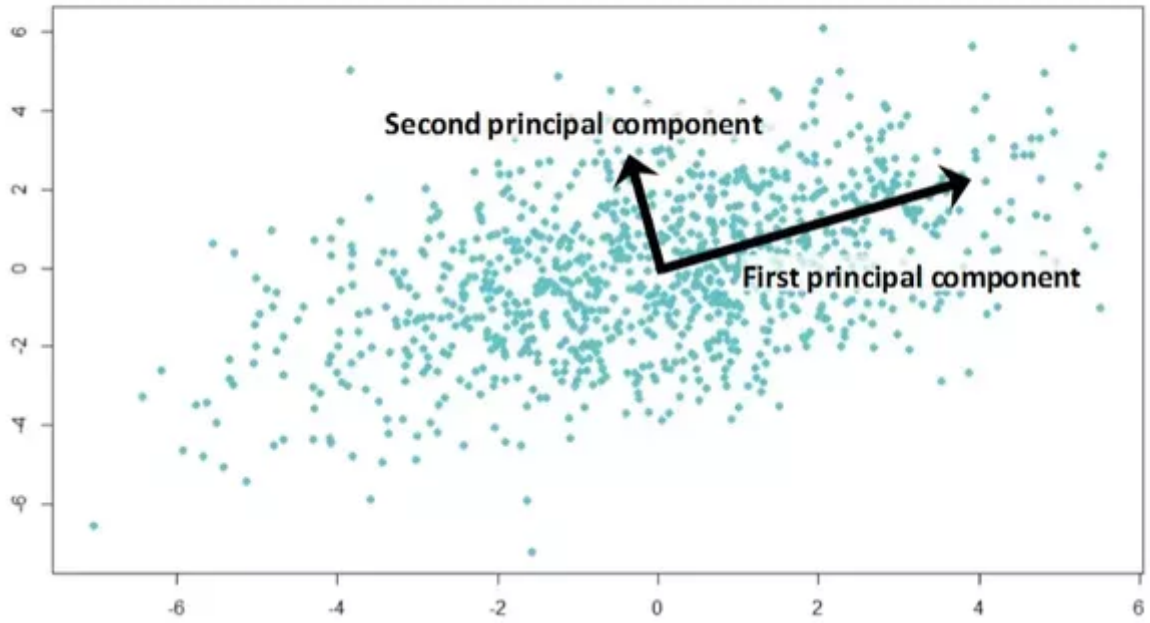
PCA Introduction

Principal Component Analysis, or PCA for short, is a method used for dimensionality reduction and the visualization of high dimensional data.

It can be thought of as a projection method where data with m-columns (features) is projected into a subspace with m or fewer columns, whilst retaining the essence of the original data.

The PCA method can be described and implemented using the tools of linear algebra. It projects the data in a lower dimensional space along the directions of maximum variance that are orthogonal to each other, while trying to preserve as much of the data's variation as possible.



Calculating PCA involves:

- Calculating the covariance matrix.
- Calculating the eigenvalues and eigenvector
- Forming Principal Components
- Projection into a new feature space

Covariance Matrix

- Representing Covariance between dimensions as a matrix e.g. for 3 dimensions:

$$C = \begin{bmatrix} cov(x,x) & cov(x,y) & cov(x,z) \\ cov(y,x) & cov(y,y) & cov(y,z) \\ cov(z,x) & cov(z,y) & cov(z,z) \end{bmatrix}$$

