
```



4. (5 points) Now, repeat the process, but this time initialize at k = 2 and present a final cluster assignment visually next to the previous search at k = 3.

[35]:
```
df2 = pd.DataFrame({'input_1': input_1, 'input_2': input_2})
get_labels2 = np.random.choice(2, 10, replace = True)
df2['k'] = get_labels2
df2
```

[35]:
```
   input_1  input_2  k
0        5        8  0
1        8        6  0
2        7        5  1
3        8        4  0
4        3        3  1
5        4        2  1
6        2        2  1
7        3        8  0
8        4        9  1
9        5        8  0
```

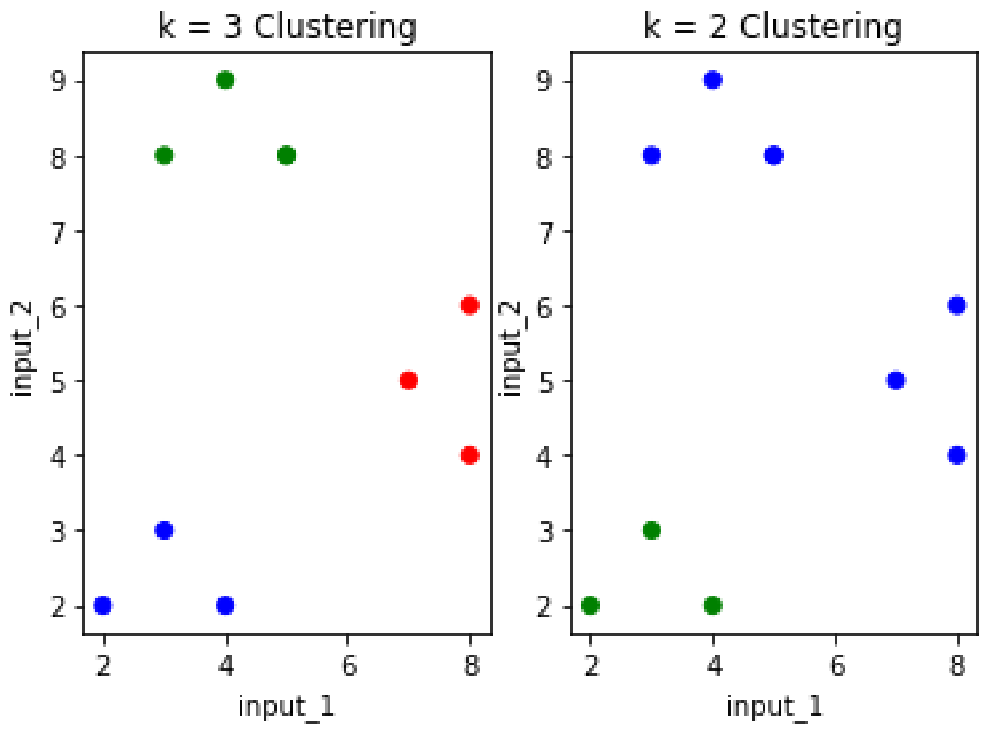
[36]:
```
df_k2 = df2.copy()
for i in range(100):
    centroids = {}
    for j in range(2):
        cen_input1 = df_k2[df_k2['k'] == j]['input_1'].mean()
        cen_input2 = df_k2[df_k2['k'] == j]['input_2'].mean()
        centroids[j] = (cen_input1, cen_input2)
    updated_k = []
    for index, row in df_k2.iterrows():
        min_distance = 100
        for key, value in centroids.items():
            distance = ((row['input_1'] - value[0]) ** 2 + (row['input_2'] -␣
 ↪value[1])**2)**0.5
            if distance < min_distance:
                min_distance = distance
                update_k = key
        updated_k.append(update_k)
    df_k2['k'] = updated_k
df_k2
```

[36]:
```
   input_1  input_2  k
0        5        8  0
1        8        6  0
2        7        5  0
3        8        4  0
4        3        3  1
5        4        2  1
6        2        2  1
7        3        8  0
8        4        9  0
9        5        8  0
```

[38]:
```
#subplots referenced from https://matplotlib.org/api/_as_gen/matplotlib.pyplot.
 ↪subplots.html
fig, axs = plt.subplots(1,2)
color = np.array(['blue', 'green','red'])
axs[0].scatter(df_k3['input_1'], df_k3['input_2'], c = color[df_k3['k']])
axs[0].set_xlabel('input_1')
axs[0].set_ylabel('input_2')
axs[0].set_title('k = 3 Clustering')

