Project 2 NLA: SVD applications

This document contains the explanation for the Least Square Problem and Principal Component Analysis experiments.

Least Squares Problem

The first problem we are asked to solved is to write a program that solves the Least Square problem for two datasets (**datafile.txt** and **datafile2.csv**) using Singular Value Decomposition. Moreover it is also required to compare the results with the ones obtained in the practice 4 in which we solved the Least Square problem using QR factorization.

(1) First we are going to tackle the problem for the dataset **datafile.txt**.

To obtain the results for the Least Square problem using Singular Value Decomposition execute **LS_SVD_1.py**.

And the results are:

Solution: [0.20000039, 1.31085632, 0.98928421]
Norm: 1.6543964762623309
Least Square Error: 10.88082840185219

Before finding the solution using QR Factorization we need to check if the matrix of our dataset is full rank or not in order to solve the Least Square problem in one way or another. When the matrix is full rank, it means that the rank is equal to the number of columns and the LS problem has unique solution. Otherwise, when the rank is lower than the number of columns, our matrix is rank deficient and the LS problem has no unique solution, so the problem will be approached in order to find the solution which minimizes the $\|.\|_2$-norm.

To know if the matrixes of the dataset **datafile.txt** is full rank or not execute **is_full_rank_1.py**. This code calculates the rank of the matrix, compare it to the number of columns and notify us if the matrix is full or rank deficient.

In this first case the matrix is full rank.

The code **LS_QR_fact_full_rank.py** finds the unique solution that makes minimum$\|Ax - b\|_2$, and the results for the **datafile.txt** are:

Solution: [0.20000039, 1.31085632, 0.98928421]
Norm: 1.6543964762623324
Least Square Error: 10.88082840185219

The same result than in the previous section has been obtained.

(2) Now we are going to face the problem for the dataset **datafile2.csv**

To obtain the results for the Least Square problem using Singular Value Decomposition execute **LS_SVD_2.py**.

And the results are:

Solution: [8.30203901e+00, 8.30408855e+00, -1.88268350e+03, 2.99498592e+04,
-2.12104358e+05, 8.37034713e+05, -2.00324191e+06, 2.97903439e+06,
-2.69206453e+06, 1.35366101e+06, -2.90442987e+05]
Norm: 4774736.291605193
Least Square Error: 1.1495978958989292

Remember to check if the matrix of this second dataset is full o rank deficient before trying to solve the LSE problem using QR Factorization.

To know if the matrixes of the dataset **datafile2.csv** is full rank or not execute **is_full_rank_2.py**

In this second case the matrix is rank deficient, and then another QR Factorization approach is required.

LS_QR_fact_rank_deficient.py solves the LSE problem using QR Factorization for rank deficient matrices. Execute it to obtain the results.

Solution: [1.66061275e+01, 0.00000000e+00, -1.88268349e+03, 2.99498591e+04,
-2.12104358e+05, 8.37034712e+05, -2.00324191e+06, 2.97903439e+06,
-2.69206452e+06, 1.35366101e+06, -2.90442987e+05]
Norm: 4774736.282833208
Least Square Error: 1.1495978959555873

The result obtained is one of the multiple solutions for this rank deficient system, but with the particularity that the solution provided minimizes the square error.

If we compare the results for this second dataset using both methods we clearly observe that the solutions are not identical, even though some of the components coincide, but the Least Square Error is practically the same.

One thing that we can conclude of this experiment is that when we are in front of a Least Square problem, and we want to solve it using QR Factorization, we need to keep in mind the rank of the matrix of our dataset. On the other side, if we want to use Singular Value Decomposition to solve it, no consideration about the rank is required.

Arrived to this point I would like to investigate a bit about the stability of solutions achieved. Stability relates how a solution *x* for a given system $Ax = b$ changes in front of small variations in the dataset matrix *A*. This fact conditions our system. A matrix is ill-conditioned if a small change in it can cause large changes in the solution *x*, otherwise it is said that the matrix is well-conditioned. The condition number of a matrix *k(A)* tells us how conditioned it is; if *k(A)* is low, the matrix is well conditioned, if *k(A)* is large, the matrix is ill-conditioned. Definitely, the accuracy of a solution depends on the condition number of the matrix.

To study the stability of each solution we have created a code that computes the condition number using the Singular Value Decomposition of the matrix dataset. The tolerance value used to obtain these condition numbers is the same used during all the previous experiments, 1e-9.

Execute **Dataset_Condition_Number.py** to see the results.

datafile.txt

Condition number: 227.5857285742449

datafile2.csv

Condition number: 75071531.9746304

In view of these results we can conclude that the second datafile is more sensible to small changes than the first one. Small changes in the **datafile2.csv** might cause huge variations in the solution. For example, if this LSE problem for the **datafile2.csv** was performed in another machine with a different precision, the solution obtained would be very different from the one obtained here. However, if the same experiment was done for **datafile.txt**, the solution obtained in the new machine would not differ so much from the current one.

Principal Component Analysis

This part consists on applying a Principal Component Analysis over two datasets

(1) The first dataset is **example.dat**, we are asked to perform PCA in two different ways; decomposing Singular Values from the covariance matrix and from the correlation matrix.

To obtain the results for this analysis execute **PCA_1.py**.

