University Of Ibn-Khaldoun Tiaret

Report of Practical Work

Principal Component Analysis Algorithm

Option: Master 1 Year SE

Course: Algorithm Complexity

Returned On 12/12/2018

Presented By:

Mr. Younes Charfaoui

Mr. Bourbai Ismail

Subject Responsible:

Mr. Chenine Abdelkader

Year of 2018-2019

Chapter I

Problem: Extracting Principal Components for a given dataset.

The Problem:

In machine learning the problem is that the data is in a n dimension and this not a problem for the mathematics, it's a problem for human that can't see this dimension, and also a problem for the machines, since we are in n dimension means that more computation are going to happen, the Principal Component Analysis (PCA) comes here to play the role of reducing the dimensionality of the data so we can visualize it in 2D or 3D, and the machine going to do less computation for finding the best machine learning model.

Principal component analysis is a dimensionality reduction technique; it lets you distill multi-dimensional data down to fewer dimensions, selecting new dimensions that preserve variance in the data as best it can.

We Made a Github [1] Repository contains all our work, you can visit it from here.

The way the algorithm work is like following:

- Standardize the data (By default it's the responsibility of the machine learning engineer).
- Obtain the Eigenvectors and Eigenvalues from the covariance matrix.
- Sort eigenvalues in descending order and choose the k eigenvectors that correspond to the k largest eigenvalues where k the number of dimensions of the new feature subspace is $(k \leq d)$.
- Construct the projection matrix W from the selected k eigenvectors.
- Transform the original dataset X via W to obtain a k-dimensional feature subspace Y.

Theory behind PCA:

Standardizing the Data: Most of the times, our dataset will contain features highly varying in magnitudes, units and range. But since, most of the machine learning algorithms use Euclidian distance between two data points in their computations, this is a problem. If left alone, these algorithms only take in the magnitude of features neglecting the units. The results would vary greatly between different units, 5kg and 5000gms. The features with high magnitudes will weigh in a lot more in the distance calculations than features with low magnitudes.

[1] https://github.com

Whether to standardize the data prior to a PCA on the covariance matrix depends on the measurement scales of the original features. Since PCA yields a feature subspace that maximizes the variance along the axes, it makes sense to standardize the data, especially, if it was measured on different scales.

Computing Eigenvectors and Eigenvalues: The eigenvectors and eigenvalues of a covariance matrix represent the "core" of a PCA: The eigenvectors (principal components) determine the directions of the new feature space, and the eigenvalues determine their magnitude. In other words, the eigenvalues explain the variance of the data along the new feature axes.

Covariance Matrix can be calculated as following:

$$\frac{1}{n-1}\sum_{i=1}^{n}(X-\bar{x})\,(Y-\bar{y})$$

The Eigen value and Eigen vector of a matrix can be calculated as following:

$$(A-\ \lambda I)X=0$$

Where λ is the Eigenvalue and X is The Eigenvector and the preceding equation means that:

$$det(A-\lambda I)\ =\ 0$$

Sort eigenvalues: The typical goal of a PCA is to reduce the dimensionality of the original feature space by projecting it onto a smaller subspace, where the eigenvectors will form the axes. In order to decide which eigenvector(s) can dropped without losing too much information for the construction of lower-dimensional subspace, we need to inspect the corresponding eigenvalues: The eigenvectors with the lowest eigenvalues bear the least information about the distribution of the data; those are the ones can be dropped. In order to do so, the common approach is to rank the eigenvalues from highest to lowest in order choose the top k eigenvectors.

Construct the projection matrix: It's about time to get to the really interesting part: The construction of the projection matrix that will be used to transform the dataset onto the new feature subspace. Although, the name "projection matrix" has a nice ring to it, it is basically just a matrix of our concatenated top k eigenvectors.

Transform the original dataset: In this last step we will use the $d*k$-dimensional projection matrix W to transform our samples onto the new subspace via the equation $Y=X\times W$, where Y is matrix of our transformed samples.

Chapter II

Algorithm Description

Specification:

- Inputs:
  - **Number components**: An Integer Number of Components or dimensions the user want to perceive in the new data
  - **Data:** The Data Matrix of Numbers the user want to reduce its dimensionality.

- Output:
  - The Data Matrix transformed to the new dimensionality.

