MAD Assignment 5

Andreas V. W. Zacchi (nzl169)

January 2, 2025

Exercise 3

a)

```python
def Handle_Nans(sample1, sample2):
    for i in range(len(sample1) - 1, -1, -1):
        if np.isnan(sample1[i]) or np.isnan(sample2[i]):
            sample1 = np.delete(sample1, i)
            sample2 = np.delete(sample2, i)

    return sample1, sample2

def Pearson_dist(sample1, sample2):
    sample1, sample2 = Handle_Nans(sample1, sample2)
    mean1 = np.mean(sample1)
    mean2 = np.mean(sample2)


    norm_sample1 = sample1 - mean1
    norm_sample2 = sample2 - mean2
    numerator = np.sum(norm_sample1 * norm_sample2)

    norm_sample1_squared = np.sum(norm_sample1 ** 2)
    norm_sample2_squared = np.sum(norm_sample2 ** 2)
    denominator = np.sqrt(norm_sample1_squared *
    norm_sample2_squared)

    return numerator / denominator
```

The implementation of the Euclidean distance and Manhattan distance can be seen in the appendix as they are straight forward.

For handling NaN entries I loop backwards over the array and delete the entry in both sample 1 and sample 2 if one contains NaN, the reason for looping backwards is to avoid index errors because we are in fact shortening the array by 1 each time we delete. This function is then called on the data before calculating any distances.

For calculating the Pearson distance I used the equation from the lecture:

$$\frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}}$$

b)

```python
def compute_silhouette_score_x(distance_map, labels, ind,
    n_clusters):
    a = []
    b = [[] for i in range(n_clusters - 1)]
    for i in range(0, len(labels)):
        if i == ind:
            continue
        if labels[i] == labels[ind]:
            a.append(distance_map[ind][i])
        else:
            b[labels[i] - 1].append(distance_map[ind][i])

    a = np.mean(a)
    b = [np.mean(z) for z in b if z]
    b = np.nanmin(b)

    if a < b:
        return 1 - a / b
    elif a > b:
        return b / a - 1
    else:
        return 0
```

To compute the silhouette score for the point x, we need to calculate both the inter- and intra-cluster distance. We do this by adding points to a if they're in the same cluster as x, and if not add to their respective cluster inside b which is an array of clusters, obviously if it's x we skip it as we don't need to compare it to itself.

Now we have array a containing all intra-cluster distances and b containing sub-arrays of inter-cluster distances. So now we take the mean of a and the mean of all non-empty sub-arrays in b, we now need to pick the cluster with the smallest inter-cluster distance to x in b. Now we can calculate the silhouette score in the way it was done in the lecture.

Lastly we need to compare the silhouette score with the sklearn version, this was done from 2 to 9 clusters and plotted as seen in Figure 1.

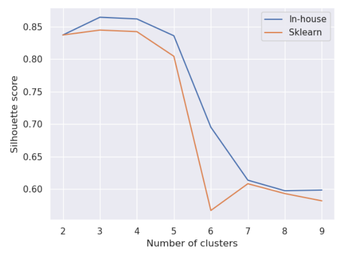


Figure 1: Plot showing in-house silhouettes score against sklearn

The percentage difference is less than 5% except for 6 clusters where it's greater than 20%, which means there is likely an error in my implementation.

c)

```python
def find_optimal_k(data, labels):
    silhouette_scores = []
    for k in range(2, 15):
        kmeans = KMeans(n_clusters=k).fit(data)
        score = silhouette_score(data, kmeans.labels_)
        silhouette_scores.append(score)

    plt.plot(range(2, 15), silhouette_scores, 'o-')
    plt.xlabel('k')
    plt.ylabel('Silhouette score')
    plt.title('Silhouette score for different k')
    plt.show()

    optimal_k = range(2, 15)[np.argmax(silhouette_scores)]
    return optimal_k
```

The above code finds the optimal amount of clusters (k) between 2 and 14, it does this by calculating the score for each k and returning the one with the highest score. It also plots the silhouettes score for each k so it can be visually seen how good each k value is, as seen in Figure 2, the resulting clustering can be seen in Figure 3.