Variances

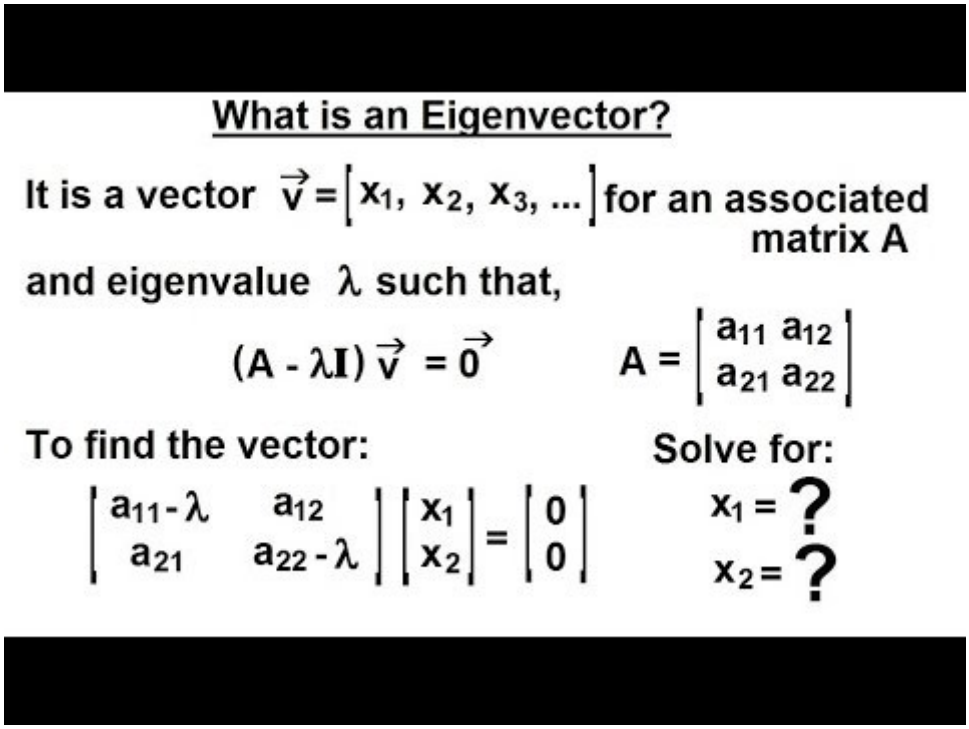
- Diagonal is the variances of x, y and z
- cov(x,y) = cov(y,x) hence matrix is symmetrical about the diagonal
- N-dimensional data will result in NxN covariance matrix

Calculating the eigen values and eigen vectors

- For every eigenvalue, there is a corresponding eigenvector.
- We will sort eigenvalue in decreasing order.
- The vector V1 corresponds to maximum eigenvalue have maximum variance implying maximum information of the dataset.
- Similarly, variance decreases as eigenvalue decreases.

λ is an eigen value for a matrix A if it is a solution of the characteristic equation: det(λ*I - A) = 0

- Where, I is the identity matrix of the same dimension as A.
- Having determined the eigenvalue λ, a corresponding eigen-vector v, can be found by solving: (λ*I-A)v = 0
- The eigenvalues λ1,λ2,...,λm are the variances of the corrdinates on each principal component axis.



Overview:

1. First, we need to center our data

$$\hat{x}_{i,j} = x_{i,j} - \mu_i$$
$$\mu_i = \frac{1}{N}\sum_{j=1}^{N} x_{i,j}$$

2. Then we compute covariance matrix *S* for centered data and find its eigenvectors *b* and eigenvalues λ.

$$S = \frac{1}{N}\hat{X}\hat{X}^T,\ S \in [D, D]$$
$$Sb = \lambda b,\ b \in [D, 1],\ \lambda \in \mathbb{R}$$

3. Sort eigenvectors by corresponding eigenvalues in descending order.
4. Then we form matrix B from first M eigenvectors and perform dimensionality reduction

$$Code = B^T\hat{X},\ \ B \in [D, M],\ \ Code \in [M, N]$$

5. For inverse transformation we need to get back to original space (find projections) and shift data back from the center adding mean values

$$X^* = BB^T\hat{X},\ X^* \in [D, N]$$
$$x^*_{i,j} = x^*_{i,j} + \mu_i$$

Transforming the data using PCA — removes the collinearity within the data. However, since PCA only removes linear dependence among variables; even after transforming the variables using

PCA, they may still be dependent — in a non linear way.

PCA Applications:

- Data Visualization
- Speed-up Machine Learning Algorithms

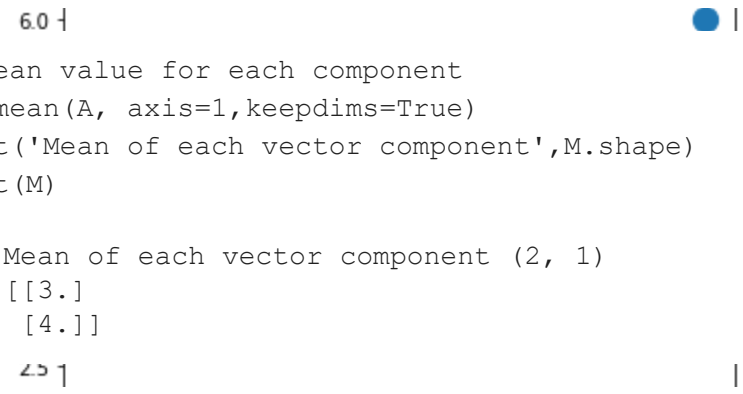
Manually Calculate Principal Component Analysis

