KTH Royal Institute of Technology

Machine Learning Advanced Course

Assignment 1

Castellano Giovanni

October 3, 2022

1.1 Principal Component Analysis

Question 1.1.1 *Explain why this data-centering step is required while performing PCA. What could be an undesirable effect if we perform PCA on non-centered data?*

When we run PCA on a dataset we perform a change of basis on the data. In particular, the goal of the algorithm is to find a sorted basis of vectors that represent the directions where the variance of the data is maximized. These direction correspond to the eigenvectors of the covariance matrix.

Notice that computing the covarariance matrix implicitly performs centering; that is because centering is embedded into the definition of covariance:

$$cov_{x,y} = \frac{\sum_{i=1}^{N}(x_i - \bar{x})(y_i - \bar{y})}{N} \tag{1}$$

This means that if we compute the eigenvectors of the matrix starting either from centered data or not centered data we get the same results.

Sometimes however, we compute the principal components minimizing the mean square error rather than computing the eigenvectors of the covariance matrix. In this case, it is essential that we perform data-centering. In other words, we cannot prove the equivalence between these two methods if we do not center the data when we minimize the mean square error.

And of course we also have to perform data centering if we use $\mathbf{YY^T}$ as covariance matrix.

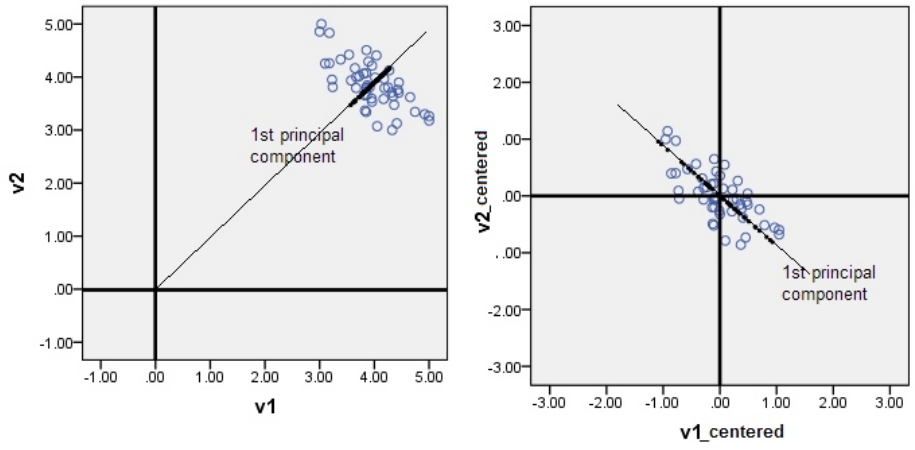

Figure 1: Not centered vs centered PCA [1].

Figure 1 clearly illustrates that the axis that we get when we minimize the mean square error without performing data-centering do not represent the direction where the variance is maximized.

Question 1.1.2 *Does the previous argument imply that a single SVD operation is sufficient to perform PCA both on the rows and the columns of a data matrix?*

It is true that if we have a data matrix $\mathbf{Y} = U\Sigma V^T$ we can compute the eigenvectors of $\mathbf{YY^T}$ and $\mathbf{Y^TY}$ with a single SVD, since in one case the eigenvectors will be the columns of $\mathbf{U}$, and in the other case the eigenvectors will be the columns of $\mathbf{V}$

However, this doesn't mean that we can compute PCA with a single SVD, in fact, we have to take in consideration that $\mathbf{YY^T}$ needs to be centered before performing PCA.

On the one hand, if we have a data matrix $\mathbf{Y}$ where the columns represent the data points and the rows the dimensions, after centering the data we will get a matrix $\mathbf{Y_{c1}}$. On the other hand, if we take the same matrix $\mathbf{Y}$ where this time the rows represent the data points and the columns the dimensions, after centering the data we will get $\mathbf{Y_{c2}}$, and in general $\mathbf{Y_{c1}} \neq \mathbf{Y_{c2}}$. Therefore, we cannot compute with a singular SVD the eigenvectors of $\mathbf{Y_{c1}Y_{c1}^T}$ and $\mathbf{Y_{c2}^TY_{c2}}$.

Question 1.1.3 *Explain why the use of the pseudo-inverse is a good choice to obtain the inverse mapping of the linear map (1).*

The matrix $\mathbf{W}$ is a rectangual matrix, therefore it is not possible to compute an inverse matrix $\mathbf{W}^{-1}$ such that $\mathbf{WW}^{-1} = \mathbf{W}^{-1}\mathbf{W} = \mathbf{I}$. However, since $\mathbf{W}$ is an orthogonal $d \times k$ matrix (with $k < d$), it is possible to define a left inverse $\mathbf{W}_L^{-1}$ such that $\mathbf{W}_L^{-1}\mathbf{W} = \mathbf{I}$ where $\mathbf{I}$ is $k \times k$.

The right inverse $\mathbf{W}_R^{-1}$ such that $\mathbf{WW}_R^{-1} = \mathbf{I}$ instead , does not exist; as it would be a function from R^d to R^k, in such function, the space collapses in a lower dimensional space, therefore, there is no way to recover the initial position of the vectors (the function is not one to one). Notice that this is not a problem for the left inverse just because the matrix $\mathbf{W}_L^{-1}$ is placed on the left and not on the right.

Many left inverse exists, however we choose the left pseudo inverse $\mathbf{W}^+$:

$$W^+ = (W^TW)^{-1}W^T \tag{2}$$

The motivation behind this choice is that although the right inverse does not exist, the left pseudo inverse has a good behavior when used as right inverse. In fact, it is the best approximation of the right inverse. This is because when we place it on the right, we get the projection matrix (link) on the columns of

$\mathbf{W}$. In this way, when we compute $\mathbf{WW}^{+}\mathbf{y}$ we will not get the vector $\mathbf{y}$ back, but we will get the nearest vector to y in the lower dimensional space. That's the best reconstruction that we can perform. This is crucial, since what we do in PCA, is optimizing this error:

$$\mathbf{E}[||y - WW^{+}y||] \tag{3}$$

where we can observe the left pseudo inverse placed on the right.

