Numerical Linear Algebra

master in fundamental principles of data science

Project 2

SVD Applications

Author
Vladislav Nikolov Vasilev

Faculty of Mathematics and Computer Science

Academic year 2021-2022

Contents

1 Introduction

The goal of this project is to discuss three common applications of the Singular Value Decomposition (SVD). First, let's briefly review what the SVD is.

Given a rectangular matrix $A \in \mathbb{R}^{m \times n}$ with $m \geq n$, we can express it as

$$A = U\Sigma V^T$$

where $U \in \mathbb{R}^{m \times m}$ and $V \in \mathbb{R}^{n \times n}$ are two orthogonal basis and $\Sigma \in \mathbb{R}^{m \times n}$ is a matrix that can be divided in the diagonal block $\Sigma\,[1:n, 1:n]$ with the singular values the singular values σ_i in the diagonal and the zero block $\Sigma\,[(m-n):m, 1:n]$. The singular values are ordered such that $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_n \geq 0$. Since U and V are orthogonal, we have that $U^{-1} = U^T$ and $V^{-1} = V^T$.

There are some cases in which we can also compute a reduced version of the SVD, which is faster and reduces the amount of memory needed to store the matrices. This can be particularly useful in scenarios where the matrix A is rank deficient.

Let $A \in \mathbb{R}^{m \times n}$ be a rectangular matrix with $rank(A) = r$, where $r < n$. For this case, the reduced SVD can be computed as

$$A = U_r \Sigma_r V_r^T \tag{1}$$

where $U_r \in \mathbb{R}^{m \times r}$ and $V_r^T \in \mathbb{R}^{r \times r}$ are the orthogonal basis and $\Sigma \in \mathbb{R}^{r \times r}$ is the diagonal matrix containing the nonzero singular values.

There are many applications of the SVD, but in this project we are going to focus on three of them: solving the Least Squares Problem, image compression and Principal Component Analysis.

2 Least Squares Problem

The first application that we are going to address is the Least Squares Problem (LSP). Recall that in this problem we have a matrix $A \in \mathbb{R}^{m \times n}$ with $m \geq n$ and a vector $b \in \mathbb{R}^m$. Our goal is to find a vector $x \in \mathbb{R}^n$ such that Ax is as close to b as possible. This can be expressed as a minimization problem, in which we have

$$\min ||Ax - b||_2 \tag{2}$$

There are many ways in which we can solve this problem: using iterative methods, normal equations, QR factorization or SVD, among many others. In this case, we are going to focus on the SVD method, which is also very appropriate for the rank deficient case.

Consider the reduced version of the SVD seen in expression (1). We can use it to compute the *Moore-Penrose inverse* (also called *pseudo-inverse*) of our matrix A as

$$A^+ = V_r \Sigma_r^{-1} U_r^T$$

with $A^+ \in \mathbb{R}^{n \times m}$. Thus, we can express the minimization problem (2) as follows:

$$x = A^+ b$$

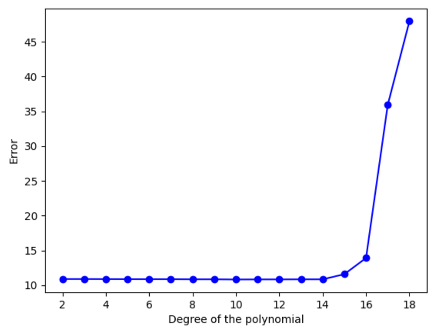
The solution obtained with this method is well-conditioned provided that the smallest nonzero singular value of A is not too small.

To solve the LSP problem, we have created a script called `lsp.py` which contains a function called `solve_lsp_svd`. This function computes the reduced SVD of an input matrix A, computes the pseudo-inverse and multiplies it by the b vector to get the value of x. For the computation of the SVD we have used the function `numpy.linalg.svd`. To get the rank of the matrix we have chosen the singular values greater than a certain tolerance, which can be fixed or is computed as `Numpy` does when computing the rank of a matrix (see implementation for further details and reference).

