Numerical Linear Algebra

Master on Fundamentals of Data science
Universitat de Barcelona

Project 2
SVD Applications

Pablo Álvarez
Blai Ras

December 9, 2020

Contents

Note: this project was coded between Pablo Alvarez and me, Blai Ras. Nevertheless, I'm the author of the following paper, meaning that each one of us did a different conclusion separately.

1 1. Least Squares problem

The Least Squares problem can appear, for example, when we have an overdetermined system, meaning that we might have no unique solution because we have more equations than unknowns. In that case LS comes in action by finding the solution that minimizes the Euclidean norm of the residuals, that is:

$$\min_{\mathbf{x}\in\mathbb{C}^n} \|\mathbf{b} - \mathbf{Ax}\|.$$

When we have a problem with a full-rank matrix, like the one in `datafile.csv`, the LS solution can be easily found as a solution of the linear system:

$$\mathbf{A}^T\mathbf{Ax} = \mathbf{A}^T\mathbf{b}.$$

In such case, $\mathbf{A}^T\mathbf{A}$ is a positive-definite matrix and therefore the LS solution is unique.

In a "tall" matrix, if we look at the this procedure numerically, we can see that its a very bad approach: we almost are squaring the condition number, so for problems with big dimensions like $m \times n \approx 16$ or more could be intractable. Thankfully, we have QR and SVD to solve the LS problem.

For the rank deficient problem, where $m > n > \text{rank}(A)$, the LS solution is not unique, so we can act in two different ways: use the Singular Value Decomposition or QR with column-pivoting. Let's dig deep onto SVD for Least Squares, $\mathbf{A} = \mathbf{U\Sigma V}^T$:

$$\|\mathbf{b} - \mathbf{Ax}\| = \|\mathbf{U}^T\mathbf{b} - \mathbf{U}^T\mathbf{Ax}\| = \|\mathbf{U}^T\mathbf{b} - \mathbf{\Sigma V}^T\mathbf{x}\|.$$

Let's define $\mathbf{c} = \mathbf{U}^T\mathbf{b}, \quad \mathbf{y} = \mathbf{V}^T\mathbf{x}$. Then $\|\mathbf{b} - \mathbf{Ax}\|$ is minimized if only $\|\mathbf{c} - \mathbf{\Sigma y}\|$ is minimal. Also, $\|\mathbf{c} - \mathbf{\Sigma y}\|$ is minimal if and only if $\mathbf{c} = \mathbf{\Sigma y}$. This means that we are transforming the LS problem with matrix A and b into a problem of diagonal matrix Σ and vector c. Hence,

$$\mathbf{x} = \mathbf{Vy} = \mathbf{V\Sigma}^{-1}\mathbf{c} = \mathbf{V\Sigma}^{-1}\mathbf{U}^T\mathbf{b}.$$

So, after the SVD, the inverse of Σ would be:

$$\mathbf{\Sigma}^{-1} = \begin{pmatrix} \tilde{\mathbf{\Sigma}}^{-1} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{pmatrix}, \quad \tilde{\mathbf{\Sigma}}^{-1} = \begin{pmatrix} 1/\sigma_1 & 0 & \dots & 0 \\ 0 & 1/\sigma_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1/\sigma_r \end{pmatrix}.$$

As stated in the PDF, we can use the Moore-Penrose pseudoinverse of matrix A and denote it as $\mathbf{A}^+$. Then, our solution is:

$$\mathbf{x} = \mathbf{A}^+\mathbf{b}.$$

We have to keep in mind that computing the SVD may be an expensive operation. QR with pivoting, therefore, must be taking in consideration. I won't be proving the whole process but the idea of applying the pivoting to the full, non-reduced QR factorization would result in:

$$\begin{aligned} A &= QR \\ A\Pi &= QR\Pi \\ \underbrace{A}_{m\times n} &= \underbrace{Q}_{m\times m} \underbrace{\begin{bmatrix} R \\ 0 \end{bmatrix}}_{m\times n} \Pi \\ \underbrace{A}_{m\times n}\underbrace{\Pi}_{n\times n} &= \underbrace{Q}_{m\times m} \begin{bmatrix} \underbrace{R_{11}}_{(r\times r)} & \underbrace{R_{12}}_{(n-r)\times r} \\ 0 & 0 \end{bmatrix} \end{aligned}$$

Where $r = rank(A)$ and $rank(R_{11}) = r$.

