DD2434 Advanced Machine Learning
2021-2022

Assignment 1: Dimensionality Reduction

Martín Iglesias Goyanes
martinig@kth.se

School of Electrical Engineering and Computer Science
Stockholm, 5 November 2021

1 Principal Component Analysis

Question 1.1.1

While developing the PCA method, we required that the data are "centered." This step is performed by subtracting the expectation of the data from each data point. Essentially, with this step, we translate the center of mass of the data to the origin of the Euclidean space.

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 performing PCA we aim to obtain the eigenvectors or directions that result in the maximum variance of the data, also known as PCs. In order to do so we can use two methods. One of them is classical EigenVector Decomposition (EVD) and the other is Singular Value Decomposition (SVD).

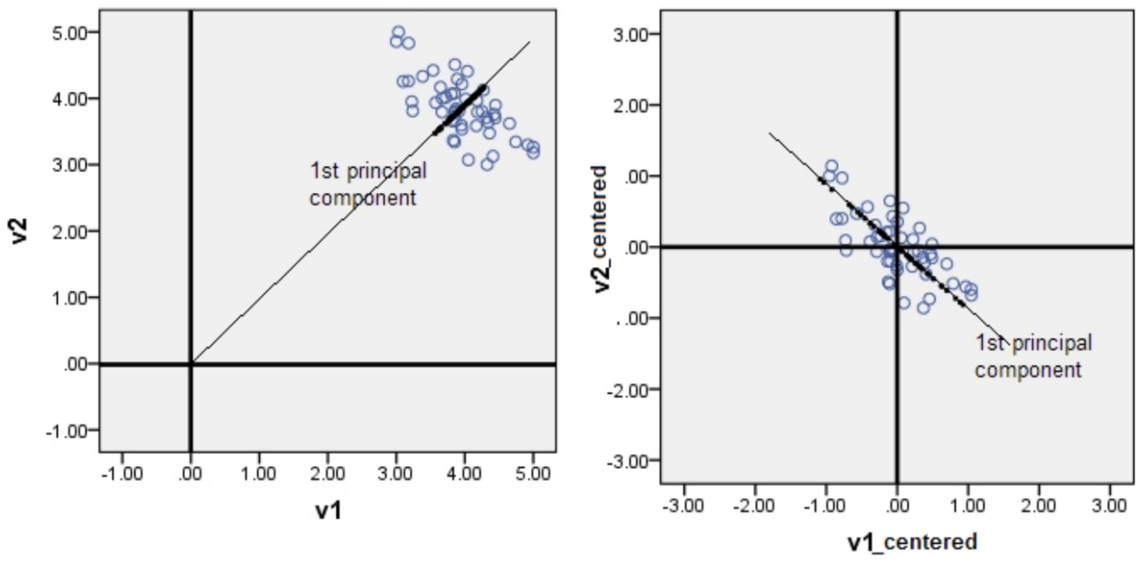
One of the differences between EVD and SVD is that when doing EVD we calculate the eigenvectors of the data's covariance matrix, whereas, when doing SVD the calculation of the eigenvectors is done over the raw data matrix. When performing EVD, since we are calculating the covariance matrix $Cov(\mathbf{X})$, we are intrinsically subtracting the mean, μ_x, from the data hence performing data-centering (Equation 1).

$$Cov(\mathbf{X}) = \sum_{i=0}^{n} (x_i - \mu_x)(x_i - \mu_x)^T \tag{1}$$

However, when doing SVD we use the raw data matrix and hence data-centering is not implicit in the process. Most of the times SVD is used instead of EVD since it is numerically more robust. Furthermore, SVD is actually the algorithm seen in class to perform PCA.

In the case when data is not center, the first eigenvector or PC-1, will just point in the direction of the mean instead of in the direction of maximum variance which is what we want (See figure 1 [1]). This is known as a generic first component. Therefore, to avoid this, we center the data before performing SVD so we get the same results as if we were using EVD but with the advantage of more numerical robustness from SVD.

Figure 1: **Left:** Data not centered, **Right:** Data is centered.



When using EVD, working with centered data provides also some advantages since the mean $\mu_x = 0$ and hence the computation of the covariance matrix, $Cov(\mathbf{X})$, results in a simple multiplication of matrixes which is more efficient (Equation 2).

$$Cov(\mathbf{X}) = \sum_{i=0}^{n} x_i x_i^T = XX^T \tag{2}$$

Question 1.1.2

Consider a data matrix of dimension mn. In some applications the role of points and dimensions can be interchanged. For example, given a document corpus represented as a matrix of type "documents × words", we may want to analyze documents based on which words occur in them, or we may want to analyze words based on which documents they appear in. So it is meaningful to perform PCA both with respect to the rows of a matrix and with respect to its columns. As we discussed in the lectures, PCA relies on SVD. Moreover, since $(\mathbf{U\Sigma V^T})^\mathbf{T} = \mathbf{V\Sigma^T U^T} = \mathbf{V\Sigma' U^T}$, where Σ' differs from $\mathbf{\Sigma}$ only in terms of size, performing SVD on a matrix gives also the SVD on its transpose.

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? Justify your answer.

No. Although we can get the SVD of the transpose $\mathbf{Y}^T$ with only one SVD operation on $\mathbf{Y}$, it does not mean we can use $\mathbf{U}^T$ for the PCA transformation from the the high dimensional data in $\mathbf{Y}^T$ to $\mathbf{X}$. This is because data might not be centered. We might have centered the rows in $\mathbf{Y}$ but it does not mean that the columns, i.e the rows in $\mathbf{Y}^T$, are centered. To perform PCA data needs to be centered, hence we would need to center $\mathbf{Y}^T$ and then apply the SVD operation.

