Assignment 1 in DD2434 Machine Learning, Advanced Course

Marie-Ange Stefanos

November 2021

1 Assignment 1

1.1 Principal Component Analysis

Question 1.1.1

Centering the data means translating the data to the origin of the graph, that is having the origin as the center of gravity of the data. Besides, PCA is a regressional model without intercept so the principal components always pass through the origin.

Hence, if the data are not centered, the principal components might go through the point cloud with incorrect slopes, as can be seen on figure 1 that comes from [2].

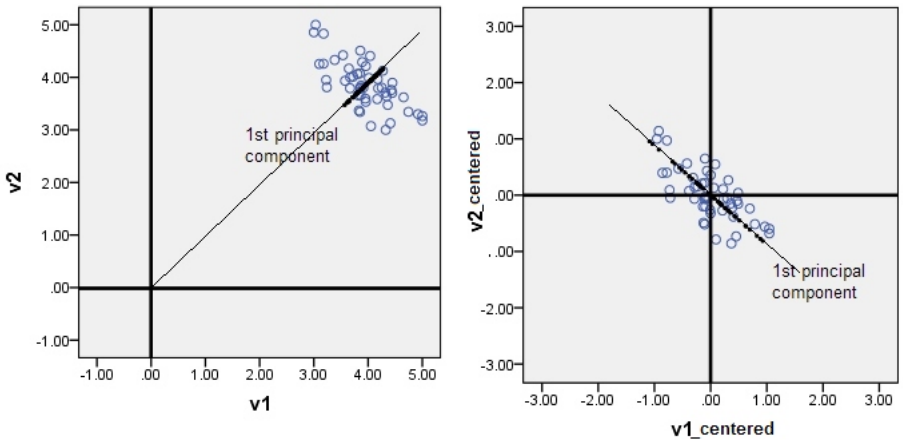


Figure 1: Computing first principal component without (left) and with (right) centering

However, it is noteworthy that the covariance matrix stays the same whether the data are centered or not. Hence, if PCA uses the eigen-decomposition of the covariance matrix, the data do not need to be centered, unlike SVD.

That is why, applying **PCA using the SVD (singular value decomposition) requires to center the data**, and not centering the data would cause an incorrect first principal component.

Question 1.1.2

Using the SVD operation on the matrix Y, the minimization of the reconstruction error is equivalent to maximizing this quantity:

$$\mathrm{Tr}(Y^T W W^T Y) = \mathrm{Tr}(V \Sigma U^T W W^T U \Sigma V^T)$$

This can be solved by taking $W = U_k$ (as it will be showed in question 1.2.4).

In the case of Y^T, using the fact that $Y^T = V \Sigma' U^T$ where Σ' only differs from Σ in terms of size, we want to maximize the expression:

$$\mathrm{Tr}(Y^T W W^T Y^T) = \mathrm{Tr}(U \Sigma' V^T W W^T V \Sigma' U^T)$$

which can be solved with $W = V_k$ similarly to previously.

So we have shown that using a single SVD operation on Y, PCA can be performed on Y and Y^T.

Question 1.1.3

We want to explain why the use of the pseudo-inverse is a good choice to obtain the inverse mapping of the linear map.

Assuming that we have $y = Wx$ where $W \in \mathbb{R}^{dxk}$ is a matrix with orthonormal columns, we want to find a way to deduce x.

The first thing is to note that W is a non-squared matrix so it cannot be invertible.

Since W is orthonormal, it has linearly independant columns, so the matrix $W^T W$ is invertible (that is $W^T W = I_k$) and $W^+ = (W^T W)^{-1} W^T = W^T$.

Moreover, by computing SVD on Y, we have : $Y = U \Sigma V^T$. Besides, the best of rank-k approximation for the matrix Y is $Y_k = U_k \Sigma_k V_k^T$.

Also, the minimisation of the MSE (mean squared error), that is the reconstruction error leads to $W = U_k$.

Therefore, x is obtained by SVD and approximating Y:

$$x = W^+ y = W^T y = U_k^T y$$

All is needed is to transpose U_k obtained by SVD, thanks to the peuso-inverse.

Question 1.1.4

Applying PCA using the criterion of variance maximization means that PCA finds the direction with the largest variance out of the overall variance in the data space. Another way to say it is that we want the principal components to explain the highest variance as possible. That is why in this case, talking about maximizing the variance concerns the overall or total variability, that is the sum of each feature variances.

That explains why maximizing the variance is maximizing the trace of the covariance matrix C_{XX} of the embedded data X, not taking into account the covariances values between different features, that is $C_{XX}[i, j]$ entries of the matrix for $i \neq j$.

As we define $W \in \mathbb{R}^{d \times k}$ as the matrix that satisfies $y = Wx \in \mathbb{R}^d$ where W is an orthogonal matrix, we have $x = W^+ y = W^T y \in \mathbb{R}^k$.

Let $Y = U\Sigma V^T \in \mathbb{R}^{d \times n}$ be the **economy-size** SVD (Singular Value Decomposition) of the matrix y. According to appendix 1 of [4], $\Sigma \in \mathbb{R}^{dxd}$ (with $p \leqslant n$) has no useless zeros. It is thus a diagonal matrix with non-zeros values on the diagonal so Σ is invertible. Moreover, by definition of the SVD, U and V are orthogonal matrices : $U^T U = I_d$ and $V^T V = I_d$.

Using the usual covariance estimator, we have :

$$\mathrm{Tr}(C_{XX}) = \frac{1}{n-1} \mathrm{Tr}(X^T X) \propto \mathrm{Tr}(X^T X)$$

We can then compute and maximize the following expression :

$$
\begin{aligned}
\mathrm{Tr}(X^T X) &= \mathrm{Tr}(Y^T W W^T Y) \\
&= \mathrm{Tr}((U\Sigma V^T)^T W W^T U\Sigma V^T) \\
&= \mathrm{Tr}(V\Sigma U^T W W^T U\Sigma V^T) \\
&= \mathrm{Tr}(V\Sigma U^T W W^T U\Sigma V^T) \\
&= \mathrm{Tr}(V\Sigma U^T W W^T U\Sigma V^T) \\
&= \mathrm{Tr}(V^T (V\Sigma U^T W W^T U\Sigma V^T) V) \\
&= \mathrm{Tr}(\Sigma U^T W W^T U\Sigma) \\
&= \|\Sigma U^T W\|_2
\end{aligned}
$$

We use at line 6 the following property of the similarity variance property of the trace :

$$\mathrm{Tr}(P^{-1}AP) = \mathrm{Tr}(A)$$

The last line involves the Frobenius norm.

