UiT

THE ARCTIC UNIVERSITY OF NORWAY

Mandatory Assignment 2

FYS-2021 - Machine Learning
Institute of Physics and Technologi
Daniel Elisabethsønn Antonsen

Problem 1

Here we will look at feature extraction with the use of multidimensional scaling or MDS.

a)

We are here asked to briefly describe the main difference between feature extraction and feature selection. We are further asked to describe the multidimensional scaling (MDS) algorithm and comment on its areas of use.

Multidimensional scaling can be seen as a family of mathematical models for analyzing and visulize the distances between different objects, where all the distances between pairs of objects is known. If we have N points with the distances between the points i and j are given as d_{ij}, for all points $i, j = 0, 1, ..., N$. Then MDS is a method for placing these points in a lower space, such as 2-dimensional space, such that the 2-norm or Euclidean distance between the points are as close to the distances d_{ij} as possible. Which is the original given distances. And so, we here require some form of projection from an unknown dimensional space, to another known dimensional space such as 2-dimensions.
We can also use MDS for dimensionality reductions by computing the pairwise 2-norm or Euclidean distances and give these values as input to the MDS algorithm, which is then projected to a lower-dimensional space with the distances preserved.
If we have a data set given by $\mathbf{x} = \{\mathbf{x}^t\}_{t=1}^N$ with $\mathbf{x}^t \in \mathbb{R}^d$. Then a matrix of these Euclidean distances is given as

$$B = \mathbf{x}^T\mathbf{x}$$

Using this as our input matrix, we can further notice that $\mathbf{x}^T\mathbf{x}$ is symmetric and real. And so, we have

$$B = \mathbf{x}^T\mathbf{x} = EDE^T = ED^{\frac{1}{2}}D^{\frac{1}{2}}E^T$$

We can find the new coordinates for the original points is given by

$$z = D^{\frac{1}{2}}E^T$$

On typical examples of the MDS algorithm is to get an approximation to a map, where we have used the road travel distance between cities which is then feed through the algorithm and finally gets an approximation of a map.

b)

We are here asked to implement a multidimensional scaling algorithm and run this on the provided data which again generates a 2-dimensional representation. This can be plotted next to a map from Google maps and be compared.

We can start by implementing the MDS algortihm as a class with the B matrix and the names of the cities as input.

```
class Multidimensional_scaling:

    def __init__(self, B:np.ndarray, labels:np.ndarray):
        """
        Args:
            B: np.ndarray, the distance matrix
```

UNIVERSITY OF TROMSØ

```
            labels: np.ndarray, the labels for each city
        """
        self.__B = B
        self.__labels = labels
```

Further we require to find the eigenvalues and eigenvectors for matrix B, and so we can implement a method for computing these using NumPy linalg library. And so

```
    def __eigenvectors(self):
        """
        Method for computing the eigenvectors and eigenvalues

        Output:
            eigenvalues: np.ndarray, array containing eigenvalues
            eigenvectors: np.ndarray, array containing corrresponding eigenvectors
        """
        # Computing eigenvalues and eigenvectors
        eigenvalues, eigenvectors = np.linalg.eig(self.__B)
        return eigenvalues, eigenvectors
```

And now that we have these values, we can continue implementing a method for computing the coordinates using the eigenvectors and eigenvalues. The method has these values as input and returns an array containing the two-dimensional coordinates. We also set the dimensions we want by setting the number of eigenvalues and eigenvectors we use for computing the coordinates. This can be implemented as

```
    def __coordinates(self, eigenvectors:np.ndarray, eigenvalues:np.ndarray, dim:int):
        """
        Method for computing the new coordinates for the points

        Args:
            eigenvalues: np.ndarray, array containing the eigenvalues for matrix B
            eigenvectors: np.ndarray, array containing the corresponding eigenvectors for
                                                        matrix B
            dim: int, spesifying the dimensions
        Output:
            np.ndarray, array containing the new coordinates
        """
        return np.sqrt(eigenvalues[:dim]) * eigenvectors[:, :dim]
```