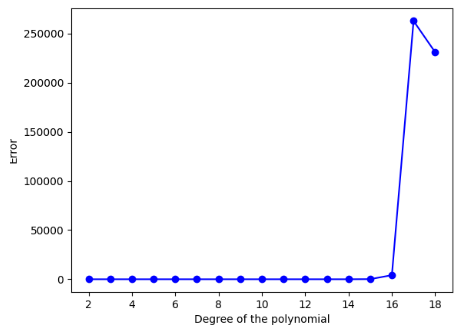
Let's now apply this method to some problems and see how it performs compared to another method: the QR factorization. The before mentioned script also contains a function called `solve_lsp_qr_full_rank`, which is used for the full rank problem, and another one called `solve_lsp_qr_rank_deficient`, which uses the QR factorization with pivoting to solve the rank deficient case.

2.1 Polynomial fitting

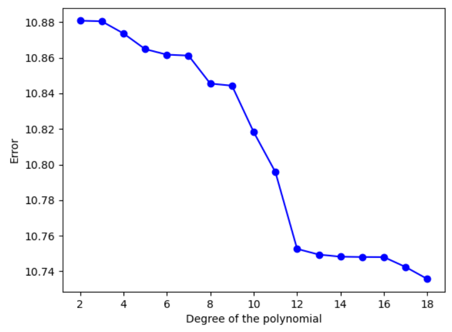
In this problem we are given a set of points x and their values b. We are going to try to approximate the values using polynomials of different degrees. To do that, we are going to use the SVD approach both with a fixed tolerance of 10^{-10} and an adaptive one based on `Numpy`'s tolerance for the computation of the rank of a matrix and the QR factorization for the full rank problem. Once we have the projection vector x, we are going to compute the error of the approximation using the 2-norm to see how good the it is and to compare the different methods.



(a) SVD with adaptive tolerance.



(b) SVD with fixed tolerance.



(c) QR factorization without pivoting.

Figure 1: Comparison of the approximation's error for different methods as the degree of the polynomial increases.

As we can see in figure 1, all of the methods yield a solution which has approximately the same error up until polynomials of degree 13. After that point, the matrix A starts to become close to rank deficient. This has a huge impact on the SVD with fixed tolerance because it doesn't take into account the magnitude of the singular values when discarding the less significant ones. Therefore, the problem is treated as a full rank one, and the problem becomes ill-conditioned because the smallest singular values are too small.

In the case of the SVD with adaptive tolerance, we can see that the error starts increasing even when adapting the tolerance to the magnitude of the singular values so that the problem is not treated as a full rank one. However, the error is not as large as in the previous case.

Finally, we can see that the QR factorization without pivoting outperforms the rest of the methods for large polynomials. It yields results with the lowest errors, and even the error seems to decrease as the degree of the polynomial increases.

2.2 The rank deficient LSP

For this case, we are going to study a problem in which we have a matrix $A \in \mathbb{R}^{15\times 11}$. However, we have that $rank(A) = 10$ since the first two columns are exactly the same. Therefore, since $r < n$, the matrix A is going to be rank deficient.

For this problem we have used two methods: the reduced SVD with adaptive tolerance and the QR factorization with pivoting since we cannot apply QR factorization directly to the matrix A.

Once again, we have solved the system and we have computed the error of the approximation using the 2-norm. In this case, both methods have produced approximately the same error, which is around 1.1496.

As we can see, both methods allow us to solve the problem with small error. However, the method based on the reduced SVD is much simpler and straightforward than the one using the QR factorization with pivoting. Therefore, for this problem we could have used the simplest one and still get a good result.

3 Image compression

Another problem where SVD can be applied is image compression, even though there are much better techniques out there. In order to compress images, we can use a **lower rank approximation** of them.

Fortunately, the SVD factorization has the property of giving the best low rank approximation matrix with respect to the Frobenius norm and the 2-norm. For a given value of $k = 1, \dots, r$, the matrix

$$A_k = \sum_{i=1}^{k} \sigma_i u_i v_i^T = U_k \Sigma_k V_k^T$$

is the best rank k approximation of A with respect to both the Frobenius norm and the 2-norm. For any of these two norms, we have that

$$\|A - A_k\| = \|\Sigma - \Sigma_k\| = \left\| \begin{bmatrix} 0 & & & & & \\ & \ddots & & & & \\ & & 0 & & & \\ & & & \sigma_{k+1} & & \\ & & & & \ddots & \\ & & & & & \sigma_r \\ & & & 0 & & \end{bmatrix} \right\| \tag{3}$$

The Frobenius norm for a given matrix A can be expressed in terms of its SVD factorization as follows:

$$\|A\|_F = \left\|U^T AV\right\|_F = \|\Sigma\|_F = \sqrt{\sum_{i=1}^{r} \sigma_i^2}$$

