Universitetet i Tromsø

Fys-2021 - Machine Learning

Take home exam 3

Candidate number: 6

November 30, 2018

1 Problem 1

1.1 a)

With multidimensional scaling(MDS) we want to scale down the features by removing the dimensions which does not contribute much to explaining the variance of the of the overall data. Here we could use the eigenspace to find the dimensions which does not change too much after a linear transformation. If we get a low eigenvalue, (the scalar of the eigenvectors) we know it doesn't change so much when we transform the data. In most cases we want to only have two features so that we can plot it on a two dimensional plot, so here we would want to choose the two features which scales the most during the transformation(the largest eigenvalues). To do this we first find the $N \times N$ matrix by taking the inner product of the data-set(this is the data-set we are given),

$$\mathbf{L} = \mathbf{X}^T\mathbf{X} \tag{1}$$

L is now the the total transformation of the data, this shows how the data-set is structures and we can now use this to find the eigenvalues and eigenvectors. The idea here is to reduce the number of features such that we have $\mathbf{X} = \mathbf{Z}$, here $\mathbf{X}$ is the data-set and $\mathbf{Z}$ will be our new data-set,

$$\mathbf{X}^T\mathbf{X} = \mathbf{EDE}^T = \mathbf{ED}^{1/2}\mathbf{D}^{1/2}\mathbf{E}^T \tag{2}$$

Here the $\mathbf{D}$ is the diagonal eigenvalue matrix, $\mathbf{E}$ is the eigenvector matrix. This means that $\mathbf{Z} = \mathbf{D}^{1/2}\mathbf{E}^T$. This is our new data-set, by having the eigenvalues in descending order we can grab the top two features from $\mathbf{Z}$, we then have our scaled data-set $\mathbf{Z}_{reduced}$. Its worth mentioning that if we have alot of features we might remove alot of information, and in some cases we might have removed to much information. Areas of use is for example what we are going to do in this task which is to map different distance (geodesic) to a 2-dimensional euclidean space by preserving the distances which contains the most information, other areas of use could be medical data which contains alot of features, and a way to visualize the difference in the features is to reduce it to three or two dimensions and plot it(multidimensional scaling could of-course be used in a wide range of field). The reason why MDS is appropriate to solve the task of mapping geodesic to 2D in this particular case is that Sweden doesn't curve too much around the globe so the geodesic distance $\approx$ euclidean distance between the cities. If we were to have cities covering larger part the geodesic we would have bigger errors at some points.

1.2 b-c)

Here we use the MDS algorithm which is given in the code below,

```python
def feature_reduction(X):
    """
    Reduced the features of data to two dimensions
    """
    # Get eigenvalues and eigenvector
    val, E = np.linalg.eig(X)
    # Sorting highest to lowest eigenvalues
    idx = val.argsort()[::-1]
    val = val[idx]
    # Taking square root of the diagonal eigenvalue matrix
    Dsqr = np.sqrt(np.diag(val))
    # Taking the inner product to form data-set Z
    Z = np.dot(E, Dsqr)
    # Return two features
    return Z[:,:2]
```

The result is given in figure(1) below,

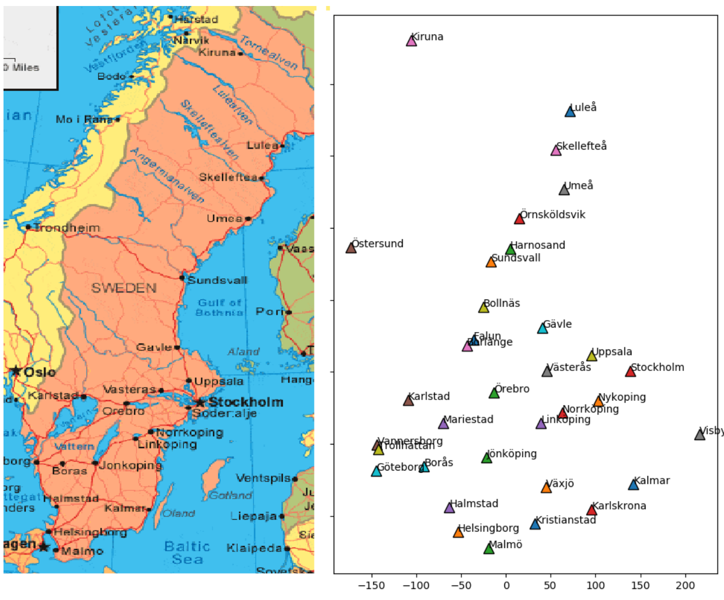

Figure 1: Here we have the plot of data which has been reduced to two features, the map on left is found here: https://geology.com/world/sweden-satellite-image.shtml

We can see that the cities seem to be placed in the right place, the distances between the cities does also seems to be fairly accurate. It can be shown why we would have some errors in the true distance if we look at how the distances actually are mapped on the geodesic as seen in the figure below,

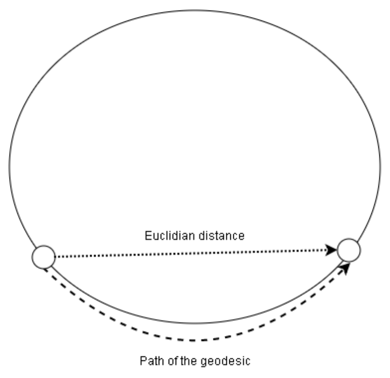

Figure 2: Difference between the geodesic and the euclidean space, as we map to 2D space we would get some loss of actual distance.

We could analyze the information loss by looking at eigenvalues which are the ones that scales our distances. The top three eigenvalues (with square root),

$$\sqrt{\lambda_1} = 2019$$
$$\sqrt{\lambda_2} = 528$$
$$\sqrt{\lambda_3} = 13$$

so here we can see that the top two eigenvalues yields the best information gain, we discard all the others and keep the top two. We now know we will lose some information.