Thus, maximizing the variance is maximizing $\|\Sigma U^T W\|_2$, which is maximized when ΣU^T and W are colinear. Since Σ is a diagonal matrix with non-zeros diagonal entries that contains the singular values of Y in the descendant order, multiplying it by U will not change the direction of the vectors constituted by the columns of U. That is why, the last expression reaches its maximum when the k columns of W are colinear with the columns of U that are associated with the k largest singular values in Σ.

Thus, $W = U_k^T$ where $k \leqslant d$ is the embedding dimension, which is the same result as with the criterion of minimizing the reconstruction error.

1.2 Multidimensional Scaling (MDS) and Isomap

Question 1.2.5

Without using the "double centering trick", the embedding data would have 2 degrees of freedom which are translation and rotation (see this chapter of deep learning course given by D. Jacobs in the University of Maryland : [3]). So there are an infinity of solutions for this problem.

"Double centering trick" consists of subtracting from each element of the distance matrix the mean of the corresponding row and the mean of the corresponding column, and then adding the mean of all entries (as explained in section 4.2.2 of [4]).

Therefore, using double centering trick enables to center the embedded data around the origin. The translation degree of freedom is thus removed from the problem and only one solution left is possible, up to a rotation.

In order to have a better visualization of the problem, the distance and similarity matrices can be plotted thanks to heatmaps, as can be seen on the figure 2 and 3, using the first 20 cities more populated in the world as data set to compute the distance and similarity matrices.

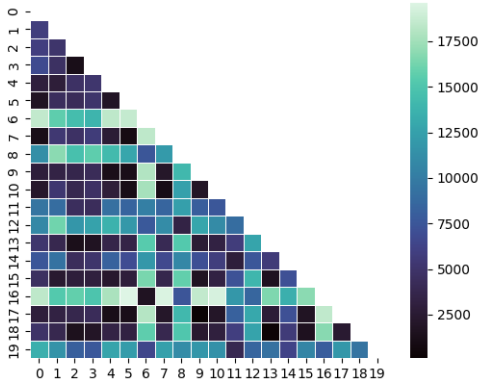


Figure 2: Distance matrix

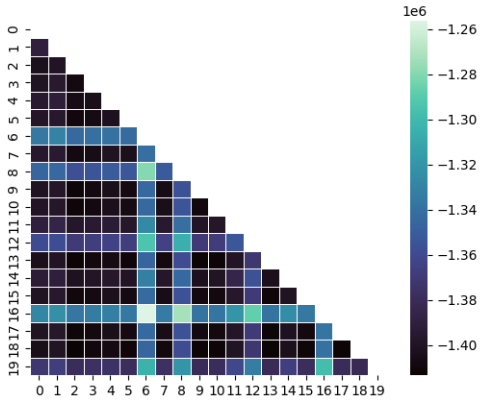


Figure 3: Similarity matrix

Since both matrices are symmetric, only the low triangle part of each have been displayed here. It can be noteworthy that similarity matrix using "double centering trick" underlines more the cities points that are close than the distance matrix.

Question 1.2.6

As explained previously in question 1.2.5, there are an infinity of solutions to the problem of dimensionality reduction because of translation and rotation degrees of freedom. Assuming that we accept the rotation degree of freedom, the embedded data can be translated ans still solve the problem. That is why, adding a constraint to the problem will force the problem to have only one solution. That is exactly what we have done in the previous question with "double centering trick" and what we can do here with the "first point trick". But instead of focusing on the origin, we will concentrate on the first data point, that is the first column of Y matrix. Indeed, we substract d_{1i} and d_{1j} to each entry d_{ij} of the distance matrix.

Another way to solve this multiple solutions issue is by using the "first point trick", that is centering the problem around the first point of the data matrix Y (instead of the origin), which would have the same consequence as before : set the embedded data around a determined point will constrain the problem and only one solution will be acceptable for the problem, with the rotation exception.

A mathematical way to go from "double centering trick" to "first point trick" is to do the translation using the point $z_i = y_i - y_1$ which is the difference between the i^{th} data point and the first point. Introducing it in the double centering trick equation shows by substitution that both equations are equivalent in order to remove the translation degree of freedom.

Question 1.2.7

We want to show that classical MDS when Y is known and PCA are two equivalent methods. We will show that both give the exact same results and thus minimize the same criterion, as explained in section 4.2.2 of [4]. Then, we will see in which cases each method is the most efficient. As mentioned in the assignment, we will focus on PCA involving SVD and MDS involving eigen-decomposition.

First and foremost, we will show that solutions given by PCA and MDS are equal.

PCA Applying SVD to Y^TY gives:

$$Y^TY = U\Sigma^T V^T V \Sigma U^T = U\Sigma^2 U^T$$

Minimizing the reconstruction error leads to $W = U_k = UI_{d\times k}$ as showed in question 1.2.4.

Therefore, the solution is:

$$\hat{X}_{PCA} = W^T Y = I_{k\times d} U^T Y$$

MDS In order to compute the MDS solution, we need the following relation between C_{XX} and C_{YY}.

$$C_{YY} = \mathbb{E}(YY^T) = \mathbb{E}(U\Sigma V^T V \Sigma U^T) = W\mathbb{E}(XX^T)W^T = WC_{XX}W^T$$

Hence, equalizing the covariances estimator leads to $Y^TY = X^TX$. Besides, applying the eigen-decomposition to Y and identifying to SVD result gives:

$$Y^TY = V\Lambda_{MDS}V^T$$

with $\Gamma_{MDS} = \Sigma\Sigma$. The solution is therefore:

$$\hat{X}_{MDS} = I_{k\times d}\Lambda_{MDS}^{\frac{1}{2}} V^T$$

PCA-MDS equivalence Now we will show that $\hat{X}_{MDS} = \hat{X}_{PCA}$

$$\hat{X}_{MDS} = I_{k\times n}\Lambda_{MDS}^{\frac{1}{2}} V^T = I_{k\times n}(\Sigma\Sigma)^{\frac{1}{2}} V^T = I_{k\times n}(U^TU)\Sigma V^T$$

$$= I_{k\times n}U^T(U\Sigma V^T) = I_{k\times n}U^TY = \hat{X}_{PCA}$$

This calculation leans on the definitions of Λ_{MDS}, the orthogonal matrix U and the SVD on Y. This shows that PCA and MDS minimize the same criterion.