Question 1.1.3

While developing the PCA method, we consider that each data point $\mathbf{y} \in R^d$ is generated by a latent vector $\mathbf{x} \in R^k$, with $k < d$, through a linear transformation

$$\mathbf{y} = \mathbf{Wx}$$

where $\mathbf{W} \in R^{d \times k}$ is a matrix with orthonormal columns. We then deduced that the inverse map is obtained by

$$\mathbf{x} = \mathbf{W}^+\mathbf{y}$$

where $\mathbf{W}^+$is the pseudo-inverse, or Moore-Penrose inverse, of $\mathbf{W}$ obtained via SVD.

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

We have that $\mathbf{y} = \mathbf{Wx}$ and we want to obtain the inverse mapping from high-dimensional data points, $\mathbf{y}$, to embeddings, $\mathbf{x}$. Therefore we need to multiply by $\mathbf{W}^{-1}$ on both sides:

$$\mathbf{W}^{-1}\mathbf{y} = \mathbf{W}^{-1}\mathbf{Wx} \tag{3}$$

However, since $\mathbf{W}$ is not a square matrix ($\in R^{d \times k}$, where $d \neq k$) it does not have an inverse $\mathbf{W}^{-1}$. Therefore we want to use the pseudo-inverse $\mathbf{W}^+$ to be able to obtain the inverse mapping:

$$\mathbf{W}^+\mathbf{y} = \mathbf{W}^+\mathbf{Wx} = \mathbf{x} \tag{4}$$

Moreover is very convenient to use the pseudo-inverse since $\mathbf{W}$ has orthonormal columns and hence the pseudo-inverse turns to be simply the transpose $\mathbf{W}^T$. ($\mathbf{W}^+ = (\mathbf{W}^T\mathbf{W})^{-1}\mathbf{W}^T = \mathbf{W}^T$).

Question 1.1.4

In the lectures, we derived PCA using the criterion of minimizing the reconstruction error. Another commonly-used criterion to derive PCA is to ask to maximize the variance of the data when projected on the lower-dimension space.

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.

From John A. Lee's book, [2], when deriving PCA using the criterion of minimizing the reconstruction error we achieve this result:

$$\hat{\mathbf{X}} = \mathbf{I}_{K\times D}\mathbf{V}^T\mathbf{Y} \tag{5}$$

where $\hat{\mathbf{X}}$ are the K-dimensional latent variables, $\mathbf{V}$ is a matrix containing left singular values of $\mathbf{Y}$, the centered data matrix.

Now, deriving PCA using the criterion of maximizing the variance of the projected data as done in Lee's book [2]: Assuming that the latent variables in $\mathbf{X}$ are uncorrelated. This means that the covariance matrix of $\mathbf{X}$, defined as

$$\mathbf{C}_{\mathbf{XX}} = E\left\{\mathbf{X}\mathbf{X}^T\right\} \tag{6}$$

provided $\mathbf{X}$ is centered, is diagonal. However, after the axis change induced by $\mathbf{W}$, it is very likely that the observed variables in $\mathbf{Y}$ are correlated, i.e., $\mathbf{C}_{\mathbf{YY}}$ is no longer diagonal. The goal of PCA is then to get back the K uncorrelated latent variables in $\mathbf{X}$. Assuming that the PCA model holds and the covariance of $\mathbf{Y}$ is known, we find that

$$\mathbf{C}_{\mathbf{YY}} = E\left\{\mathbf{Y}\mathbf{Y}^T\right\} = E\left\{\mathbf{W}\mathbf{X}\mathbf{X}^T\mathbf{W}^T\right\} = \mathbf{W}E\left\{\mathbf{X}\mathbf{X}^T\right\}\mathbf{W}^T = \mathbf{W}\mathbf{C}_{\mathbf{XX}}\mathbf{W}^T \tag{7}$$

Since $\mathbf{W}^T\mathbf{W} = \mathbf{I}$, left and right multiplications by, respectively, $\mathbf{W}^T$ and $\mathbf{W}$ lead to

$$\mathbf{C}_{\mathbf{XX}} = \mathbf{W}^T\mathbf{C}_{\mathbf{YY}}\mathbf{W} \tag{8}$$

Next, the covariance matrix $\mathbf{C}_{\mathbf{YY}}$ can be factored by eigenvalue decomposition:

$$\mathbf{C}_{\mathbf{yy}} = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^T \tag{9}$$

where $\mathbf{V}$ is a matrix of normed eigenvectors $\mathbf{v}_d$ and $\mathbf{\Lambda}$ a diagonal matrix containing their associated eigenvalues λ_d, in descending order. Because the covariance matrix is symmetric and semipositive definite, the eigenvectors are orthogonal and the eigenvalues are nonnegative real numbers. Substituting in Eq. (8) finally gives

$$\mathbf{C}_{\mathbf{XX}} = \mathbf{W}^T\mathbf{V}\mathbf{\Lambda}\mathbf{V}^T\mathbf{W} \tag{10}$$

This equality holds only when the K columns of $\mathbf{W}$ are taken colinear with K columns of $\mathbf{V}$, among D ones. If the PCA model is fully respected, then only the first P eigenvalues in $\mathbf{\Lambda}$ are strictly larger than zero; the other ones are zero. The eigenvectors associated with these K non-zero eigenvalues must be kept:

$$\mathbf{W} = \mathbf{V}\mathbf{I}_{K\times P} \tag{11}$$

yielding

$$\mathbf{C}_{\mathbf{XX}} = \mathbf{I}_{K\times D}\mathbf{\Lambda}\mathbf{I}_{K\times P} \tag{12}$$

This shows that the eigenvalues in $\mathbf{\Lambda}$ correspond to the variances of the latent variables (the diagonal entries of $\mathbf{C}_{\mathbf{XX}}$).