Question 1.1.4 *Derive PCA using the criterion of variance maximization and show that one gets the same result as with the criterion of minimizing the reconstruction error. Show this result for projecting the data into k dimensions, not just 1 dimension.*

We already proved during the lectures (slides module 2) that the principal components computed using the mean square error criterion correspond to the columns of the $\mathbf{U}$ matrix that we get performing SVD on the centered data matrix $\mathbf{Y}$. We will prove here, that we get the same result choosing as principal components the axis where the variance is maximized; Let's suppose for a moment to know that such axis correspond to the eigenvectors $\mathbf{u_1}, \mathbf{u_2}, ..., \mathbf{u_d}$ of the covariance matrix $\mathbf{S}$.

If our data is centered we can compute the eigenvectors from the eigendecomposition of $\mathbf{YY}^T$; notice that we are ignoring the division by N in equation 1 since multiplying a matrix by a constant doesn't change its eigenvectors. Performing SVD we get equation 4:

$$YY^T = U\Sigma V^T V \Sigma^T U^T = U\Sigma_1 U^T \tag{4}$$

Where the last equality leads to the eigendecomposition of the matrix, therefore, the eigenvectors of the covariance matrix correspond to the columns of the $\mathbf{U}$ matrix.

Let's prove now, that the variance is maximized along the directions of the eigenvectors $\mathbf{u_1}, \mathbf{u_2}, ..., \mathbf{u_d}$:

We want to find a set of vectors $\mathbf{w_1}, \mathbf{w_2}, ..., \mathbf{w_p}$ with $p <= d$ which span the subspace where we want to project the data. We want, moreover, that the variance of the projected data is maximized. Starting from the covariance matrix $\mathbf{S}$ we can compute the variance of the projection on a single vector $\mathbf{w_1}$ as:

$$w_1^T S w_1 \tag{5}$$

what we want to do is to maximize this quantity, however, we notice that we can make it as large as we want increasing the length of the vector $\mathbf{w_1}$. To avoid

this problem, we introduce the constraint $||\mathbf{w_1}|| = 1$. In other words, we just care about the direction of the vector $\mathbf{w_1}$.

This problem can be solved introducing a Lagrange multiplier λ_1 and then optimizing with respect to $\mathbf{w_1}$:

$$w_1^T S w_1 + \lambda_1(1 - w_1^T w_1) \tag{6}$$

By setting the derivative equal to zero we get:

$$S w_1 = \lambda_1 w_1 \tag{7}$$

Therefore $\mathbf{w_1}$ is an eigenvector. Moreover, substituting 7 in 5 we get:

$$w_1^T \lambda_1 w_1 \tag{8}$$

we can conclude that to maximize the objective function, λ_1 must be as large as possible. Therefore $\mathbf{w_1}$ is the eigenvector with the larger eigenvalue.

We can now proceed with the second principal component $\mathbf{w_2}$. This time however, we will add the following constraint to the optimization problem:

$$w_2^T w_1 = 0 \tag{9}$$

i.e. we want the next principal component to be orthogonal to the first one. We get:

$$w_2^T S w_2 + \lambda_2(1 - w_2^T w_2) - \beta(w_2^T w_1) \tag{10}$$

taking the derivative of 10 with respect to $\mathbf{w_2}$ and setting it to zero we can calculate the value of β:

$$S w_2 - \lambda_2 w_2 - \beta w_1 = 0 \tag{11}$$

multiplying by $\mathbf{w_1^T}$:

$$w_1^T S w_2 - \lambda_2 w_1^T w_2 - \beta w_1^T w_1 = 0 \tag{12}$$

since $\mathbf{S}$ is symmetric, substituting 7:

$$\lambda_1 w_1^T w_2 - \lambda_2 w_1^T w_2 - \beta w_1^T w_1 = 0 \tag{13}$$

imposing the constraints:

$$\lambda_1 0 - \lambda_2 0 - \beta 1 = 0 \tag{14}$$

$$\beta = 0 \tag{15}$$

substituting 15 in 10:

$$w_2^T S w_2 + \lambda_2(1 - w_2^T w_2) - 0 \tag{16}$$

finally, taking the derivative of 16 with respect to $\mathbf{w_2}$:

$$S w_2 = \lambda_2 w_2 \tag{17}$$

As we can see also $\mathbf{w_2}$ is an eigenvector. This result can be generalized by induction to the remaining vectors $\mathbf{w_i}$

This proof is partially based on the proof presented by Bishop in his book [**?**].

1.2 Multidimensional Scaling (MDS) and Isomap

Question 1.2.5 *Explain in English what is the intuitive reason that the "double centering trick" is necessary in order to be able to solve for* **S** *given* **D**.

The problem is that knowing the distance between two points in an euclidean space, is not enough to find the dot product between these points. This is because if we translate the space the distance is preserved but the dot product changes. In the same way therefore, we cannot go from the distance matrix to the similarity matrix.

What we do with the double centering trick, is translating the center of mass of the points to the origin of our coordinate system (or equivalently, we move the origin to the center of mass of the points). In other words, we are fixing the position of the points somewhere. Now that the position is fixed, we can compute the dot products starting from the distances. Notice that, the choice of the center of mass is not casual, since it is one of the few points from which we can calculate s_{ii} and s_{jj} easily using the distance matrix.

For instance suppose to have three points in a two dimensional space as in figure 2. Figure 3 shows what happens to the points when we reconstruct them using classical MDS starting from the distance matrix and using the double centering trick.

As we can observe, the center of mass (that in this case correspond to the green point), moved to the origin.

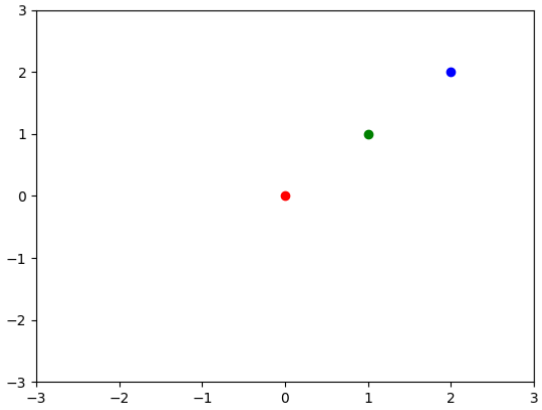

Figure 2: 3 points on a 2 dimensional space.