Program:

```
1.  # The Main Algorithm Of PCA
2.
3.  import numpy as np
4.
5.  def pca(number_compontents, data):
6.      if not 0 <= number_compontents <= data.shape[1]:
7.          raise ValueError('The number of features are less than the number of components')
8.
9.      # calculate the covariance matrix
10.     cov_matrix = np.cov(data.T)
11.
12.     # calculate the eigen things
13.     eig_vals , eig_vecs = np.linalg.eigh(cov_matrix)
14.
15.     # codes are the same
16.     eig_pairs = [(np.abs(eig_vals[i]) , eig_vecs[:,i]) for i in range(len(eig_vals))]
17.
18.     # Sorting All of Them
19.     eig_pairs.sort(key = lambda x : x[0] , reverse= True)
20.
21.     # Getting the selected vector in a form of matrix
22.     final = [eig_pairs[i][1].reshape(data.shape[1],1) for i in range(number_compontents)]
23.
24.     # Creating the Projection Matrix, multiplying by -identity is an addons
25.     projection_matrix = np.hstack((final))
26.
27.     # transforming the data
28.     Y = data.dot(projection_matrix)
```

```
29.
30.    return Y
```

Tools Used:

During this practical work we have used:

- Python as a programming language to implements PCA Algorithm.
- Numpy library for doing math calculations like matrix multiplication, covariance matrix calculations and so one, this library it's highly optimized for math operations.
- Matplotlib library for visualizing the data and graphs (plotting in general).
- Sklearn library, from which we used Standard Scaler for scaling our data in a proper range, and The PCA Algorithm to compare our result to its.
- Pandas library which is in the background of this algorithm, it help us load the data from different source (CSV, TSV, Excel) and convert it into numpy matrix we can us in our algorithm.

Comparing to Already Implemented PCA:

We had compared our pretty algorithm to the one implemented in the Scikit-learn[2] Library, the result was the same in term of meaning, we noticed just that the exact mirror image of the plot from out step by step approach. This is due to the fact that the signs of the eigenvectors can be either positive or negative, since the eigenvectors are scaled to the unit length 1, both we can simply multiply the transformed data by $\times (-1)$ to revert the mirror image but it's not so important because the purpose is to reduce the dimensionality and by that we could find a boundary to separate the data for classification task by some machine learning algorithm like Logistic regression or doing Support Vector machine or Even Naïve Bayes Algorithm. The Other side of comparison is time and this time the one implemented in the Library beats our algorithm like the figure describe bellow, This Library Use the Singular Value Decomposition instead of calculating the covariance matrix and doing the Eigen Things and for we things it is faster than ours.

[2] https://scikit-learn.org/

We Did Some Dimensionality reduction to 2D and 3D for some dataset and make the comparison between Our Pca and Their Pca (Sklearn), here is some screen shot and the data corresponding:

MNIST Dataset: This is A Well Know Dataset that almost every beginner use it during his first models. Here is result of reduction to 2D and 3D Spaces by both algorithm.

First with 2D:

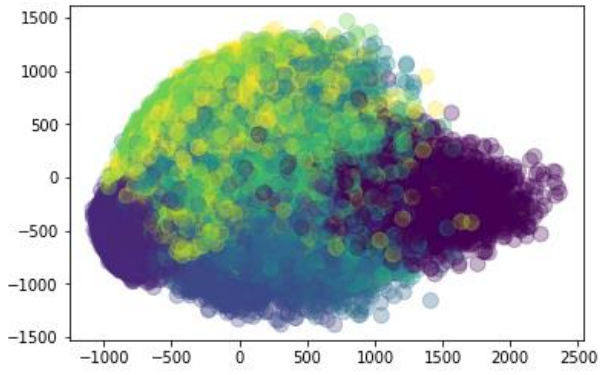


Figure 1 Sklearn PCA Result

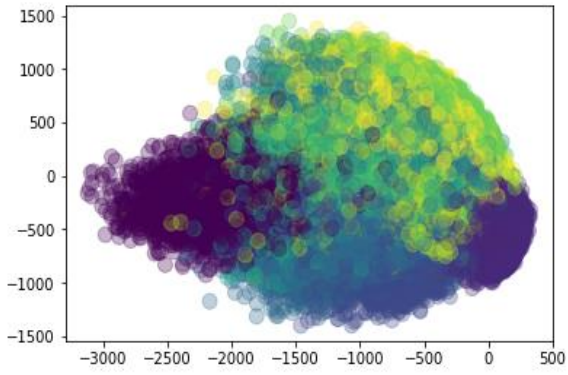


Figure 2 Our PCA Result

We See That Results are mirror of each other, we can obtain like Sklearn PCA by multiplying the result by $-\mathbf{1}$, but the purpose of PCA is not that.