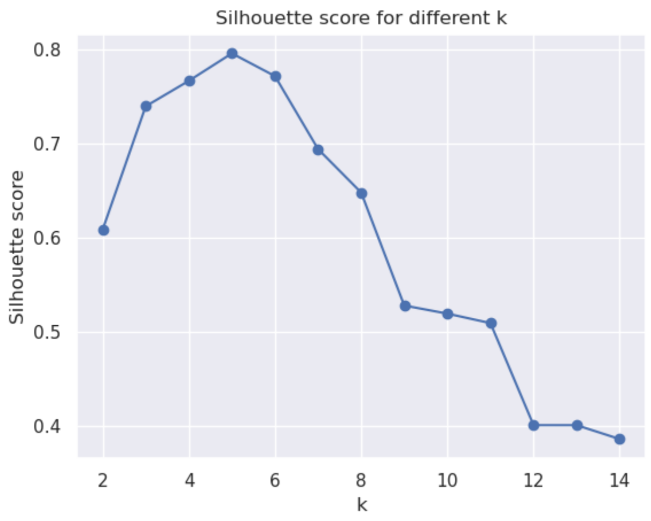


Figure 2: Plot showing silhouette score for each amount of clusters k

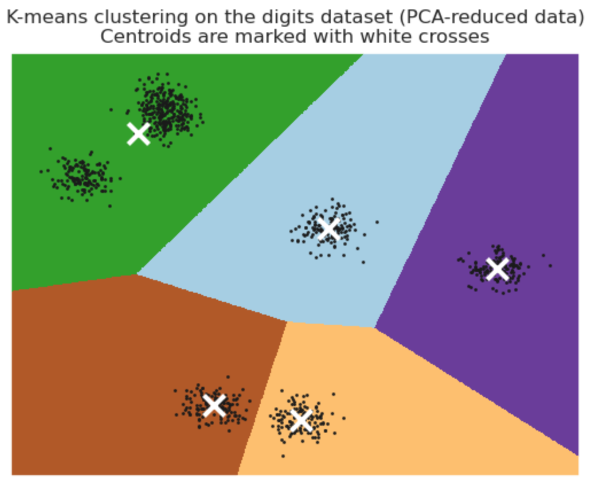


Figure 3: Plot showing the clustering found by kmeans with 5 clusters

Visually it would seem like it would be better to have 6 clusters, splitting green into two clusters. This also happens sometimes but usually has a lower silhouette score, this stems from the randomisation in kmeans. Therefore we cannot solely rely on the silhouette score.

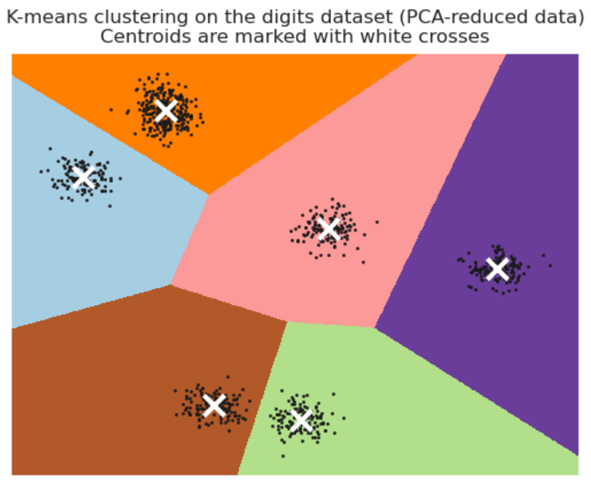


Figure 4: Plot showing the clustering found by kmeans with 6 clusters

Appendix

Assignment: Clustering

This assingment aims to test your understanding of clustering.

Exercise 1

Please implement and test the following distance metrics:

a) Euclidian distance

b) Manhattan distance

c) Pearson vorrelation coefficient

Please take into account that some features for some samples could be missing.