Following the results obtained in (3), we get that the Frobenius norm can be computed as

$$\|A - A_k\|_F = \sqrt{\sum_{i=k+1}^{r} \sigma_i^2}$$

The 2-norm can also be expressed in terms of the SVD:

$$\|A\|_2 = \left\|U^T AV\right\|_2 = \|\Sigma\|_2 = \sigma_1$$

Hence, using again the results from expression (3), we get the 2-norm can be computed as

$$\|A - A_k\|_2 = \sigma_{k+1}$$

We have created a script called `image_compression.py` which implements the compression of both black and white and color images using a low rank approximation of the image. It automatically detects whether the input image is black and white or a color one. For the black and white case, it is as easy as computing the SVD of the input image, selecting the first k singular values and then multiplying matrices accordingly. For the color case, the low rank approximation is computed for each channel. After that, the approximations are stacked again so that the output image has 3 channels. The functions that perform the previous operations are called `bw_compression` and `color_compression`. The script can only compress one image at a time. In order to run the script, run the following in the command line:

```
$ python3 image_compression.py -i IMAGE_PATH -r RANK
```

To see how good the low rank approximation is, we have computed the relative error of the approximation using the Frobenius norm as follows:

$$\frac{\|A - A_k\|_F}{\|A\|_F} = \frac{\sqrt{\sum_{i=k+1}^{r} \sigma_i^2}}{\sqrt{\sum_{i=1}^{r} \sigma_i^2}}$$

To compute the error for color images we have computed it for each channel and then we have averaged the individual results.

To test how the compression works, we have used four images. Three of them are black and white and the other one is a color one. The first one of them is a picture of two deer in a forest, which has a size of 800×491 pixels. The second one is a picture of New York, with a size of 1121×700 pixels. The third one is a picture of a woman which has a size of 1000×780 pixels. Lastly, we have a color picture of a dog, which has a size of 480×360 pixels. For each one of them, we are going to try different ranks. In this case, we will use 1, 5, 10, 25 and 100.

The results of the low rank approximations can be seen down below in figures 3, 4, 5 and 6. In general, and as it is expected, we can see that for $k = 1$ we cannot tell what any of the images is. As the value of k increases, we can appreciate more and more details of the original image.

Generally speaking, when the rank of the approximation is around $k = 25$, we start to see that the approximation resembles the original image, although depending on the case it is more or less detailed. For instance, in figures 3, 5 and 6 we can see that the approximations resemble the original image quite a lot. On the other hand, the image of New York (figure 4) is still quite blurry and a big part of the details are lost. The image of the woman is a particular case, because starting from $k = 10$ we can clearly tell what its content is. This might be due to the fact that it doesn't have as much details as the other ones.

For $k = 100$, we can see that the image of the deer, the woman and the dog have almost the same frequencies as the original one. The photo of New York, however, still lacks some details, although it is much better than before. However, this is not surprising at all considering that this is the image that has the most details.

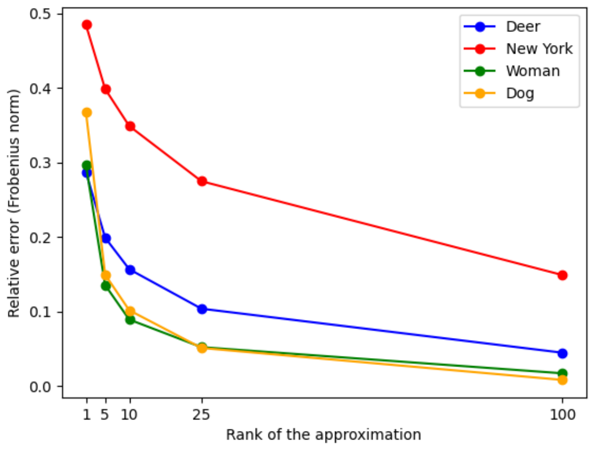
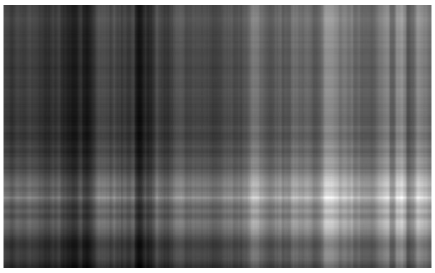
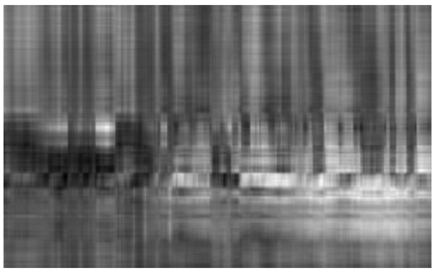


Figure 2: Evolution of the relative error of the low rank approximation as the rank increases for all of the images.