2 Problem 2

2.1 a)

K-means aims to partition the data into different clusters that may belong to different classes. This is done by first placing N number of centroids into the vector-space of our data. Then we find the distance from each data-point to all the centroids. When all distances have been found we assign the data-points which is closest to a centroid to this cluster. Then the mean of the cluster is found and then we place the centroid at the mean. This processed is done iteratively until the centroids converge. So we have as follows,

Given a set of observations (x1, x2, ..., x_n), where each observation is a d-dimensional real vector, k-means clustering aims to partition the n observations into k (n) sets S = S1, S2, ..., S_k so as to minimize the within-cluster sum of squares,

$$\arg\min_{\mathbf{S}} \sum_{i=1}^{k} \sum_{\mathbf{x}\in S_i} \|\mathbf{x} - \boldsymbol{\mu}_i\|^2 \tag{3}$$

where μ_i is the mean of points in S_1.[1]

Some problems with k-means is that there could be more then one minimum in our data-set. This means that the centroids could end up at different coordinates for which the centroids finds the minimum distance, to illustrate this, the code has been run with a couple of times until we get a different result with $N = 3$. This problem is shown in figure(3) below,

K-means could be used in medicine where the data-set contains blood data and other measures, then if we know that some of the patients have cancer or some other illness we can see if there is a given data measures that will group people into the non-cancer or cancer cluster, or compressing of images.

2.2 b)

The implementation of the clustering is done with a few functions in python, here we have 8 functions, (this is not really needed, but here we also have cost function etc.) the main function is the k-means function which uses most of the other functions,

```
def k_means(data, N, label = [None], savefigure = False,
                                          max_iters = 150):
```

Here we use the data-set, number of centroids, labels(if we want to use labels), savefigure to save plots if we are in 2D space. and max iteration where default is 150.

This function then uses the "initate_cents" function to place the centroids at random positions in our vector space,

```
centroids = initate_cents(data, N)
```

[1]see reference Kriegel, Hans-Peter; Schubert, Erich; Zimek, Arthur

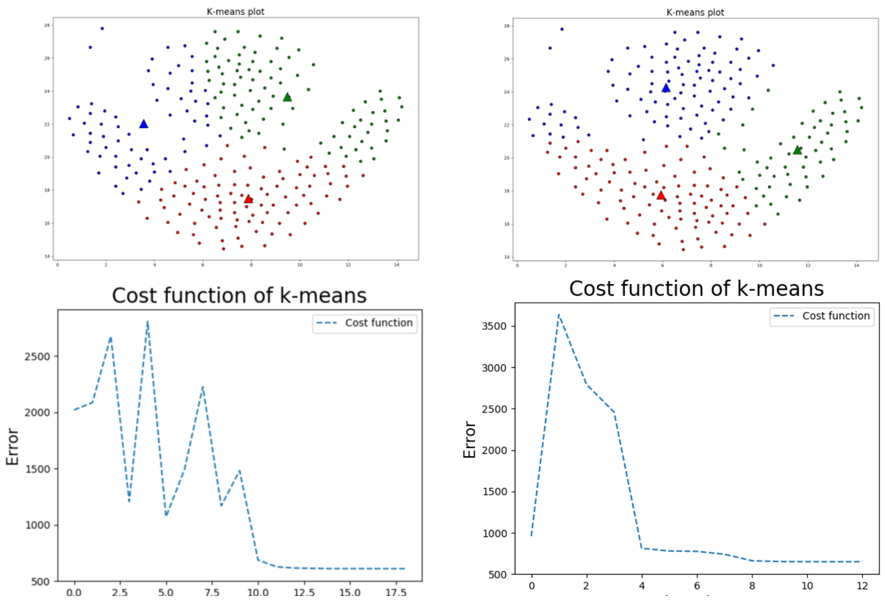

Figure 3: Here we used N = 3, as we can see on this 2D data-set, both errors has minimum at ≈ 500, and we get two different result. This means that the data-set have different areas of convergence when the amount of clusters are 3. This ambiguity holds together with the initialization of the centroids,(random star values). Here it would be a good idea to remake the k-means a couple of time to see if we might get a better result. The cost function here is the sum of all the distances from the centroid to the points inside their respective cluster.

This function uses a random point in the data-set to place the centroids, this ensures that the centroids always finds a cluster, code is shown below,

```
def initate_cents(data, N):
    """
    Make random centroids in the vectorspace of the
                               datapoints
    """
    global reset
    features = len(data[0])
    # map the min and max value of the dataset to contain the
                                centroid spawn into
    centroids = np.zeros((N,features))

    # place centroids in N dimensional space
    for j in range(N):
        for i in range(features):
            # make random coordinate
```

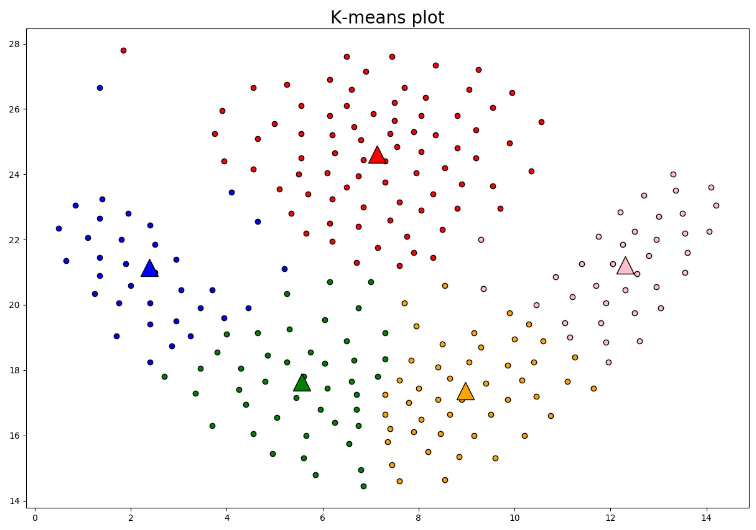

Figure 4: Clustered data using N = 5, here we find 5 groups that might be of interest. This may of-course not actually be actually true because we will always find groups, so we need to have some intuition of how many clusters there might be depending on our goal, or use loss function to find the best amount of clusters.

```
            rand = np.random.choice(data[random.randint(0,len
                                            (data)-1)],
                                            replace=False)
            centroids[j,i] = rand
    reset = 0
    return centroids
```

After we return the centroids matrix we use the "get_distance" function, this function finds all the distances to all the points for each centroids and pluck it into a matrix. Code for it is below,

```
def get_distance(data, centroids, N):
    """
    Find distance between the datapoints to the centroids.
    """
    # find shape
    rows_data, cols_data = data.shape
    temp = np.zeros((rows_data,N))
    # fill temporary matrix with the distances in the N
                                dimensional space of the
                                dataset
    for length, row in enumerate(data):
        for cent, row_cent in enumerate(centroids):
            # [centroid1, centroid2]
            #[length    , length]
            temp[length, cent] = np.linalg.norm(row-row_cent)
    return temp
```

After returning the distance matrix, we use "get_cents" function, this is the function which groups the data for each centroid. Here we use dictionaries with the *key* as the centroid and the *values* are the data-points.