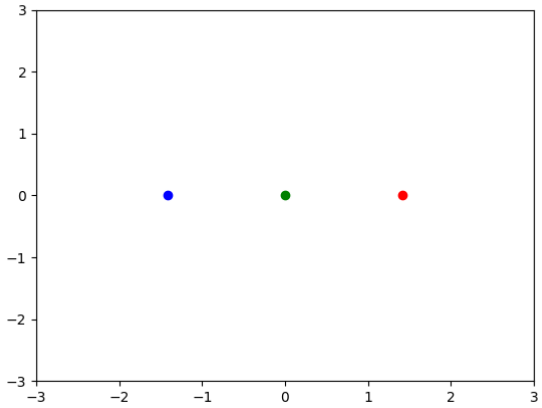

Figure 3: Reconstruction of the position of 3 points using classical MDS and the double centering trick

Notice that we also observe a rotation, the reason is that if we have two points in a space and we rotate the space, the scalar product between them doesn't change. This means that we can always get a rotation when we perform classic MDS, even if start directly from the similarity matrix.

Question 1.2.6 *Use the same reasoning as in the previous question (1.2.5) to argue that* s_{ij} *can be computed as* $s_{ij} = -\frac{1}{2}(d_{ij}^2 - d_{1i}^2 - d_{1j}^2)$*, where* d_{1i} *and*

d_{1j} are the distances from the first point in the data set to points i and j, respectively. In particularly, argue that although the solution obtained by the "first point trick" will be different than the solution obtained by the "double centering trick", both solutions are correct.

As already explained in the previous answer, the trick consists in placing the origin of the coordinate system on a point from which we can compute s_{ii} and s_{jj} just using the distance matrix. We can observe that, if we move the origin to the position of the first data point, s_{ii} will be equal to the squared distance from the first data point to the ith point, and we get the formula of the first point trick. Notice that, it is not essential that we choose the first point of the data set. It can be anyone of them, or alternatively, the center of mass.

Figure 4 shows what happens to the same points of Figure 2 when we reconstruct them using classical MDS starting from the distance matrix and using the first point trick (in this case the first point is the blue point).

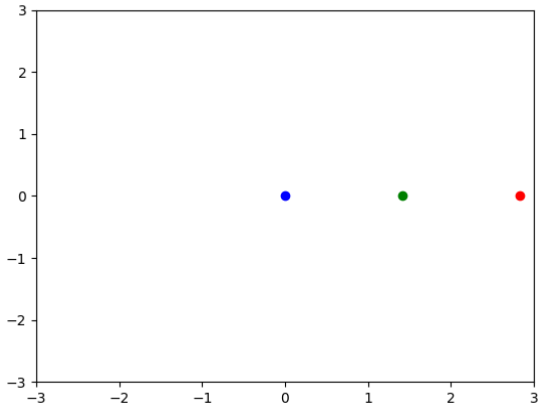

Figure 4: Reconstruction of the position of 3 points using classical MDS and the first point trick

As we can observe, the blue point moved to the origin.

Therefore, both solutions are correct, we just observe a different translation of the space.

Question 1.2.7 *Show that the two methods, i.e., classical MDS when $\mathbf{Y}$ is known and PCA on $\mathbf{Y}$, are equivalent. Which of the two methods is more efficient?*
Suppose to start from a centered data matrix $\mathbf{Y} \in R^{d \times n}$ where d is the number of initial dimensions and n is the number of points in the space. We can

decompose the matrix using SVD:

$$Y = U\Sigma V^T \tag{18}$$

When we perform PCA, we first perform a change of basis on the space; subsequently, we drop some of the dimensions. We already proved that the vectors of the new basis are the eigenvectors of the covariacnce matrix. Moreover, we know that these eigenvectors correspond to the columns of the **U** matrix in equation 18. Therefore, the new data matrix **X** after performing PCA is:

$$X = I_{p\times d}U^T Y \tag{19}$$

where p is the new number of dimensions. Notice that we are supposing that the eigenvectors in **U** are sorted according to their eigenvalues. Moreover, since **U** is a unitary matrix, we basically just rotated the space and dropped the dimensions with less variance.
When we compute classical MDS instead, we find **X** starting from the gram matrix **S**:

$$S = Y^T Y = V\Sigma_2 V^T = V\Sigma_2^{1/2}\Sigma_2^{1/2}V^T = (\Sigma_2^{1/2}V^T)^T(\Sigma_2^{1/2}V^T) \tag{20}$$

where we decomposed the matrix **S** thanks to the spectral theorem (since it is symmetric). Also in this case, the eigenvectors are sorted. Notice that in this equation **V** is the same **V** matrix of equation 18.

Finally:

$$X = I_{p\times n}\Sigma_2^{1/2}V^T \tag{21}$$

Taking in consideration that:

$$\Sigma_2 = (\Sigma^T\Sigma) \tag{22}$$

i.e., the singular values of **Y** are the square root of the eigenvalues of the Gram matrix (we proved it during the lectures).

We can prove the equivalence between equation 21 and equation 19

$$I_{p\times d}U^T Y = I_{p\times n}\Sigma_2^{1/2}V^T \tag{23}$$

$$I_{p\times d}U^T U\Sigma V^T = I_{p\times n}(\Sigma^T\Sigma)^{1/2}V^T \tag{24}$$

$$I_{p\times d}\Sigma V^T = I_{p\times d}\Sigma V^T \tag{25}$$

where we just used equation 18 and equation 22.

Regarding the computational complexity of the algorithms, to perform cMDS we have to compute the eigenvalue decomposition of the gram matrix $S = Y^T Y \in R^{n \times n}$ where n is the number of points. Many algorithms exist for this purpose, but in general the complexity is $\mathcal{O}(n^3)$.

