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.
    - **Scaling**: A Boolean value indicates weather to standardize the data or not.
- Output:
    - The Data Matrix transformed to new dimensionality.

Program:

```
import numpy as np
from sklearn.preprocessing import StandardScaler

def pca(number_compontents, data , is_scaled = True):
    if not is_scaled:
        scaler = StandardScaler()
        data = scaler.fit_transform(data)

    # calculate the covariance matrix
    cov_matrix = np.cov(data.T)

    # calculate the eigen things
    eig_vals , eig_vect = np.linalg.eigh(cov_matrix)

    # codes are the same
    eig_pairs = [(np.abs(eig_vals[i]) , eig_vect[:,i]) for i in range(len(eig_vals))]

    # Sorting All of Them
    eig_pairs.sort(key = lambda x : x[0] , reverse= True)

    # Choosing The Highest Ones
    final = []
    for i in range(number_compontents):
        final.append(eig_pairs[i][1].reshape(data.shape[1],1))

    # Creating the Projection Matrix
    projection_matrix = np.hstack((final))

    # transforming the data
    Y = data.dot (projection_matrix)

    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/

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, $(\boldsymbol{n}, \boldsymbol{m})$ 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 $\boldsymbol{O}\ (\boldsymbol{m^3})$, all of this plus the sorting algorithm to use for sorting the eigen values array of size m, we can use an algorithm with $\boldsymbol{O}\ (\boldsymbol{m}\ \boldsymbol{Log}(\boldsymbol{m}))$, the time for doing projection on the new axis with matrix multiplication which is about $\boldsymbol{O}\ (\boldsymbol{nm^2})$; all of this can be written as following:

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

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

Empirically

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

n	T(n , n)	n	T(n , n)
100		2100	
200		2200	
300		2300	
400		2400	
500		2500	
600		2600	
700		2700	
800		2800	
900		2900	
1000		3000	
1100		3100	
1200		3200	
1300		3300	
1400		3400	
1500		3500	
1600		3600	
1700		3700	
1800		3800	
1900		3900	
2000		4000	

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[3] 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[4]: $\boldsymbol{O\ (nm^2)}$.**
- **Calculation of Eigen Things[5] : $\boldsymbol{O\ (n^3)}$**
- **Matrix Multiplication of $\boldsymbol{n \times m}$ Matrix by $\boldsymbol{m \times c}$: $\boldsymbol{O\ (n \times m \times c)}$ and we can say in the worst case $\boldsymbol{c = m}$.**

The Result is corresponding to the empirical calculation as we can see in the following screen shots.

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