PCA vs. MDS regarding computational efficiency On the one hand, SVD on $Y \in \mathbb{R}^{d\times n}$ has a computational efficiency of $O(n^2d)$. On the other hand, the eigen-decomposition of Y^TY has computational complexity $O(n^3)$. Considering that PCA is based on SVD and MDS on eigen-decomposition, which method is the most efficient depends on the number of dimensions and the number of points. If the number of dimensions is not too high and the number of points is huge (d << n), PCA will be more efficient and thus prefered. Otherwise, if the number of points is reasonable but the number of dimensions is enormous (d >> n), then MDS would be better.

Question 1.2.8

In the Isomap method, the construction a neighborhood graph G can lead to a disconnected graph. Indeed, the construction is based on the use of k-NN (k-nearest neighbors) algorithm. In case of data that contains multiple classes/clusters or lying on multiple manifolds, the constructed graph can be disconnected if k is too small. The graph will results in several smaller disconnected graphs, for instance one per class or manifold.

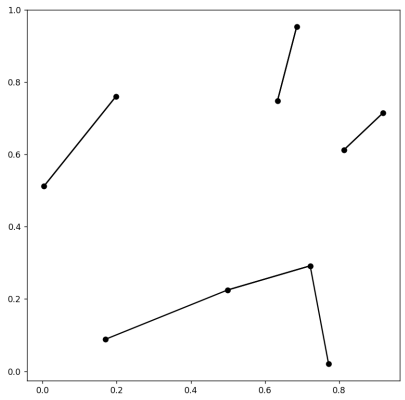


Figure 4: Random data set with applied 2_{NN}

As we can see on the figure 4, the constructed graph using k-NN with k=2 is disconnected. This dataset encompasses 10 random points in the plane.

In this case, the problem is that in Isomap algorithm, the graph G constructed by k-NN is used to compute the shortest path distance between all data points (using the Floyd-Warshall algorithm for instance). However, if the graph is disconnected, there are points that will not be connected, so the distance between them will be unknown. Therefore, it will not be possible to apply on the distance matrix since it will be incomplete.

Question 1.2.9

In order to patch the problem arising in the case of a disconnected neighborhood graph, we should find a way to obtain a connected graph by keeping the neighborhood relations between the points. One way to do so is to increase the value of k parameter of the k-NN algorithm, repeat this until the graph is connected, and chose the lowest k that makes a connected graph.

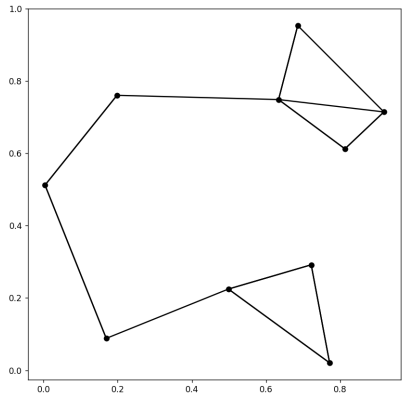


Figure 5: Random data set with applied 3_{NN}

Let us take the previous example to illustrate the solution. Here, we have increase k value until the graph was connected. The value k=3 was enough so we stopped here. If it was not, we would have iterated the operation and increased k until we obtain a connected graph.

Checking if the graph is disconnected is not a big deal since the distance matrix can be initialized with negative values for instance. As long as it contains at least one negative value, there will be some missing distance values, that is to say some paths do not exist, so the graph is disconnected. Hence, checking that the graph is disconnected is not an issue here.

Even if trouble concerning high values of k in k-NN may appear, this would solve the disconnected graph issue and would make the Isomap algorithm work correctly. This would conserve at the same time the relationships between data points.

1.3 PCA vs. Johnson-Lindenstrauss random projections

Question 1.3.10

This question is about comparing 2 linear mapping methods (PCA and Johnson-Lindenstrauss (JL) random projections) according 3 criteria: projection error, computational efficiency and intended use cases.

i. Projection error

The two approaches are chosen for different purposes.

JL random projections gives the low dimensional space that get the distances upto a given error, while PCA gives the *best* possible embedding, given the dimension of the embedded space.

Thus, JL random projection provide a worst-case pairwise garantee (that is for each pair of points), unlike PCA that gives the "best" embedding on average, which means that the total error between true similarity and projected similarity is minimized.

As showed in these experiments [1], the maximum accuracy obtained by using PCA is higher than the maximum accuracy obtained by using RP (random projections). Therefore, JL random projection and PCA do not solve the same problems. The choice of the algorithm depends not only on what parameters the user wants to focus on (error vs. dimension) but also on what type of error (pair-wise or total) and guarantees the user needs.

ii. Computational efficiency

Let's consider the data matrix is $Y \in \mathbb{R}^{d \times n}$ and the embedded data $X \in \mathbb{R}^{k \times n}$ where d is the number of dimensions and n the number of points. As said above, PCA has computational complexity $O(n^3)$, whereas it is $O(n \times k)$ for JL random projections. Thus, if the number of dimensions of the initial data is huge, JL random projections will be prefered. However, if the dataset is huge and the number of dimensions is not too high, then PCA will be prefered.

iii. Intended use cases

Several points that have not been already mentioned will be summarized here. With a sparse matrix, JL random projections will be prefered over PCA because it is faster. Hence, sparse data in a non linear subspace should be studied using JL random projections rather than PCA. Another point about very large data concerning memory, is that PCA need to hold a lot in memory, while it is not necessary the case for JL random projections. That is another situation where JL random projection should be chosen.