So, in our code both methods are applied on each dataset as the PDF dictates. As you can see when running the `LS.py`, the execution time isn't long at all, a couple of seconds at most. Nevertheless, my conclusions are:

- Normally, I can always know the rank of the matrix of my problem. If, in any case, I could not know this rank, I would always use SVD, given the fact that it provides a solution as long as the data vector is not in the null space.
- On a daily, I would use QR factorization, given the fact that is more stable than performing $\mathbf{A}^T\mathbf{A}\mathbf{x} = \mathbf{A}^T\mathbf{b}$ with "tall" matrices.
- Nevertheless, if I know I can encounter with rank-deficient matrices, I would take in consideration SVD before QR with column pivoting.

The only differences between both methods are seen in our vector of solutions x when the rank of our matrix A is not full. As we said, there's no unique solution(s), so each method can find different vectors. The norm of both methods (and the error norm), as seen in the output that produces our code, is the same till certain degree.

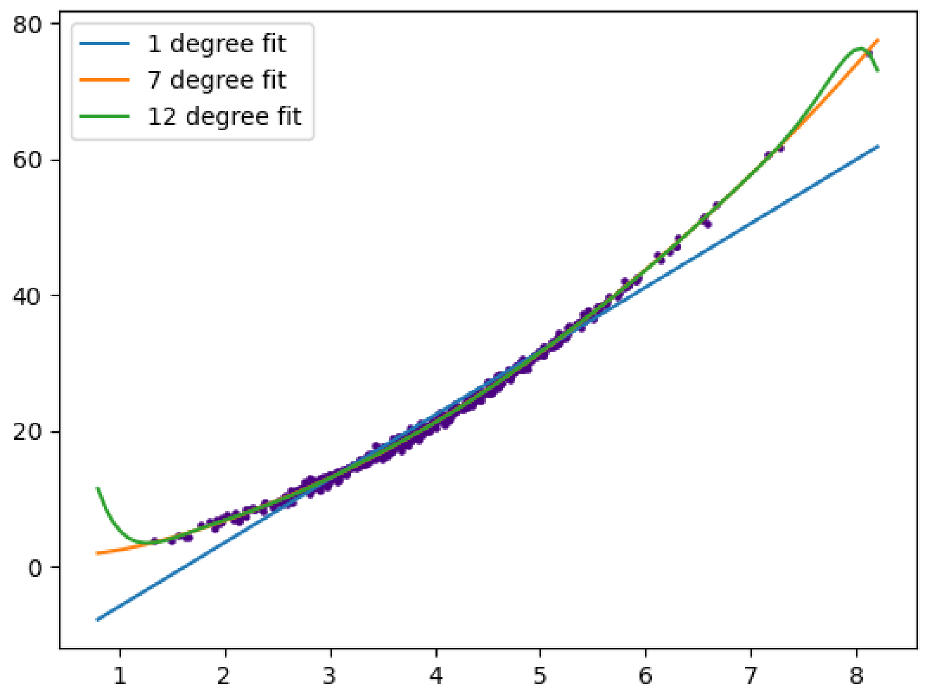

Figure 1.1: LS solution for the `datafile.csv` with 3 polynomials of degrees 1, 7 and 12.

2 2. Graphics compression

2.1 The SVD factorization has the property of giving the best low rank approximation matrix with respect to the Frobenius and/or the 2-norm to a given matrix. State properly the previous statement and write down the corresponding proofs for the Frobenius norm and the 2-norm.

At first I was going to describe two different proves for each method but I discovered an alternative proof which works for both Frobenius and L2-norm. It's a proof based on the Weyl's inequality for eigenvalues, or, more precisely, its alternative for singular values.