In real situations, some noise may corrupt the observed variables in $\mathbf{y}$. As a consequence, all eigenvalues of $\mathbf{C}_{\mathbf{YY}}$ are larger than zero, and the choice of K columns in $\mathbf{V}$ becomes more difficult. Assuming that the latent variables have larger variances than the noise, it suffices to choose the eigenvectors associated with the largest eigenvalues. Hence, the same solution as in Eq. (11) remains valid, and the latent variables are estimated exactly as in the method using the criterion for minimum reconstruction error:

$$\mathbf{Y} = \mathbf{W}\hat{\mathbf{X}} \tag{13}$$

Using the left pseudo-inverse of $\mathbf{W}$, $\mathbf{W}^{+} = (\mathbf{W}^T\mathbf{W})^{-1}\mathbf{W}^T = \mathbf{W}^T$:

$$\mathbf{W}^{+}\mathbf{Y} = \mathbf{W}^{+}\mathbf{W}\hat{\mathbf{X}} \tag{14}$$

$$\mathbf{W}^{T}\mathbf{Y} = \mathbf{W}^{T}\mathbf{W}\hat{\mathbf{X}} \tag{15}$$

$$\mathbf{W}^{T}\mathbf{Y} = \hat{\mathbf{X}} \tag{16}$$

Substituting with Eq. (11):

$$\mathbf{I}_{K\times D}\mathbf{V}^T\mathbf{Y} = \hat{\mathbf{X}} \tag{17}$$

Reordering we get the same as with the minimum reconstruction error in Eq. (5):

$$\hat{\mathbf{X}} = \mathbf{I}_{K\times D}\mathbf{V}^T\mathbf{Y} \tag{18}$$

2 Multidimensional Scaling (MDS) and Isomap

Question 1.2.5

In the derivation of classical MDS with distance matrix, our goal is to derive the Gram matrix (similarity matrix) $\mathbf{S} = \mathbf{Y}^T\mathbf{Y}$ from the distance matrix $\mathbf{D}$, while $\mathbf{Y}$ is unknown. We get $s_{ij} = -\frac{1}{2}\left(d_{ij}^2 - s_{ii} - s_{jj}\right)$. In the lectures we mention the "double centering trick," and how this can be used to solve for matrix $\mathbf{S}$ given $\mathbf{D}$. The mathematical derivation for the "double centering trick" is given in the textbook of Lee and Verleysen, Section 4.2.2.

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

By definition MDS assumes $\mathbf{Y}$ to be centered. The distance matrix $\mathbf{D}$ contains distances who have been calculated from vectors $\mathbf{y}$ whose origin of coordinates is unknown. Therefore, the cosine product s_{ij} can have multiple values depending on the origin of coordinates of the two vectors i and j. By applying the double-centering trick we remove this translational degree of freedom, fixing the center of origin of all the points to the mean of the data, which ensures the embeddings we get will have zero mean.

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 $s_{ij} = -\frac{1}{2}\left(d_{ij}^2 - d_{1i}^2 - d_{1j}^2\right)$, where d_{1i} and d_{1j} are the distances from the first point in the dataset 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.

In this case we do a similiar thing as in the double centering trick. This time, however, instead of removing the translational freedom by fixing the points' center of coordinates to the mean of the data, we do it by centering them around the first point of the data. Both solutions are correct since both are invariant to translation, however, they are different since the resulting embeddings will be centered around different points.

Question 1.2.7

Consider now the classical MDS algorithm when $\mathbf{Y}$ is known. In that case, we form $\mathbf{S} = \mathbf{Y^TY}$ and obtain the MDS embedding by the eigen-decomposition of $\mathbf{S}$. Observe that PCA involves a singular-value decomposition (SVD) operation, while classical MDS involved an eigenvector decomposition (EVD) operation.

Show that the two methods, i.e., classical MDS when Y is known and PCA on Y, are equivalent. Which of the two methods is more efficient? (Hint: Your answer may involve a case analysis.)

We can see in [2] how to demonstrate that metric MDS and PCA give the same solution:

The data coordinates $\mathbf{Y}$ are assumed to be known - this is mandatory for PCA, but not for metric MDS - and centered. Moreover, the singular value decomposition of $\mathbf{Y}$ is written as $\mathbf{Y} = \mathbf{V\Sigma U}^T$. On one hand, PCA decomposes the covariance matrix, which is proportional to $\mathbf{YY}^T$, into eigenvectors and eigenvalues:

$$\hat{\mathbf{C}}_{\mathbf{yy}} \propto \mathbf{YY}^T = \mathbf{V\Sigma}^T\mathbf{U}^T\mathbf{U\Sigma V}^T = \mathbf{V\Sigma\Sigma}^T\mathbf{V}^T = \mathbf{V\Lambda}_{\text{PCA}}\mathbf{V}^T$$

where the division by N is intentionally omitted in the covariance, and $\mathbf{\Lambda}_{\text{PCA}} = \mathbf{\Sigma\Sigma}^T$. The solution is $\hat{\mathbf{X}}_{\text{PCA}} = \mathbf{I}_{K\times D}\mathbf{V}^T\mathbf{Y}$. On the other hand, metric MDS decomposes the Gram matrix into eigenvectors and eigenvalues:

$$\mathbf{S} = \mathbf{Y}^T\mathbf{Y} = \mathbf{U\Sigma V}^T\mathbf{V\Sigma U}^T = \mathbf{U\Sigma}^T\mathbf{\Sigma U}^T = \mathbf{U\Lambda}_{\text{MDS}}\mathbf{U}^T$$

where $\mathbf{\Lambda}_{\text{MDS}} = \mathbf{\Sigma}^T\mathbf{\Sigma}$. The solution is $\hat{\mathbf{X}}_{\text{MDS}} = \mathbf{I}_{K\times N}\mathbf{\Lambda}_{\text{MDS}}^{1/2}\mathbf{U}^T$. By equating both solutions and by again using the singular value decomposition of $\mathbf{Y}$:

$$\begin{aligned}
\hat{\mathbf{X}}_{\text{PCA}} &= \hat{\mathbf{X}}_{\text{MDS}} \\
\mathbf{I}_{K\times D}\mathbf{V}^T\mathbf{Y} &= \mathbf{I}_{K\times N}\mathbf{\Lambda}_{\text{MDS}}^{1/2}\mathbf{U}^T \\
\mathbf{I}_{K\times D}\mathbf{V}^T\mathbf{V\Sigma U}^T &= \mathbf{I}_{K\times N}\left(\mathbf{\Sigma}^T\mathbf{\Sigma}\right)^{1/2}\mathbf{U}^T \\
\mathbf{I}_{K\times D}\mathbf{\Sigma U}^T &= \mathbf{I}_{K\times D}\mathbf{\Sigma U}^T
\end{aligned}$$