Then with 3D:

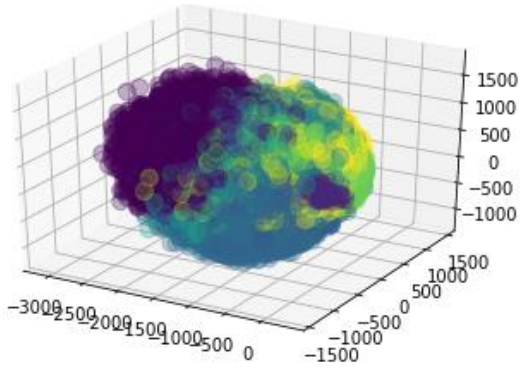


Figure 3 Sklearn PCA Result in 3D

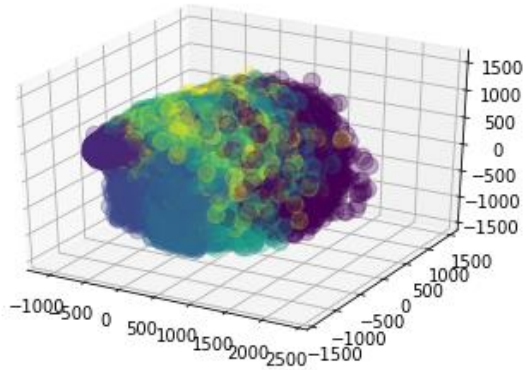


Figure 4 Our PCA Result in 3D

The rest of the example are in the Github Repository, just use this link.

Chapter III

Complexity Calculation

Formally:

The Algorithm is of complexity $\boldsymbol{O}\ (\boldsymbol{n^3})$ of polynomial class. The calculation is easy for this algorithm, we have two main parameters (rows and columns of dataset, $\boldsymbol{(n, m)}$ respectively) and the number of principal components we want to obtain buy running the algorithm, the algorithm make the covariance matrix of the input data which take $\boldsymbol{O}\ (\boldsymbol{nm^2})$, and then the eigen decomposition run on the result matrix of the covariance operation, which is of shape $\boldsymbol{m} \times \boldsymbol{m}$ and for this the Eigen's calculation will take $\boldsymbol{O}\ (\boldsymbol{m^3})$, creation of tuple (Eigen Value , Eigen Vector) takes $\boldsymbol{O}\ (\boldsymbol{m})$,all of this plus the sorting algorithm to use for sorting the Eigen values array of size m, we can use an algorithm with $\boldsymbol{O}\ (\boldsymbol{m\ Log(m)})$,The Standard[3] Library Use Time Sort[4] algorithm, then a reshaping of the vector has been established, to take vector from $(\boldsymbol{1 \times m})$ to $(\boldsymbol{m \times 1})$, this operation takes a Constant Time, another operation we used is the hstack function, which make an horizontal stack of columns in a matrix, this takes a linear time $\boldsymbol{O}\ (\boldsymbol{m})$; the time for doing projection on the new axis with matrix multiplication which is about $\boldsymbol{O}\ (\boldsymbol{nmc})$, in this case we made $\boldsymbol{c = m}$ because we want to search the worst case, all of this can be written as following:

$$
\begin{aligned}
&= \boldsymbol{O}\ (\boldsymbol{nm^2 + m^3 + m\ Log(m) + nmc + m}) \\
&= \boldsymbol{O}\ (\boldsymbol{nm^2 + m^3 + nm^2}) \\
&= \boldsymbol{O}\ \left(\boldsymbol{(n + m) \times m^2}\right)
\end{aligned}
$$

And since we are talking about the worst case here, if $\boldsymbol{m}$ is equal two $\boldsymbol{n}$ that will cause a huge calculation for Eigen's things and for covariance matrix which will lead to a complexity of $\boldsymbol{O}\ (\boldsymbol{2n^3}) = \boldsymbol{O}\ (\boldsymbol{n^3})$

[3] https://en.wikipedia.org/wiki/Timsort
[4] https://wiki.python.org/moin/TimeComplexity