Now that we have the coordinates, we can implement a method for plotting the results together with the names for each. Here we can simply use matplotlib.pyplot library and make use of the scatter plot. Further we can use the annotate method from matplotlib to plot the names of each cities at each coordinates. This is implemented as

```
    def _plot(self, coordinates:np.ndarray, labels:np.ndarray):
        """
        Method for plotting the points

        Args:
            coordinates: np.ndarray, array containing the new points
            labels: np.ndarray, array containing the labels for for each cities
        """
        plt.scatter(coordinates[:, 1], coordinates[:, 0])
        for i, txt in enumerate(labels[:, 0]):
            plt.annotate(txt, (coordinates[i, 1] + 3, coordinates[i, 0] + 3))
        plt.show()
```

And lastly, we can implement a method for running all the other methods.

```
    def run(self):
        """
        Method for running the MDS algorithm
        """
        __eigenvalues , __eigenvectors = self.__eigenvectors()
```

```
        __coordinates = self.__coordinates(__eigenvectors, __eigenvalues, 2)
        self._plot(__coordinates, self.__labels)
```

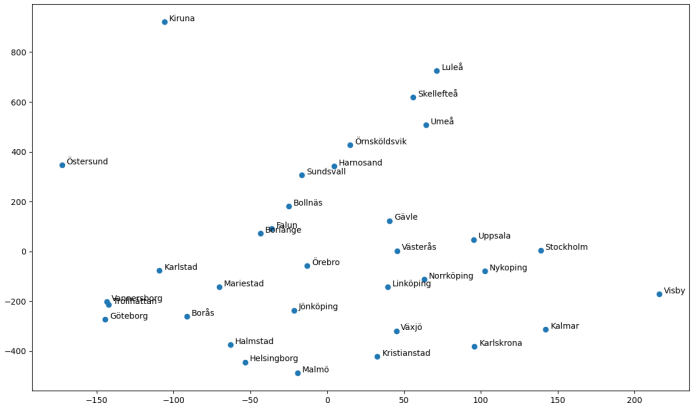

Figure 1: *Illustration of the plot found using MDS.*

Figure 2:
Illustration of the results on top of map over Sweden. Source for map: https://vemaps.com/sweden/se-02.

In figure 1 we have plotted the results after applying the MDS algorithm, and in figure 2 we have combined the results from MDS together with a map over Sweden.
As we can see from the figures, the coordinates make sence in the fact that the distances somewhat is accurate. But also the scaling of the results don't match with the scale of the map. So, we have to rotate

and re-scale to be able to compare. It might be a little difficult to see, but we can see that the position of the cities to the north, is located around the right area.

Problem 2

a)

We are here asked to briefly describe the k-means algorithm.

K-means clustering is an unsupervised learning algorithm, that groups together unlabelled datasets into different clusters. The number of these clusters are predefined, and the k in k-means clustering describes this number. The algorithm is iterative and divides the inputed dataset into k different clusters, so that each point belongs to at most one cluster. It does this by computing the distances and minimize the distances between each point and each centroids or centre of each clusters. And further the algorithm assign each input to the nearest k-center and the points closest to a centroid creates a cluster.
If we have a dataset x, we start the algorithm by randomly choosing startvalues from χ where $x \in \chi$, for the centroids m_k where k denotes the number of centroids defined at the start. We then assign each point in the dataset to a centriod C_j such that

$$||x - m_j||$$

is minimized. Where $||x - m_j||$ is the Euclidean distance. Further we update the centroids by computing the mean of the points assign to each cluster,

$$m_j = \frac{1}{C_j} \sum_{y \in C_j} y$$

for $j = 0, ..., k$. This is done iteratively until we have little to no change in centriods m_j.

b)

We are here asked to implement the k-mean algorithm and run the k-mean for $k = 2$, $k = 4$ and $k = 10$ clusters on the given dataset. We are further asked to plot the centroids and some of the images closest to each centroid using a grayscale colourmap.