In terms of efficency this equivalence comes also very handy when the size of the data matrix $\mathbf{Y}$ becomes problematic. If data dimensions, D, are low but the number of points, N, is huge, PCA spends fewer memory resources than MDS since the product $\mathbf{YY}^T$ (resulting matrix would be $D \times D$) has a smaller size than $\mathbf{Y}^T\mathbf{Y}$ (resulting matrix would be $N \times N$). By contrast, MDS is better when the dimensionality is very high but the number of points rather low because product $\mathbf{Y}^T\mathbf{Y}$ has a smaller size than $\mathbf{YY}^T$. Summarizing:

- When data is low-dimensional but many data points PCA ($\mathbf{O}(D^2)$) is more efficient than MDS ($\mathbf{O}(N^2)$).
- When data is high-dimensional but few data points PCA ($\mathbf{O}(D^2)$) is less efficient than MDS ($\mathbf{O}(N^2)$).

Question 1.2.8

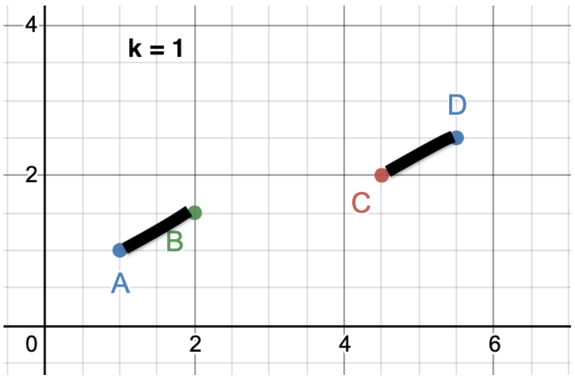
Consider the Isomap method used to reduce the dimensionality of a given dataset. Isomap requires constructing a neighborhood graph G, as discussed in the lectures and the textbook.

Argue that the process to obtain the neighborhood graph G in the Isomap method may yield a disconnected graph. Provide an example. Explain why this is problematic

The graph G for Isomap is constructed by first calculating the euclidean distances between all points and then connecting each point with the k nearest neighbors. In some cases if k is too small some points will not connect.

In the simple example of figure 2 it can be seen how in the case of using $k = 1$ in the ISOMAP algorithm we would end with a disconnected graph because we are connecting points only to its closest neighbor. We would neeed to increase k, i.e increase robustness, so we don't end with a disconnected graph.

(a) 4 points in 2-Dimensional space forming a 1-D manifold



(b) 10 points in 3-Dimensional space forming a 2-D manifold

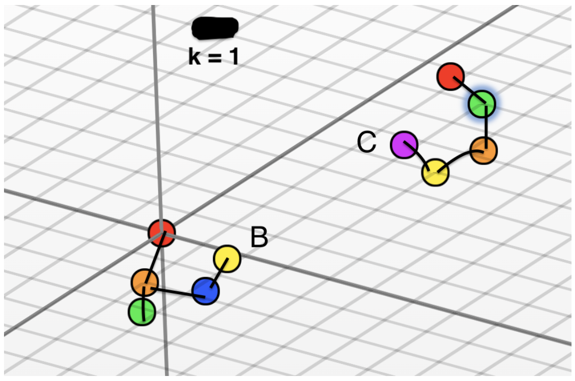


Figure 2: Disconnected graphs example.

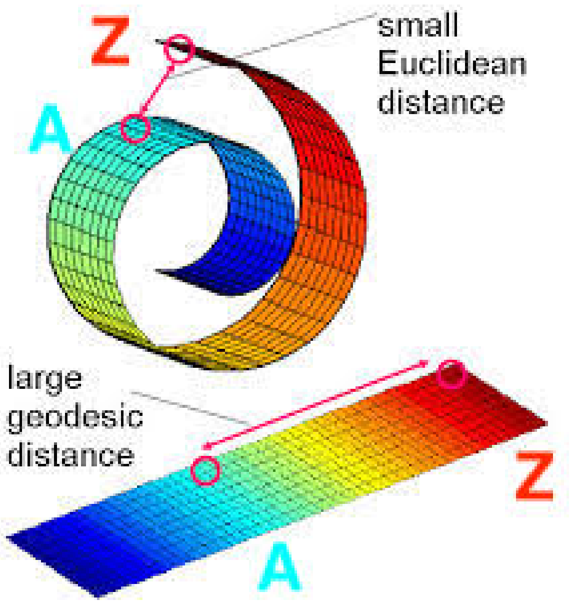
The graphs like the ones in figure 2 yield a geodesic distance of infinity between point B and C. When calculating the similarity between B and C we obtain minus infinity. Therefore the disconnection in the graph leads to an incorrect result where the embedding representation does not maintain the data point relationships of the original dimensional space.

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?

A possible heuristic would be to replace the infinity distance between points i and j by the euclidean distance between points. It would work well for some manifolds like the ones in figure 2 but not so well in other, more complex, manifolds such as the one in figure 3. In the Swish roll manifold of figure 3 we can see how if point Z is the one disconnected from the graph then, points like A and its surroundings would have a metric distance smaller to what the actual distance (geodesic distance) in the low-dimensional embedding should be. This is due to the fact that euclidean distance in that case does not accurately represent the distances between some points in the manifold.