axs[1].scatter(df_k2['input_1'], df_k2['input_2'], c = color[df_k2['k']])
axs[1].set_xlabel('input_1')
axs[1].set_ylabel('input_2')
axs[1].set_title('k = 2 Clustering')
```

[38]:
```
Text(0.5, 1.0, 'k = 2 Clustering')
```



5. (10 points) Did your initial hunch of 3 clusters pan out, or would other values of k, like 2, fit these data better? Why or why not?

According to the k=3 clusting plot, the red cluster located in the middle between the green and blue ones. The distance between red and green and that between red and blue should be very similar. Hence, it make sense that the graph should have three clusters. Nonetheless, in the the k = 2 clustering graph, the original green and red clusters are now clustered together. As a result, the distance among points within each cluster in the right graph is higher. Therefore, k = 3 performs better than k = 2.

6. (15 points) Perform PCA on the dataset and plot the observations on the first and second principal components. Describe your results, e.g.,

What variables appear strongly correlated on the first principal component?

What about the second principal component?

[49]:
```
df_wiki = pd.read_csv('wiki.csv')
predictors = df_wiki.columns
df_wiki = StandardScaler().fit_transform(df_wiki)
pca = PCA(n_components = 2, random_state = 456123)
df_pca = pd.DataFrame(data = pca.fit(df_wiki).components_.T, index =␣
 ↪predictors, columns = ['PC1', 'PC2'])
```

[50]:
```
df_pca.nlargest(5, 'PC1')
```

```
[50]:            PC1       PC2
      bi2   0.230924  0.083430
      bi1   0.226193  0.056372
      use3  0.218809  0.155158
      use4  0.214558  0.160868
      pu3   0.210863  0.028776
```

```
[51]: df_pca.nsmallest(5,'PC2')
```

```
[51]:            PC1       PC2
      peu1  0.061228 -0.271740
      inc1  0.104667 -0.245440
      sa3   0.120376 -0.242315
      sa1   0.121658 -0.229925
      enj2  0.131110 -0.227596
```

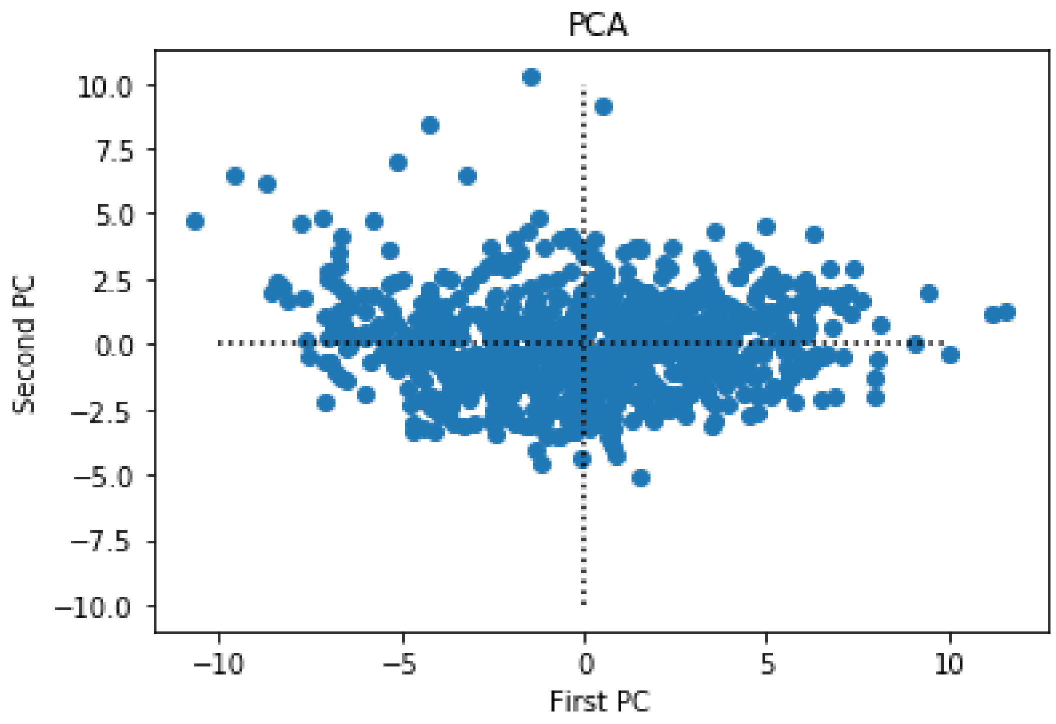
```
[52]: pca = PCA(n_components = 2, random_state = 456123)
      PC = pca.fit_transform(df_wiki)
      df_PC = pd.DataFrame(data = PC, columns = ['PC1', 'PC2'])
      df_PC
```