We can start by implementing the k-means clustering as a class, which takes in the dataset x and the number of clusters k. We initialize the cenroids by choosing random values from the inputed dataset, and creates arrays for the different clusters and distances which have not been found yet.

```
class k_mean:

    def __init__(self, X:np.ndarray, k:int):
        """
        Args:
            X: np.ndarray, matrix containg unlabeld dataset
            k: int, number of clusters
        """
        self.__X = X
        self._k = k
        self._centroids = X[np.random.choice(X.shape[0], k, replace=False)]
        self._labels = np.zeros(X.shape[0])
        self._distance = np.zeros((X.shape[0], k))
```

Since we know that we need the Euclidean distance, we can make a method in the class that computes the 2-norm or Euclidean distance between the data points and the centroids using the NumPy library. We iterate over the number of clusters and compute the distances. This is implemented as

```
    def _comp_distance(self):
        """
        Method for computing the Euclidean distance matrix
        """
        for i in range(self._k):
            self._distance[:, i] = np.linalg.norm(self.__X - self._centroids[i, :], axis=
                                                  1)
```

Now that we have computed the distances, we can assign each datapoint in the inputed dataset to one of the clusters by finding the minimum value fopr the distances to each cluster, from each point. This is implemented as a method of the form

```
    def _assign(self):
        """
        Method for assigning points to clusters
        """
        self._labels = np.argmin(self._distance, axis=1)
```

We can now implement a method for updating the centroids using the mean of all the point assigned to each cluster. This can be implemented as

```
    def _update_centroids(self):
        """
        Method for updating the centroids
        """
        for i in range(self._k):
            self._centroids[i] = np.mean(self.__X[self._labels == i], axis=0)
```

We can further implement a method for running the algorithm until the difference in distance to the centroids is so smaller than some tolorance, which here is sat to 1×10^{-5}. The algorithm consists of running all the methods described above, and we will here return the arrays consisting of the clusters, the centroids and the sorted arrays of the last distances. This is implemented as

```
    def run(self):
        """
        Method for running the algorithm

        Output:
            centroids: np.ndarray, array containing the centroids
            labels: np.ndarray, array containing the labels for each point
            Sorted array containg the distances from min to max
        """
        while True:
            self._comp_distance()
            self._assign()
            self._update_centroids()
            old_distance = self._distance
            if np.all(np.linalg.norm(self._distance - old_distance)< 1E-5):
                break
        return self._labels, self._centroids, np.argsort(self._labels)
```

And lastly we can implement a method for plotting the results, where we input arrays for the images and the centroids found. This is then plotted as a subplot, and byu iterating over the number of images we plot each centroid in the first column and the next four images is the closest to the centroid. This is implemented as

```
    def plot(self, imgs:np.ndarray, centroids:np.ndarray):
        """
        Method for plotting the clusters

        Args:
            img: np.ndarray, array containing the images to be plotted
            centroids: np.ndarray, array containing the centroids
        """
        fig, ax = plt.subplots(self._k, imgs.shape[1], figsize=(12, 12), tight_layout=
                                                        True, \
                               subplot_kw={'xticks': [], 'yticks': []}, sharex=True,
                                                                    sharey=True)
        for j in range(self._k):
            im_centroid = centroids[j].reshape(28, 20)
            ax[j, 0].imshow(im_centroid, cmap='gray')
            for i in range(1, imgs.shape[1]):
                im = imgs[j, i].reshape(28, 20)
                ax[j, i].imshow(im, cmap='gray')
        fig.suptitle(f"K-means clustering with k = {str(self._k)}")
```