Figure 3: Swish roll manifold



However in some other cases, like the ones in figure 2 the heuristic would work great. This is due to the fact that if we had drawn the graph passing from B to C and calculated the distances through the graph, the euclidean and geodesic distance would be very similar.

3 PCA vs. Johnson-Lindenstrauss random projections

Question 1.3.10

Both PCA and Johnson-Lindenstrauss random projections are linear maps. Given data $\mathbf{Y} = (\mathbf{y}_1, \dots, \mathbf{y}_n) \in R^{d \times n}$, both methods find a matrix $\mathbf{A} \in R^{k \times d}$ and reduce the dimension of the data from d to k by computing the projection $\mathbf{X} = \mathbf{AY}$.

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

In Johnson-Lindenstrauss random projections the projection error is minimized for all pairs of points (worst case error). However, in PCA, we are minimizing the error for the expected distance, i.e, the average error for all pairs of points.

In terms of computational efficiency, random projections has a computational-complexity of $O(nkd)$, where n = number of points in $\mathbf{Y}$, k = the embeddings dimension and d = the original dimensions of data. PCA has a computational-complexity of $O(d^2n)+O(d^3)$ [3]. Therefore, random projections method is much more efficient than PCA.

In terms of use cases, we would like to always use PCA since its more robust. However, when the data has a lot of dimensions, we might consider using random projections. As we have seen above, computational efficiency is much better with random projections. Therefore, we might want to sacrifice the robustness of the solution given by PCA for the efficiency of random projections.

4 Programming tasks- 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 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.

I have used the API provided by `https://www.distance24.org/` which returns distances in miles between 2 pairs of cities. In total I have used 122 cities across 4 continents.

Question 1.4.12

Apply classical MDS to compute an (x, y) coordinate for each city in your dataset, given the distance matrix 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.

The reconstructed map of figure 4 is pretty good. In general it has kept North, West, East and South directions except in the case of North America where it seems to have been rotated 90 degrees to the East. On the other hand, if we take the case of Europe we can see it has not only kept cardinal direction but also recreates the real map very well, Madrid, Barcelona and Rome are very close together in the South, Reykjavik is off shore in Iceland and Helsinki and northern capital cities are displayed where they should. Africa and Asia also look good. South America looks good but Brasilia, Montevideo and Buenos Aires might be a little bit to close. In general, what matters is that the relative distances between cities have been maintained.

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.

I have employed sklearn's implementation [4]. With this implementation the main parameters that can be tuned are these two: n_components and n_init. N_components has to be 2 since we are asked to replicate question 1.4.12 where a set of coordinates (x, y) are requested and n_init is the number of times the optimization algorithm (SMACOF) is ran with different initialization, then the algorithm returns the best one among those. I have ran the algorithm with n_init=[2,16].

As we can see in both maps, figure 5 and figure 6, the quality is the same as in classical MDS as the relative distances are still preserved despite the rotations that can be seen among the continents/areas.