```
[52]:           PC1       PC2
      0   -0.150216 -1.983507
      1   -3.314020 -0.791893
      2   -4.682484 -0.312228
      3    1.774200  1.986370
      4    7.254695  2.013757
      ..        …         …
      795  0.227143  1.474793
      796  4.434784 -0.931543
      797  1.449455 -0.170504
      798 -2.888282  2.721490
      799 -7.000656  2.805481

      [800 rows x 2 columns]
```

```
[56]: plt.figure()
      plt.scatter(df_PC['PC1'], df_PC['PC2'])
      plt.xlabel('First PC')
      plt.ylabel('Second PC')
      plt.title('PCA')
      plt.hlines(0, -10, 10, linestyles = 'dotted', colors = 'black')
      plt.vlines(0, -10, 10, linestyles = 'dotted', colors = 'black')
```

```
[56]: <matplotlib.collections.LineCollection at 0x1f2bf554e88>
```



According to the tables above, the variables'bi1', 'bi2', 'use3', 'use4', and pu3 appear most strongly correlated on the first pricipal component. 'peu1', 'inc1', 'sa3', 'sa1', and 'enj2' are the most strongly correlated on the second component.

7. (5 points) Calculate the proportion of variance explained (PVE) and the cumulative PVE for all the principal components. Approximately how much of the variance is explained by the first two principal components?

[57]:
```
percentage1 = 100 * pca.explained_variance_ratio_[0]
percentage2 = 100 * pca.explained_variance_ratio_[1]
sum_per = 100 * (sum(pca.explained_variance_ratio_))
print(percentage1, 'percent of the variance is explained by the first principal␣
 ↪component')
print(percentage2, 'percent of the variance is explained by the second␣
 ↪principal component')
print(sum_per, 'percent of the variance is explained by the two components␣
 ↪together')
```

```
22.810627785663385 percent of the variance is explained by the first principal
component
6.37247429597453 percent of the variance is explained by the second principal
component
29.183102081637912 percent of the variance is explained by the two components
together
```

8. (10 points) Perform *t*-SNE on the dataset and plot the observations on the first and second dimensions. Describe your results.

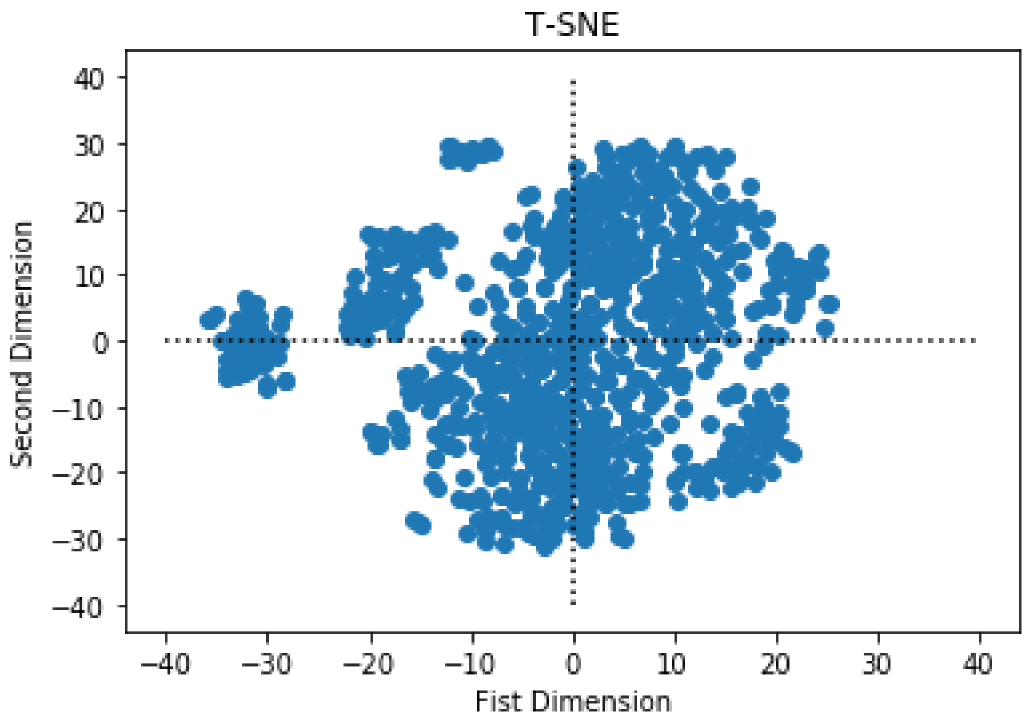
```
[58]: tsne = TSNE(n_components = 2, random_state = 456123)
      df_tsne = pd.DataFrame(data = tsne.fit_transform(df_wiki), columns = ['dim1',␣
       ↪'dim2'])
      df_tsne
```