Conclusion about PCA vs. JLRP

PCA is more effective for severe dimensionality reduction, wheras JL is more suitable when a high number of dimensions is kept, although a high number is not always optimal accuracy as showed in the experiences led by Sampath Deegalla and Henrik Boström in this article [1]. They show that the use of PCA results in the highest accuracy for most of their datasets (90%).

1.4 Programming task — MDS

Question 1.4.11 (Data collection)

In this part we will use the data set provided here [5]. It contains in particular the cities name, latitude and longitude coordinates, the country and the population.

Since the raw data contains approximately 40 000 points, only the two most populated cities of each country have been extracted to have a useable subset. It now countains 431 cities.

In order to compute the distance between cities, we will use Haversine formula ((cf fig. 6) that determines the great-circle distance between two points on a sphere given their longitudes and latitudes. Hence, the distance matrix of the data can be computed by using Haversine formula and the deduced distance expression (cf fig. 7). It can be visualized thanks to a heatmap as can be seen on the figure 8. We can notice that the matrix is symmetric and that the diagonal of the distance matrix contains only zero values as it should, since they are the distances between a point and itself.

$$\operatorname{hav}\left(\frac{d}{r}\right) = \operatorname{hav}(\varphi_2 - \varphi_1) + \cos(\varphi_1)\cos(\varphi_2)\operatorname{hav}(\lambda_2 - \lambda_1)$$

Figure 6: Haversine formula

$$\begin{aligned} d &= 2r \arcsin\left(\sqrt{\operatorname{hav}(\varphi_2 - \varphi_1) + \cos(\varphi_1)\cos(\varphi_2)\operatorname{hav}(\lambda_2 - \lambda_1)}\right) \\ &= 2r \arcsin\left(\sqrt{\sin^2\left(\frac{\varphi_2 - \varphi_1}{2}\right) + \cos(\varphi_1)\cos(\varphi_2)\sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right)}\right) \end{aligned}$$

Figure 7: Distance using Haversine formula

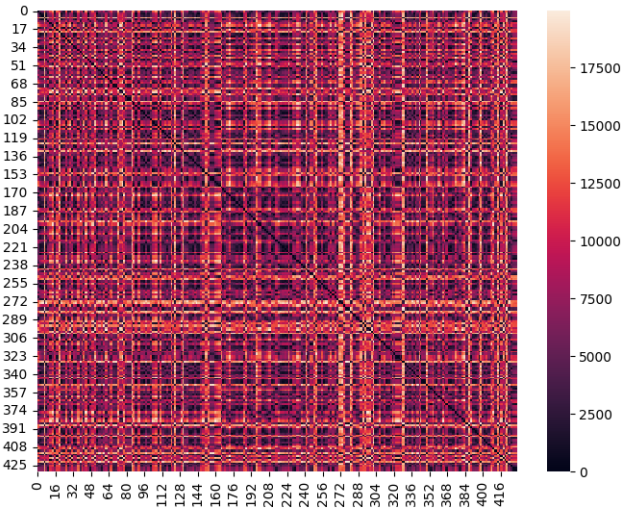


Figure 8: Distance matrix heatmap

Question 1.4.12 (Classical MDS)

Classical MDS have been implemented on the subset described in previous question. It is noteworthy that the double centering trick is handmade as can be seen in the code (see Appendix B). The coordinates of the embedded data have been defined by the real part of the results, since the eigen-decomposition can provide complex eigen vectors.

This implementation of classical MDS gives the reconstructed map illustrated on the figure 9. Each city is annotated by the ISO 3166-1 alpha-3 code of the respective country and is colored according to the continent it belongs to. The legend contains the following entries: AS (Asia), EU (Europe), AF (Africa), OC (Oceania), NA (North America), SA (South America), AN (Antartica) and "Not found" (for some countries for which no continent have been found in the chosen database). The continents have been found on an another database (on this website [6]) and a mapping have been done to associate each city of the first dataset to the continent it belongs to.

We can see that cities that belong to a same continent are indeed gathered in the same zone. That shows that close cities on the Earth have been represented close to each other on the scatter plot, so cities that are truly far from each other are also represented far from each other on this plot. Therefore, it seems like distances have been well preserved. To have a better idea of this "well" preserved, we have computed the mean error between the true distances between points and the distances that separate them in the embedded data. We have computed an mean error of 13km on this dataset when using classical MDS.

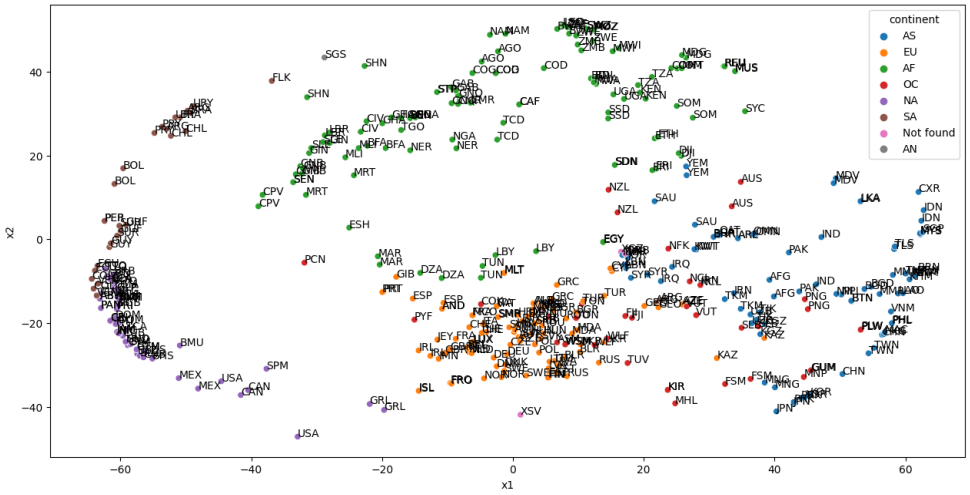


Figure 9: Reconstructed map obtained by classical MDS

As said previously in question 1.2.5, there are an infinity of solutions according to how you rotate the embedded data. That is the main reason why the result does not fit with the exact planisphere you have in mind. However, the results coherent to the extent that North America is to the north of South America and to the West of Europa, which is to the West of Asia and to the North of Africa. However, Oceania cities are strangely sparsed over Europe and Asian cities and should be improved.