I need to prove that if $\text{rank}(B) = k$ then $\|A - B\|_2 \geq \|A - A_k\|_2$. Given two matrices X and Y with dimension $m \times n$ $(m \geq n)$ and the singular values are ordered in decreasing order, we have:

$$\sigma_{i+j-1}(X + Y) \leq \sigma_i(X) + \sigma_j(Y) \quad \text{for } 1 \leq i, j \leq n, \ i + j - 1 \leq n \tag{1}$$

As said, if B has rank $k, \sigma_{k+1}(B) = 0$. Setting $j = k + 1, Y := B$ and $X := A - B$ in (1) gives:

$$\sigma_{i+k}(A) \leq \sigma_i(A - B) \quad \text{for } 1 \leq i \leq n - k.$$

Now, for the L2, we just need to set $i = 1$. For the Frobenius norm, if $B = A_k$:

$$\|A - B\|_F^2 \geq \sum_{i=1}^{n-k} \sigma_i^2(A - B) \geq \sum_{i=k+1}^{n} \sigma_i^2(A)$$

I found out about this prove in this link. I opted for this demonstration instead of the one in the Wikipedia due that is simpler and I think the Wikipedia one is incomplete or that assumes some complex things without explaining them. I also do not have a strong mathematical background.

2.2 Use the previous results to obtain a lossy compressed graphic image from a .jpeg graphic file.

As said, we tried several different images, all type `.jpeg` and tried to create approximations of them using a lower rank matrix with SVD. All images were in color although our code checks the dimension of the given image and converts it to gray scale if needed. This is our set:

Name	**Width (px)**	**Height (px)**	**Size (KB)**
Digit	300	388	7
GeorgFrobenius	201	296	9
Euler	440	512	48
Lenna	630	630	122
Dog	6000	4000	7410

As we can see, we opted for an small image containing a digit, as suggested, and then 3 different images with relatively short dimensions. Afterwards, we added one more, `dog.jpeg`, which has been produced from a good camera and therefore has a big dimension and size. With that set we can see the performance of the SVD in various different cases.

If you run `GraphicsCompression.py` you get a folder named `Compressed` with all the reduced images with the percentage of the Frobenius norm captured. For each image this code generates 10 different compressed images. Each one of them has been produced with a different value of k that goes from 5 to 50, jumping 5 by 5. With this values of k we always get a percentage around 80-90%, a known *rule of thumb.*

Let me explain one interesting comparison of my results. I compare my `Dog.jpeg` image compression and my `Lenna.jpeg` compression. As we can see in the figure 2.1, with k = 5 we almost do not recognize Lenna. Nevertheless, with only k=20 we start seeing its face and with k=50 we would say that there's almost no difference with the original one. Its Frobenius norm captured percentages are 89.71, 96.59 and 98.57. This behavior repeats with the other 3 small images.

Now look at figure 2.2. Its behavior with k=5 and 20 is more or less the same with the `Lenna.jpeg` example, but with k=50 we can still notice that a compression has been performed, while in the `Lenna.jpeg` example or all the other 3 images with k=50 and above we almost can't notice the compression.

That's due to the image size: `Dog.jpeg` has a width of 6000 pixels and a height of 4000 pixels (a matrix of shape 6000x4000) while the other 4 images are a lot smaller. Its a logic behavior: the bigger the image the more noticeable will be the compression. Take also in consideration the execution time of the compression of `Dog.jpeg` vs. the others: 30 seconds for `Dog.jpeg` and 0.3 seconds for `Lenna.jpeg`!