Figure 4: Map obtained when using Classical MDS

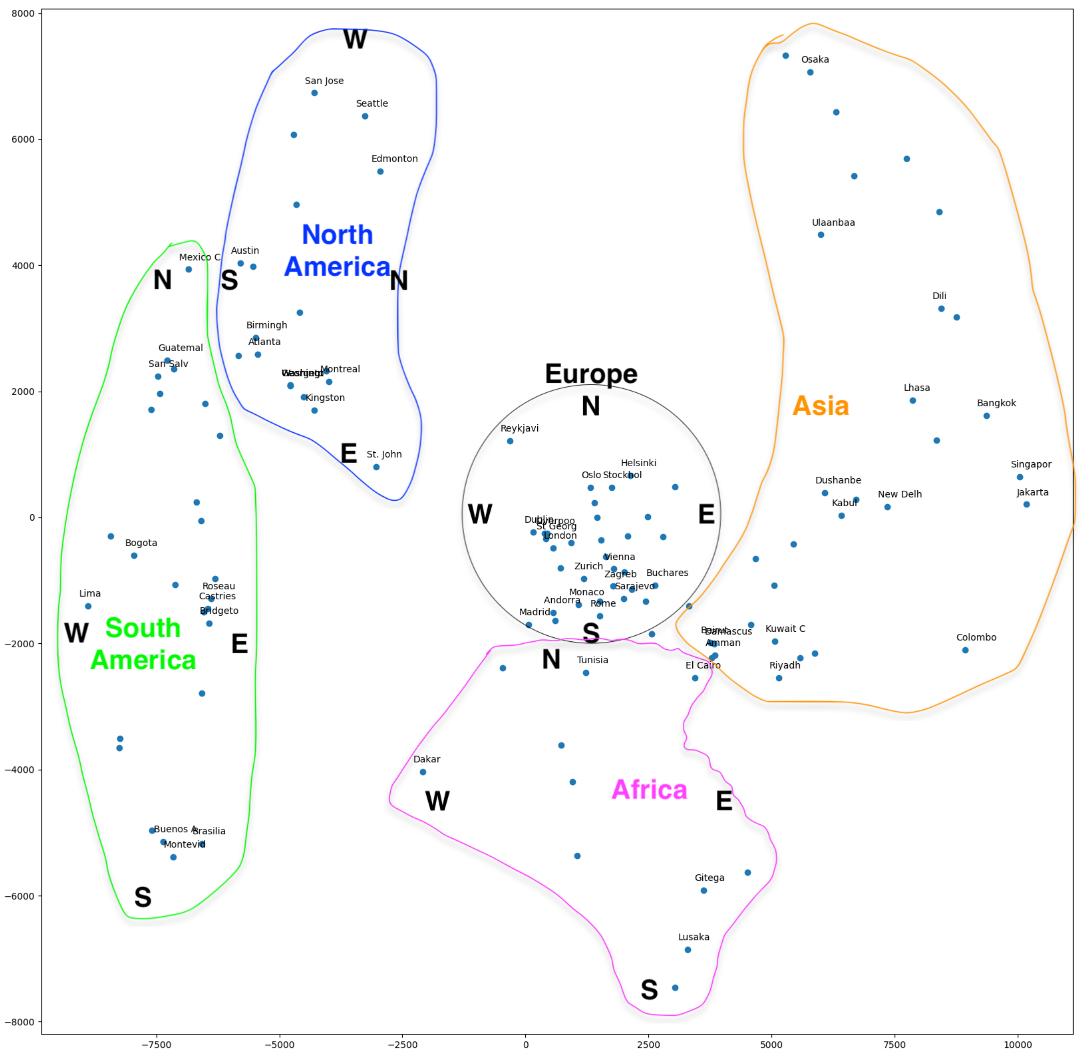


Figure 5: Map obtained