Question 1.4.13 (Metric MDS)

For this question, metric MDS have been implemented using *manifold.MDS* function from *sklearn* module. It particularly has 2 parameters that we have tuned to study the corresponding results: the maximum number of iterations *max_iter* and the precision *eps*. As you can see in the appendix A.a, the results are quite the same up to a rotation whatever values you choose for those parameters, except when you take extreme values such as too low *max_iter* (for instance $max\_iter \leqslant 40$, see Appendix A.b) or too high *eps* (like $eps \geqslant 1$, see Appendix A.c).

However,unlike classical MDS, metric MDS gives better results concerning Oceania, as you can see on the figure 10 that shows the results obtained for $max = 3000$ and $eps = 1e-9$. The rest of the continents have similar results to what we have obtained with classical MDS.

The points are not annoted by the name of the countries for a better readability, although they are colored by continent as in the previous question. The title

of each plot contains the parameter taken by MDS function. It is noteworthy that *metric* argument is *True* by default so it is not necessary to precise it in the code.

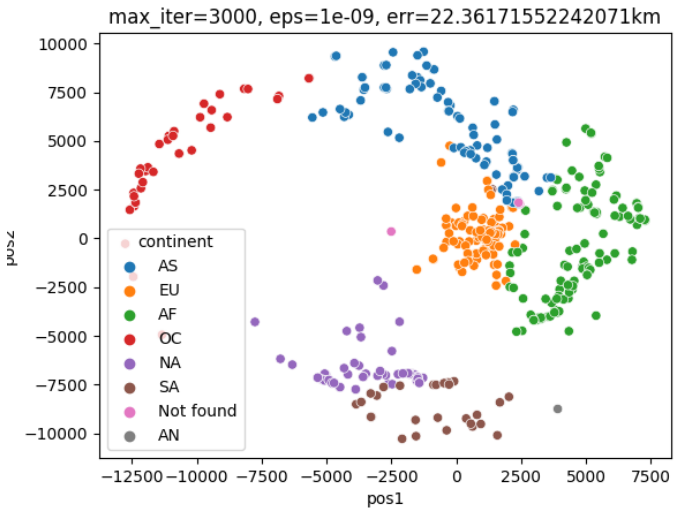


Figure 10: Reconstructed map obtained by classical MDS

The computed error is between 22 and 23km whatever the values for the parameters, which is higher than what we obtained with classical MDS but is still steady and low.

The code that was used to achieve this assignment can be found in Appendix B, including an example of script to use it and repeat all the results that have been obtained here.

References

[1] Sampath Deegalla and Henrik Boström. *Reducing High-Dimensional Data by Principal Component Analysis vs. Random Projection for Nearest Neighbor Classification*. URL: https://people.dsv.su.se/~henke/papers/deegalla06.pdf.

[2] *How does centering the data get rid of the intercept in regression and PCA?* 2012. URL: https://stats.stackexchange.com/questions/22329/how-does-centering-the-data-get-rid-of-the-intercept-in-regression-and-pca.

[3] David W. Jacobs. *Multidimensional Scaling: More complete proof and some insights not mentioned in class*. URL: http://www.cs.umd.edu/~djacobs/CMSC828/MDSexplain.pdf.

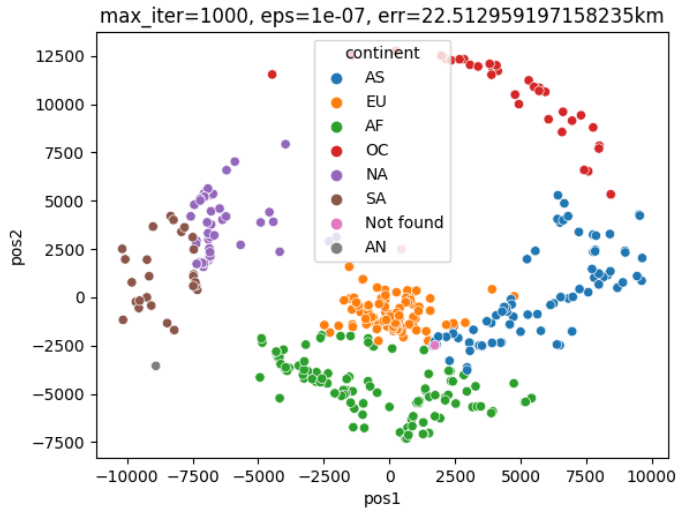
[4] Michel Verleysen John A. Lee. *The Principles of Quantum Mechanics*. Springer Publishing Company, Incorporated, 2007. ISBN: 978-0-387-39350-6.

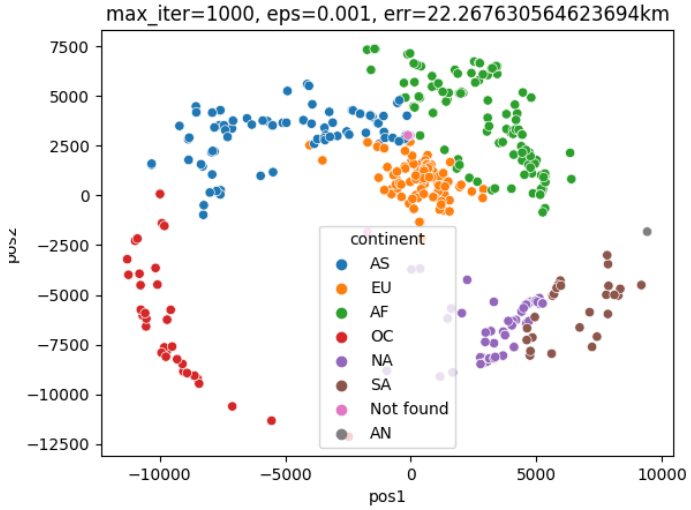
[5] *Simple Maps*. URL: https://simplemaps.com/data/world-cities.