PCA instead, can be summarized in two steps:

1) Compute the covariance matrix

2) Compute the eigenvalue decomposition of the covariance matrix $YY^T \in R^{d \times d}$ where d is the number of dimensions

The first step has complexity $\mathcal{O}(d^2 n)$, the second instead has complexity $\mathcal{O}(d^3)$, therefore in total we get $\mathcal{O}(d^2 n + d^3)$.
In general, n is much larger than d therefore we will expect PCA to be faster; however, if d is much larger than n (i.e. we have a few points living in a space with a large number of dimensions), cMDS will be faster.

Question 1.2.8 *Argue that the process to obtain the neighborhood graph* $\mathbf{G}$ *in the Isomap method may yield a disconnected graph. Provide an example. Explain why this is problematic.*

The first step of the isomap algorithm is to determine the neighbors of each point, therefore it is necessary to define when 2 points can be considered neighbors. For this purpose, we can choose a threshold value; if the euclidean distance between two points is less than the threshold, then we can conclude that they are neighbors. Afterwards, we create a graph where each point is represented by a node, and neighbors are linked by means of a weighted link representing the euclidean distance between them. We can therefore compute the geodesic distance between two points finding the shortest path in the graph.

But how do we choose the threshold?

On the one hand, if the value of the threshold is too large, we could end up considering too many points as neighbors, this means that the shortest path in the graph will not be representative of the geodesic distance. On the other hand, if the threshold is too small, we could get a disconnected graph.

Figure 5 for instance, illustrates some points in a two dimensional space, living on a 1-dimensional manifold. In particular, the manifold is the function $f(x) = x^4$. To preserve the structure of the manifold, each point should have a maximum of two neighbors, therefore, the threshold should be very small. Proceeding in this way however, we will get a disconnected graph. In particular, the graph will probably be divided in 5 disconnected subgraphs, since we can observe 5 clusters of points.

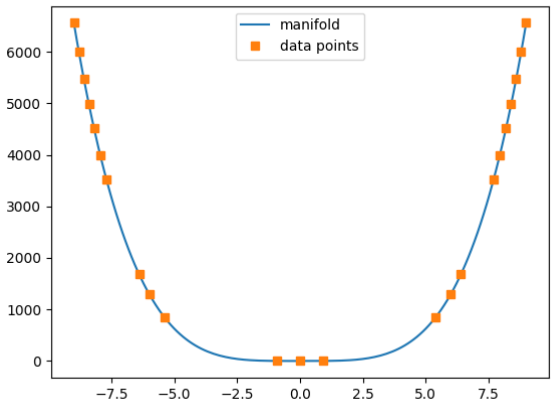

Figure 5: An example of 2-dimensional points living on a 1-dimensional manifold

Question 1.2.9 *Propose a heuristic to patch the problem arising in the case of a disconnected neighborhood graph. Explain the intuition of your heuristic and why it is expected to work well. How does your heuristic behave in the example you provided in the previous question?*

If we get a disconnected graph, the only thing to do is to join the disconnected subgraphs. To join two disconnected subgraphs we need to choose two nodes (one for each subgraph) and link them. The weight of the edge will be equal to the euclidean distance between the points. Moreover, we can choose the two points in order to minimize such distance.

But how do we choose which subgraphs to join?

An interesting idea is to compute the centers of mass of the points represented by each disconnected subgraph; afterwards, we link two subgraphs if their associated centers of mass can be considered neighbors. This however, can still lead to a disconnected graph, therefore, we could need to repeat it iteratively. The algorithm is formalized below.

Suppose to perform Isomap and to get a disconnected graph G_1, composed by N disconnected subgraphs $S_1^1, S_1^2, ..., S_1^N \in SETdsg(G_1)$ where $SETdsg(G)$ is the set of disconnected subgraphs of a graph G.

Let's call a cluster C_1^i the set of points in the space represented by the nodes in the disconnected graph S_1^i . Therefore we have N clusters $C_1^1, C_1^2, ..., C_1^N \in SETc(G_1)$ where $SETc(G)$ is the set of clusters associated with the graph G, and therefore there is a one-to-one association between the elements of

$SETdsg(G_1)$ and the elements of $SETc(G_1)$.

Proceed in this way:

1) $S = SETdsg(G_1)$

2) $C = SETc(G_1)$

3) compute the centers of mass of each cluster in C.

4) create a new graph G_2 where each node represents the center of mass of the clusters, and two nodes are linked only if the two centers of mass are neighbors; also in this case we can define the concept of neighbors choosing a threshold T.

5) for each pair i and j of linked nodes in G_2, link the corresponding S_1^i and S_1^j in G_1 (the graph G_1 now has $p <= N$ disconnected subgraphs and therefore we have p clusters).

6) if G_1 is still disconnected, repeat from step 1 increasing the threshold T.

This algorithm should work well if we start from a low value of T and increase it gradually because also in this case the low value of T preserves the structure of the manifold. We will probably need to increase T after each iteration but we should still be able to preserve the manifold since at each iteration we have less disconnected subgraphs.

Figure 6 illustrates the position of the five centers of mass in our previous example.

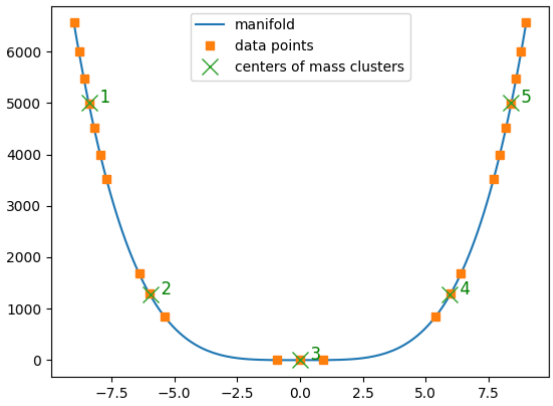

Figure 6: Centers of mass clusters

Figure 7 illustrates the situation after the first iteration (if we start with a good value T), clusters 2 was linked with 3 and the same happened with cluster 4; now

we have only one center of mass for them. Afterwards, increasing the threshold T the three remaining clusters will appear neighbors and therefore we will get a connected graph.