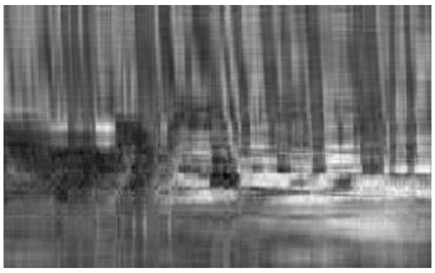
In figure 2 we can appreciate how the relative error changes as the rank of the approximations increase. As it is expected, the higher the rank of the approximation, the lower the relative error. The image of the dog and the woman have very similar errors as the rank increases. They are followed by the image of the deer in the forest, which has a low relative error. As we could have imagined, the image of New York is the one that has the highest relative error. Nonetheless, as we saw before, it actually manages to decrease quite a lot as the rank of the approximation increases up to $k = 100$.



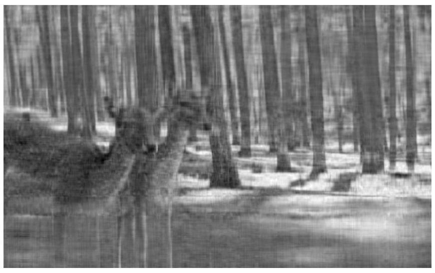
(a) $k = 1$



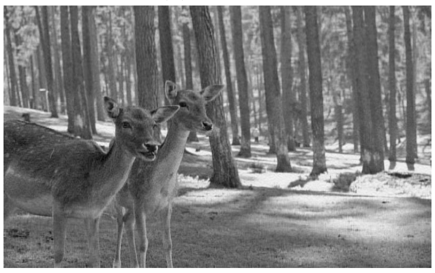
(b) $k = 5$