PCA is an operation applied to a dataset, represented by an n x m matrix A that results in a projection of A which we will call B. Let's walk through the steps of this operation.

```
from numpy import array
from numpy import mean
#Function that computes covariance matrix
from numpy import cov
#Function that computes the Eigen decomposition
from numpy.linalg import eig
import matplotlib.pyplot as plt

# define a matrix
#
#     a11, a12
#A = (a21, a22)
#     a31, a32
# 2x3 matrix
# 3 vectors with two components
A = array([[1, 3, 5], [2, 4, 6]])
print('A:')
print(A)

plt.scatter(A[0,:], A[1,:], s=150)
plt.show()
```

```
A:
```

The first step is to calculate the mean values of each column.



```
#A mean value for each component
M = mean(A, axis=1,keepdims=True)
print('Mean of each vector component',M.shape)
print(M)
```

```
Mean of each vector component (2, 1)
[[3.]
 [4.]]
```

PCA is largely affected by scales and different features might have different scales. So it is better to standardize data before finding PCA components. Next, we need to center the values in each column by subtracting the mean column value.

```
C = A - M
print('Normalized matrix',C.shape)
print(C)
```

```
Normalized matrix (2, 3)
[[-2.  0.  2.]
 [-2.  0.  2.]]
```

The next step is to calculate the covariance matrix of the centered matrix C.Correlation is a normalized measure of the amount and direction (positive or negative) that two columns change together. Covariance is a generalized and unnormalized version of correlation across multiple columns. A covariance matrix is a calculation of covariance of a given matrix with covariance scores for every column with every other column, including itself.

```
V = cov(C)
print('Covariance matrix V:')
print('C',C.shape,'x C^T ',C.T.shape,'->',V.shape)
print(V)
```

```
Covariance matrix V:
C (2, 3) x C^T  (3, 2) -> (2, 2)
[[4. 4.]
 [4. 4.]]
```

Finally, we calculate the eigendecomposition of the covariance matrix V. This results in a list of eigenvalues and a list of eigenvectors.

The eigendecomposition is calculated on a square matrix using an efficient iterative algorithm, of which we will not go into the details.

```
# eigendecomposition of covariance matrix
values, vectors = eig(V)
print('\nEigenvectors:')
print(vectors)
print('\nEigenvalues:')
print(values)
```

```
Eigenvectors:
[[ 0.70710678 -0.70710678]
 [ 0.70710678  0.70710678]]

Eigenvalues:
[8. 0.]
```

```
# project data
P = vectors.T.dot(C)
print('\nProjected data:')
print(P)
```

```
Projected data:
[[-2.82842712  0.          2.82842712]
 [ 0.          0.          0.        ]]
```

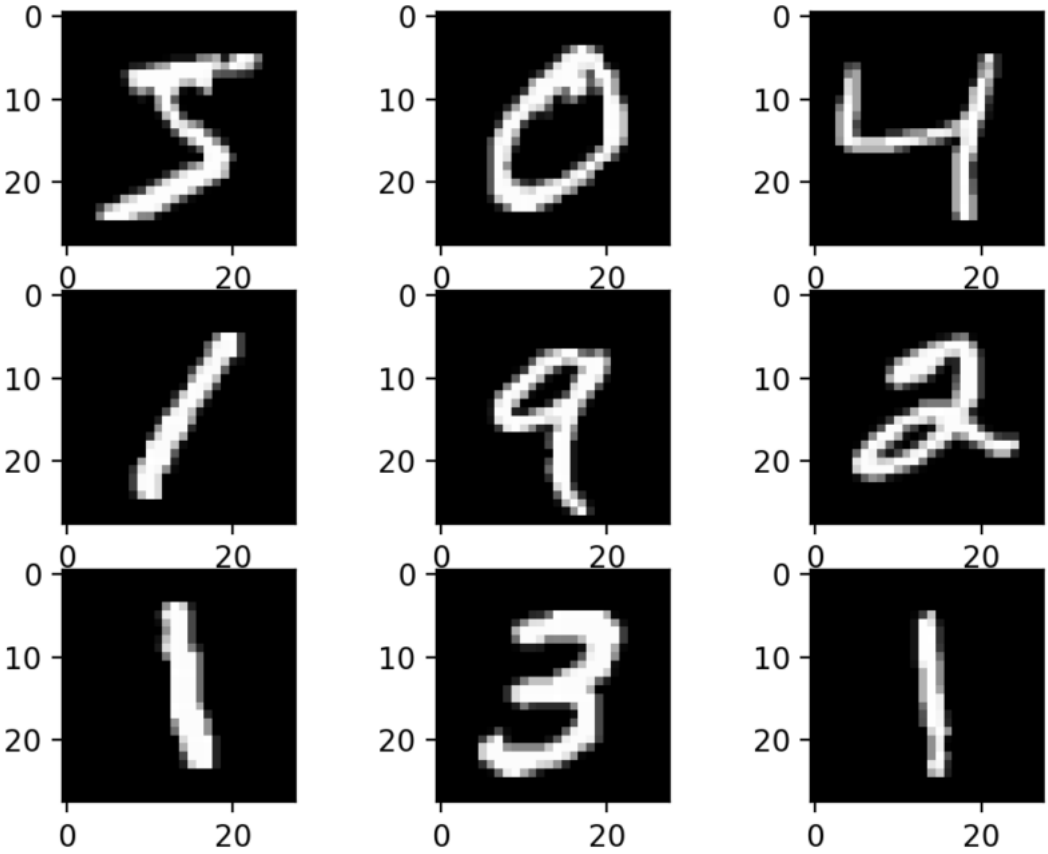
Interestingly, we can see that only the first eigenvector is required, suggesting that we could project our 2×3 matrix onto a 1×3 matrix with little loss, i.e., we lose no information when projecting to a line. Essentially project along the X axis.