This function also uses the "error_func" which takes sums up all the distances to the centroids and for plotting the error function, (we want to minimize the overall length to all the centroids). If we use labels with our data, this function will also keep track of the labels in its own dictionary. After grouping all the data to the centroids, the group dictionary and label dictionary will be returned. We then have a plotting function, but this will not be relevant and will not be used in this task because we use high dimensional data-set. We then have the "get_means" function,

```
# Get mean coordinates for the centroids
groups_mean = get_means(data, groups)
```

This function finds the mean of the cluster returns the coordinates

```
def get_means(data, groups):
    """
    Get the mean of the clusters and assign the centroids
                                        this new coordinate in a
    dictionary.
    """
    features = len(data[0])
    groups_mean = {}
    for i in groups:
        for n in range(features):
            if i not in groups_mean:
                groups_mean[i] = []
            groups_mean[i].append(np.mean(groups[i][:,n]))

    return groups_mean
```

The "groups_means" dictionary is then sent into "remake_cents" function to assign the centroids to this new coordinate.

```
def remake_cents(data, groups_mean, N):
    """
    Since the distance function uses matrix, we need to fill
                                        up a new
    matrix with the new mean coordinates.
    """

    global error_func_holder
    global error_index
    features = len(data[0])
    centroids = np.zeros((N,features))

    try:
        # assign centroids to new matrix for finding distance
                                                    if not all centroids
                                                  (N) have at least one
        # data point assigned to it, this test will fail and
                                                  go to except to make
                                                  new random centroids.
        for j in range(N):
            centroids[j,:] = groups_mean[j]
    # If some centroids fails tp find a cluster we make new
                                              random centroids
    # this could happen there is one centroid which didnt
                                              have a closest datapoint
    except:
        if not os.path.exists('plots'):
            os.makedirs('plots')
        # remove plots added to folder
        shutil.rmtree('plots')
        if not os.path.exists('plots'):
            os.makedirs('plots')

        error_index.append(len(error_func_holder)-1)
        # initiate new random centroids
        return initate_cents(data, N)
    return centroids
```

Here we also have some *safety measure* were we reset the centroids in the case where we don't have centroids assigned to a group(or if they get placed on top of each other) and also then resets the plots. After this function we run all the function over again until the centroids converge, which is tested with snippet below,

```
# if converged or reached max iterations return the groups of
                                                  clusters
if change == np.sum(temp) or savefi == max_iters:
```

We also have the "plot_centroids" function which plots the coordinate of the centroids, code is shown below,

```
def plot_centroids(groups,title,v,h):
    """
    Plots the images from groups.
    """
    plt.figure(figsize= (8,5))
    for i in sorted(groups):
        try:
            plt.subplot(int(len(groups)/2),int(len(groups)/4)
                                                ,i+1)
        except:
            plt.subplot(2,4,i+1)
        number = groups[i]
        number = np.reshape(number,(v,h))
        plt.imshow(number, cmap = 'Greys_r')
        plt.suptitle(title, fontsize = 20)
    plt.savefig('plots/%s.png'%title)
    plt.close()
```

This function saves the plot in the folder */plot*. When running this algorithm on the *frey-faces.csv* data with $N = 3$ we get the following,

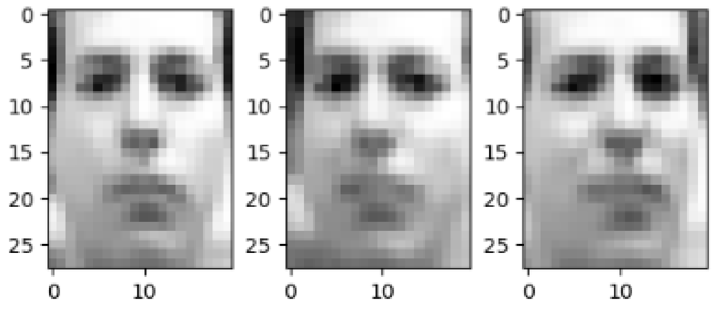

Figure 5: Frey faces made by using the coordinates of the centroids. These faces might be a bit more "blurry" then the original faces as these are "synthetic" faces of were the centroid is placed.

2.3 c)

By changing the amount of centroids to $N \in \{2, 4, 10\}$ we get,

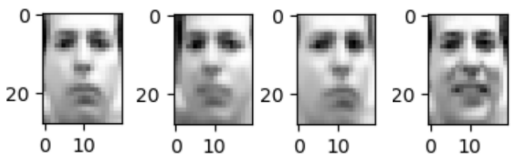

Figure 7: Frey faces with N = 4 using the position of the centroids

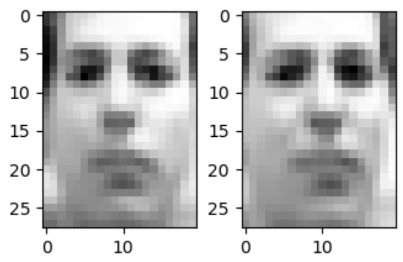

Figure 6: Frey faces with N = 2 using the position of the centroids

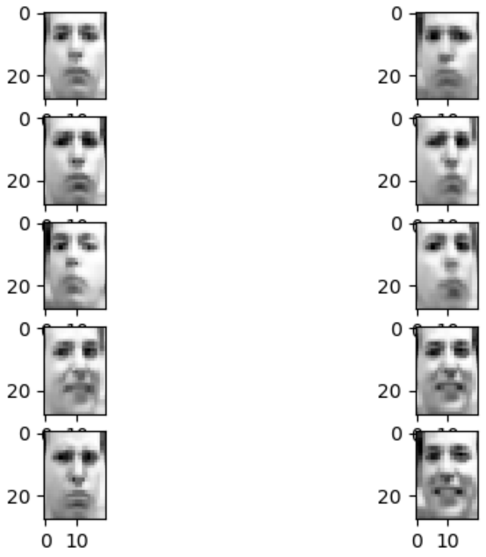

Figure 8: Frey faces with N = 10 using the position of the centroids

If we inspect the same image from N = 10 and N = 3, we can see that the one with N = 3 is less sharp,