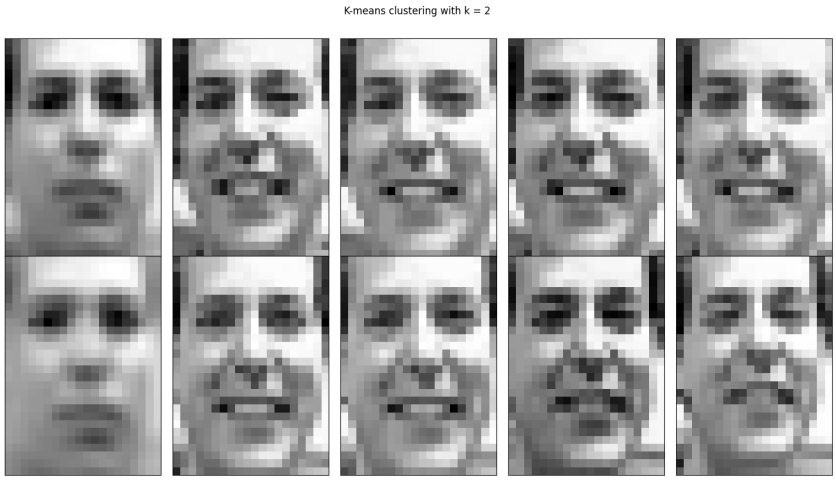

Figure 3: *K-means clustering with* $k = 2$.

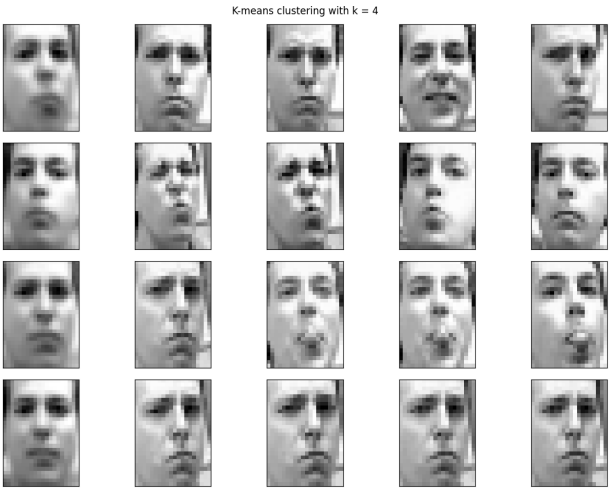

Figure 4: *K-means clustering with* $k = 4$.

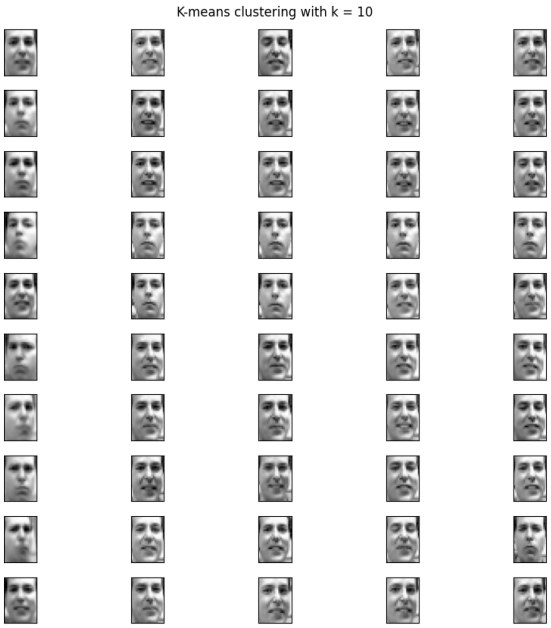

Figure 5:
K-means clustering with $k = 10$. *NB! I don't know why the spacing is the way it is, but I did not find a solution in time for this submission.*

In figure 3 we have the results from running the algorithm using $k = 2$, in figure 4 we have the results using $k = 4$. And in figure 5 we have the results using $k = 10$.
We can see somewhat of a pattern where each cluster represent different types of facial expression, some where the person is smiling. Other where the person is sad, and so on. This may not be so easy to see from the result with only 2 clusters, but as we add on more cluster this pattern can be more easily recognisable.

