Image Compression using Principal Component Analysis (PCA)

August 30, 2021

Topics: Machine Learning

Principal Component Analysis (PCA), is a dimensionality reduction method used to reduce the dimensionality of a dataset by transforming the data to a new basis where the dimensions are non-redundant (low covariance) and have high variance.

This tutorial aims to make the reader understand the concept of PCA mathematically by providing them with one of the use cases of PCA, image compression.

Table of contents

Prerequisites

The reader should have basic knowledge of linear algebra (matrix operations and their properties) and statistics. Also, readers should be familiar with few terms of linear algebra, like basis vectors, vector spaces, orthogonality, and covariance. Make sure you understand these terms before going through this blog.

Introduction

What is Dimensionality Reduction?

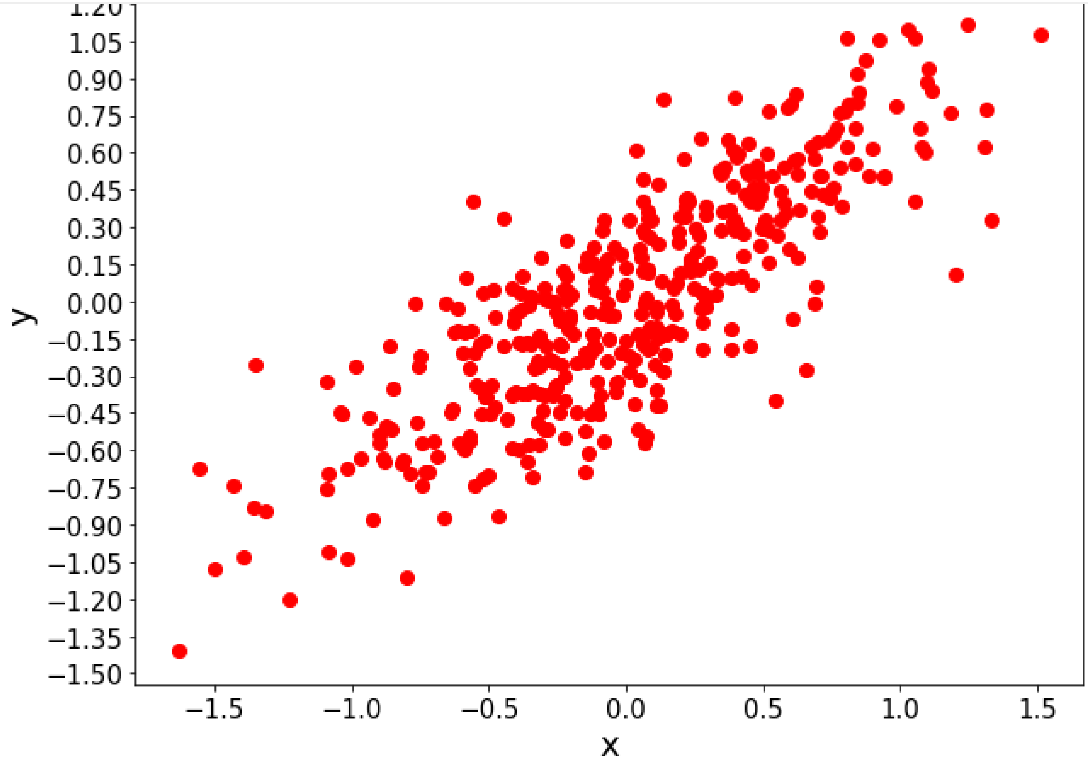
Dimensionality: The number of input variables or features for a dataset is

> **Dimensionality reduction** is the transformation of data from a high-dimensional space into a low-dimensional space so that the low-dimensional representation retains some meaningful properties of the original data, ideally close to its intrinsic dimension (number of variables needed in a minimal representation of the data).

Dimensionality reduction refers to techniques that reduce the number of input variables in a dataset. More input features often make a predictive modeling task more challenging to model, generally referred to as the **curse of dimensionality**.