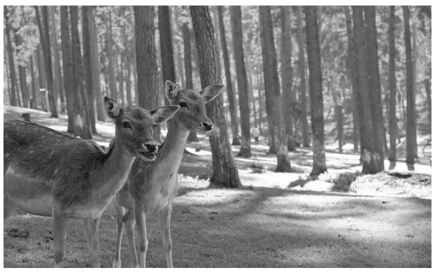
(c) $k = 10$



(d) $k = 25$

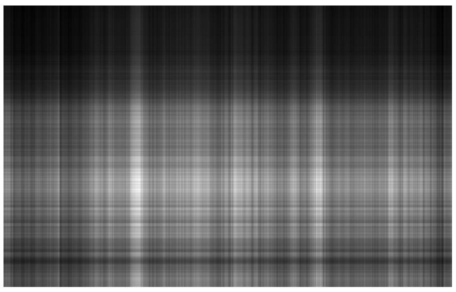


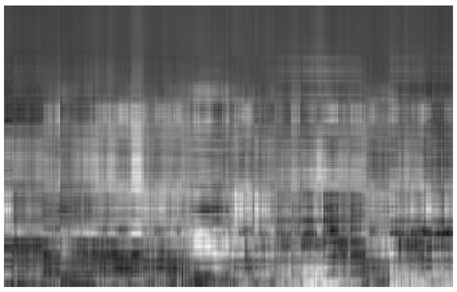
(e) $k = 100$

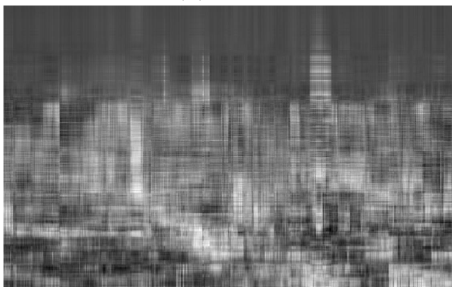


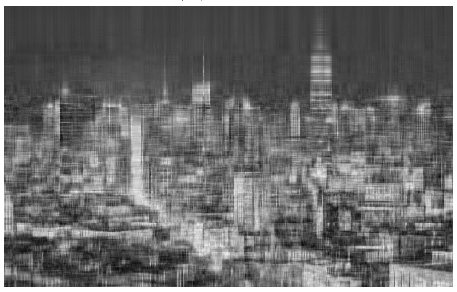
(f) Original

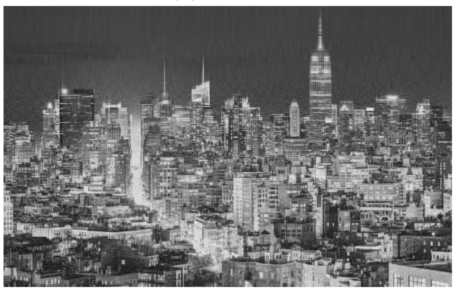
Figure 3: Low rank approximation of the image containing deer.


(a) $k = 1$


(b) $k = 5$


(c) $k = 10$


(d) $k = 25$


(e) $k = 100$

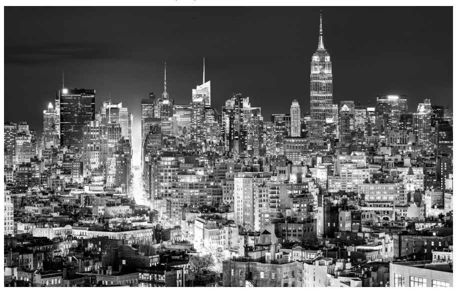
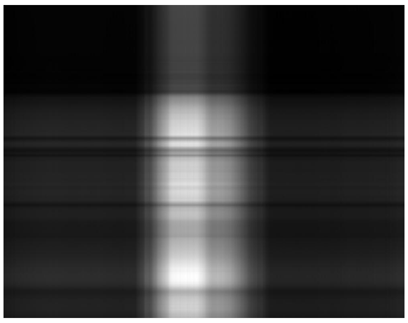
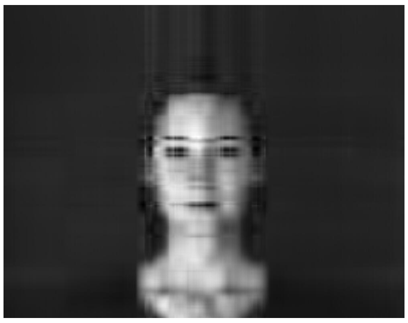
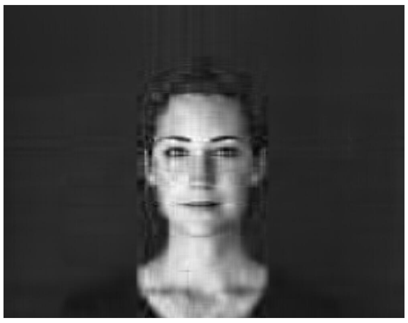
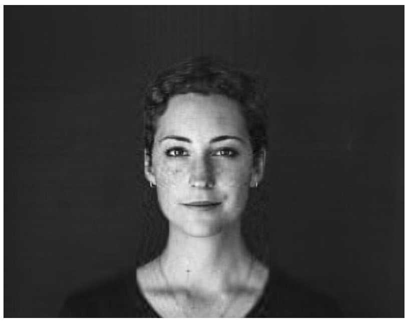

(f) Original

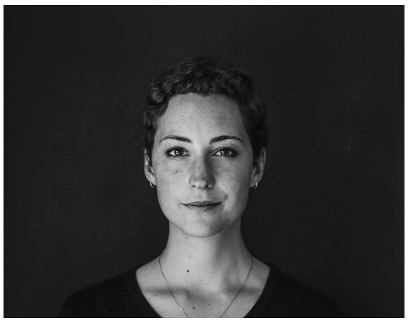
Figure 4: Low rank approximation of the image of New York.