when using Metric MDS and n_init=2

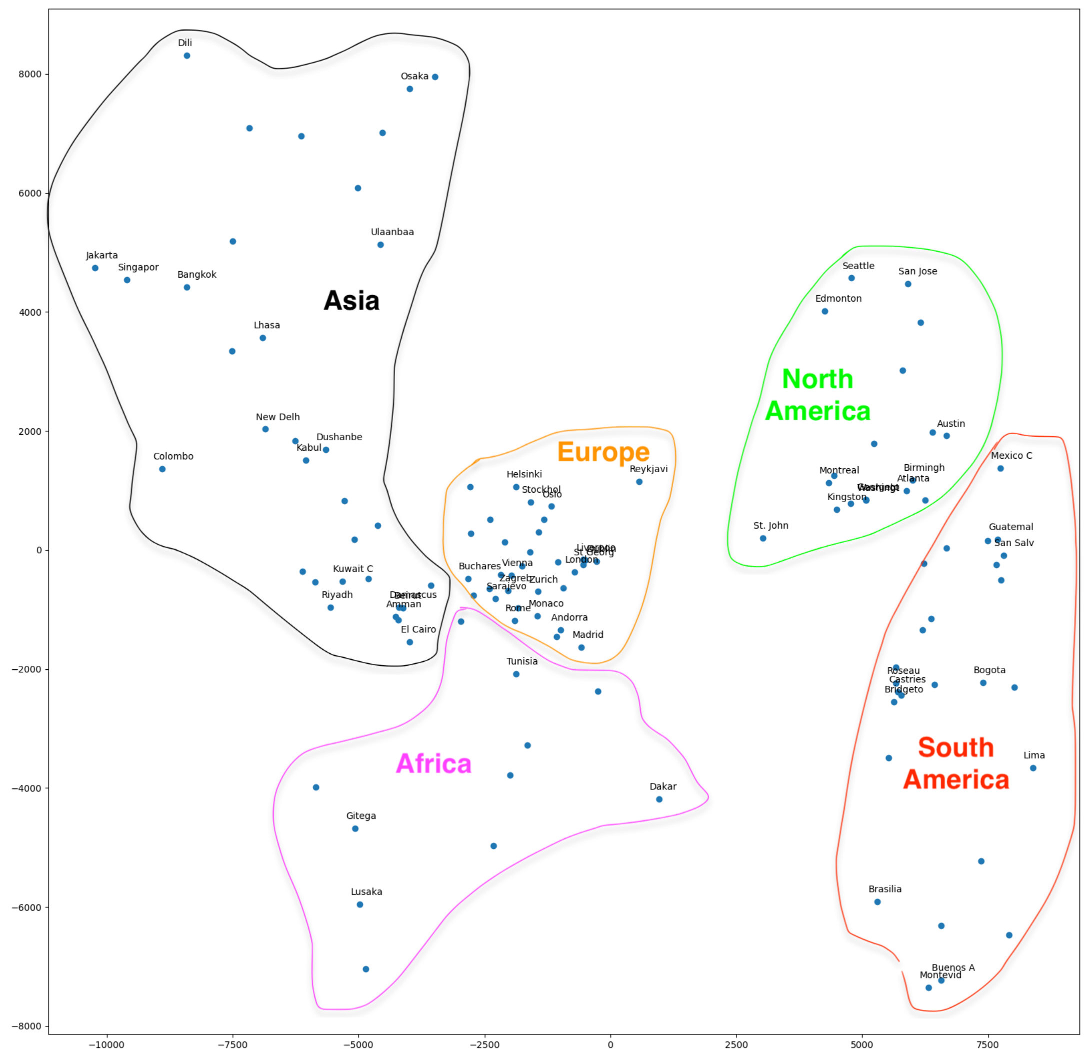
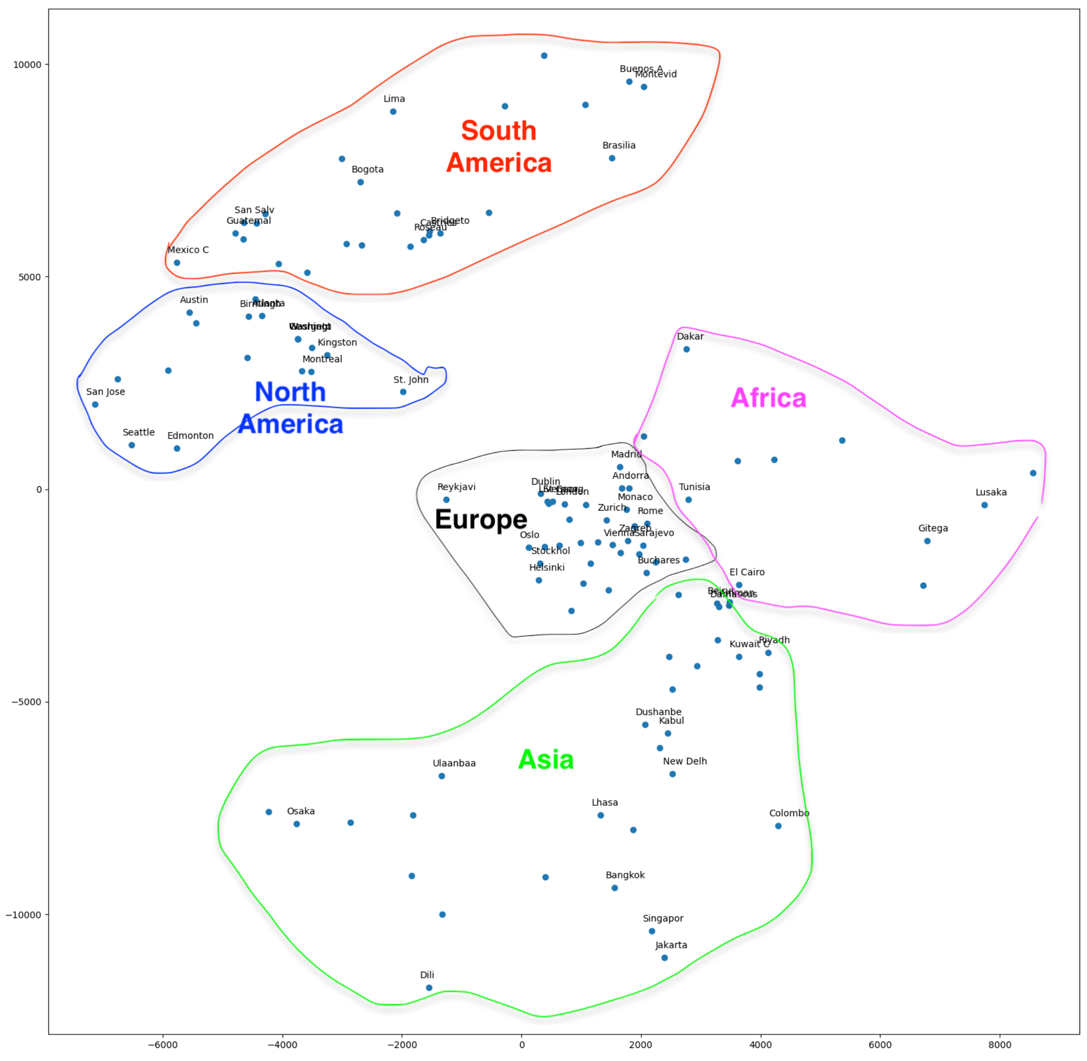