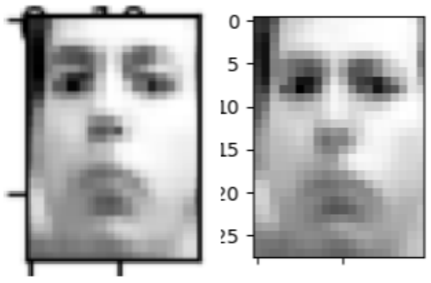

Figure 9: N = 10 to the left, N = 3 to the right using the position of the centroids

The reason for this is because when we groups with less centroids we center our centroids to the mean of data which contains the different faces, this makes the face a mix of different faces. When we use 10 centroids, and if we assume that the different frey faces are easy to group, then all the centroids will be made of mostly that ”face coordinate” which makes the picture more sharp. If we plot the border cases, we get

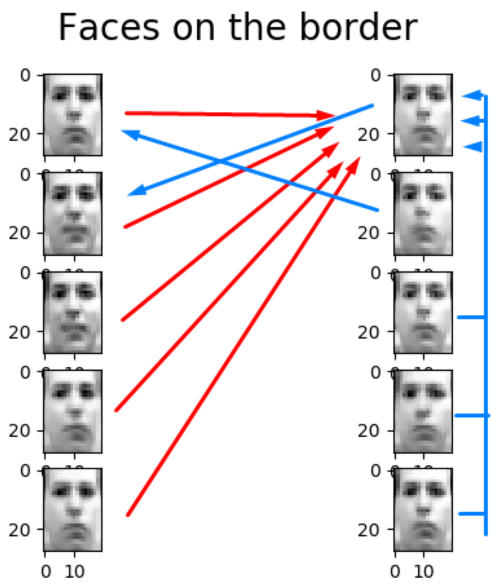

Figure 10: N = 10 using the position of the cluster border, here we see which cluster that borders to the other cluster, column1: {0,2,4,6,8}, column2: {1,3,5,7,9}(cluster nr.). It looks like most faces transition to the default face at the top right. In examples discussed in figure 14, 15 and 16 we have taken other units vector to get more border cases. These border cases are ”synthetic” as they are not real faces because we use the coordinate of the border.

These are the images which the clusters are having a hard time to cluster with good *confidence*. As we see in figure(10) the images are very similar to each-other which causes this ambiguity. In this case we only used one of vector for each centroid even though they had N vectors each, this generality is done using the,

```
border_control_all_boundaries(centroid_placement)
```

function, although this process takes a long time and will result in alot of images so we have used the function,

```
border_control(centroid_placement)
```

. Finding the border cases were done by finding the vector from each centroid to another,

$$\mathbf{v}_i = \mathbf{r}_i + \lambda \hat{\mathbf{d}}_{i,j}$$

Here $\mathbf{v}$ is the vector line we follow as we traverse from centroid i to centroid j, $\mathbf{r}$ is the vector to the centroid we want to traverse from, λ is a scalar, and d is the unit vector given by,

$$\hat{\mathbf{d}}_{i,j} = \frac{\mathbf{r_i} - \mathbf{r_j}}{||\mathbf{r_i} - \mathbf{r_j}||} \tag{4}$$

And the code is given in appendix. Keep in mind that these images are just the bordering coordinate and not necessarily an actually images from the data, we could use this information to keep in mind what kind of picture would be a border-case. We could find the nearest image to the traversing point to find the actual image border case.

We can also look at the ones which are far away from the centroid which could be of interest, as we can see in the distance plot below,

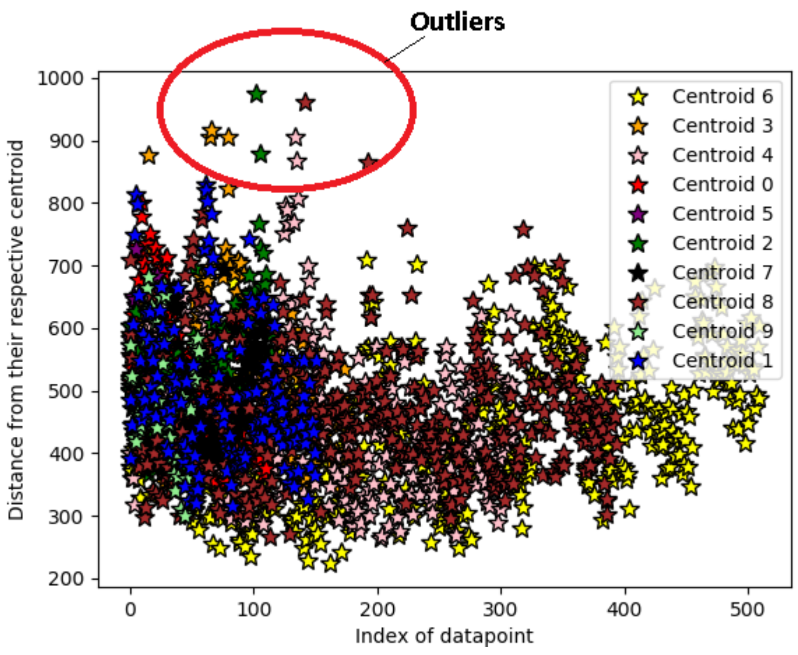

Figure 11: Here we see all the data-points and their distances, we can observe some points are abnormally far away from a centroid, we use this as the boundary centroids.