```
[58]:           dim1       dim2
      0   -19.583126  11.007133
      1   -15.840316  14.153754
      2   -32.024673   5.357511
      3   -33.440903  -4.613192
      4   -10.368248 -28.905149
      ..         …          …
      795  17.824806  -2.804761
      796   6.833607 -19.981953
      797   5.579870 -12.514900
      798   1.697928   5.827499
      799  12.196272  28.150850

      [800 rows x 2 columns]
```

```
[59]: plt.figure()
      plt.hlines(0, -40, 40, linestyles = 'dotted', colors = 'black')
      plt.vlines(0, -40, 40, linestyles = 'dotted', colors = 'black')
      plt.scatter(df_tsne['dim1'], df_tsne['dim2'])
      plt.xlabel('Fist Dimension')
      plt.ylabel('Second Dimension')
      plt.title('T-SNE')
```

```
[59]: Text(0.5, 1.0, 'T-SNE')
```

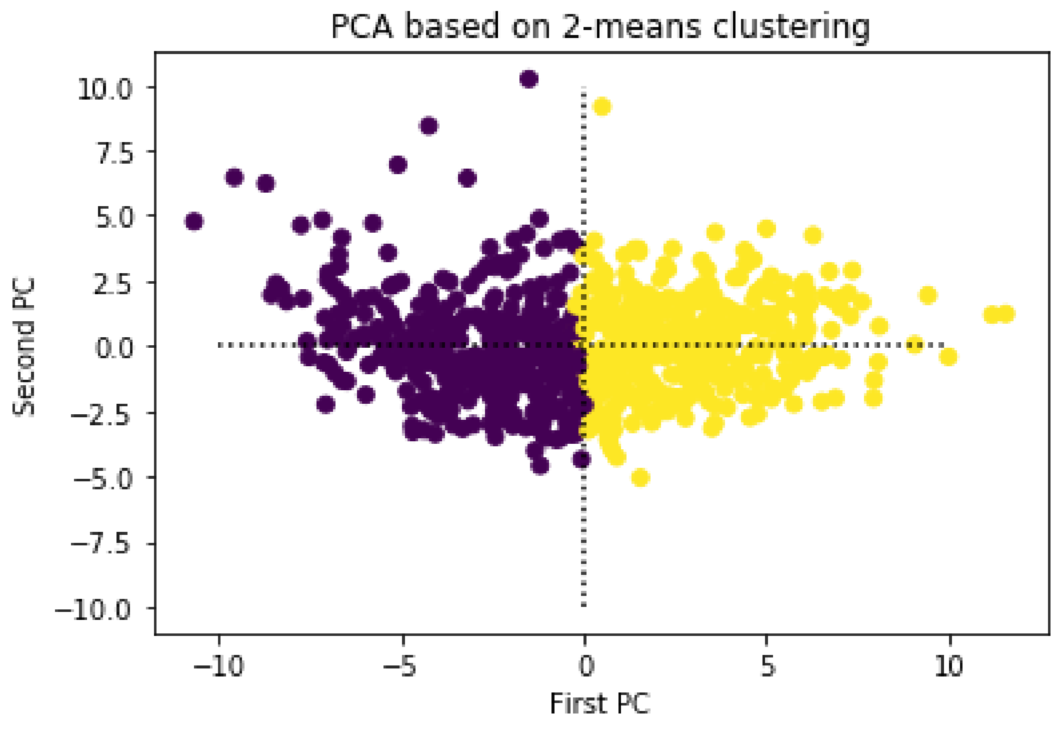


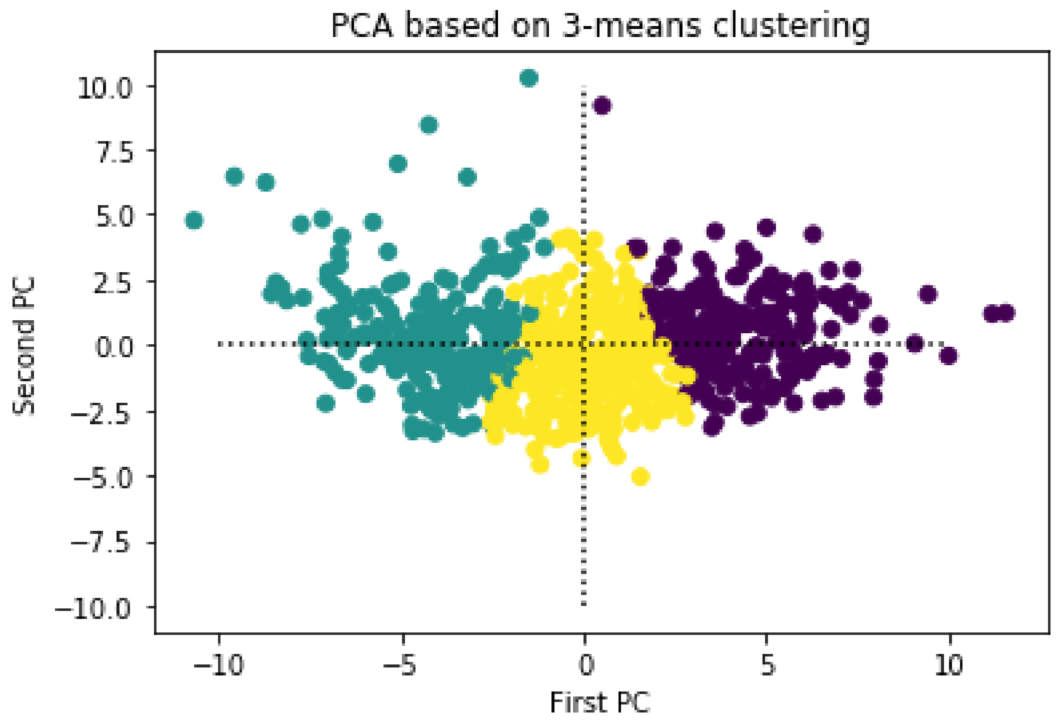
According to the plot, we can see a large cluster exists in the middle and several small ones surround.The large cluster implies that there are a majority of features that are similar to one another.

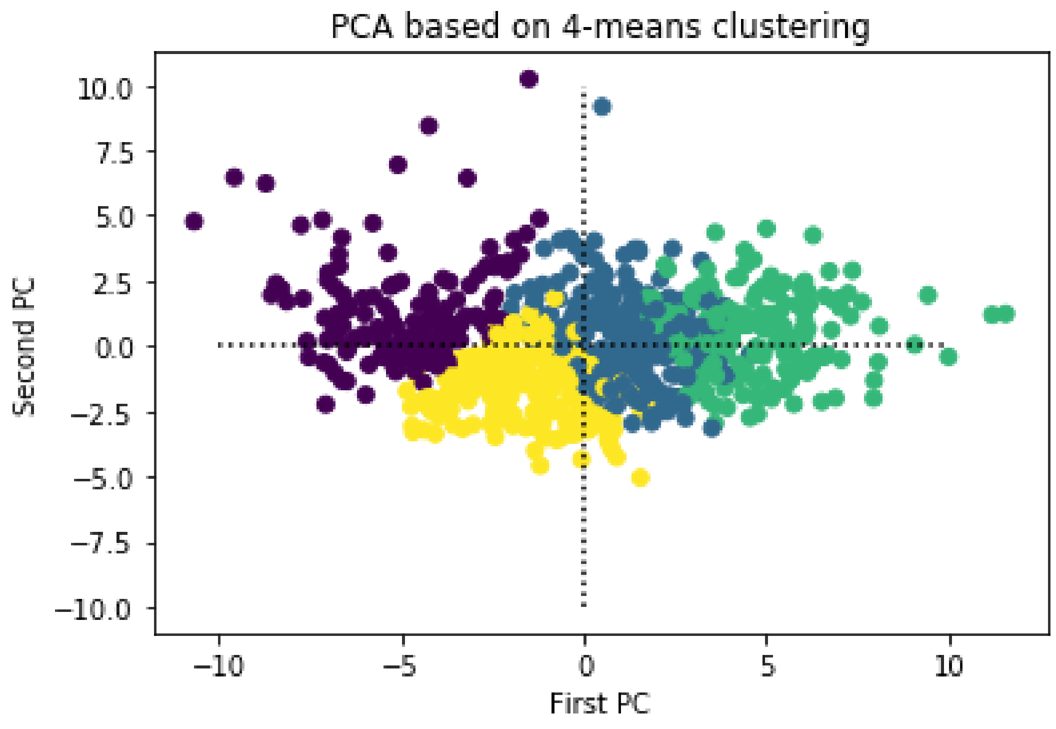