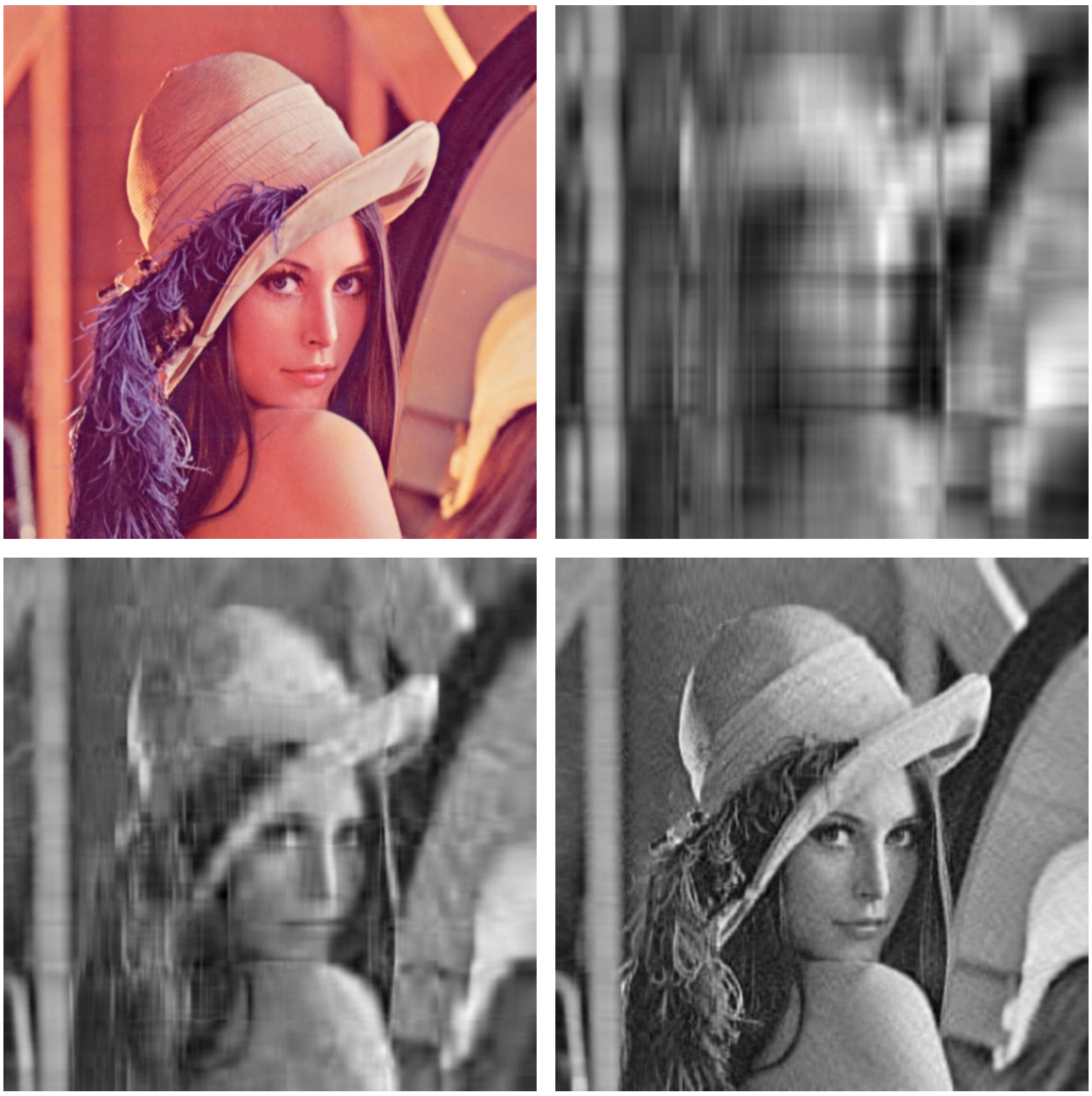

Figure 2.1: Original `Lenna.jpeg` and then with k = 5, 20 and 50

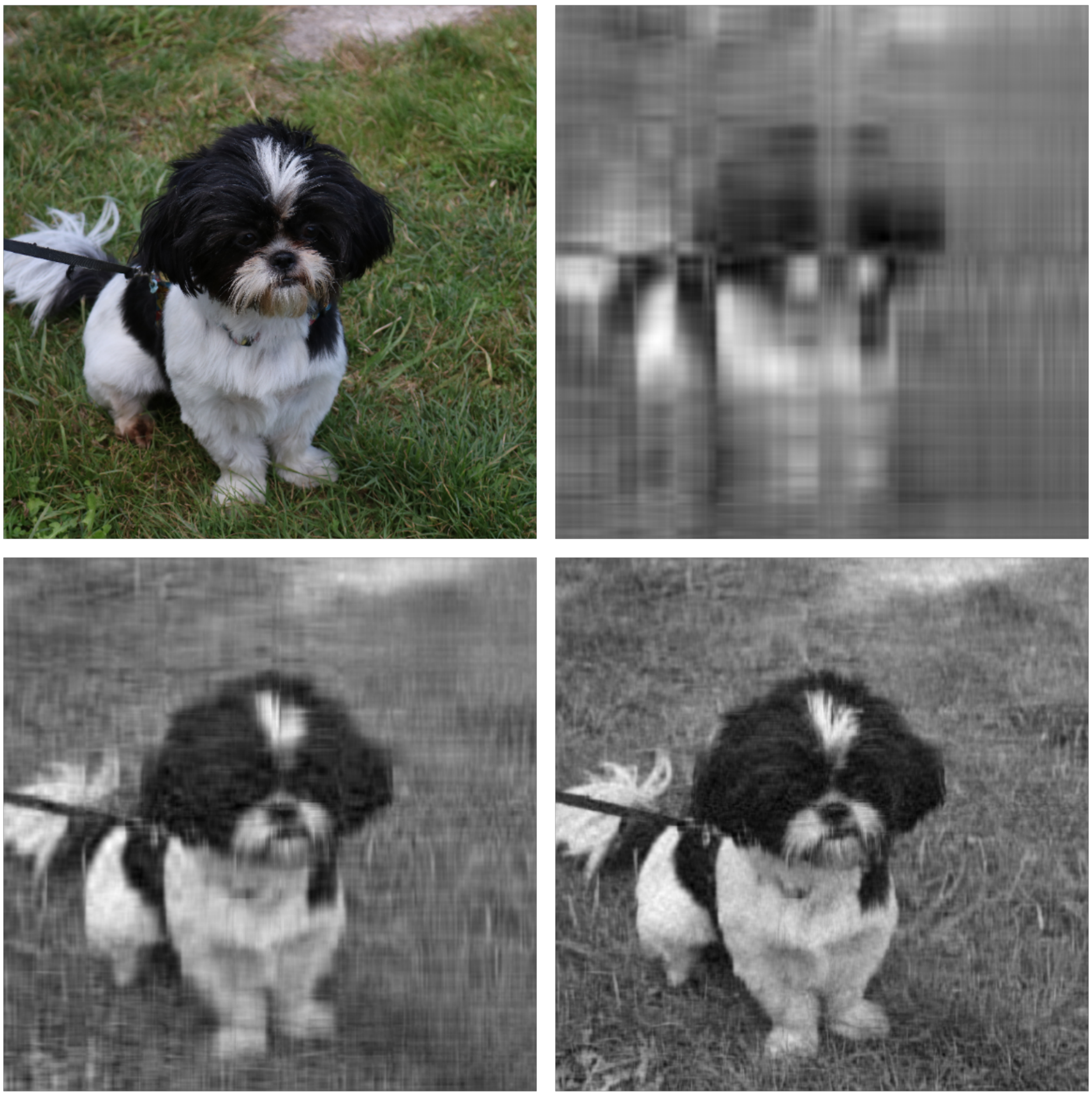

Figure 2.2: Original `Dog.jpeg` and then with k = 5, 20 and 50

3 Principal Component Analysis (PCA)

We designed a code, `PCA.py`, which reads boths datasets and performs PCA on each one of them. For the first dataset, `example.dat`, the user has the option to choose which type of matrix wants to use. As indicated, the two options are the correlation matrix or the covariance matrix. The user can switch between them writing `python PCA.py <name>` where *<name>* can only be `"covariance"` or `"correlation"`. If the user does not specify the matrix type, the covariance matrix is used by default.

For the `example.dat` dataset, the code outputs the following:

- Dataset in the PCA coordinates: the PCA matrix obtained with shape 16x4. The user can see the expression of the original dataset in the new PCA coordinates.
- The portion of the total variance accumulated in each of the principal components: an array of 4 elements, one for each component is print.
- Portion of the total variance accumulated computed by the PCA module of Sklearn.
- Standard deviation of each of the principal components: again, an array of 4 elements, one for each component is print.

For the `RCsGoff.csv` dataset, the code ignores the user matrix type input. As indicated, for this dataset we must use the covariance matrix. If not, we get an overflow event. The output of the code in this case is a `.csv` file, following the scheme indicated in the .PDF. The name of this file is `PCA_to_RCsGoff.csv` and it will be created in the same folder where the code is executed.