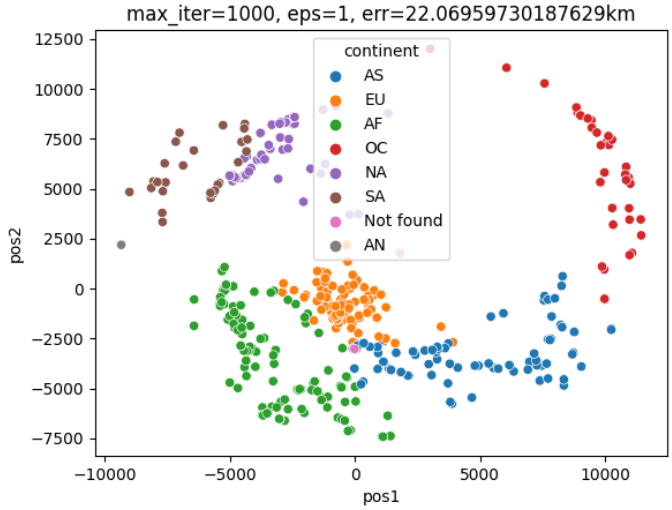
[6] Steve Withington. *Country and Continent Codes List*. URL: https://gist.github.com/stevewithington/20a69c0b6d2ff846ea5d35e5fc47f26c#file-country-and-continent-codes-list-csv-csv.

Appendix A

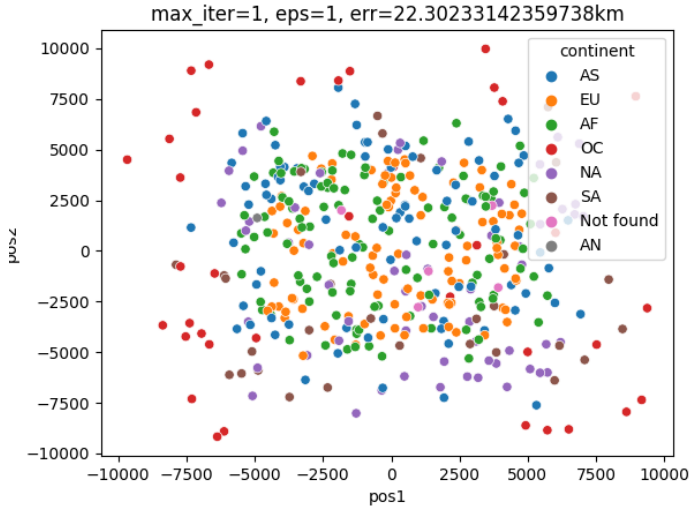
a. Different parameter values for acceptable results