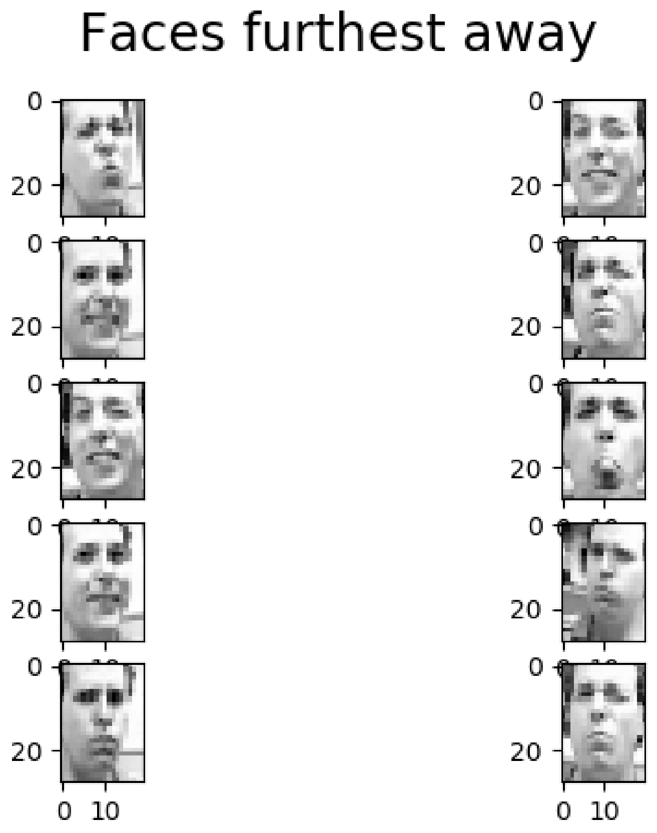

Figure 12: We can see that there are some noise in the background as the face doesn't cover the whole image, this may cause the images to be further away from the rest of the images in each centroid.

If we take a look at figure(12) below we can see that the faces have some artifacts or some background noise, this makes the images alot different than the rest of the images which sets them far away from the centroid. There could also be the case that the frey face is just alot more different than the other faces in the data. We can also observe that some of the faces look very similar which means that the the features of these images where hard to cluster. In figure(11) below we have the faces which are closest to the centroids,

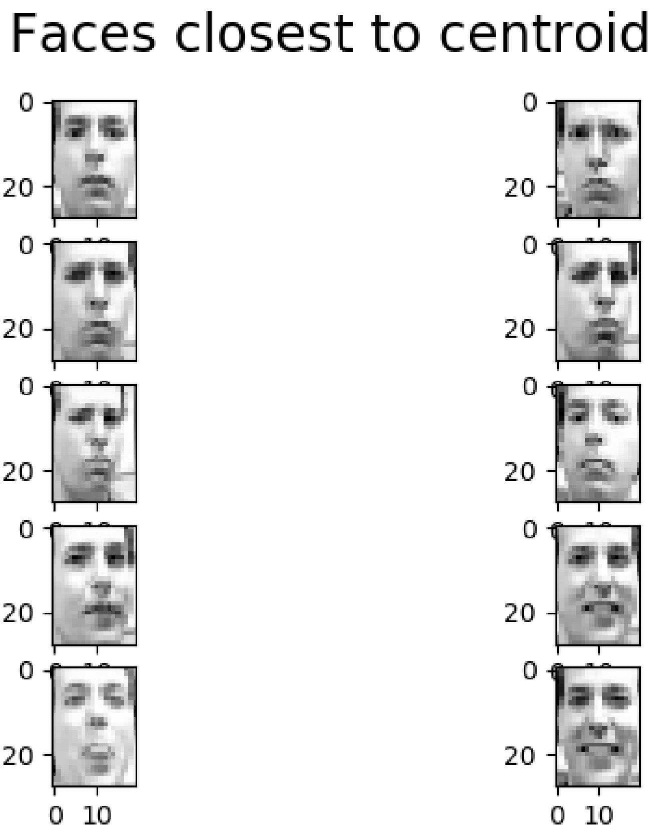

Figure 13: N = 10, images closest to the centroids, here we see that we have less background noise in the images then in figure(12) assuming most of the images are like these faces, then the centroids will center around these for which they have done.

In figure(14) below there is a more detailed picture of how the images transition as we move to the border. If we take a look at the borders and compare with $N = 2$, $N = 4$ and $N = 10$,

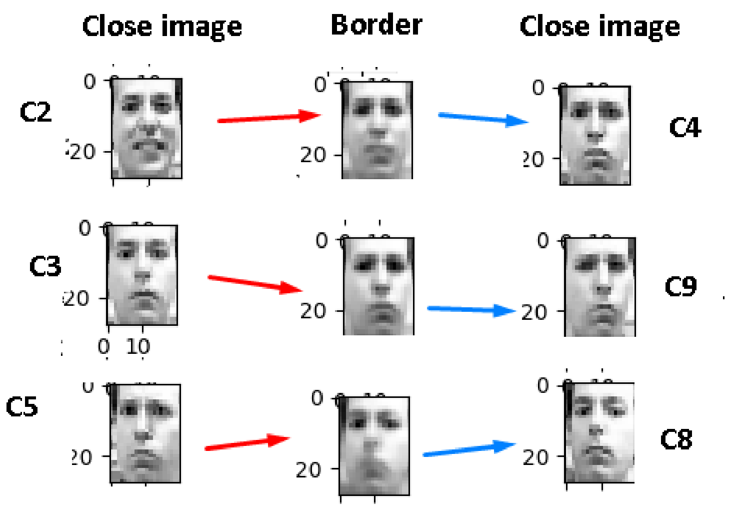

Figure 14: With N = 10 we have taken out some of the border cases, as we can see, the faces transition from being a face with a given feature to the border face which has a "middle ground" face between both the close image faces which has a smiling or some other feature at the different centroids.

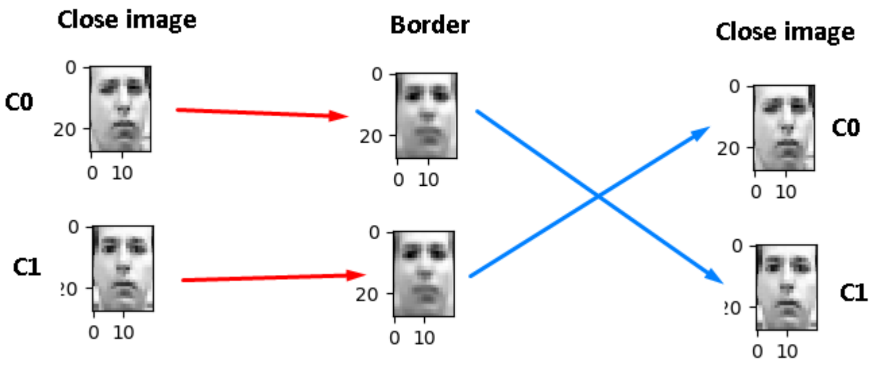

Figure 15: With N = 2 we see that the transition is not as good as with N = 10, this is because the borders are less defined with less centroids. It might be worth looking at the actual image close to the border to see the difference.