(a) $k = 1$


(b) $k = 5$


(c) $k = 10$


(d) $k = 25$


(e) $k = 100$

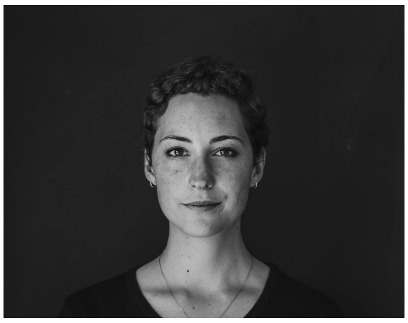
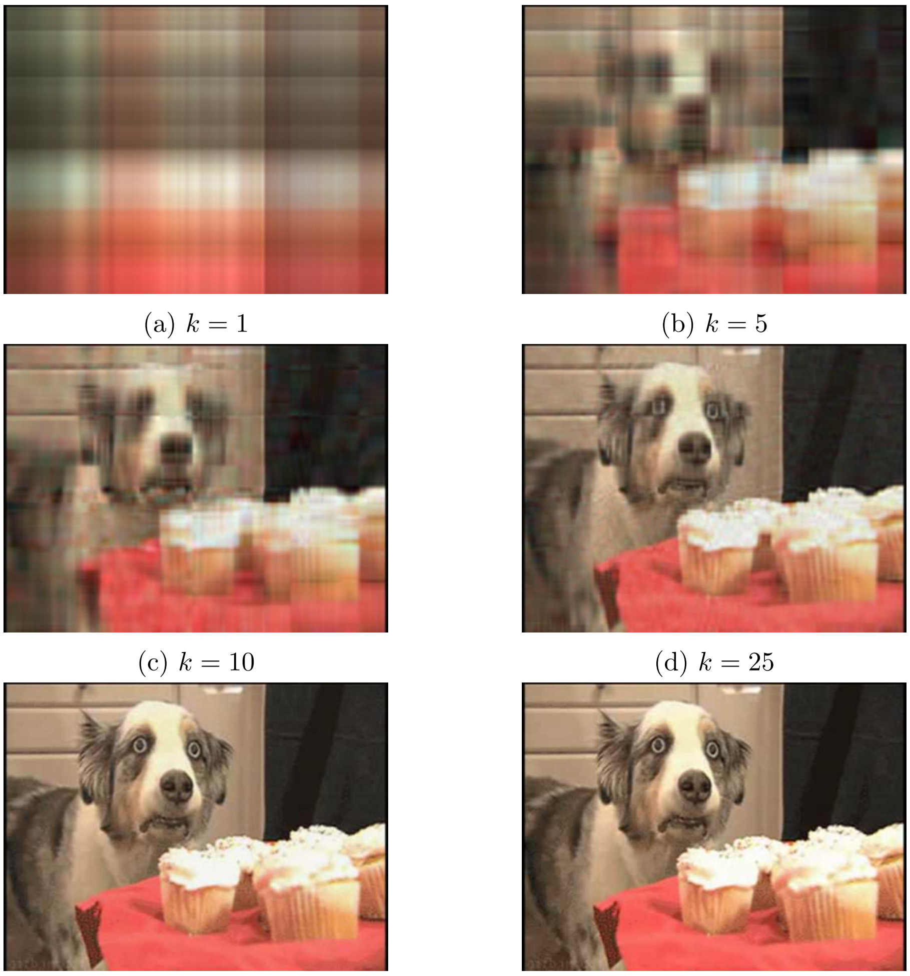

(f) Original

Figure 5: Low rank approximation of the picture of a woman.



(a) $k = 1$ (b) $k = 5$

(c) $k = 10$ (d) $k = 25$

(e) $k = 100$ (f) Original

Figure 6: Low rank approximation of the image of a dog that has seen too many things (and some cupcakes).

4 Principal Component Analysis

The last application of the SVD that we are going to see is the Principal Component Analysis (PCA). Very simply put, PCA is a technique used to detect the principal components of a data set in order to reduce it into fewer dimensions that still contain most of the information. These principal components form an orthonormal basis and are sorted according to their variance. Using them, we can project our data into a space where the variables are uncorrelated. Usually, one keeps the most relevant principal components, i.e. the ones that contain most of the variance of the original dataset, whereas the rest of them can be considered as noise.

Given a matrix $X \in \mathbb{R}^{m \times n}$, where m is the number of variables and n the observations that make up the dataset, one can compute the covariance matrix as $C_X = \frac{1}{n-1} X X^T$. This matrix contains the relationship between each pair of variables (the covariance), and its diagonal contains the variance of each variable, which is in fact the covariance of a given with itself. Then, we could compute the eigenvalues and eigenvectors of this matrix. However, the computation of the covariance matrix is highly numerically unstable.