Empirically

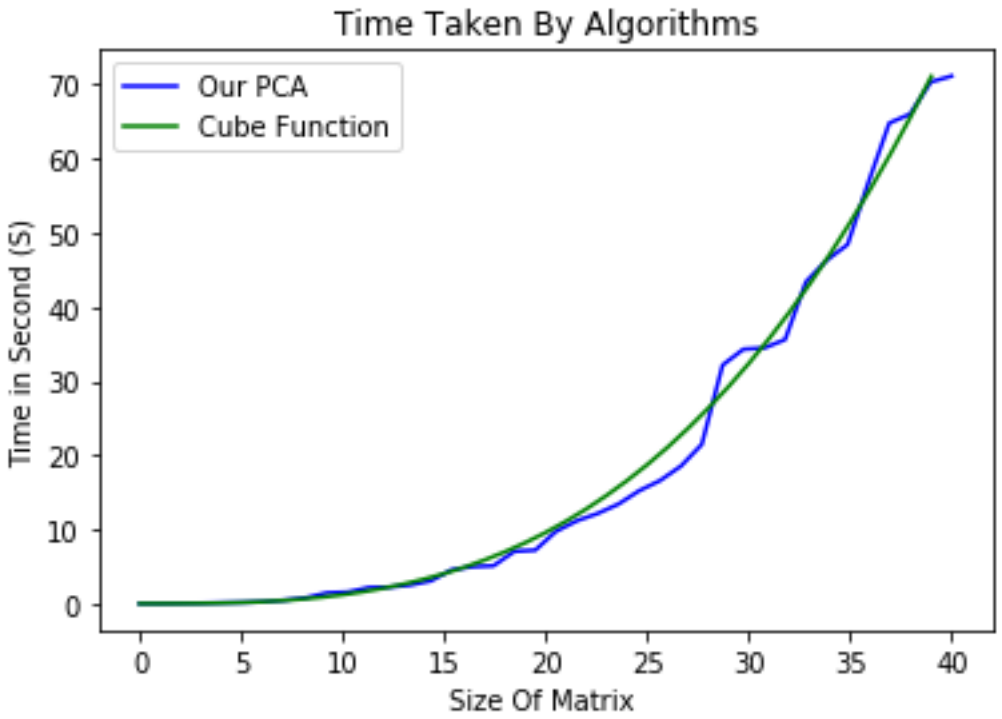
The next table shows the time in seconds for Pca Algorithm with data of shape $(n \times n)$ with n in range $[\mathbf{0}, \mathbf{4000}]$ with $\boldsymbol{step} = \mathbf{100}$

n	T(n , n) in Second	n	T(n , n) in Second
100	0.0010004	**2100**	9.75153
200	0.00700021	**2200**	11.1717
300	0.0369999	**2300**	12.1567
400	0.060997	**2400**	13.4285
500	0.121995	**2500**	15.2357
600	0.169	**2600**	16.6041
700	0.268997	**2700**	18.5548
800	0.479514	**2800**	21.4775
900	0.798524	**2900**	32.1727
1000	1.44898	**3000**	34.3103
1100	1.55898	**3100**	34.4796
1200	2.16397	**3200**	35.6105
1300	2.23397	**3300**	43.3833
1400	2.51096	**3400**	46.3093
1500	3.08162	**3500**	48.4198
1600	4.58208	**3600**	56.8012
1700	4.98221	**3700**	64.7981
1800	5.10407	**3800**	66.0101
1900	7.05277	**3900**	70.35
2000	7.20914	**4000**	71.143

We see when is getting to be bigger the algorithm is take a lot of time, in fact this because the number of features is equal to the number of examples, in real world dataset we will not have this kind of sparse dataset and most of the time the number of features will be smaller and even more smaller for the number of principal components wanted by the final user of this algorithm, in the MNIST data set for example , you have 60k rows , but only 768 columns, so the algorithm will not take a lot for this dataset (it take 7.17 Seconds , and 7.08 by the Sklearn Algorithm) , and as a final mentions , the machine learning and deep algorithm does heavily computation on the machine, so by default it will take time, and the optimization in this fields is related heavily on the hardware, for example NVidia Taking Up This field by its GPU, Google By Their TPU[5] and so one.

[5] https://cloud.google.com/tpu/

If we plot this result with the graph of our and with the function $f(x) = \frac{1}{900}\ \boldsymbol{x^3}$ we get the following:



From the graph we see that the curve of the time complexity for our PCA is approximately equal to the $\boldsymbol{f(x)}$.