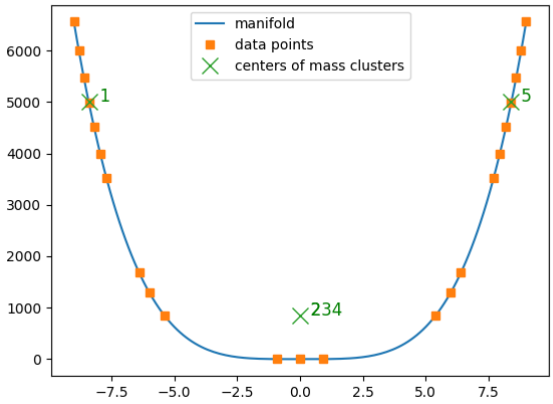

Figure 7: Centers of mass after the first iteration

Finally, figure 8 shows how the point would be connected in the final graph, in this case, we preserved perfectly the structure of the manifold, and the geodesic distance was approximated with the euclidean distance.

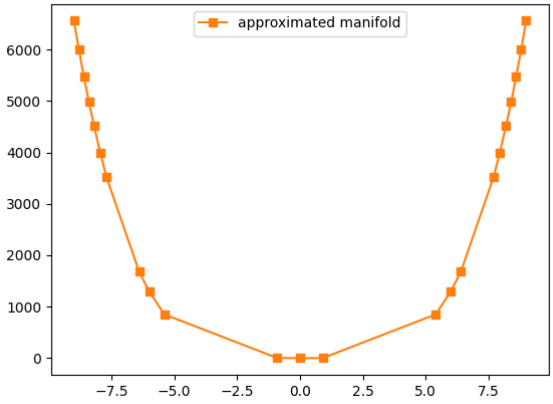

Figure 8: Approximation of the geodesics distances with euclidean distances

1.3 PCA vs. Johnson-Lindenstrauss random projections

Question 1.3.10 *Provide a qualitative comparison (short discussion) between the two methods, PCA vs. Johnson-Lindenstrauss random projections, in terms of (i) projection error; (ii)computational efficiency; and (iii) intended use cases.*

When we project the data on a k dimensional subspace using PCA we get a projection that respect the Johnson-Lindenstrauss lemma (in terms of projection error), but not only this, the subspace that we find is the best one that we can find. It means that in general the projection error is smaller when we use PCA. PCA however, can be very slow from a computational point of view, therefore, sometimes it is better to use random projections, since they will return a good solution (a solution that respect the Johnson-Lindenstrauss lemma) in less time. For instance, we could prefer using random projections when we have a very high number of dimensions since it would take much less time. Random projections are also extremely useful when we work with a stream of data. In this case, the projection that we find with PCA for an initial set of points could become very bad as we get new points from the stream. This means that we would need to compute a new subspace every time that we get new data; and of course, it will much faster to do it using random projections. In conclusion, we will not use PCA when it is too expensive from a computational point of view.

1.4 Programming task — MDS

Question 1.4.11 *Search the internet for an API for calculating distances between world cities. Distances could be estimated by the geodesic, flight time of an actual flight, or other heuristic, it does not matter as long as it is a reasonable approximation.Use the API to compute the pairwise distance matrix* $\mathbf{D}$ *for at least 100 different world cities of your choice. Choose the cities so that they cover the whole globe, for instance all 5 continents,or even a larger number of regions subdividing continents, e.g., Middle East Asia, Central Asia,South East Asia, etc., it is up to you to pick your criteria.*

To collect the data, I used two dataset:

1) a csv file containing the name and the continent of each country (link) [**?**]

2) a csv file containing the names of the most important towns of each country (link) [**?**]

To begin, I used the first dataset to get some random countries from each of the 6 continents:

1) Europe

2) Africa

3) Asia

4) South America

5) North America

6) Oceania

Afterwards, for each of the country I picked one random town from the second dataset. The dataset however, does not contain the coordinates of the towns, therefore, I retrieved them using Nominatim API. Finally, I wrote all the information about the towns on a new csv file in order to make it easier to access the data, and I plotted the towns using Cartopy.

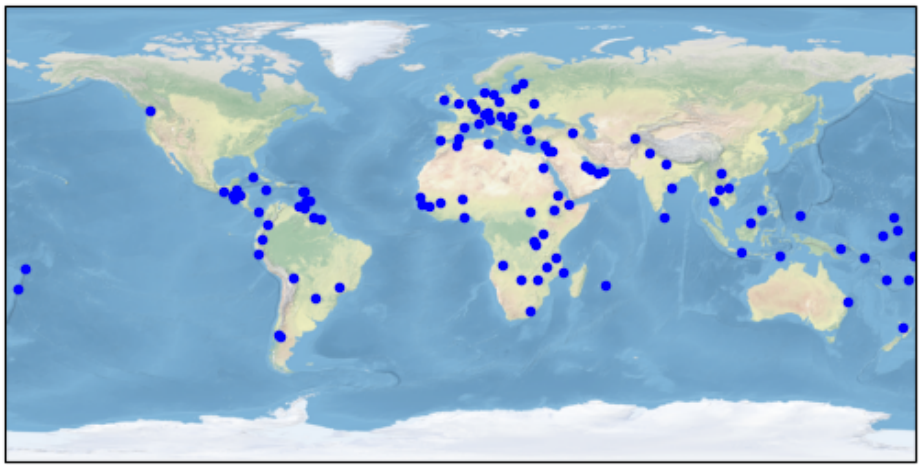

Figure 9: Initial Towns plotted using Cartopy

Figure 9 shows the position of the collected towns. As we can observe, there are not many towns in big countries such as USA, Russia and Canada. Moreover, in some zones such as Europe we can notice an high density of towns. This is because I just collected one town for each country that I chose. To make the distribution of the towns a bit more uniform, I manually removed some of the towns from the zones with many countries and I added some towns in the big countries. The final result is shown in 10. The total number of towns is 100.