In [22]:
```
import numpy as np


def Handle_Nans(sample1, sample2):
    # implement some algorithm that will remove elements in sample1 and s
    # if sample1[i] is Nan or sample2[i] is Nan
    for i in range(len(sample1) - 1, -1, -1): # iterate backwards to avoi
        if np.isnan(sample1[i]) or np.isnan(sample2[i]):
            sample1 = np.delete(sample1, i)
            sample2 = np.delete(sample2, i)

    return sample1, sample2

def Euclidian_dist(sample1, sample2):
    sample1, sample2 = Handle_Nans(sample1, sample2)
    return np.linalg.norm(sample1 - sample2)

def Manhattan_dist(sample1, sample2):
    sample1, sample2 = Handle_Nans(sample1, sample2)
    return np.sum(np.abs(sample1 - sample2))

def Pearson_dist(sample1, sample2):
    sample1, sample2 = Handle_Nans(sample1, sample2)
    mean1 = np.mean(sample1)
    mean2 = np.mean(sample2)


    norm_sample1 = sample1 - mean1
    norm_sample2 = sample2 - mean2
    numerator = np.sum(norm_sample1 * norm_sample2)

    norm_sample1_squared = np.sum(norm_sample1 ** 2)
    norm_sample2_squared = np.sum(norm_sample2 ** 2)
    denominator = np.sqrt(norm_sample1_squared * norm_sample2_squared)

    return numerator / denominator
```

```python
def main():
    sample1 = np.asarray([17, 28, 37, 23, 8, float('Nan')])
    sample2 = np.asarray([21, 35, float('Nan'), 23, 2, 5])

    print(Euclidian_dist(sample1, sample2))
    print(Manhattan_dist(sample1, sample2))
    print(Pearson_dist(sample1, sample2))

main()
```

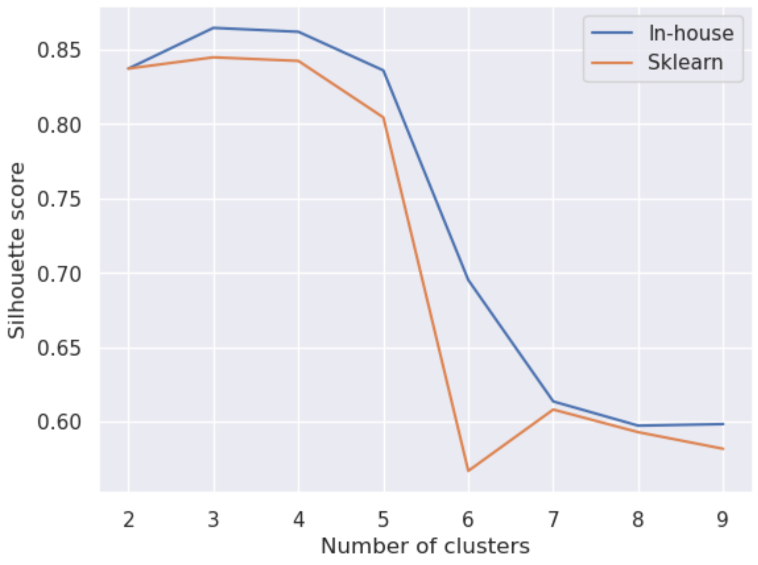
```
10.04987562112089
17.0
0.9738876639603918
```

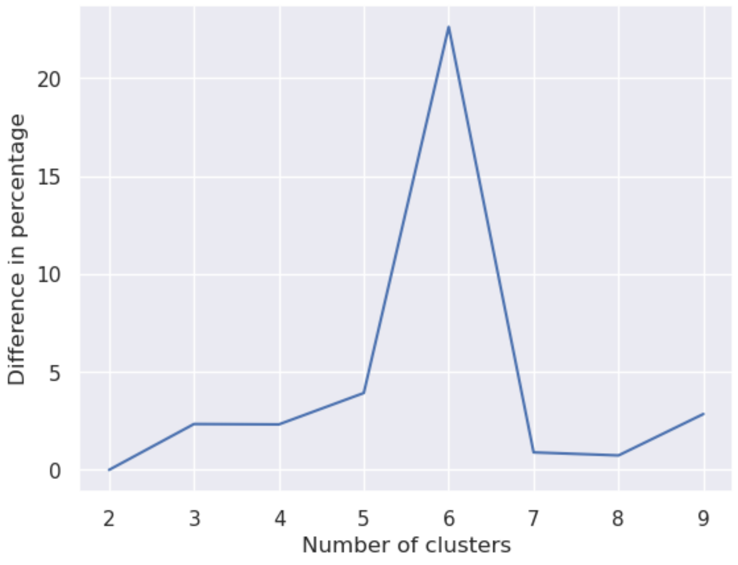
Exercise 2

Please implement and test the silhouette score and compare its performance to silhouette_score from sklearn.metrics