3.1 Results

example.dat

We can see that the portion of the total variance accumulated in each of the principal components is exactly the same as the one computed using the PCA of Scikit, even if we choose the correlation or the covariance matrix. After performing the PCA on this dataset thanks to `PCA.py` we can conclude this extra following facts:

- The Kaiser rule for selecting components in PCA should not be used in this dataset. Kaiser dictates that a component should not be retained unless it has an eigenvalue greater than or equal to one, and, in this case, all the eigenvalues of the 4 components are greater or equal than 1. Consequently, we would be keeping all the components and we would not be reducing the dataset. We can see this analysis in the Scree plot 3.1 where the "Y" axis has logarithmic scale due to the dimension of the eigenvalues.
- Instead, I would be using the 3/4 of the total variance rule, meaning that I would only keep the first two components. The first component has an accumulated variance of 66.97%, which is lower than 0.75 (a.k.a. 3/4). The second one has 82.86%

which is greater than 0.75 and therefore I have to stop selecting components. We can see this analysis in the figure 3.2.

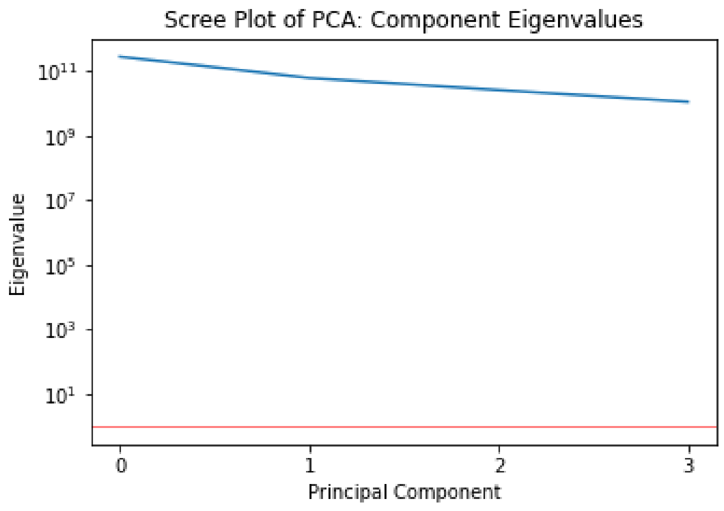

Figure 3.1: Scree plot of the dataset `example.dat`. All the eigenvalues are 1. The red line is the "1" but its zoomed out.

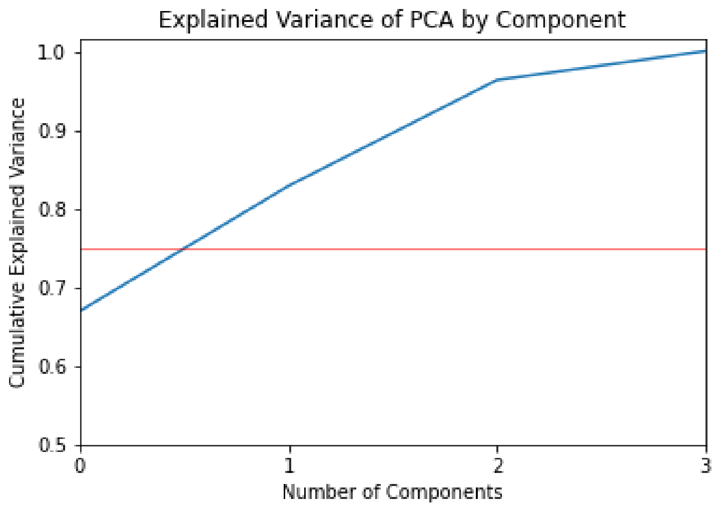

Figure 3.2: Accumulative variance of the 4 components on the dataset `example.dat`. Approximately, 66.97% of the variance is explained by the first component, and about 96.31% is explained by the first 3 components.

I also computed how informative has each variable present in the dataset for the principal component. To do so, I designed the function that gets the 4 components (I get all becasue they are only 4) and shows the magnitude of each component for each variable, thanks to the `.components_` method of Scikit-PCA (for example). Then, I compute the absolute value of each magnitude so we can compare each one on an scale of [0,1]. The output can be seen in the table 3.1.

In order to know what components are more informative, we look for the values near to one. We can see that the third variable, variable "2", is very informative of the first component, slightly less informative for the second component and not informative for the other 2 components. The opposite happens for the fourth variable: not informative at all at components 1 & 2 and very informative on components 3 and 4.