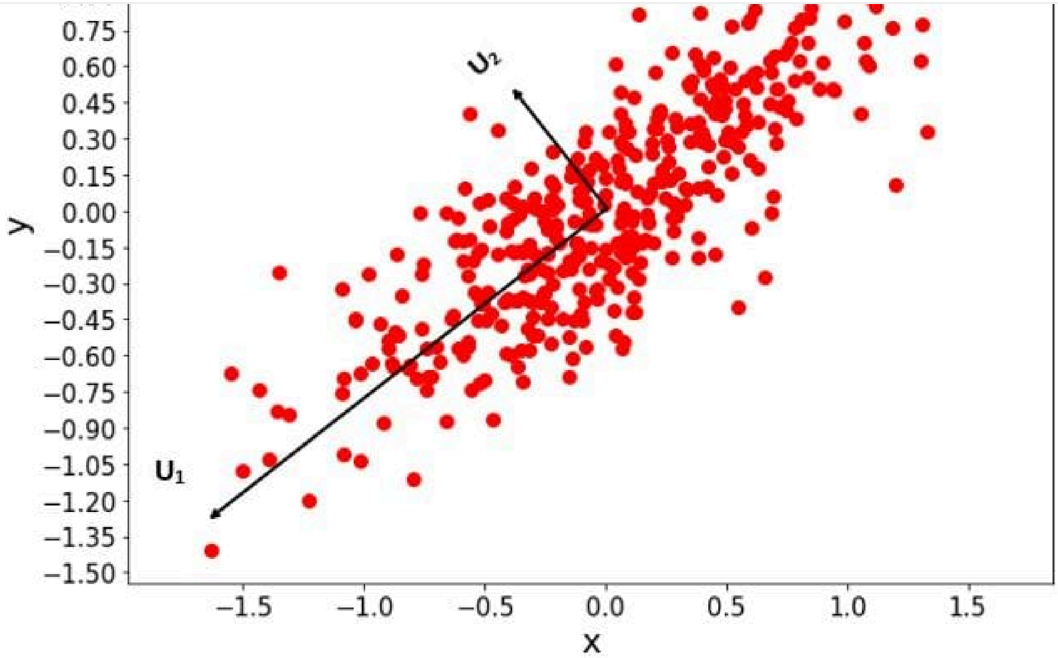
To explain the concept of dimensionality reduction, I will take an example, consider the following data where each point (vector) is represented using a linear combination of the x and y axes:

Postively correlated data



Motivation for Dimensionality Reduction

Now, what if we choose a different basis?



Here I have used u_1 and u_2 as a basis instead of x and y. We can observe that all the points have a minimal component in the direction of u_2 (almost noise).

It seems that the same data that was initially in $R_2 : (x, y)$ can now be represented in $R_1 : (u_1)$ by making an intelligent choice of basis.

Why not care about u_2.

Because the variance in the data in this direction is minimal (all data points have almost the same value in the u_2 direction), if we were to build a classifier on top of this data, then u_2 would not contribute to the classifier as the points are not distinguishable along this direction. In this way, we have reduced the dimensionality.

> In general, we are interested in representing the data using fewer dimensions such that the data has higher variance along these dimensions.

Correlation

columns are highly correlated (or have high covariance), one is redundant since it is linearly dependent on the other column.

We can normalize the correlation to get the correlation coefficient. The formula for the correlation coefficient is defined as:

Equation

Requirements for Dimensionality Reduction

In general, we are interested in representing the data using fewer dimensions such that,

The data has **high variance** along these dimensions.

The dimensions are linearly **independent** (uncorrelated).

If we want to reduce dimensions by transforming the data into a new basis, the resulting basis should be **orthogonal**.

Principal Component Analysis (PCA)

> **Principal Component Analysis (PCA)** is a statistical procedure that uses an orthogonal transformation that converts a set of correlated variables to a set of uncorrelated variables.

To explain the concept of PCA mathematically, I will go over an example:

Orthogonal transformation

Assumptions

Let $p_1, p_2, p_3, \ldots, p_n$ be a set of n orthonormal vectors. Let p be a square

Let $x_1, x_2, x_3, \ldots, x_n \in R^n$ be n data points and let X be a square matrix such that $x_1, x_2, x_3, \ldots, x_n$ are the rows of a matrix

(Assumption: Data is zero-mean and unit variance), we will go over why we have this assumption at a later part.

Transformation

Suppose we want to represent x_i using the new basis P.

$$x_i = \alpha_{i1} p_1 + \alpha_{i2} p_2 + \ldots \ldots . + \alpha_{in} p_n$$

As we have assumed P as an orthonormal basis, for an orthonormal basis, we can find α_i using, $\alpha_{ij} = x_i^T p_j$.