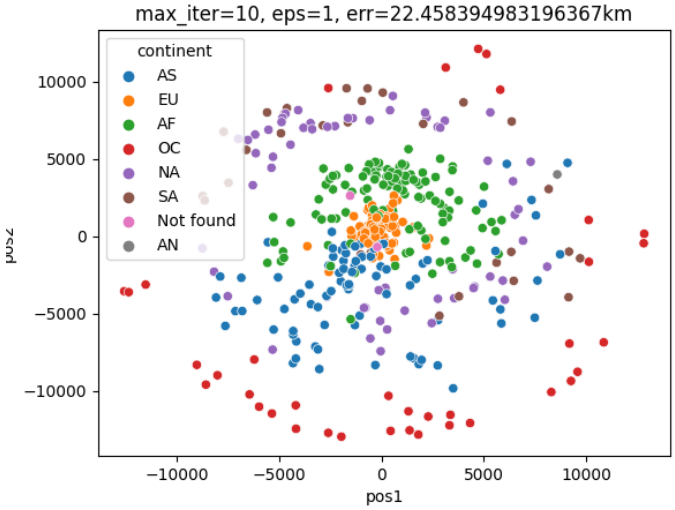


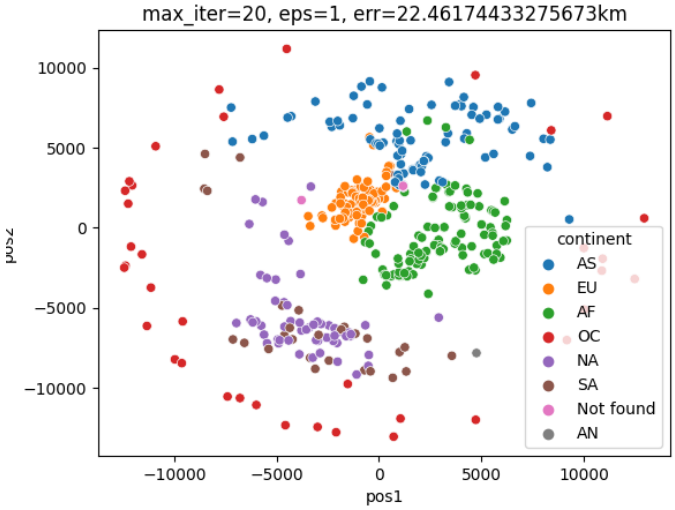




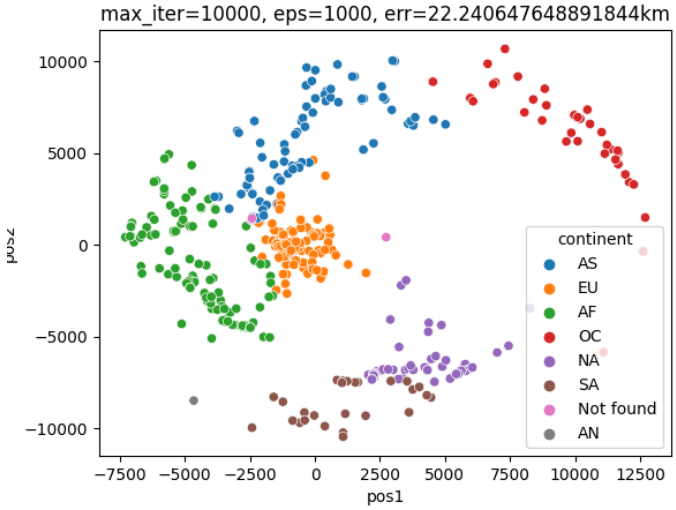
b. Varying *max_iter* parameter

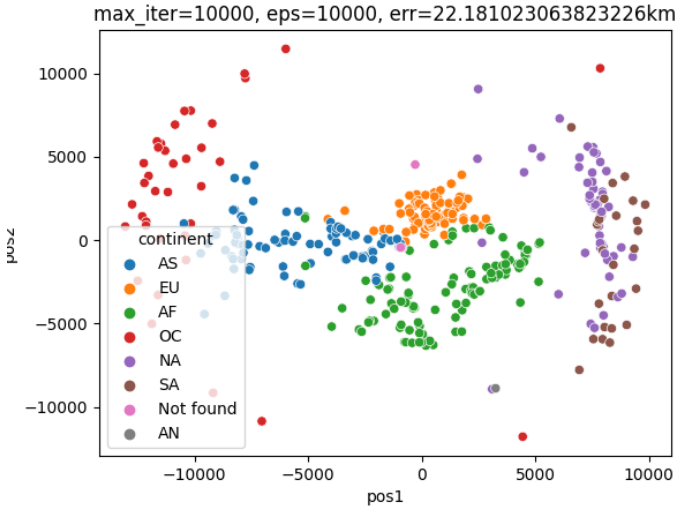






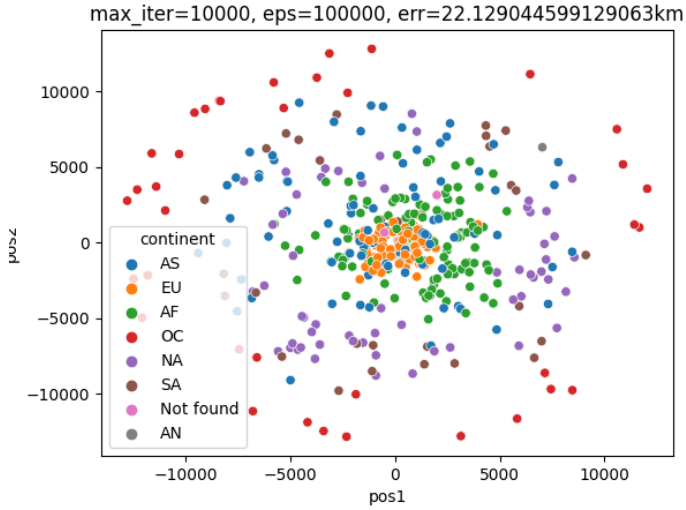
c. Varying *eps* parameter





Appendix B: The code

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from scipy.linalg import sqrtm, eig
from sklearn.manifold import MDS
```




```
## Disconnected graph
"""
from scipy.spatial.distance import pdist, squareform
from matplotlib.collections import LineCollection

np.random.seed(10)
N = 10
X = np.random.rand(N,2)
k = 3

# matrix of pairwise Euclidean distances
distmat = squareform(pdist(X, 'euclidean'))

# select the kNN for each datapoint
neighbors = np.sort(np.argsort(distmat, axis=1)[:, 0:k])

# get edge coordinates
coordinates = np.zeros((N, k, 2, 2))
for i in np.arange(N):
    for j in np.arange(k):
        coordinates[i, j, :, 0] = np.array([X[i,:][0], X[neighbors[i, j], :][0]])
        coordinates[i, j, :, 1] = np.array([X[i,:][1], X[neighbors[i, j], :][1]])

# create line artists
lines = LineCollection(coordinates.reshape((N*k, 2, 2)), color='black')
```

```
fig, ax = plt.subplots(1,1,figsize = (8, 8))
ax.scatter(X[:,0], X[:,1], c = 'black')
ax.add_artist(lines)
plt.show()
"""

## Load the data given the path of the dataset (.csv file)
def load_data(path):
    data = pd.read_csv(path, keep_default_na=False)
    return data


## Compute geodesic distance (haversin formula)
def haversin(phi1, phi2, lbda1, lbda2):
    r = 6371 #earth radius (km)
    t1 = (np.sin((phi2-phi1)/2))**2
    t2 = np.cos(phi1)*np.cos(phi2)*(np.sin((lbda2-lbda1)/2))**2
    arg = t1 + t2
    arcsin = np.arcsin(np.sqrt(arg))
    return 2*r*arcsin


## Extract 2 most populated cities of each country
def preprocess(rawdf):
    dfVar = rawdf[['city', 'lat', 'lng', 'country', 'iso3', 'population']] #useful variables
    sortdf = dfVar.sort_values(['country', 'population'], ascending=[True, False])
    groupdf = sortdf.groupby('country').head(2).reset_index(drop=True)
    df = groupdf[['city', 'lat', 'lng', 'country', 'iso3']]
    return df


## Compute the distance matrix
def dist_matrix(df):
    nbCities = df.shape[0]
    D = np.zeros((nbCities, nbCities))
    for i in range(len(D)):
        for j in range(len(D)):
            phi1 = df.iloc[i].lat
            phi2 = df.iloc[j].lat
            lbda1 = df.iloc[i].lng
            lbda2 = df.iloc[j].lng
            D[i, j] = haversin(phi1*np.pi/180, phi2*np.pi/180, lbda1*np.pi/180, lbda2*np.pi/
    return D
```

```python
# Associate each country to its continent
def find_continents(df):
    country_continent = load_data("country_continent.csv")
    continents = []
    for country_idx in range(n):
        country = df['iso3'][country_idx]
        country_found = False
        continent_cpt = 0
        while (not (country_found) and continent_cpt < 262):
            current_country = country_continent['Three_Letter_Country_Code'][continent_cpt]
            if current_country == country:
                continents.append(country_continent['Continent_Code'][continent_cpt])
                country_found = True
            else:
                continent_cpt += 1
        if continent_cpt == 262:
            continents.append('Not found')
    return continents


## Classical MDS Algorithm
def classical_mds(d):
    n = d.shape[0]
    # double centering trick
    one = np.ones((n, 1))
    # s = -0.5*(d - (1.0/n)*d@one@one.T - (1.0/n)*one@one.T@d + (1.0/n**2) * one@one.T@d@one

    C = np.tile(np.mean(d, axis=0), (n, 1))
    R = np.tile(np.mean(d, axis=1), (n, 1))
    M = np.tile(np.mean(d), (n, n))
    s = -0.5 * (d - R - C + M)