9. (15 points) Perform k-means clustering with $k = 2, 3, 4$. Be sure to scale each feature (i.e.,mean zero and standard deviation one). Plot the observations on the first and second principal components from PCA and color-code each observation based on their cluster membership. Discuss your results.

[60]:
```
def plotting_kmean(k):
    k_mean = KMeans(n_clusters = k, random_state = 456123).fit(df_wiki)
    plt.figure()
    plt.scatter(df_PC['PC1'], df_PC['PC2'], c = k_mean.labels_)
    plt.xlabel('First PC')
    plt.ylabel('Second PC')
    plt.title(f'PCA based on {k}-means clustering')
    plt.hlines(0, -10, 10, linestyles = 'dotted', colors = 'black')
    plt.vlines(0, -10, 10, linestyles = 'dotted', colors = 'black')
```

[61]:
```
k = [2,3,4]
for i in k:
    plotting_kmean(i)
```


PCA based on 2-means clustering
Second PC
First PC
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
−10
−5
0
5
10


PCA based on 3-means clustering
Second PC
First PC
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
−10
−5
0
5
10

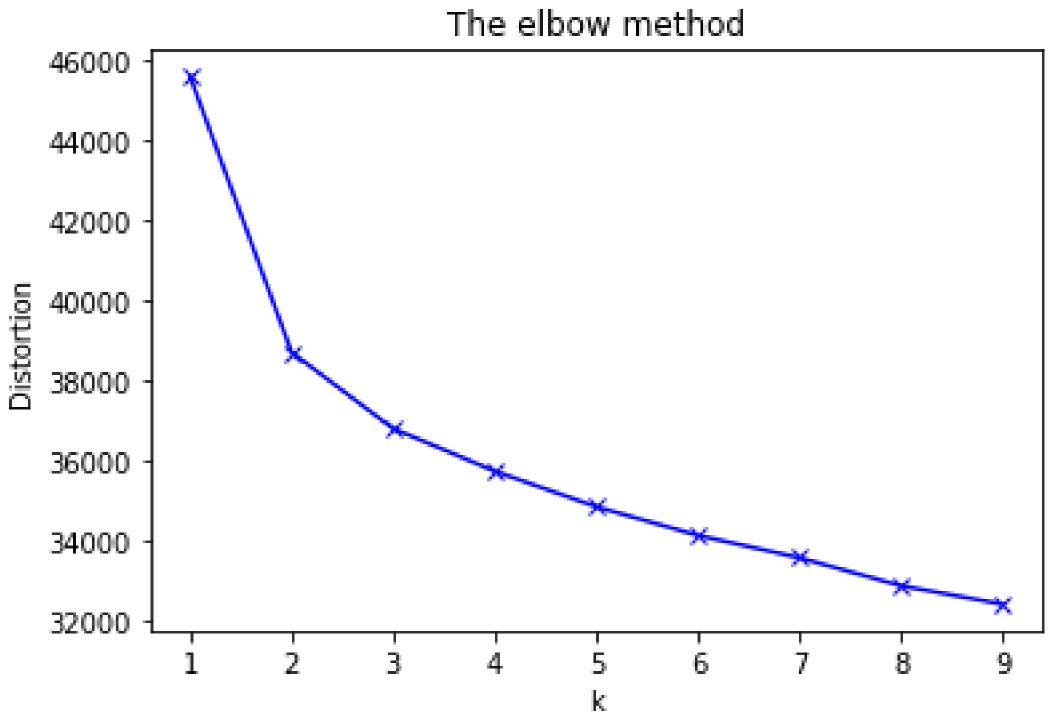