Table 3.1: Absolute value of the loadings of the dataset `example.dat` for each component

	Component 1	Component 2	Component 3	Component 4
0	0.595766	0.108547	0.605301	0.516615
1	0.378618	0.834263	0.267511	0.298482
2	0.706467	0.540168	0.317939	0.328691
3	0.051138	0.021017	0.678943	0.732106

RCsGoff.csv

After performing the PCA on this dataset thanks to `PCA.py` we can conclude the following facts:

- The Kaiser rule for selecting components in PCA should not be used in this dataset. Kaiser dictates that a component should not be retained unless it has an eigenvalue greater than or equal to one, and, in this case, all the eigenvalues of the 20 components except one are greater or equal than 1. Consequently, we would be keeping 19 components and we would not be reducing the dataset. We can see this analysis in the Scree plots 3.3 and 3.4, where the "Y" axis has logarithmic scale due to the dimension of the eigenvalues.
- Instead, I would be using the 3/4 of the total variance rule, meaning that I would only keep the first two components. The first component has an accumulated variance of 72.29%, which is lower than 0.75 (a.k.a. 3/4). The second one has 88,07% which is greater than 0.75 and therefore I have to stop selecting components. We can see this analysis in the figure 3.5.

Again, I represented the magnitude of each component for each variable present in the dataset in the table 3.2. We can see some weird things: the first component is not near 1 at all, so, in theory, its not very informative. But, following the criteria of the rule of the 3/4 of the total variance, I would pick the component... We can also see that the second component is the most representative, followed by the third one.

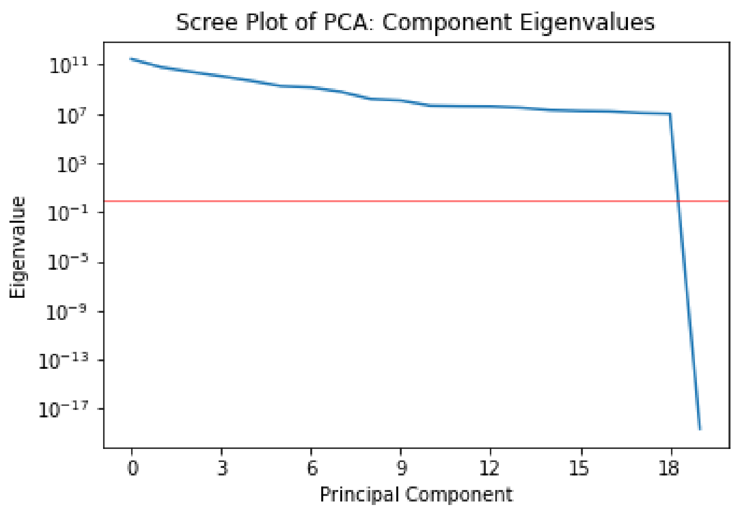

Figure 3.3: Scree plot of the dataset `RCsGoff.csv`. All the eigenvalues except one is $\geq$ 1. The red line is the "1" but its zoomed out.

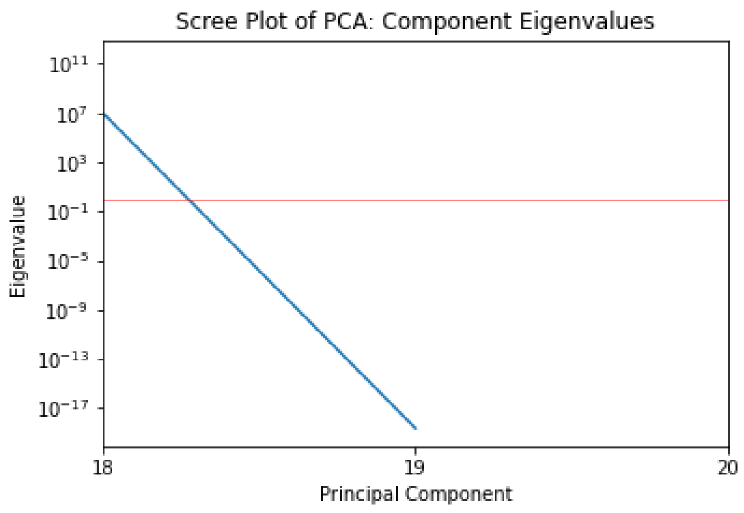

Figure 3.4: Scree plot of the 20th component of `RCsGoff.csv`, with value 2.39e-19.