    # eigen-decomposition of s
    eigVal, eigVect = np.linalg.eig(s)
    # sorting eigenVal and eigenVect to have eigenVal in the descending order
    idx = eigVal.argsort()[::-1]
    eigVal = eigVal[idx]
    eigVect = eigVect[:, idx]

    # data embedding
    k = 2
    ikn = np.zeros((n, n))
    for i in range(k):
        ikn[i, i] = 1
    eigVal[-1] *= -1
    x = ikn @ (np.diag(np.sqrt(eigVal))) @ eigVect.T
```

```
    # embedded data dataframe
    x = x.real
    x1 = np.reshape(x[0, :], (1, n))
    x2 = np.reshape(x[1, :], (1, n))
    X = np.concatenate((x1.T, x2.T), axis=1)
    dfX = pd.DataFrame(X, columns=['x1', 'x2'])
    # adding countries and iso3
    dfX = dfX.set_index(df.index)
    dfX.insert(2, 'country', df['country'])
    dfX.insert(3, 'iso3', df['iso3'])

    # Adding continents to the dataframe
    continents = find_continents(df)
    dfCont = pd.DataFrame(np.array(continents), columns=['continent'])
    dfCont = dfCont.set_index(df.index)
    dfX.insert(3, 'continent', dfCont['continent'])

    return dfX


## Metric MDS Algorithm
def metric_mds(max_iter, eps):
    mds = MDS(n_components=2, max_iter=max_iter, eps=eps, dissimilarity="precomputed")
    pos = mds.fit(d).embedding_

    continents = find_continents(df)
    dfPos = pd.DataFrame(list(zip(pos[:, 0], pos[:, 1], np.array(continents))), columns=['po
    dfPos.insert(3, 'iso3', df['iso3'])

    title = "max_iter=" + str(max_iter) + ", eps=" + str(eps)

    return dfPos, title


## Compute the error between computed and true distances
def compute_error(d, dfRes):
    x1 = dfRes.iloc[:, 0]
    x2 = dfRes.iloc[:, 1]
    nbCities = dfRes.shape[0]

    error = 0
    # Compute the distance matrix of the embedded data and the error
    embedded_d = np.zeros((nbCities, nbCities))
    for i in range(len(embedded_d)):
        for j in range(i+1):
```

```
            phi1 = x1[i]
            phi2 = x1[j]
            lbda1 = x2[i]
            lbda2 = x2[j]
            new_dist = haversin(phi1*np.pi/180, phi2*np.pi/180, lbda1*np.pi/180, lbda2*np.pi
            embedded_d[i, j] = new_dist
            error += (new_dist-d[i, j])**2
    return np.sqrt(error)/(nbCities*(nbCities+1)/2)

path = "worldcities.csv"
rawdf = load_data(path)
df = preprocess(rawdf) # Extracting 2 most populated cities of each country


## Testing haversin formula
"""
prs = 127
tky = 187
phi1 = df.iloc[prs].lat
phi2 = df.iloc[tky].lat
lbda1 = df.iloc[prs].lng
lbda2 = df.iloc[tky].lng
print(haversin(phi1, phi2, lbda1, lbda2))
"""

## Computing the distance matrix
# (done once and saved in dist.npy file)
"""
D = dist_matrix(df)
new_name = "dist.npy"
np.save(new_name, D)
"""

## Loading the distance matrix D (saved before)
saved_d = "dist.npy"
d = np.load(saved_d)
n = d.shape[0]
# sns.heatmap(d) # plotting d heatmap
# plt.show()

## 2 methods of data displaying:
"""
# using matplotlib
plt.scatter(x=rawdf['lng'], y=rawdf['lat'])
plt.show()
# using seaborn
```

```
sns.scatterplot(data=rawdf, x='lng', y='lat', hue='capital')
"""

## Double_centering illustration
"""
    ## Preprocessing raw data
reducedf = rawdf[['city', 'lat', 'lng', 'country', 'iso3', 'population']]  # useful variable
sortdf = reducedf.sort_values(['population'], ascending=False)
subdf = sortdf.iloc[:20]
subD = dist_matrix(subdf)
# np.save("dist_subdf.npy", D)
n = subD.shape[0]
one = np.ones((n, n))
subS = -0.5*(subD - (1.0/n)*subD@one@one.T - (1.0/n)*one@one.T@subD + \
          (1.0/n**2) * one@one.T@subD@one@one.T)
## Displaying heatmaps
mask = np.triu(np.ones_like(subS, dtype=bool))
# cmap = sns.diverging_palette(0, 225, s=80, l=65, as_cmap=True)
cmap = sns.color_palette("mako", as_cmap=True)
# sns.heatmap(subD, mask=mask, cmap=cmap, square=True, linewidths=.5)
# sns.heatmap(subS, mask=mask, cmap=cmap, square=True, linewidths=.5)

subS1 = np.ones((n, n))
for i in range(n):
    for j in range(n):
        subS1[i, j] = -0.5*(subD[i, j]**2 - subD[0, i]**2 - subD[0, j]**2)
# sns.heatmap(subS1, mask=mask, cmap=cmap, square=True, linewidths=.5)
"""

## Classical MDS
dfX = classical_mds(d)
sns.scatterplot(data=dfX, x='x1', y='x2', hue='continent')
for i in range(n):
    plt.annotate(dfX['iso3'][i], (dfX['x1'][i], dfX['x2'][i]))
plt.show()
err = compute_error(d, dfX)


## Metric MDS

max_iter_list = [1, 10, 20, 30, 40, 50, 100, 1000, 3000, 6000, 10000]
eps_list = [100000, 10000, 1000, 100, 10, 1, 1e-1, 1e-3, 1e-5, 1e-7, 1e-9, 1e-11, 1e-13]
for max_iter in max_iter_list:
    for eps in eps_list:
        dfPos, title = metric_mds(max_iter, eps)
        err = compute_error(d, dfPos)
```

```
plt.figure()
plt.title(title+", err="+str(err)+"km")
sns.scatterplot(data=dfPos, x='pos1', y='pos2', hue="continent")
plt.savefig(title + ".png")

# for i in range(n):
#     plt.annotate(dfPos['iso3'][i], (dfPos['pos1'][i], dfPos['pos2'][i]))
# plt.show()
```