This code performs SVD over both matrices; first over covariance and then over correlation. For each analysis, the code writes down:

- The portion of the total variance accumulated in each of the Principal Components.
- The standard deviation of each of the Principal Components.
- The components of the original dataset in the new PCA coordinates.

One interesting thing that can be appreciated from the comparison of the results is that the proportion of the total variance in the first Principal Component is higher when PCA is performed on covariance matrix (66,97%) than in correlation matrix (60,75%). The explanation that I provide to this fact is the following. A Principal Component is a linear combination of the variables with the purpose to maximize the variance for this component, so when the data is not standardized (using covariance matrix) the variables with higher variance will tend to dominate the first component and increase the portion of variance accumulated on it.

(2) The second dataset is **RCsGoff.csv**. In this case we are just performing the SVD over the correlation matrix.

To obtain the results for this analysis execute **PCA_2.py**.

As in the previous case the code writes down:

- The portion of the total variance accumulated in each of the Principal Components.
- The standard deviation of each of the Principal Components.
- The components of the original dataset in the new PCA coordinates.

After the execution of this code a csv file is generated, **PCA_dataset2.csv**. This file contains the new coordinates of the dataset as a dataframe, whose columns are the Principal Components and indexes the different observations. Moreover, another row has been concatenated, the percentage of total variance accumulated in each Principal Component.

To visualize this dataframe execute the following lines of code:

```
import pandas as pd
pd.read_csv('PCA_dataset2.csv', sep=',', index_col=0)
```

Once the results have been acquired, further analysis is required. This dataset consists of 20 observations measuring the amount of 58581 genes present on each one. So, which is the number of Principal Components needed to explain the dataset?

Different criterions have been applied.

3/4 of the total variance rule

The rule 3/4 of the total variance encourages to use the Principal Components that maintain the third fourth parts of the whole variance.

To obtain the number of Principal Components for this rule execute **3_4_rule.py**.

Under this rule 2 Principal Components are needed to explain the dataset.

Kaiser rule

The Kaiser rule is to drop all components with eigenvalues under 1, considering the eigenvalue equal to the information accounted for a single component.

To obtain the number of Principal Components for this rule execute **Kaiser_rule.py**.

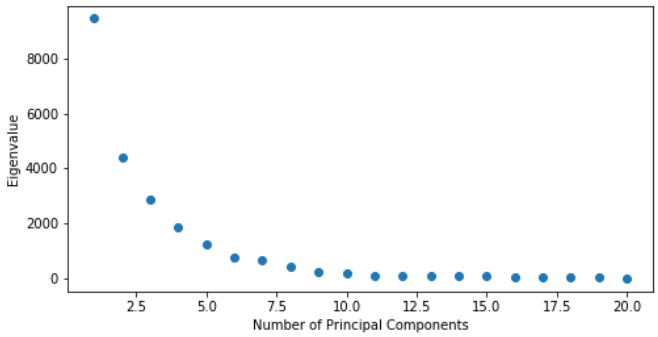
Under this rule 19 Principal Components are needed to explain the dataset.

Scree plot

The Scree plot rule represents the eigenvalues on the y-axis and the number of Principal Components on the x-axis. The number of Principal Components needed is decided observing the plot described previously, at the point when the slope of the curve clearly starts to tend to 0 ("the elbow") the value at the x-axis indicated the number of Principal Components required.

To obtain the number of Principal Components for this rule execute **Scree_plot_rule.py**.

The result is the following plot:

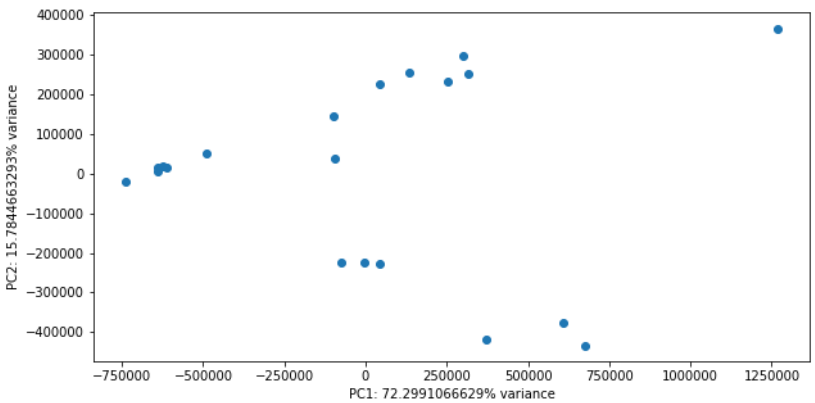


We can infer from it that the optimal number of Principal Components could be 5.

It is important to keep in mind that one of the reasons for running a PCA is to reduce the large number of variables that describe complex observations. In other words, we would like to find a smaller number of interpretable components that explain the maximum amount variability in the data.

Due to this reasoning in order to be able to plot the dataset we use the 3/4 of the total variable rule that recommend us to use the two Principal Components.

Execute the code **plot_RCsGoff.py** to visualize the new representation of RCsGoff dataset.



The X-axis corresponds to the PC1, which accumulates the 72.2991% of the variance, and the Y-axis is the PC2, which accumulates the 15.7845% of the variance. To verify the correct performance of the implementation, we verify that the plot above is the same as the one provided in the statement of this project.