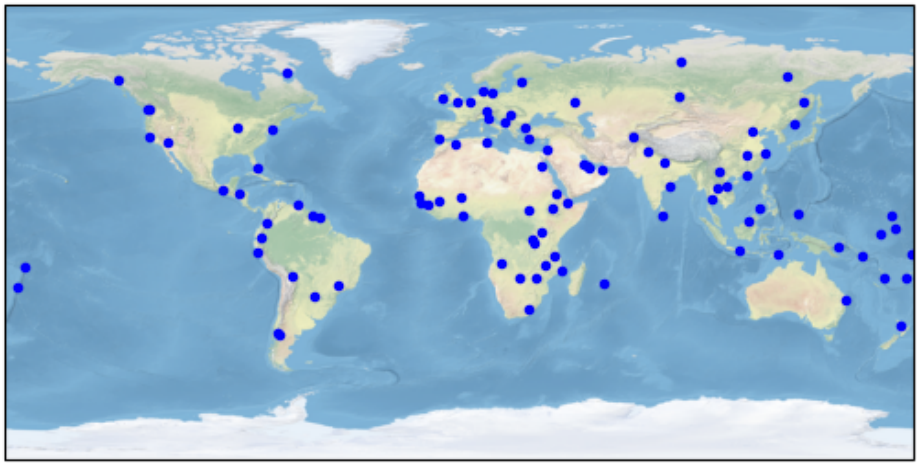

Figure 10: Final Towns plotted using Cartopy

To estimate the distances between them, I used the distance() function in the geopy library which offers an implementation of the Vincenty distance.

Question 1.4.12 *Apply classical MDS to compute an (x, y) coordinate for each city in your dataset, given the distance matrix* $\mathbf{D}$. *Plot the cities on a plane using the coordinates you computed. You may want to annotate the cities by their name, or abbreviation, use different colors to indicate continents, or regions, etc.Discuss how good is the reconstructed map your created using classical MDS. For this task, you should implement MDS by yourself, by relying only on a package for eigenvector decomposition, that is, do not try to find a library that implements MDS.*

To perform cMDS I computed the similarity matrix from the distance matrix using the double centering trick. Afterwards, I computed the eigenvalue decomposition of the similarity matrix and I used it to reconstruct the positions of the towns. It was interesting to note that the eigenvectors decomposition returned some negative eigenvalues. This is due to the fact that we are using non-euclidean distances.

I plotted the result in both 2 and 3 dimensions using different colors for towns of different continents.

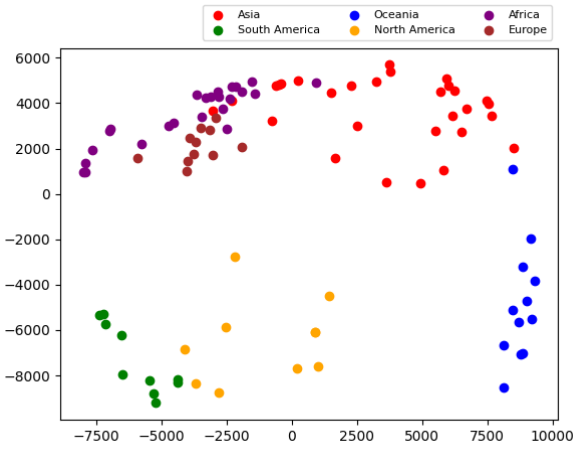

Figure 11: Classical mds 2D

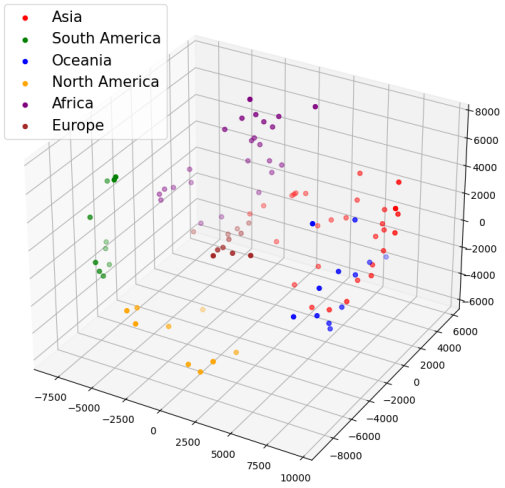

Figure 12: Classical mds 3D

It was interesting that although I reconstructed the positions of the towns in 3 dimensions, the result is not perfect, this is because cMDS is supposed to work with euclidean distances but we are using geodesic distances.

Regarding the 2 dimensional reconstruction, the results are consistent with reality; we can observe that north and south America are close to each other and at the same time they are far from all the other continents. Furthermore, Asia is close to Africa, Europe and Oceania. And at the same time Oceania is far from Europe and Africa. We can also notice that north America is trying to be close to Asia, this is because Russia and Canada are very close countries therefore we observe this rotation of the american continent in the map. We have to remember in fact, that there is no way to map the points that live on the surface of a sphere to a 2 dimensional plane preserving all the geodesic distances.

Question 1.4.13 *Repeat the task of 1.4.12, but using metric MDS this time. You may tune the parameters of the method until you are satisfied with the results.Discuss the parameters that you tuned and their effect on the end result. Discuss the quality of your map, and compare it with the one you obtained by classical MDS. For this task, you are free to search for an implementation of metric MDS and use it as a black box.*

Scikit learn offers an implementation of metric MDS with the following parameters to tune:

1) n_init: Number of times the SMACOF algorithm will be run with different initializations. The final results will be the best output of the runs, determined by the run with the smallest final stress.

2) max_iter: Maximum number of iterations of the SMACOF algorithm for a single run.

3) eps: Relative tolerance with respect to stress at which to declare convergence.

I plotted the final stress of the algorithm for different values of them.