Appendix

MDS.py

```
"""
File containg the Multidimensional scaling algorithm
"""

__author__ = 'Daniel Elisabeths nn Antonsen'

# Importing libraries and modules
import numpy as np
import matplotlib.pyplot as plt

class Multidimensional_scaling:

    def __init__(self, B:np.ndarray, labels:np.ndarray):
        """
        Args:
            B: np.ndarray, the distance matrix
            labels: np.ndarray, the labels for each city
        """
        self.__B = B
        self.__labels = labels

    def __eigenvectors(self):
        """
        Method for computing the eigenvectors and eigenvalues

        Output:
            eigenvalues: np.ndarray, array containing eigenvalues
            eigenvectors: np.ndarray, array containing corrresponding eigenvectors
        """
        # Computing eigenvalues and eigenvectors
        eigenvalues, eigenvectors = np.linalg.eig(self.__B)
        return eigenvalues, eigenvectors

    def __coordinates(self, eigenvectors:np.ndarray, eigenvalues:np.ndarray, dim:int):
        """
        Method for computing the new coordinates for the points

        Args:
            eigenvalues: np.ndarray, array containing the eigenvalues for matrix B
            eigenvectors: np.ndarray, array containing the corresponding eigenvectors for
                                                                matrix B
            dim: int, spesifying the dimensions
        Output:
            np.ndarray, array containing the new coordinates
        """
        return np.sqrt(eigenvalues[:dim]) * eigenvectors[:, :dim]


    def _plot(self, coordinates:np.ndarray, labels:np.ndarray):
        """
        Method for plotting the points

        Args:
            coordinates: np.ndarray, array containing the new points
            labels: np.ndarray, array containing the labels for for each cities
        """
        plt.scatter(coordinates[:, 1], coordinates[:, 0])
        for i, txt in enumerate(labels[:, 0]):
            plt.annotate(txt, (coordinates[i, 1] + 3, coordinates[i, 0] + 3))
        plt.show()
```

```
    def run(self):
        """
        Method for running the MDS algorithm
        """
        __eigenvalues , __eigenvectors = self.__eigenvectors()
        __coordinates = self.__coordinates(__eigenvectors, __eigenvalues, 2)
        self._plot(__coordinates, self.__labels)
```

KMC.py

```
"""
File containg the k-means clustering algorithm
"""

__author__ = 'Daniel Elisabeths nn Antonsen'

# Importing libraries and modules
import numpy as np
import matplotlib.pyplot as plt

class k_mean:

    def __init__(self, X:np.ndarray, k:int):
        """
        Args:
            X: np.ndarray, matrix containg unlabeld dataset
            k: int, number of clusters
        """
        self.__X = X
        self._k = k
        self._centroids = X[np.random.choice(X.shape[0], k, replace=False)]
        self._labels = np.zeros(X.shape[0])
        self._distance = np.zeros((X.shape[0], k))

    def _comp_distance(self):
        """
        Method for computing the Euclidean distance matrix
        """
        for i in range(self._k):
            self._distance[:, i] = np.linalg.norm(self.__X - self._centroids[i, :], axis=
                                                  1)

    def _assign(self):
        """
        Method for assigning points to clusters
        """
        self._labels = np.argmin(self._distance, axis=1)

    def _update_centroids(self):
        """
        Method for updating the centroids
        """
        for i in range(self._k):
            self._centroids[i] = np.mean(self.__X[self._labels == i], axis=0)

    def run(self):
        """
        Method for running the algorithm

        Output:
            centroids: np.ndarray, array containing the centroids
            labels: np.ndarray, array containing the labels for each point
```

```
            Sorted array containg the distances from min to max
        """
        while True:
            self._comp_distance()
            self._assign()
            self._update_centroids()
            old_distance = self._distance
            if np.all(np.linalg.norm(self._distance - old_distance)< 1E-5):
                break
        return self._labels, self._centroids, np.argsort(self._labels)

    def plot(self, imgs:np.ndarray, centroids:np.ndarray):
        """
        Method for plotting the clusters

        Args:
            img: np.ndarray, array containing the images to be plotted
            centroids: np.ndarray, array containing the centroids
        """
        fig, ax = plt.subplots(self._k, imgs.shape[1], figsize=(12, 12), tight_layout=
                                                        True, \
                               subplot_kw={'xticks': [], 'yticks': []}, sharex=True,
                                                                    sharey=True)
        for j in range(self._k):
            im_centroid = centroids[j].reshape(28, 20)
            ax[j, 0].imshow(im_centroid, cmap='gray')
            for i in range(1, imgs.shape[1]):
                im = imgs[j, i].reshape(28, 20)
                ax[j, i].imshow(im, cmap='gray')
        fig.suptitle(f"K-means clustering with k = {str(self._k)}")
```