According to the graphs above, we can see that 2-means clusting performs the best as there are not many overlappings between the clusters. The 3-means clustering also perform well that we can observe clear boundaries for the clusters. For these two plots, the first PC has influential effects as the boundaries are quite vertical for 2-means and for 3-means. The boundaries for 3-means are little bit tilt, which suggest the second pc also has influence on clustering. The overlapping in the 4-means that the current two pcs are not enough to get four clusters.

10. (10 points) Use the elbow method, average silhouette, and/or gap statistic to identify the optimal number of clusters based on k-means clustering with scaled features.

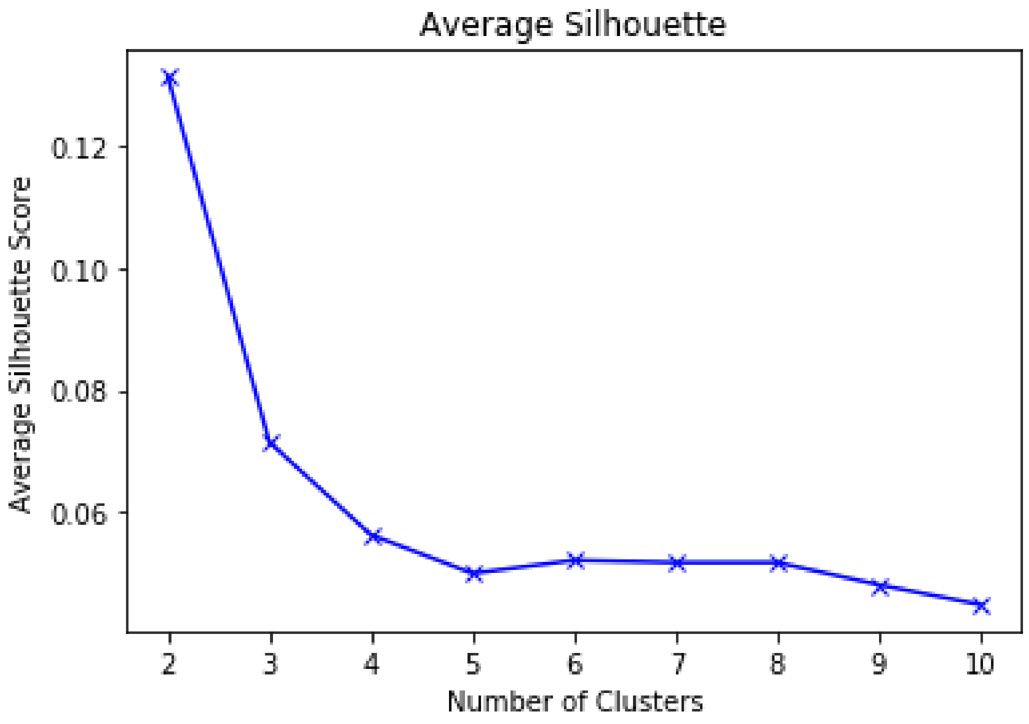
[62]:
```
#referenced from https://predictivehacks.com/
 ↪k-means-elbow-method-code-for-python/
distortions = []
k = range(1,10)
for i in k:
    kmeanModel = KMeans(n_clusters = i)
    kmeanModel.fit(df_wiki)
    distortions.append(kmeanModel.inertia_)
plt.figure()
plt.plot(k, distortions, 'bx-')
plt.xlabel('k')
plt.ylabel('Distortion')
plt.title('The elbow method')
```