In [ ]:
```python
import numpy as np
from sklearn.datasets import make_blobs
from scipy.spatial import distance_matrix
from sklearn.metrics import silhouette_score, silhouette_samples

def compute_elemtwise_distance(data):
    return distance_matrix(data, data)

def compute_silhouette_score_x(distance_map, labels, ind, n_clusters):
    a = []
    b = [[] for i in range(n_clusters - 1)]  # n_clusters - 1 as we have
    for i in range(0, len(labels)):
        if i == ind:  # skip the current element
            continue
        if labels[i] == labels[ind]:  # same cluster so add to a
            a.append(distance_map[ind][i])
        else:  # different cluster so add to respective cluster
            b[labels[i] - 1].append(distance_map[ind][i]) # subtract 1 as

    a = np.mean(a) # mean distance of element to all other elements in th
    b = [np.mean(z) for z in b if z] # mean distance of element to all ot
    b = np.nanmin(b) # get the cluster with the smallest mean distance to

    if a < b:
        return 1 - a / b
    elif a > b:
        return b / a - 1
    else:
        return 0

def compute_silhouette_score(data, labels, n_clusters):
    distance_map = compute_elemtwise_distance(data)
    score = []
    for i in range(0, len(labels)):
        score.append(compute_silhouette_score_x(distance_map, labels, i,
    return np.mean(score)
```

```python
def generate_data(n_samples, n_clusters):
    data, labels = make_blobs(n_samples = n_samples, centers = n_clusters
                              random_state = 3, cluster_std = 0.6)
    return data, labels

def main():
    in_house_scores = []
    sklearn_scores = []
    diff_percentage = []

    n_samples = 1000
    n_max_clusters = 10
    for i in range(2, n_max_clusters):
        data, labels = generate_data(n_samples, i)
        in_house_scores.append(compute_silhouette_score(data, labels, i))
        sklearn_scores.append(silhouette_score(data, labels))
        diff_percentage.append((in_house_scores[-1] - sklearn_scores[-1])


    plt.figure()
    plt.plot(range(2, n_max_clusters), in_house_scores, label = 'In-house
    plt.plot(range(2, n_max_clusters), sklearn_scores, label = 'Sklearn')
    plt.legend()
    plt.xlabel('Number of clusters')
    plt.ylabel('Silhouette score')
    #plt.savefig('silhouette_score.png')
    plt.show()

    plt.figure()
    plt.plot(range(2, n_max_clusters), diff_percentage)
    plt.xlabel('Number of clusters')
    plt.ylabel('Difference in percentage')
    plt.show()
main()
```