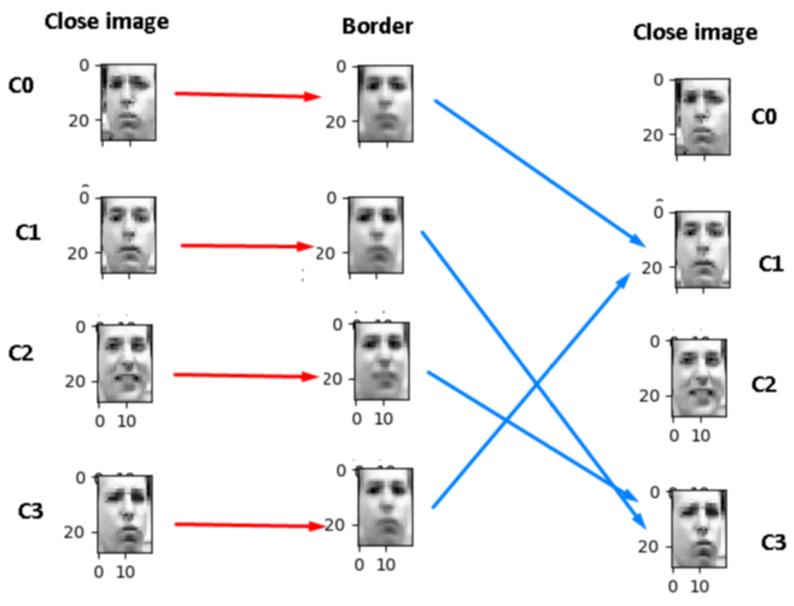

Figure 16: With N = 4 we see that the transition is better towards the image in the bordering cluster than the N = 2. Here we picked out one of the unit vector from each, and by following the vector from the centroid we met another centroid border which was in the way, therefore we did not have a border case in this particular case.