Chapter IV

Difficulties encountered and possible solutions

During this cool project we've learned a lot of things, first we brush up our math skills about linear algebra, things like Eigen Value and Eigen Vector, and also statistics like calculating the covariance matrix. As this concepts are really cool and hard to compute we've used some library to do this task, like numpy for example, it's very optimize for doing this calculation, but the problem with this is we can't calculate the time complexity for this functions even as this library is open source[6] but it's really hard to read and understand what they are doing and then calculate the time complexity, so the solution proposed by our professor is to calculate the time complexity empirically and try to approximates the well know function. We found also some response on the stack overflow and other stack exchange community a general complexities for well know operation, and when we write the algorithm our self we saw that it's equivalent but the problem is that our analysis for the general algorithm and not the numpy implementation so the solution was hybrid, here is what we did:

Consider the following:

- $\boldsymbol{n}$: The Number of Row or Examples.
- $\boldsymbol{c}$: is the number of components the user want.
- $\boldsymbol{m}$: is the number of features or columns

What we found is that:

- **Covariance Matrix Calculation[7]: $\boldsymbol{O}\ (\boldsymbol{nm^2})$.**
- **Calculation of Eigen Things[8] : $\boldsymbol{O}\ (\boldsymbol{n^3})$**
- **Matrix Multiplication of $\boldsymbol{n}\ \times \boldsymbol{m}$ Matrix by $\boldsymbol{m}\ \times \boldsymbol{c}$: $\boldsymbol{O}\ (\boldsymbol{n} \times \boldsymbol{m} \times \boldsymbol{c})$ and we can say in the worst case $\boldsymbol{c}\ =\ \boldsymbol{m}$.**

Here is some screen shoot describing some functions behavior:

[6] https://github.com/numpy/numpy
[7] https://cstheory.stackexchange.com/questions/14734/what-is-computational-complexity-of-calculating-the-variance-covariance-matrix?rq=1
[8] https://cstheory.stackexchange.com/questions/2611/complexity-of-finding-the-eigendecomposition-of-a-matrix

Hstack Function:

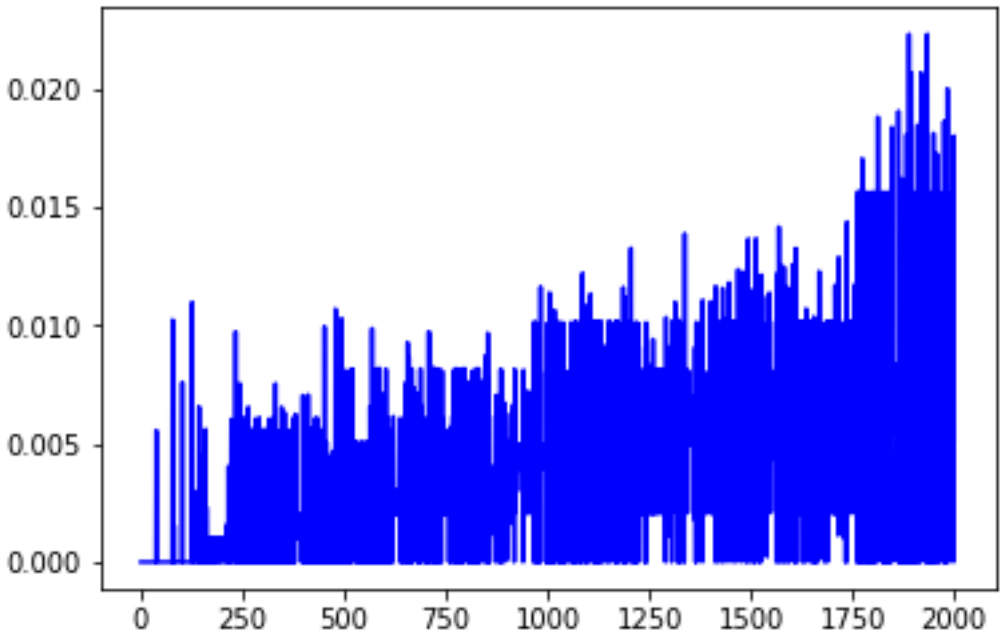


Figure 5 Empirical calculation for hstack function

Reshape Function:

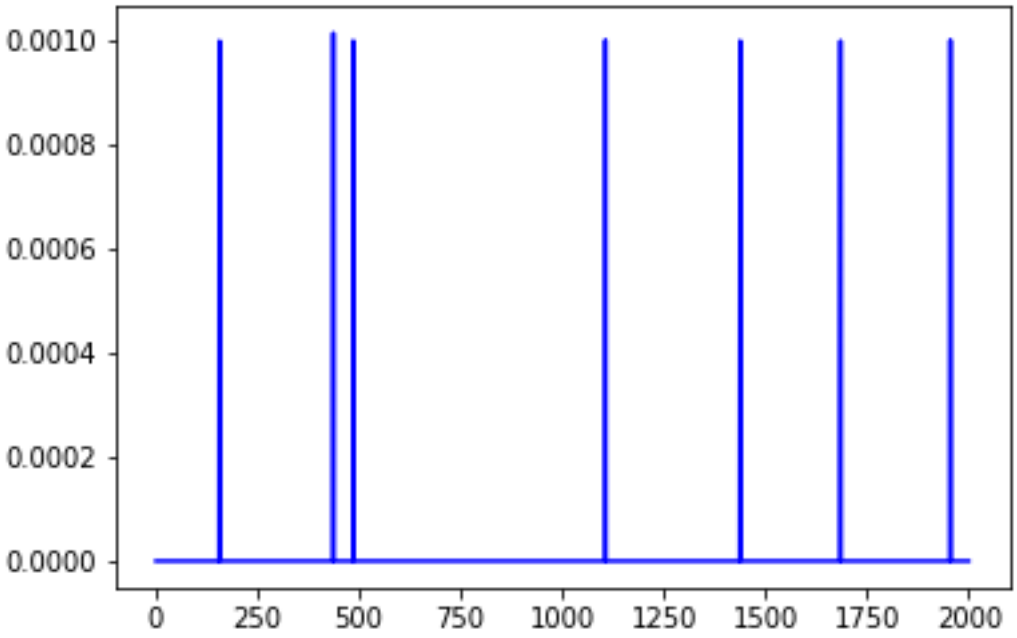


Figure 6 Empirical calculation for reshape function

Covariance Function: we gave the methods matrices of shape $\boldsymbol{n} \times \boldsymbol{n}$

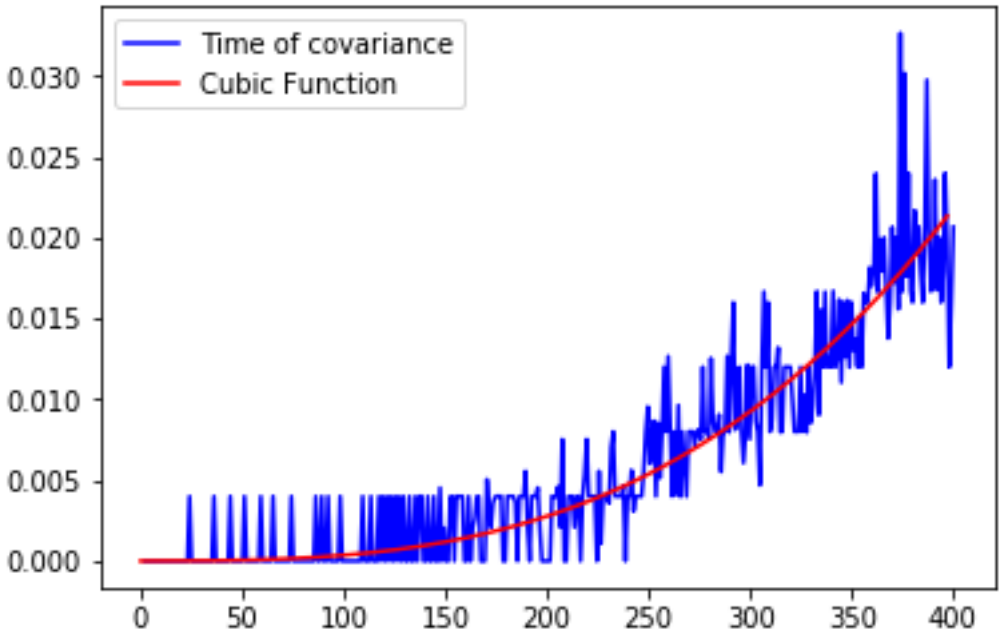


Figure 7 Empirical Calculation for Covariance function

Conclusion:

During this practical work we demonstrate the why and the how of the principal component analysis, the math behind it, the time complexity and the different result we obtained, we made also a comparison between Our Implemented PCA and The Pca from Sklearn Library which provide a lot of tools and models for machine learning and deep learning tasks. The Purpose was to find the complexity of this algorithm, we found it and it was $\boldsymbol{O}\ (\boldsymbol{n^3})$ but isn't that much faster, we could do better by changing the methodology of finding the axes the describe the variance of the data, one method called Singular Value Decomposition (SVD) which can be used instead of Eigen Decomposition Task, most PCA implementations perform SVD to improve the computational efficiency.