Instead of doing that, we can consider the matrix $Y = \frac{1}{\sqrt{n-1}} X^T$. We can clearly see that $Y^T Y = C_X$. With this, we can compute the reduced SVD of the matrix Y:

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

where $U \in \mathbb{R}^{n \times r}$, $\Sigma \in \mathbb{R}^{r \times r}$ and $V \in \mathbb{R}^{r \times m}$, being $r = rank(Y)$.

From here, we can see that $\sigma_i^2 = \lambda_i$, and the values are sorted in descending order, hence giving us the order of the principal components. If $\text{var}_T = \sum_i \lambda_i$ accounts for the total variance, then $\sigma_i^2 / \text{var}_T$ accounts for the portion of the total variance in each principal component. Also, the matrix V contains the eigenvectors of $Y^T Y = C_X$ as columns, and the coordinates of the dataset in this new space are given by $V^T X$.

That being said, let us now put this into practice. We have created a script called `pca.py` which contains the function `PCA`. This function computes the PCA of a given input matrix using the covariance or the correlation matrix depending on the boolean parameter `covariance`. First, it adjusts the matrix so that the variables are in the rows and the observations are represented as columns. Then, according to the selected matrix, it transforms the data accordingly. After that, it computes the Y matrix and applies the reduced SVD to it. To compute the rank of the matrix, we have used a tolerance based on the the machine ϵ for floating point numbers. Finally, we project the data into the new space and compute some additional information (proportion of the variance explained by each principal component, cumulative percentage of the variance explained and standard deviation).

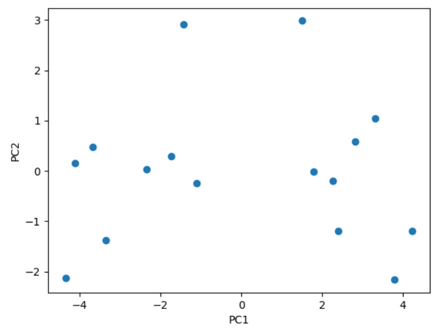
Also, we have to find out how many of the principal components we need to keep to explain the data. There are several techniques that we can apply, but we are going to focus on two of them:

- **Kaiser rule**: Select the principal components whose eigenvalue is greater than 1. This can be done, for instance, using a Scree plot.

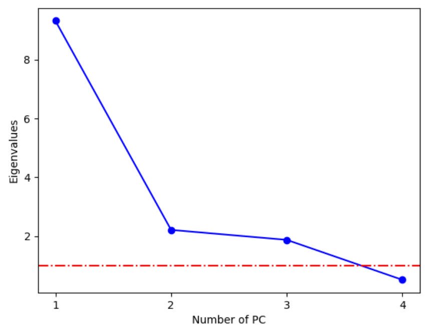
- **The 3/4 of the total variance rule**: Select the principal components that explain 3/4 of the total variance. This can be done by checking the cumulative values of the explained variance.

4.1 Example problem

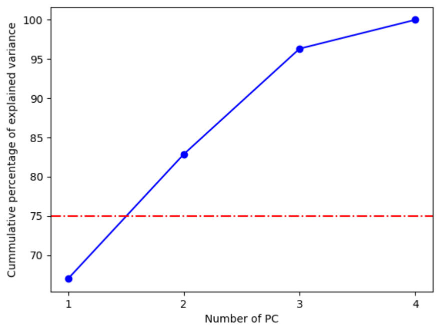
First, we are going to apply PCA to an example problem. Here, we have 4 variables and 16 observations. We are going to apply PCA on both the covariance and the correlation matrix and see how each one of them performs.



(a) Representation of the data using the two most important principal components.



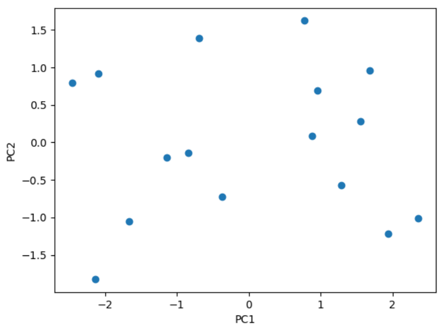
(b) Kaiser rule.



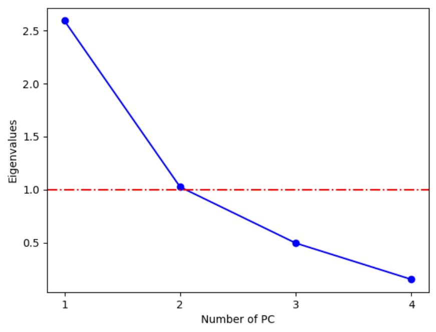
(c) 3/4 of the total variance rule.