```
[62]: Text(0.5, 1.0, 'The elbow method')
```



```
[64]: #referenced from https://scikit-learn.org/stable/modules/generated/sklearn.
       ↪metrics.silhouette_score.html
      sil = []
      for i in range(2,11):
          kmeanModel = KMeans(n_clusters = i)
          sil.append(silhouette_score(df_wiki,kmeanModel.fit_predict(df_wiki)))
      plt.figure()
      plt.plot(range(2,11), sil, 'bx-')
      plt.ylabel('Average Silhouette Score')
      plt.xlabel('Number of Clusters')
      plt.title('Average Silhouette')
```

```
[64]: Text(0.5, 1.0, 'Average Silhouette')
```



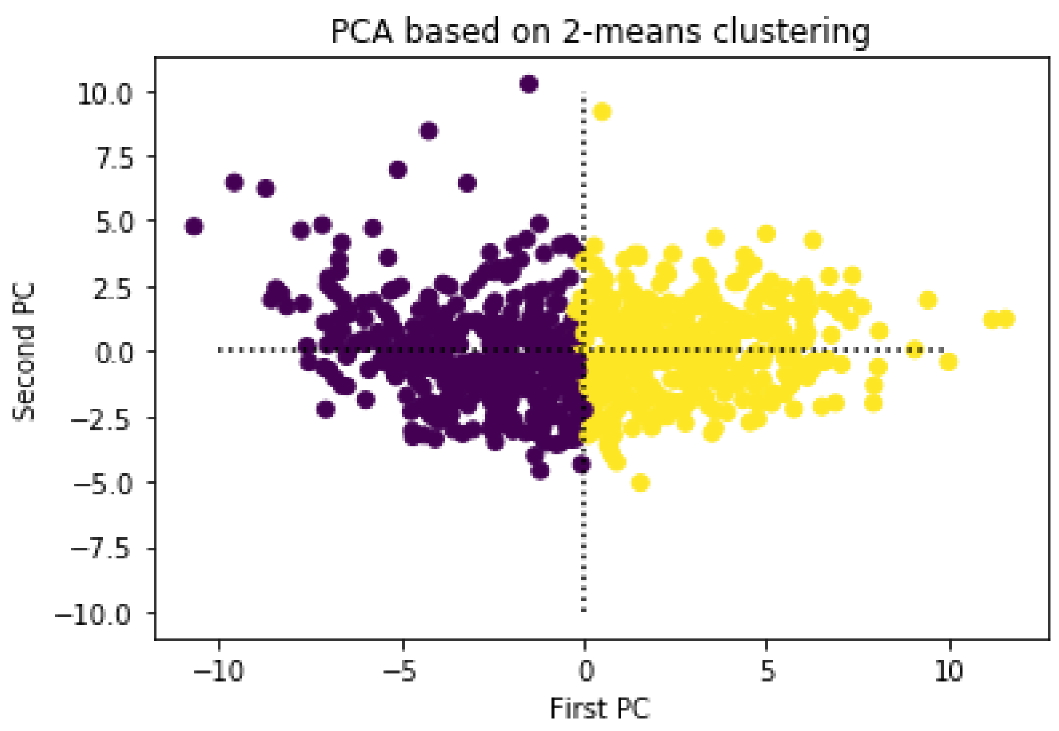
[73]:
```
from gap_statistic import optimalK
opt = optimalK.OptimalK()
opt_clusters = opt(df_wiki, cluster_array = np.arange(2,11))
print("the optimal number of cluster based on gap statistic is", opt_clusters)
```