In general, the transformed data x_i' is given by $x_i' = x_i^T P$, thus the transformed matrix will be given as:

$$X' = XP$$

Covariance matrix (Σ)

If X is a matrix with zero mean, then $\Sigma = 1/m * (X'^T X)$ is the covariance matrix. In other words, Σ_{ij} stores the covariance between columns i and j of X.

Explanation: Let C be the covariance matrix of X. Let μ_i and μ_j denote the means of i^{th} and j^{th} column of X, respectively.

Then by the definition of covariance, we can write:

$C_{ij} = 1/m * \sum_{k=1}^{m} (X_{ki}-\mu_i)(X_{kj}-\mu_j)$

$C_{ij} = 1/m * X_i^T X_j = 1/m * (X^T X)_{ij}$

So far we know that,

$X' = XP<$

Covariance matrix of transformed data can be written as:

$= 1/m * X'^T X'$

$= 1/m * (XP)^T XP$

$= 1/m * P^T X^T XP$

$= 1/m * P^T (X^T X)P$

$= P^T (1/m * (X^T X))P$

$= P^T \Sigma P$

What do we want from the covariance matrix of transformed data?

Ideally we want,

$(1/m * X'^T X')_{ij} = 0$ if $i \neq j$ (Covariance = 0)

$(1/m * X'^T X')_{ij} \neq 0$ if $i = j$ (Variance = 0)

In other words, we want $P^T \Sigma P = D$ (where D is a diagonal matrix).

It will have distinct non-negative eigenvalues, and thus, linearly independent eigenvectors.

Eigenvectors of a symmetric matrix are orthogonal, which can be turned into an orthonormal basis.

Principal components

Now we know that Σ is a symmetric matrix, and the eigenvectors of Σ can be used as a suitable orthonormal basis. From the Diagonalization of matrix principle, we can say that to make $P^T \Sigma P$ a diagonal matrix, P will be the matrix of eigenvectors of the matrix Σ.

Now we can perform orthonormal transformation:

$$x_i = \Sigma_{j=1}^{n} (\alpha_{ij} p_j)$$

The n orthogonal eigenvectors p_j are the **Principal Components**.

Dimensionality reduction

As discussed already, we want to retain uncorrelated dimensions that have a maximum variance. Therefore for dimensionality reduction, we will sort the eigenvectors according to eigenvalues in descending order and keep top k eigenvectors to represent x_i:

$$x_i' = \Sigma_{j=1}^{k} \alpha_{ij} p_j$$

Where x_i' is a reconstructed vector with k dimensions, earlier we were using n dimensions to represent it. Hence the dimensionality is being reduced.

Use case: Image compression

(For illustration, I have used the Olivetti dataset, available at:

Consider we are given a large number of images of human faces (for this dataset, we have 400 images), each image is $64 \times 64 (4096$ **dimensions).** Now, we would like to represent and store the images using much fewer dimensions(say 100 dimensions).

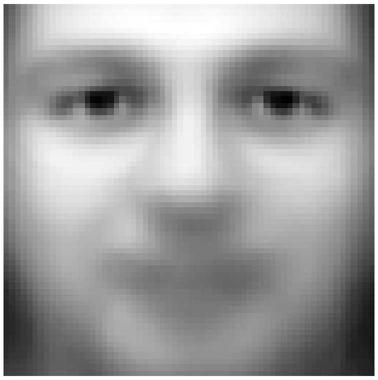
```
# Importing necessary libraries
import numpy as np
import matplotlib.pyplot as plt
from numpy import linalg as LA

dirname = '/content/gdrive/My Drive/Kaggle'
fileName = 'olivetti_faces.npy'
faces = np.load(os.path.join(dirname,fileName))
```

Let us see a sample image from the dataset.

```
plt.imshow(faces[1], cmap='gray')
plt.axis('off')
```

Let's see what the average of all images looks like:



First, we will make all our images zero centered, subtracting the average image from each image in the matrix for zero centering.

```
X = faces
X = X.reshape((X.shape[0], X.shape[1]**2)) #flattening the image
X = X - np.average(X, axis=0) #making it zero centered

#printing a sample image to show the effect of zero centering
plt.imshow(X[0].reshape(64,64), cmap='gray')
plt.axis('off')
```