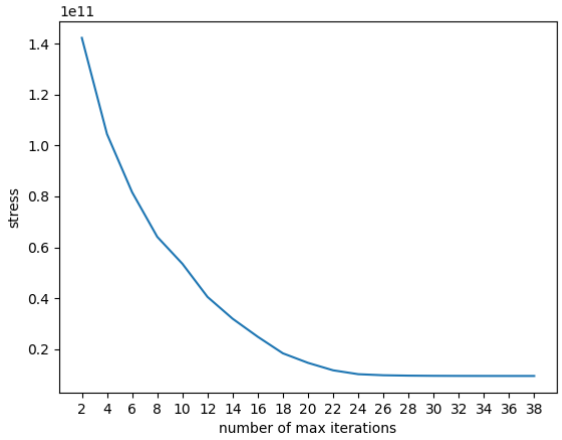

Figure 13: How the stress changes as max_iter increases for some fixed value of the other parameters

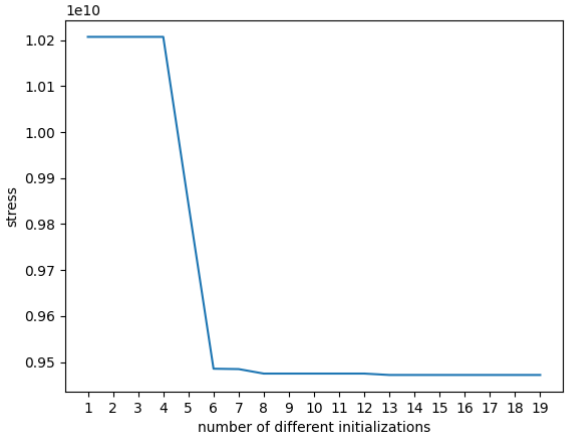

Figure 14: How the stress changes as n_init increases for some fixed value of the other parameters

As we can observe from the figures, in general the stress decreases when we increase the number of max iterations or the the number of different initialization of the algorithm. In figure 14 we can observe that if the number of initializations is too low we have a significant stress, instead if the number of initialization is greater than 5 the stress is much lower, but it doesn't keep decreasing if we increase the number of iterations. Similarly in figure 13 the stress decreases as

the the number of maximum iterations increases, but in any case we cannot go below about 0.2.

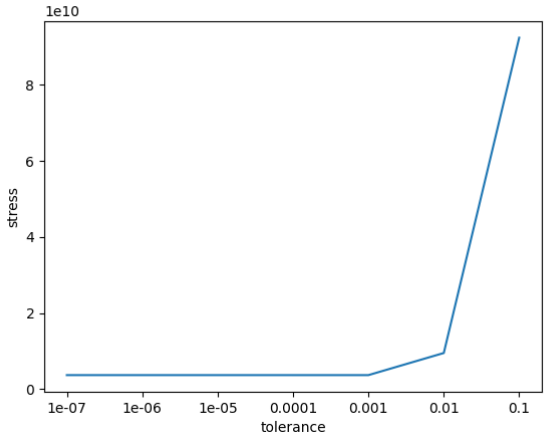

Figure 15: How the stress changes as eps increases for some fixed value of the other parameters (notice that the distances between the points in the x axis is not representative of the real distance between the values)

Figure 15 at the same way, shows that we get a smaller stress for low values of epsilon; also in this case the stress stabilizes at a certain value.

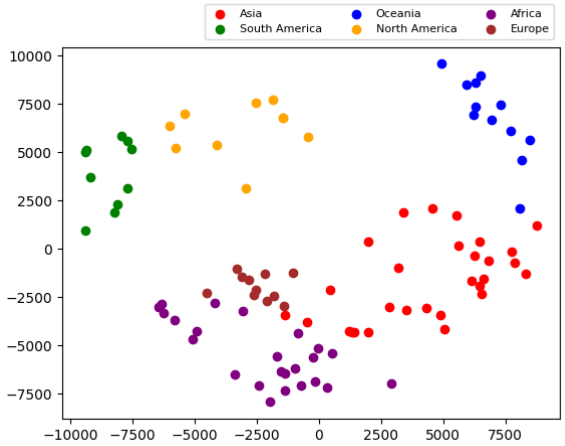

Figure 16: Metric mds 2D. n_init=1000; max_iter=8000; eps=1e-8

The results for metric MDS is similar to the classical one, also in this case,

north and south America are far from the other continents and we can observe the rotation of the american continent towards Asia. Also the other continents are placed in a very similar way; however, in the cMDS, Asia is at the top of the plot , and the american continent is at the bottom. In figure 16 instead, we observe exactly the opposite: the american continent is on the top, whereas, Asia and Africa are on the bottom; we basically observe a reflection accross the horizontal axis.

For completeness, I also plotted the 3D reconstruction for the metric MDS.

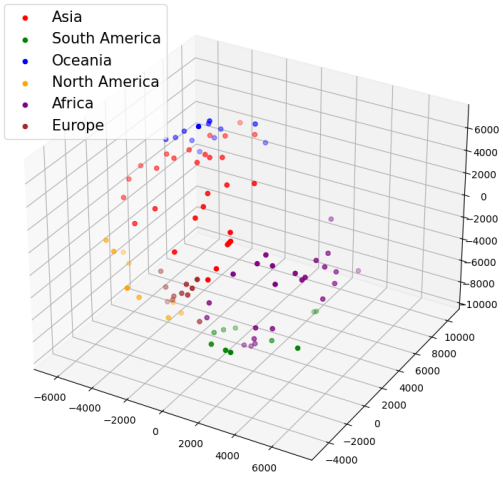

Figure 17: Metric mds 3D. n_init=1000; max_iter=8000; eps=1e-8

References

[1] ttnphns (https://stats.stackexchange.com/users/3277/ttnphns), "How does centering the data get rid of the intercept in regression and pca?." Cross Validated. URL:https://stats.stackexchange.com/q/22331 (version: 2016-09-16).