```
the optimal number of cluster based on gap statistic is 10
```

According to the two graphs, the optimal number of clusters is 2.

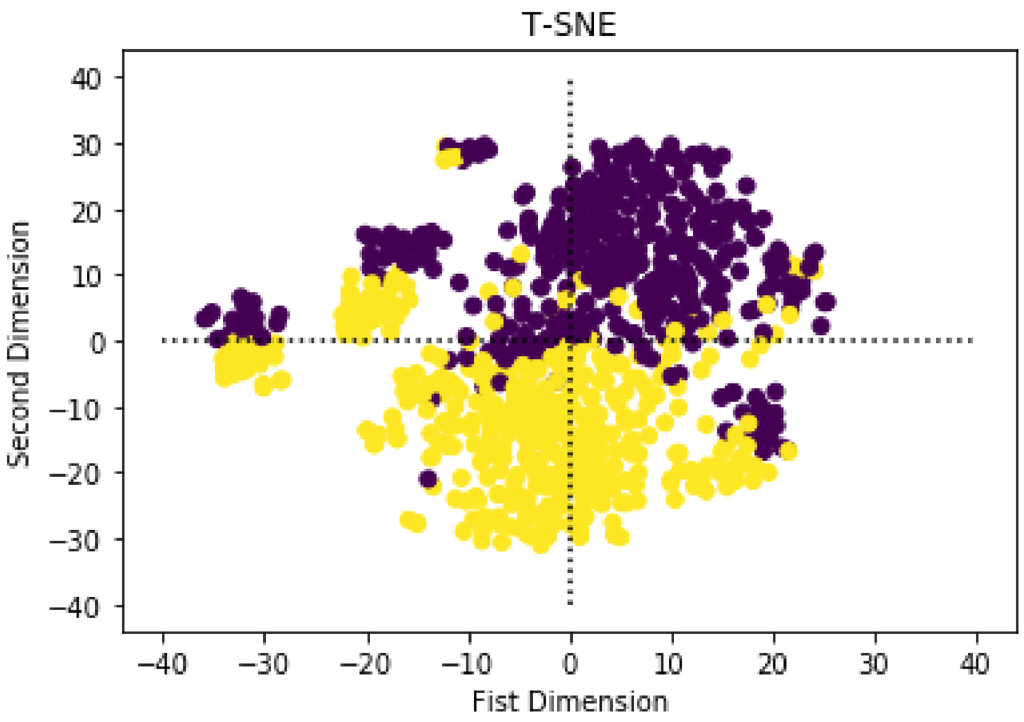
11. (15 points) Visualize the results of the optimal $\hat{k}$-means clustering model. First use the first and second principal components from PCA, and color-code each observation based on their cluster membership. Next use the first and second dimensions from t-SNE, and color-code each observation based on their cluster membership. Describe your results. How do your interpretations differ between PCA and t-SNE?

[70]:
```
plotting_kmean(2)
```



```
[71]: tsne = TSNE(n_components = 2, random_state = 456123)
      df_tsne = pd.DataFrame(data = tsne.fit_transform(df_wiki), columns = ['dim1',␣
       ↪'dim2'])
      k2 = KMeans(n_clusters=2, random_state=456123).fit(df_wiki)
      plt.figure()
      plt.hlines(0, -40, 40, linestyles = 'dotted', colors = 'black')
      plt.vlines(0, -40, 40, linestyles = 'dotted', colors = 'black')
      plt.scatter(df_tsne['dim1'], df_tsne['dim2'], c = k2.labels_)
      plt.xlabel('Fist Dimension')
      plt.ylabel('Second Dimension')
      plt.title('T-SNE')
```

```
[71]: Text(0.5, 1.0, 'T-SNE')
```



From the graphs above we can see that PCA does a better job in sperating the data in 2 cluster. By comparison, the t_SNE has a lot overlaps for the two clusters. The reason why that t-SNE only preserves local similarities whereas PCA perserves large pairwise distance maximized variance. Since t-SNE only finds the local neighbors, it may not be able to get clear cluster boundaries. Moreover, the t-SNE finds patterns in the data by identify the similarity, while PCA focus on placing dissimilar data point far apart. Therefore, this can be another reason for why PCA does a better job at seperating different clusters.