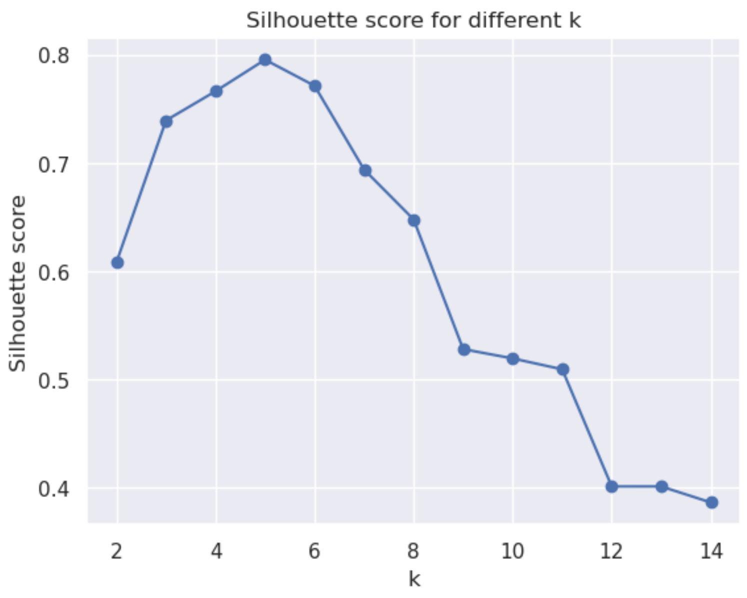


Exercise 3

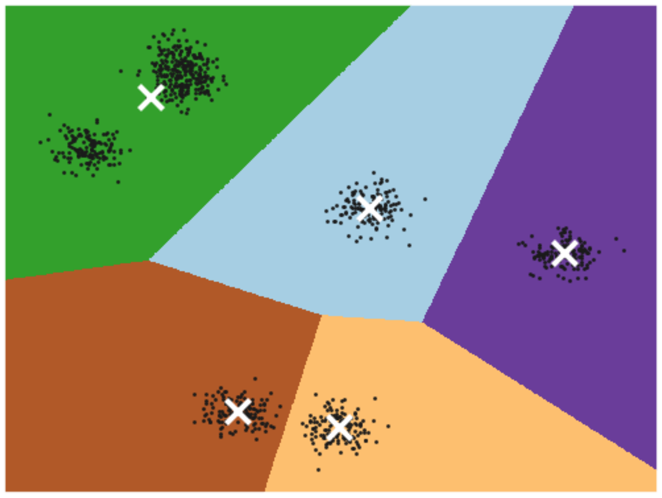
Please compute silhouette score for different number of clusters in k-Mean algorithm. Select the optimal k with the highest silhouette score.

You are allowed to use KMeans and silhouette_score from sklearn.

In [131…
```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats

# use seaborn plotting defaults
import seaborn as sns
sns.set()
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score

def visualizeKMeans(data, kmeans):
    h = .02     # point in the mesh [x_min, x_max]x[y_min, y_max].

    # Plot the decision boundary. For that, we will assign a color to eac
    x_min, x_max = data[:, 0].min() - 1, data[:, 0].max() + 1
    y_min, y_max = data[:, 1].min() - 1, data[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_m

    # Obtain labels for each point in mesh. Use last trained model.
    Z = kmeans.predict(np.c_[xx.ravel(), yy.ravel()])

    # Put the result into a color plot
    Z = Z.reshape(xx.shape)
    plt.figure(1)
    plt.clf()
    plt.imshow(Z, interpolation='nearest',
               extent=(xx.min(), xx.max(), yy.min(), yy.max()),
               cmap=plt.cm.Paired,
               aspect='auto', origin='lower')

    plt.plot(data[:, 0], data[:, 1], 'k.', markersize=2)
    # Plot the centroids as a white X
    centroids = kmeans.cluster_centers_
    plt.scatter(centroids[:, 0], centroids[:, 1],
                marker='x', s=169, linewidths=3,
                color='w', zorder=10)
    plt.title('K-means clustering on the digits dataset (PCA-reduced data
              'Centroids are marked with white crosses')
    plt.xlim(x_min, x_max)
    plt.ylim(y_min, y_max)
    plt.xticks(())
    plt.yticks(())
    plt.savefig('kmeans_clusters.png')
    plt.show()


def find_optimal_k(data, labels):
    silhouette_scores = []
    for k in range(2, 15):
        kmeans = KMeans(n_clusters=k).fit(data)
        score = silhouette_score(data, kmeans.labels_)
        silhouette_scores.append(score)

    plt.plot(range(2, 15), silhouette_scores, 'o-')
    plt.xlabel('k')
    plt.ylabel('Silhouette score')
    plt.title('Silhouette score for different k')
    plt.savefig('silhouette_score_3.png')
    plt.show()

    optimal_k = range(2, 15)[np.argmax(silhouette_scores)] # get the k wi
    return optimal_k

def main():
    n_clusters = 7
    plt.figure(0)
    data, labels = make_blobs(n_samples = 1000, centers = n_clusters,
                              random_state = 5, cluster_std = 0.5)

    k = find_optimal_k(data, labels)

    plt.figure(1)
    kmeans = KMeans(n_clusters = k).fit(data)
    visualizeKMeans(data, kmeans)

main()
```



K-means clustering on the digits dataset (PCA-reduced data)
Centroids are marked with white crosses



```
In [ ]:
```