Now, after making the images zero-centered, we will calculate the covariance

```
cov_mat = np.cov(X, rowvar = False)

#now calculate eigen values and eigen vectors for cov_mat
cov_mat = np.cov(X, rowvar = False)

#sort the eigenvalues in descending order
sorted_index = np.argsort(eigen_values)[::-1]

sorted_eigenvalue = eigen_values[sorted_index]
#similarly sort the eigenvectors
sorted_eigenvectors = eigen_vectors[:,sorted_index]
```

Initially, the image had 4096 dimensions. Let's reduce the dimension from 4096 to 100.

```
n_components = 100
eigenvector_subset = sorted_eigenvectors[:,0:n_components]
print(eigenvector_subset.shape)
```

```
(4096, 100)
```

These 100 dimensions are the **Principal Components**. Now, as we can see, the shape of eigenvector_subset is $(4096, 100)$. If represented as a 64×64 image after performing transpose on this matrix, we can get 100 such images.

These 100 images will be called **Eigenfaces**, and one can represent any image as a linear combination of these 100 images.

Let's print first 16 eigenfaces.

```
  fig.add_subplot(4,4,i+1)
  plt.imshow(eigenvector_subsetT[i].reshape(64,64) , cmap= 'gray')
  plt.axis('off')
plt.show()
```

How is the image compressed?

Initially, the image had dimensions of 4096. Let's reduce the dimension from

```
x_reduced = np.dot(eigenvector_subset.transpose(),X.transpose()).transp
print(x_reduced.shape)
```

```
(400, 100)
```

Earlier, to store 400 images, we required a 400×4096 matrix. We need to store 400×100 matrix for x_reduced and 4096×100 matrix for storing principal components.

As the image is gray-scale, let's suppose it requires 2 bits to store each pixel of an image. Therefore, after compressing the image, we will be able to save:

$= ((400 \times 4096) \times 2)] - ((400 \times 100) + (4096 \times 100)) \times 2$

$= 2377600$ bits

$= 297200$ bytes

≈ 290 KB

For our example, the images were gray-scale and had a low resolution; that is why we could save only 290 KB. On the other hand, suppose images have very high resolution with more than one channel, then one can use this method to save lots of space.

Reconstructing images using less information

```
# Reconstructing the first image
```

```
temp= np.matmul(eigenvector_subset,x_reduced[1])
```

```
plt.axis('off')
```

Reconstructed Image:

Original Image:

Conclusion

As we have seen in this tutorial, using the concept of PCA, we have compressed the images and stored the eigenfaces. And to retain the image, we reconstruct it using the stored eigenfaces.

But we have to note that there is extra calculation overhead for reconstructing

Happy coding!

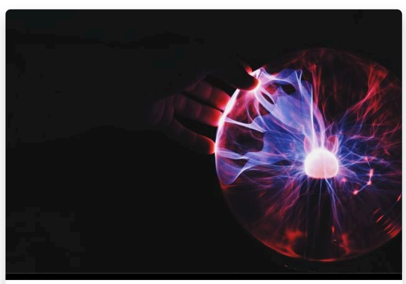
Peer Review Contributions by: Lalithnarayan C

Similar Articles

Machine Learning

Guide to Probability Predictions on Multi-class Labels

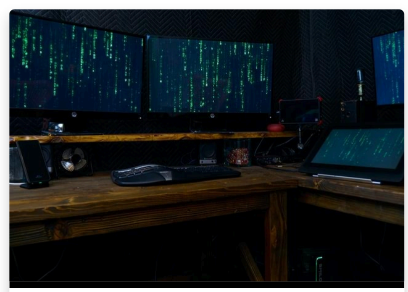
Read More



Machine Learning

Getting Started with Kernel PCA in Python

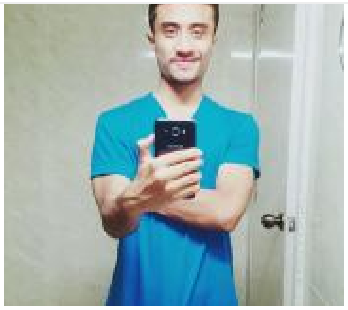
Read More



Machine Learning, Languages

System of Linear Equations using Gaussian Elimination Algorithm

Read More



Deewakar Chakraborty

Deewakar Chakraborty is currently pursuing his Masters in Data Science at Defence Institute of Advanced Technology, DRDO Pune. He likes to build things and understand the problems that one cannot see through a strictly theoretical perspective. Apart from studies, he listens to Greenday and supports Real Madrid.

View author's full profile →

Sect