3 Appendix

Machine learning module

```
import numpy as np
import random
import matplotlib.pyplot as plt
import shutil
import os
import pandas as pd

# Hold global variables for error function
reset = 0
error_func_holder = []
error_index = []
def error_func(error):
    """
    Sums up alle the distances for each centroids and stores
                                  it in global variable to
                                  plot
    at the end of algorithm.
    """
    global error_func_holder
    summed = 0
    for key in error:
        summed += np.sum(error[key])
    error_func_holder.append(summed)








def get_distance(data, centroids, N):
    """
    Find distance between the datapoints to the centroids.
    """

    # find shape
    rows_data, cols_data = data.shape
    temp = np.zeros((rows_data,N))
    # fill temporary matrix with the distances in the N
                                  dimensional space of the
                                  dataset
    for length, row in enumerate(data):
        for cent, row_cent in enumerate(centroids):
            # [centroid1, centroid2]
            #[length    , length]
            temp[length, cent] = np.linalg.norm(row-row_cent)

    return temp



def get_cents(data, temp, label = [None]):
    """
    Assigns the datapoints which are nearest a centroid and
                              put them in a dictionary
    which belongs a the centroid.
    """
    groups = {}
    error_dist = {}
    label_dic = {}
    for i, row in enumerate(temp):
        #assign centroid to datapoints with lowest distance
                              to a dictionary
        # argmin will give index of the lowest value (index =
                              centroid)
        centroid_index = np.argmin(row)


        #if not exist in centroid dictionary assign new list
        if centroid_index not in groups:
            groups[centroid_index] = []

        if centroid_index not in error_dist:
            error_dist[centroid_index] = []

        #fills up the centroid with the datapoints
        groups[centroid_index].append(data[i,:])

        # If enabled (label != None) then add all the labels
                              to the centroids
                              aswell.
        if label[0] != None:
            if centroid_index not in label_dic:
                label_dic[centroid_index] = []
            label_dic[centroid_index].append(label[i])


        #fills up the error dictionary for the cost function
        error_dist[centroid_index].append(row[centroid_index]
                              )

    for key in groups:
        #Make them numpy arrays to be able to use numpy on
                              the datapoints
        groups[key] = np.array(groups[key])
    error_func(error_dist)

    return groups, label_dic



def get_means(data, groups):
    """
    Get the mean of the clusters and assign the centroids
                                   this new coordinate in a
    dictionary.
    """
    features = len(data[0])
    groups_mean = {}
    for i in groups:
        for n in range(features):
            if i not in groups_mean:
                groups_mean[i] = []
            groups_mean[i].append(np.mean(groups[i][:,n]))

    return groups_mean


def initate_cents(data, N):
    """
    Make random centroids in the vectorspace of the
                               datapoints
    """
    global reset
    features = len(data[0])
    # map the min and max value of the dataset to contain the
                                 centoid spawn into
    centroids = np.zeros((N,features))

    # place centroids in N dimensional space
    for j in range(N):
        for i in range(features):
            # make random coordinate
            rand = np.random.choice(data[random.randint(0,len
                                      (data)-1)],
                                      replace=False)
            centroids[j,i] = rand
    reset = 0
    return centroids


def remake_cents(data, groups_mean, N):
    """
    Since the distance function uses matrix, we need to fill
                                   up a new
    matrix with the new mean coordinates.
    """

    global error_func_holder
    global error_index
    features = len(data[0])
    centroids = np.zeros((N,features))

    try:
        # assign centroids to new matrix for finding distance
                                        if not all centroids
                                       (N) have atleast one
        # data point assigned to it, this test will fail and
                                       go to except to make
                                       new random centroids.
        for j in range(N):
            centroids[j,:] = groups_mean[j]

    # If some centroids fails tp find a cluster we make new
                                   random centroids
    # this could happen there is one centroid which didnt
                                   have a closest datapoint
    except:
        if not os.path.exists('plots'):
            os.makedirs('plots')
        # remove plots added to folder
        shutil.rmtree('plots')
        if not os.path.exists('plots'):
            os.makedirs('plots')

        error_index.append(len(error_func_holder)-1)
        # initiate new random centroids
        return initate_cents(data, N)

    return centroids



def make_plots(savefi, groups, groups_mean, N, savefigure =
                                   False):
    """
    If savefigure = True then plot the dataset and centroids
                                   with their respective
                                   color
    and save the figures.
    """
    plt.figure(figsize = (15,10))
    colors = ['red','blue','green','orange','pink','purple','
                                   yellow','black','brown','
                                   lightgreen','k', 'w','c','
                                   lightblue']
    for i in groups:
        plt.plot(groups[i][:,0],groups[i][:,1], 'o', color =
                                       colors[i],
                                       markeredgecolor= '
                                       black')


    for i in groups_mean:
        plt.plot(groups_mean[i][0],groups_mean[i][1],'^',
                                       color = colors[i],
                                       markersize = 20,
                                       markeredgecolor= '
                                       black')

    #make new folder if it does not exist (for plots)
    if not os.path.exists('plots'):
        os.makedirs('plots')
    # # Save plots
    # make_plots(savefi, groups, groups_mean, N)
    plt.title('K-means plot', fontsize = 20)
    if savefigure == True:
        plt.savefig('plots/k-means_%d_cents%d'%(savefi,N))
    plt.close()



def k_means(data, N, label = [None], savefigure = False,
                               max_iters = 150):
    """
    This is the main k-means algo, this is the function to
                                   run,
    this function "initate_cents", "get_distance", "get_cents
                                   ", "remake_cents"
                                   functions.

    returns grousp dictionary, centroid placement dictionary,
                                    and labels (labels are
                                   empty dictionary if
    labels are not given)
    """

    #make new folder if it does not exist (for plots)
    if not os.path.exists('plots'):
        os.makedirs('plots')
```

```
# make a loader for to see that the program is working
loader = ['|','/','-','|','/','-']

# copy to avoid pointer
data = np.copy(data)

rows_data, cols_data = data.shape

# Make random placed centroids
centroids = initate_cents(data, N)

#some variables to change in the whole loop to keep track
                                of loader etc.
change = 0
savefi = 0
load = 0
global reset

# Begin iteration
while True:

    # Find distance from all points
    temp = get_distance(data, centroids, N)

    # Group the data to the centroids
    groups, label_dic = get_cents(data, temp, label)

    if reset >= 2:
        make_plots(savefi, groups, groups_mean, N,
                                        savefigure)

    # Get mean coordinates for the centroids
    groups_mean = get_means(data, groups)

    # make matrix with coordinates for distance measure
    centroids = remake_cents(data, groups_mean, N)



    # if converged or reached max iterations return the
                                groups of clusters
    if change == np.sum(temp) or savefi == max_iters:
        global error_index
        global error_func_holder
        # get the values from error function where new
                                        centroids where
                                        generated
        error_values = [error_func_holder[x] for x in
```

```
                                                error_index]

    #Plots the error function and the generated
                                                centroids
    plt.plot(error_func_holder,'--', label = 'Cost
                                                function')
    #plt.plot(error_index, error_values,'*', label =
                                                'Generated new
                                                centroids', color
                                                = 'red',
                                                markersize = 13)
    plt.title('Cost function of k-means', fontsize =
                                                20)
    plt.ylabel('Error', fontsize = 15)
    plt.xlabel('Iteration', fontsize = 15)
    plt.legend(loc = 'best')
    # save errorfunction plot
    plt.savefig('plots/errorfunction.png')
    plt.close()
    # return the centroids with their given dataset
                                                and the placement
                                                of the centroids
    return groups, groups_mean, label_dic


# To change name of savefig file
savefi += 1
reset += 1
change = np.sum(temp)

# The loader
load += 1
try:
    print('{0} {0} {0} Generating k-means {0} {0} {0}
                                                '.format(loader[
                                                load]), end = '\r'
                                                )
except:
    load = 0
    print('{0} {0} {0} Generating k-means {0} {0} {0}
                                                '.format(loader[
                                                load]), end = '\r'
                                                )
```

```
def plot_centroids(groups, title , v, h):
    """
    Plots the images given.
    """
    for i in sorted(groups):

        plt.subplot(5,2,i+1)
        number = groups[i]
        number = np.reshape(number,(v,h))
        plt.imshow(number, cmap = 'Greys_r')
        plt.suptitle(title, fontsize = 20)
    plt.savefig('plots/%s.png'%title)
    plt.close()




def feature_reduction(X):
    """
    Reduced the features of data to two dimensions
    """
    # Get eigenvalues and eigenvector
    val, E = np.linalg.eig(X)
    # Sorting highest to lowest eigenvalues
    idx = val.argsort()[::-1]
    val = val[idx]
    # Taking square root of the diagonal eigenvalue matrix
    Dsqr = np.sqrt(np.diag(val))
    print(Dsqr)
    # Taking the inner product to form data-set Z
    Z = np.dot(E, Dsqr)
    # Return two features
    return Z[:,:2]




def minmax_control(groups, centroid_placement):
    """
    Finds the datapoints closest to the centroids and the
                                    ones on the border
    returns two dictionaries, first is datapoint closest to
                                    centroids and second
                                    dictionary
    is the ones on the border. The plots are saved to folder
                                    /plots
    """
```

```
colors = ['red','blue','green','orange','pink','purple','
                                     yellow','black','brown','
                                     lightgreen','k', 'w','c','
                                     lightblue']

distances_to_centroids = {}
for i in groups:
    for datapoint in groups[i]:
        if i not in distances_to_centroids:
            distances_to_centroids[i] = []
        distances_to_centroids[i].append(np.linalg.norm(
                                          datapoint -
                                          centroid_placement
                                          [i]))

borders = {}
closest = {}
for i in distances_to_centroids:
    if i not in borders:
        borders[i] = []
    max_index = np.argmax(distances_to_centroids[i])
    borders[i].append(groups[i][max_index])

    if i not in closest:
        closest[i] = []
    min_index = np.argmin(distances_to_centroids[i])
    closest[i].append(groups[i][min_index])



for i in distances_to_centroids:
    plt.plot(distances_to_centroids[i], '*',
                                       markeredgecolor= '
                                       black', markersize =
                                       10 , color = colors[i]
                                       , label = 'Centroid %d
                                       '%i)
plt.xlabel('Index of datapoint')
plt.ylabel('Distance from their respective centroid')
plt.legend(loc = 'best')
plt.savefig('plots/Distance_Plot')
plt.close()

return closest, borders
```

```
def border_control(centroid_placement):

    """
    Finds the border cases from each centroid
    """

    # finds all the unit vectors
    borders = {}
    for i in centroid_placement:
        if i not in borders:
            borders[i] = []
            for l in centroid_placement:
                if i == l:
                    continue
                else:
                    borders[i].append((np.array(
                                                            centroid_placement
                                                            [l]) - np.
                                                            array(
                                                            centroid_placement
                                                            [i]))\
                    /(np.linalg.norm(np.array(
                                                            centroid_placement
                                                            [l]) - np.
                                                            array(
                                                            centroid_placement
                                                            [i]))))
    # dictionary for which cluster it has the boundary to
    bound_change_cluster = {}
    # dictionary for the position for which the boundary is
    bound_change_position = {}
    for indec in sorted(centroid_placement):
        # lock to start with cluster when we have found the
                                           boundary
        lock = 0
        while True:
            if lock != 1:
                # Find the current position on the vector
                try:
                    current_position = centroid_placement[
                                                            indec] + i
                                                            *np.array(
                                                            borders[
                                                            indec][
                                                            indec])
                except:
                    current_position = centroid_placement[
                                                            indec] + i
                                                            *np.array(
                                                    borders[
                                                    indec][0])
                # list to append all the distances to each
                                                 centroid from
                                                 the point on
                                                 the vector
                lst = []
                for j in sorted(centroid_placement):
                    lst.append(np.linalg.norm(
                                                    current_position
                                                     -
                                                    centroid_placement
                                                    [j]))
                # find which centroid the point belongs too
                                                 by finding the
                                                  minimum
                                                 distance
                # and chech which index(cluster) it belongs
                                                 to
                index_min = np.argmin(lst)
                i += 0.1
                if indec != index_min:
                    if indec not in bound_change_position:
                        bound_change_position[indec] = []
                        bound_change_cluster[indec] = []
                    bound_change_cluster[indec].append(
                                                    index_min)
                    bound_change_position[indec].append(
                                                    current_position
                                                    )
                    lock = 1
            else:
                break

    return bound_change_cluster, bound_change_position



def border_control_all_boundaries(centroid_placement):

    """
    Finds the border cases from each centroid
    """

    # finds all the unit vectors
    borders = {}
    for i in sorted(centroid_placement):
        if i not in borders:
            borders[i] = []
            for l in sorted(centroid_placement):
                if i == l:
                    continue
                else:
                    borders[i].append((np.array(
                                                    centroid_placement
                                                    [l]) - np.
                                                    array(
                                                    centroid_placement
                                                    [i]))\
                    /(np.linalg.norm(np.array(
                                                    centroid_placement
                                                    [l]) - np.
                                                    array(
                                                    centroid_placement
                                                    [i]))))
    # dictionary for which cluster it has the boundary to
    bound_change_cluster = {}
    # dictionary for the position for which the boundary is
    bound_change_position = {}
    for indec in sorted(centroid_placement):
        for border_values in borders[indec]:
            # lock to start with cluster when we have found
                                                    the boundary
            lock = 0
            while True:
                if lock != 1:
                    # Find the current position on the vector
                    current_position = np.array(
                                                    centroid_placement
                                                    [indec]) +
                                                     i*np.
                                                    array(
                                                    border_values
                                                    )
                    # list to append all the distances to
                                                    each
                                                    centroid
                                                    from the
                                                    point on
                                                    the vector
                    lst = []
                    for j in sorted(centroid_placement):
                        lst.append(np.linalg.norm(
                                                        current_position
                                                         -
                                                        centroid_placement
                                                        [j]))
                    # find which centroid the point belongs
```

```
                                                       too by
                                                       finding
                                                       the
                                                       minimum
                                                       distance
                    # and chech which index(cluster) it
                                                       belongs to
                    index_min = np.argmin(lst)
                    i += 0.1
                    if indec != index_min:
                        if indec not in bound_change_position
                                                          :
                            bound_change_position[indec] = []
                            bound_change_cluster[indec] = []
                        # cluster represents which the
                                                          centroid

                                                          borders
                                                           to
                        bound_change_cluster[indec].append(
                                                          index_min
                                                          )
                        # position is the border positions
                        bound_change_position[indec].append(
                                                          current_position
                                                          )
                        lock = 1
                else:
                    break

    return bound_change_cluster, bound_change_position
```