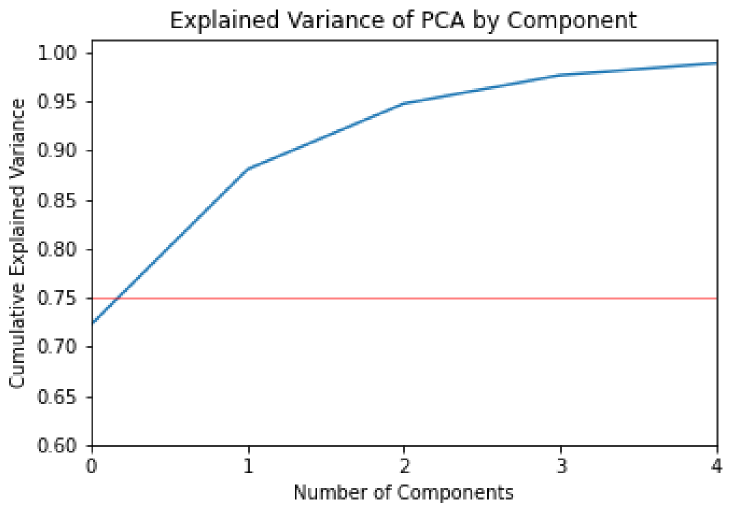

Figure 3.5: Accumulative variance of the 3 first components on the dataset `RCsGoff.csv`. Approximately, 72.29% of the variance is explained by the first component, and about 88% is explained by the first 2 components.

Table 3.2: Absolute value of the loadings of the dataset `RCsGoff.csv` for the first four components

	Component 1	Component 2	Component 3	Component 4
day0_rep1	5.594512e-07	0.001661	0.000017	5.565657e-07
day0_rep2	1.870004e-07	0.000680	0.000003	2.362530e-06
day0_rep3	9.740057e-07	0.000113	0.000015	4.457856e-06
day1_rep1	4.228256e-07	0.000817	0.000004	3.116306e-06
day1_rep2	2.847092e-06	0.001518	0.000028	9.719115e-07
day1_rep3	1.643411e-06	0.001433	0.000010	2.592712e-05
day2_rep1	4.643632e-09	0.000275	0.000050	3.124582e-06
day2_rep2	1.176034e-06	0.002175	0.000016	4.562673e-06
day2_rep3	1.261381e-05	0.001407	0.000150	8.339558e-06
day4_rep1	1.101093e-05	0.002504	0.000034	3.675542e-05
day4_rep2	2.332067e-05	0.003262	0.000333	3.831106e-05
day4_rep3	4.702655e-05	0.003502	0.000027	2.047400e-05
day5_rep1	1.910378e-05	0.006554	0.000044	1.465132e-05
day5_rep2	5.571387e-05	0.003805	0.000320	6.342373e-05
day11_rep1	3.520349e-06	0.003778	0.000228	5.453415e-06
day11_rep2	1.093122e-05	0.003532	0.000269	3.770131e-06
day11_rep3	9.953847e-06	0.007693	0.000334	6.829269e-05
day18_rep1	1.818369e-06	0.000809	0.000145	8.126254e-05
day18_rep2	2.575097e-05	0.003501	0.000011	1.817621e-04
day18_rep3	9.581217e-01	0.117001	0.000350	9.848273e-02

4 Difficulties

We got some trouble with the LS problem using QR. We did not get the same norm as the one using SVD and spent some time finding out what was the problem. At the end was only a matrix transposition that we forgot. Also, the mathematical part on this exercise was very hard for me.

For the second exercise, Graphics Compression, the hardest part was creating the reduced image. At the end, we just needed the `np.matrix` function at the start of each matrix outputed by the SVD. In the second place, we encountered a *warning* when saving the reduced image saying that "the conversion of float to int was lossy". We removed this warning by converting the image first to int using `.astype(np.uint8)`. Finally, I did find hard the theoretical part of this exercise where we had to state that SVD always gives the best low rank matrix approximation.

For the last exercise, the hardest part was finding out what matrix did we need to transpose for the `RCsGoff.csv` dataset. If you do not transpose the input, you are computing the PCA's as if the observations were the variables. I also found hard to understand the explained variance of each component when performing PCA. The concept was a bit abstract for me so a graphical representation like the one in figure 3.5 finally did the job.