Figure 6: Map obtained when using Metric MDS and n_init=16



List of Figures

References

[1] *How does centering the data get rid of the intercept in regression and pca?* [Online]. Available: `https://stats.stackexchange.com/questions/22329/how-does-centering-the-data-get-rid-of-the-intercept-in-regression-and-pca`.

[2] J. A. Lee and M. Verleysen, *Nonlinear dimensionality reduction.* Springer, 2007.

[3] S. Deegalla and H. Bostrom, "Reducing high-dimensional data by principal component analysis vs. random projection for nearest neighbor classification," *2006 5th International Conference on Machine Learning and Applications (ICMLA'06)*, 2006. DOI: `10.1109/icmla.2006.43`.

[4] F. Pedregosa *et al.*, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

5 Appendix

```
import requests
import numpy as np
import json
import matplotlib.pyplot as plt
from sklearn.manifold import MDS

distances = {
    ”Stockholm”: {}, ”Budapest”: {}, ”Oslo”: {}, ”Copenhagen”: {}, ”Helsinki”:
    ”Berlin”: {}, ”London”: {}, ”Amsterdam”: {},
    ”Vienna”: {}, ”Warsawa”: {}, ”Madrid”: {}, ”Paris”: {}, ”Rome”: {}, ”Mosco
    ”Dublin”: {}, ”Athens”: {}, ”Zurich”: {}, ”Prague”: {}, ”Reykjavik”: {},
    ”Kiev”: {}, ”Bucharest”: {}, ”Minsk”: {}, ”Monaco”: {}, ”San␣Marino”: {},
    ”Sofia”: {}, ”Zagreb”: {}, ”Belgrade”: {},
    ”Andorra␣la␣Vella”: {}, ”Gothenburg”: {}, ”Birmingham”: {}, ”Manchester”:
    ”Liverpool”: {}, ”Barcelona”: {}, ”Amman”: {},
    ”Ashgabat”: {}, ”Dili”: {}, ”Jerusalem”: {}, ”Dushanbe”: {}, ”Doha”: {},
    ”Lhasa”: {}, ”Baghdad”: {}, ”Beirut”: {}, ”Baku”: {}, ”Colombo”: {},
    ”Islamabad”: {}, ”Singapore”: {}, ”Tokyo”: {}, ”Kabul”: {}, ”Beijing”: {},
    ”Damascus”: {}, ”Seoul”: {}, ”New␣Delhi”: {}, ”Ankara”: {},
    ”Bangkok”: {}, ”Hanoi”: {}, ”Riyadh”: {}, ”Abu␣Dhabi”: {}, ”Kuwait␣City”:
    ”Dhaka”: {}, ”Osaka”: {}, ”Tehran”: {}, ”Ulaanbaatar”: {},
    ”Taipei”: {}, ”Jakarta”: {}, ”Manila”: {}, ”Washington”: {}, ”Ottawa”: {},
    ”Mexico␣City”: {}, ”Belmopan”: {}, ”Guatemala␣City”: {},
    ”Tegucigalpa”: {}, ”San␣Salvador”: {}, ”Managua”: {}, ”San␣Jose”: {},
    ”Panama␣City”: {}, ”St.␣John's”: {}, ”Nassau”: {}, ”Bridgetown”: {},
    "Havana": {}, "Roseau": {}, "Santo Domingo": {}, "St Georges": {},
    "Port-au-Prince": {}, "Kingston": {}, "Basseterre": {}, "Castries": {},
    "Kingstown": {}, "Buenos Aires": {}, "Sucre": {}, "Brasilia": {},
    "Santiago": {}, "Bogota": {}, "Quito": {}, "Georgetown": {}, "Asuncion": {
    "Lima": {}, "Paramaribo": {}, "Montevideo": {}, "Caracas": {},
    "WashingtonDC": {}, "NewYork": {}, "Montreal": {}, "Chicago": {}, "Atlanta
    "Dallas": {}, "Austin": {}, "Denver": {}, "Seattle": {}, "Las Vegas": {},
    "Edmonton": {}, "Casablanca": {}, "Tunisia": {}, "Tamanrasset": {},
    "El Cairo": {}, "Agadez": {}, "Dakar": {}, "Duala": {}, "Gitega": {},
    "Nairobi": {}, "Lusaka": {}, "Johannesburgo": {}
}


def get_distances():
    endpoint = "https://www.distance24.org/route.json?stops="

    for start_city in distances.keys():
        print(f"Calculating distances for {start_city}")
        for end_city in distances.keys():
            if start_city not in distances[end_city].keys():
                cities = f"{start_city}|{end_city}"
                url = f"{endpoint}{cities}"
                response = requests.get(url).json()
                dist = response["distance"]
            else:
                dist = distances[end_city][start_city]

            distances[start_city][end_city] = dist
            print(f"{start_city} -> {end_city}: {dist}")

    print(distances)

    distList = []

    for start_city in distances.keys():
        dists = []
        for d in distances[start_city].values():
            dists.append(d)
        distList.append(dists)

    distMatrix = np.array(distList)

    print("\n\n\t##Matrix##")
    print(distMatrix)

    with open('distances.txt', 'w') as outfile:
        json.dump(distances, outfile)


def classical_mds():
    '''
    Apply classical MDS to compute an (x, y) coordinate
```

```
for each city in yourdataset,
given the distance matrix D. Plot the cities on a plane
using the coor-dinates you computed.
You may want to annotate the cities by their name,
orabbreviation, use different colors
to indicate continents, or regions, etc.
Discusshow good is the reconstructed map your
created using classical MDS. For thistask, you should
implement MDS by yourself, by relying only
on a package foreigenvector decomposition, that is, do not
try to find a library that implements MDS
'''
# Loading downloaded distances
with open('distances.txt') as json_file:
    distances_json = json.load(json_file)

# Converting distances to numpy matrix
distList = []

for start_city in distances_json.keys():
    dists = []
    for d in distances_json[start_city].values():
        dists.append(d)
    distList.append(dists)

distMatrix = np.array(distList)

print(
    f"\t##␣Distance␣Matrix␣##\t({distMatrix.shape[0]}x{distMatrix.shape[1]
print(distMatrix)

distMatrix = distMatrix**2
print(
    f"\t##␣Distance␣Matrix␣##\t({distMatrix.shape[0]}x{distMatrix.shape[1]
print(distMatrix)

N = distMatrix.shape[0]
K = 2  # Since we want to plot (x, y) coordingates, i.e, 2-dimesnions

# Compute Centering Matrix for double centering
centeringMatrix = np.identity(N) - (np.ones((N, 1)) @ np.ones((1, N)))/N

# Compute Gram matrix with double centering from distance matrix
gramMatrix = -(1/2)*(centeringMatrix @ distMatrix @ centeringMatrix)

# Eigen-decomposition
w, v = np.linalg.eigh(gramMatrix)
# The values are returned in ascending order, and we want the K greatest s
w = w[-K:]
v_T = v.T[-K:]

assert np.all(w > 0)
```

```
    latentMatrix = np.diag(np.sqrt(w)) @ v_T

    print(
        f"\t## Latent Matrix ##\t({latentMatrix.shape[0]}x{latentMatrix.shape[
    print(latentMatrix)

    plt.figure(figsize=(20, 20))

    plt.scatter(latentMatrix[0, :], latentMatrix[1, :])
    labels = [city[:8] for city in distances_json.keys()]
    for (index, label), x, y in zip(enumerate(labels), latentMatrix[0, :], lat
        if index % 2 == 0:
            plt.annotate(
                label,
                xy=(x, y),
                xytext=(-10, 10),
                textcoords='offset points'
            )
    plt.savefig('classical-map.png')


def metric_mds():
    '''
    Repeat the task of 1.4.12, but using metric MDS this time.
    You may tune theparameters of the method until you are
    satisfied with the results.
    Discuss theparameters that you tuned and their effect
    on the end result.
    Discuss the qualityof 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 useit as a black box.
    '''
    # Loading downloaded distances
    with open('distances.txt') as json_file:
        distances_json = json.load(json_file)

    # Converting distances to numpy matrix
    distList = []

    for start_city in distances_json.keys():
        dists = []
        for d in distances_json[start_city].values():
            dists.append(d)
        distList.append(dists)

    distMatrix = np.array(distList)

    mds = MDS(n_components=2, metric=True, n_init=16,
              verbose=3, dissimilarity="precomputed")
    latentMatrix = mds.fit_transform(distMatrix)
    print(
        f"\t##␣Latent␣Matrix␣##\t({latentMatrix.shape[0]}x{latentMatrix.shape[
    print(latentMatrix)

    plt.figure(figsize=(20, 20))

    plt.scatter(latentMatrix[:, 0], latentMatrix[:, 1])
    labels = [city[:8] for city in distances_json.keys()]
    for (index, label), x, y in zip(enumerate(labels), latentMatrix[:, 0], lat
        if index % 2 == 0:
            plt.annotate(
                label,
                xy=(x, y),
                xytext=(-10, 10),
                textcoords='offset␣points'
            )
    plt.savefig('metric-map.png')

    print(mds.get_params())


def main():
    # get_distances()
    classical_mds()
    metric_mds()


main()
```