Figure 7: Results for the covariance matrix.

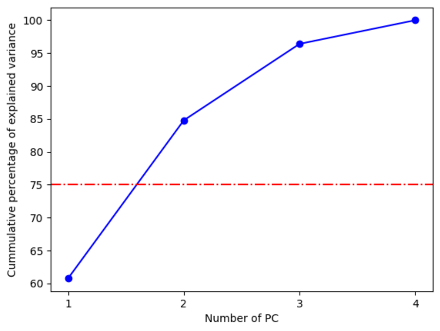
In figure 7 we can see the results obtained when using the covariance matrix. We can see the data represented using the two most important principal components. In this case, according to the Kaiser's rule we should use the first three principal components, whereas the 3/4 of the total variance rule tells us that we should only consider the two first principal components. In this case, it makes more sense to follow the 3/4 rule, because using only two of them we can explain approximately 85% of the total variance. Thus, we are not losing a log of information in the process, and we are actually using a smaller representation of the data.



(a) Representation of the data using the two most important principal components.



(b) Kaiser rule.



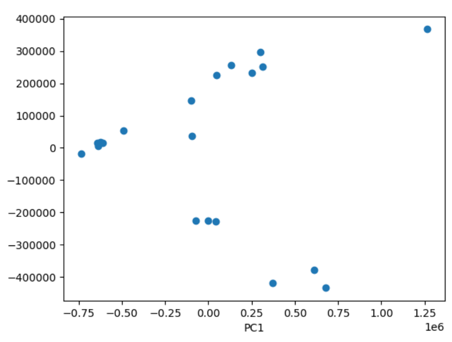
(c) 3/4 of the total variance rule.

Figure 8: Results for the correlation matrix.

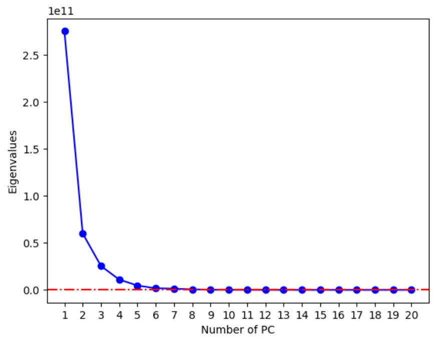
In the figure above we can observe the results that we have obtained when using the correlation matrix. In this case, the representation of the data using only the two most important principal components changes a little bit from the previous case. Here, both the Kaiser rule and the 3/4 of the total variance rule suggest that we should use the first two principal components to represent the data. Therefore, in this case we can choose any of them. However, we have a stronger preference for the 3/4 rule because it allows us to explain a big part of the variance of the data keeping in general less principal components than when using Kaiser rule (see the previous case).

4.2 Genes problem

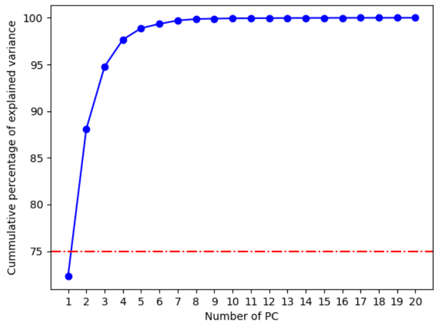
Let's apply now PCA to a bigger problem. We have a dataset with 58581 variables and 20 observations. In this case, we are only going to use the covariance matrix, because when the correlation matrix is used, SVD is not able to converge to a solution.



(a) Representation of the data using the two most important principal components.



(b) Kaiser rule.



(c) 3/4 of the total variance rule.

Figure 9: Results for the genes problem using the covariance matrix.

In figure 9 we can observe how the data is represented using only the first two principal components. Also, we can see how many principal components we have to chose according to the two rules. According to the Kaiser rule, we should choose the first 6 principal components, whereas if we use the 3/4 rule, we would only need to select the first 2 principal components. These two components already attain approximately 90% of the total variance, so it doesn't make sense to use more, because we are not losing a significant amount of information. In this case, we can see that the 3/4 rule allows us to represent a good amount of information in less dimensions than the Kaiser rule, which makes it more suitable for this problem.