Task 1

```


import machinepy as ml
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

B = np.genfromtxt('city-inner-sweden.csv') # this is NXN
names = pd.read_csv('city-names-sweden.csv', header=None)
names = np.array(names)

Z = ml.feature_reduction(B)



for x,y in zip(names, Z):
    plt.plot(y[1], y[0],'^',markersize = 10 , markeredgecolor
                                    = 'black') # y is the MDS
                                    output
    plt.annotate(x[0], ([y[1],y[0]]))
plt.show()
```

Task 2

```

import machinepy as ml
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

data = np.genfromtxt('frey-faces.csv')


N = 10


groups, centroid_placement, labels = ml.k_means(data, N,
                              savefigure = False, max_iters
                              = 5000)

closest, furthest = ml.minmax_control(groups,
                              centroid_placement)

bound_change_cluster, bound_change_position = ml.
                              border_control(
                              centroid_placement)
#bound_change_cluster1, bound_change_position1 = ml.
                              border_control(
                              centroid_placement, 1)





# (A) This will find all the border cases
#bound_change_cluster, bound_change_position = ml.
                              border_control_all_boundaries(
                              centroid_placement)

# prints out the border of which centroids border to Key is
                              centroid which
# we iterate from, the values are the centroids we go over to
                              , index of the
# list tells which index corresponds to what data-point in
                              the bound_change_position
# array it is.
print(bound_change_cluster)
```

```
ml.plot_centroids(centroid_placement,'Faces from centroids',
                              28, 20)
ml.plot_centroids(closest,'Faces closest to centroid', 28, 20
                              )
ml.plot_centroids(furthest,'Faces furthest away', 28, 20)
ml.plot_centroids(bound_change_position,'Faces at boundary',
                              28, 20)
```

References

Ethem Alpaydin. *Introduction to machine learning.* The MIT Press, third edition, 2014.

Kriegel, Hans-Peter; Schubert, Erich; Zimek, Arthur. (2016). ”The (black) art of runtime evaluation: Are we comparing algorithms or implementations?”., 2016. DOI:10.1007/s10115-016-1004-2.