main.py

```
"""
Main file for the course FYS-2021 Machine Learning, UiT The Arctic University.
"""

__author__ = 'Daniel Elisabeths nn Antonsen'

# Importing modules and libraries
import numpy as np
import os
from MDS import Multidimensional_scaling
import pandas as pd
from KMC import k_mean
import matplotlib.pyplot as plt

## Problem 1
# Opening data containing the B-matrix
data_inner_sweden = np.loadtxt(os.path.join("resources", "city-inner-sweden.csv"))
names_inner_sweden = pd.read_csv(os.path.join("resources", "city-names-sweden.csv"),
                                                    header=None).to_numpy()
faces_data = np.loadtxt(os.path.join("resources", "frey-faces.csv"))

# Running multidimensional scaling algorithm
Multidimensional_scaling(data_inner_sweden, names_inner_sweden).run()

# Running k-mean clustering algorithm for k = 2
mean_2 = k_mean(faces_data, 2)
labels_2, centroids_2, sorted_2 = mean_2.run()
# Sorte the images and choose the once closest to the centroids
sorted_2_0 = faces_data[sorted_2[labels_2 == 0], :]
sorted_2_1 = faces_data[sorted_2[labels_2 == 1], :]
```

```
im_2 = np.array([sorted_2_0[:5, :], sorted_2_1[:5, :]])

# Plotting the images for k = 2
mean_2.plot(im_2, centroids_2)

# Running k-mean clustering for k = 4
mean_4 = k_mean(faces_data, 4)
labels_4, centroids_4, sorted_4 = mean_4.run()
# Sorte the images and choose the once closest to the centroids
sorted_4_0 = faces_data[sorted_4[labels_4 == 0], :]
sorted_4_1 = faces_data[sorted_4[labels_4 == 1], :]
sorted_4_2 = faces_data[sorted_4[labels_4 == 2], :]
sorted_4_3 = faces_data[sorted_4[labels_4 == 3], :]

# Plotting the images for k = 4
im_4 = np.array([sorted_4_0[:5, :], sorted_4_1[:5, :], sorted_4_2[:5, :], sorted_4_3[:5,
                                                    :]])
mean_4.plot(im_4, centroids_4)

# Runnning k-mean clustering for k = 10
mean_10 = k_mean(faces_data, 10)
labels_10, centroids_10, sorted_10 = mean_10.run()

# Sorte the images and choose the once closest to the centroids
sorted_10_0 = faces_data[sorted_10[labels_10 == 0], :]
sorted_10_1 = faces_data[sorted_10[labels_10 == 1], :]
sorted_10_2 = faces_data[sorted_10[labels_10 == 2], :]
sorted_10_3 = faces_data[sorted_10[labels_10 == 3], :]
sorted_10_4 = faces_data[sorted_10[labels_10 == 4], :]
sorted_10_5 = faces_data[sorted_10[labels_10 == 5], :]
sorted_10_6 = faces_data[sorted_10[labels_10 == 6], :]
sorted_10_7 = faces_data[sorted_10[labels_10 == 7], :]
sorted_10_8 = faces_data[sorted_10[labels_10 == 8], :]
sorted_10_9 = faces_data[sorted_10[labels_10 == 9], :]

# Plotting the images for k = 10
im_10 = np.array([sorted_10_0[:5, :], sorted_10_1[:5, :], sorted_10_2[:5, :], sorted_10_3
                                                    [:5, :],
                  sorted_10_4[:5, :], sorted_10_5[:5, :], sorted_10_6[:5, :], sorted_10_7
                                                          [:5, :],
                  sorted_10_8[:5, :], sorted_10_9[:5, :]])
mean_10.plot(im_10, centroids_10)

if __name__ == "__main__":
    plt.show()
```