Data collection

```
from time import sleep
from geopy.geocoders import Nominatim
import pandas as pd
import numpy as np
import csv

df = pd.read_csv (r'Countries-Continents.csv', index_col=False)
continents = []
countries = []
towns = []

#select some random countries for each continent
for continent in list(set(df["Continent"])):
    continent_df = df[df["Continent"] == continent]
    random_countries = continent_df.take(np.random.permutation(len(continent_df))[:30])
    for index, row in random_countries.iterrows():
        continents.append(continent)
        countries.append(row["Country"])

df = pd.read_csv (r'world-cities.csv', index_col=False)

#for each country select a random town
for country in countries[:]:
    country_df = df[df["country"] == country]
    #this is necessary if the names of the countries in the
    #two datasets are not exactly the same
    if country_df.empty:
        del countries[i]
        del continents[i]
        continue
    random_town = country_df.take(np.random.permutation(len(country_df))[:1])
    for index, row in random_town.iterrows():
        towns.append(row["name"])


geolocator = Nominatim(user_agent="advanced-machine-learning-a1")

loc= []
i = 0
#find the coordinates of each town
for town in towns[:]:
    t = geolocator.geocode(town)
    # if geolocator doens't find the town, do not take in consideration the country
    if t is None:
```

```
            del countries[i]
            del continents[i]
            del towns[i]
            sleep(1)
            continue
        check_country = geolocator.reverse([t.raw["lat"],t.raw["lon"]], language='en')

        #if for some motivation the geolocator found the wrong town
        if countries[i] != check_country.raw["address"]["country"]:
            del countries[i]
            del continents[i]
            del towns[i]
            sleep(1)
            continue
        loc.append(t.raw)
        sleep(1)
        i = i+1

#insert all the information of each town in a list
locations = []
i = 0
for l in loc:
    locations.append([continents[i],countries[i],towns[i],l["lat"],l["lon"]])
    i += 1

#write everything in a csv file
with open("towns.csv", "w") as f:
    writer = csv.writer(f)
    writer.writerow(["continent","country","town","latitude","longitude"])
    writer.writerows(locations)
    f.close()
```

MDS

```
import numpy as np
from numpy import linalg as lg
import matplotlib.pyplot as plt
import pandas as pd
import geopy.distance
from mpl_toolkits import mplot3d
from sklearn.datasets import load_digits
from sklearn.manifold import MDS

#read data from the file and compute the distance matrix
def generate_distance_matrix():
    df = pd.read_csv(r'towns2.csv')
    D = np.zeros((len(df),len(df)))
    i = j = 0
    for index, town in df.iterrows():
        j = 0
        for index, town2 in df.iterrows():
            #use geopy to compute the distance
            D[i][j] = geopy.distance.distance((town["latitude"],
            town["longitude"]),(town2["latitude"],
            town2["longitude"]) ).km
            j = j +1
        i = i+1
    return D

#compute the simiarity matrix from the distance matrix
def compute_similarity_matrix(D, size):
    D = D**2
    mean = []
    sum = 0
    #compute rows, columns, and total mean of the matrix
    for i in range(size):
        sum1 = 0
        for j in range(size):
            sum += D[i][j]
            sum1 += D[i][j]
        mean.append(sum1 / size)
    total_mean = sum / (size * size)

    S = np.zeros((size, size))
    for i in range(size):
        for j in range(size):
            #first centering trick
            #S[i][j] = -0.5 * (D[i][j] - D[1][j] - D[1][i])
```

```
            #double centering trick
            S[i][j] = -0.5 * (D[i][j] - mean[i] - mean[j] + total_mean)

    return S


#compute classical MDS from the similarity matrix
def cMDS(S, k):
    eigenvalues, eigenvectors = lg.eig(S)
    #sort the eigenvectors
    idx = eigenvalues.argsort()[::-1]
    eigenvalues = eigenvalues[idx]
    eigenvectors = eigenvectors[:, idx]
    eigenvalues = np.sqrt(np.abs(eigenvalues))
    sqrt_diagonal = np.diag(eigenvalues)
    Y = sqrt_diagonal @ eigenvectors.T
    return Y[:k]

#classical MDS
D = generate_distance_matrix()
S = compute_similarity_matrix(D, 100)
X = cMDS(S, 2)

#metric MDS
embedding = MDS(dissimilarity="precomputed", random_state=0,
                n_init=1000, max_iter=8000, eps=1*10**(-8),
                 n_jobs=8)
X2 = embedding.fit_transform(D)


#2D plot
# plt.scatter(X[0][:29], X[1][:29], c="red", label="Asia")
# plt.scatter(X[0][30:40], X[1][30:40], c="green", label="South America")
# plt.scatter(X[0][41:53], X[1][41:53], c="blue",label="Oceania")
# plt.scatter(X[0][54:64], X[1][54:64], c="orange", label="North America")
# plt.scatter(X[0][64:88], X[1][64:88], c="purple", label="Africa")
# plt.scatter(X[0][89:100], X[1][89:100], c="brown", label="Europe")
# plt.legend(
#             ('Asia', 'South America', 'Oceania', 'North America', 'Africa', 'Europe'),
#              bbox_to_anchor=(1, 1.13),
#             loc='upper right',
#             ncol=3,
#             fontsize=8)

#3D plot
# fig = plt.figure(figsize=(9, 9))
```

```
# ax = plt.axes(projection="3d")
# ax.scatter3D(X[0][:29], X[1][:29],X[2][:29], c="red", label="Asia")
# ax.scatter3D(X[0][30:40], X[1][30:40], X[2][30:40], c="green", label="South America")
# ax.scatter3D(X[0][41:53], X[1][41:53],X[2][41:53], c="blue", label="Oceania")
# ax.scatter3D(X[0][54:64], X[1][54:64],X[2][54:64], c="orange", label="North America")
# ax.scatter3D(X[0][64:88], X[1][64:88],X[2][64:88], c="purple", label="Africa")
# ax.scatter3D(X[0][89:100], X[1][89:100],X[2][89:100], c="brown", label="Europe")
# ax.legend(loc=2,prop={'size': 15})
# plt.show()

# code used to generate the plots of the stress value for
#different parameters (this is for epsilon)
# stress = []
# x = []
# for i in reversed(range(1,8)):
#     x.append(i)
#     embedding = MDS(dissimilarity="precomputed", random_state=0,
#       n_init=20, max_iter=100, eps=1*10**(-i),n_jobs=8)
#     X = embedding.fit_transform(D)
#     X = X.T
#     stress.append(embedding.stress_)
# x.reverse()
# plt.plot(x,stress)
#
# plt.xticks(x,[10**(-i) for i in reversed(range(1,8))])
# plt.xlabel("tolerance")
# plt.ylabel("stress")
# plt.show()
```