This is called the covariance method for calculating the PCA, although there are alternative ways to to calculate it.

Other matrix decomposition methods can be used such as Singular-Value Decomposition, or SVD.

PCA on MNIST dataset with code

Loading the MNIST dataset images and their labels. Load only the training set normalizing the data.



Each number is a 28x28 image which corresponds to a 784 dimensional vector. This is difficult to visualize so we need to reduce the dimensionality. Lets try and reduce the dimensionality to 3!

```
import numpy as np
import matplotlib.pyplot as plt
from keras.datasets import mnist

from sklearn.preprocessing import StandardScaler

(X_train, label), (_, _) = mnist.load_data()
X_train = X_train.reshape(len(X_train), np.prod(X_train.shape[1:]))
data = X_train
standardized_data = StandardScaler().fit_transform(data)
print(standardized_data.shape)
```

```
(60000, 784)
```

```
#find the co-variance matrix which is : A^T * A
sample_data = standardized_data
# matrix multiplication using numpy
covar_matrix = np.matmul(sample_data.T , sample_data)
print ("The shape of variance matrix =", covar_matrix.shape)
```

```
The shape of variance matrix = (784, 784)
```

```
# finding the top two eigen-values and corresponding eigen-vectors
# for projecting onto a 3-Dim space.
#https://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.linalg.eigh.html
from scipy.linalg import eigh
# the parameter 'eigvals' is defined (low value to heigh value)
# eigh function will return the eigen values in asending order
# this code generates only the top 3 eigenvalues.
values, vectors = eigh(covar_matrix)
vectors = vectors[:,-3:]
print("Shape of eigen vectors = ",vectors.shape)
# converting the eigen vectors into (3,d) shape for easyness of further computation
vectors = vectors.T
print("Updated shape of eigen vectors = ",vectors.shape)
# here the vectors[1] represent the eigen vector corresponding 1st principal eigen
# here the vectors[0] represent the eigen vector corresponding 2nd principal eigen
```

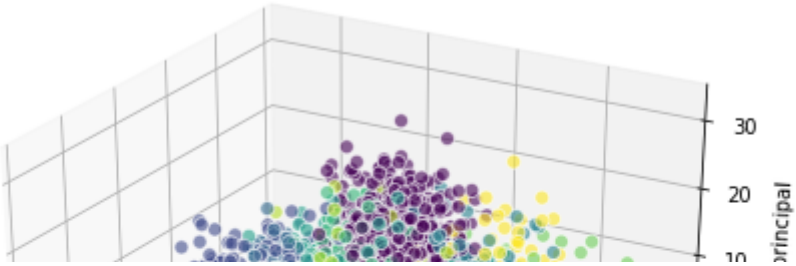
```
Shape of eigen vectors =  (784, 3)
Updated shape of eigen vectors =  (3, 784)
```

Projecting the original data sample on the plane formed by three principal eigenvectors by vector-vector multiplication.

```
import matplotlib.pyplot as plt
new_coordinates = np.matmul(vectors, sample_data.T)
# also add the label as a third dimension to use it for color visualization
new_coordinates = np.vstack((new_coordinates, label)).T
print(new_coordinates.shape)
```

```
(60000, 4)
```

```
fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(111, projection='3d')

ax.scatter(new_coordinates[:,0], new_coordinates[:,1], new_coordinates[:,2], c=new_

ax.set_xlabel('1st_principal')
ax.set_ylabel('2nd_principal')
ax.set_zlabel('3d_principal')

plt.show()
```

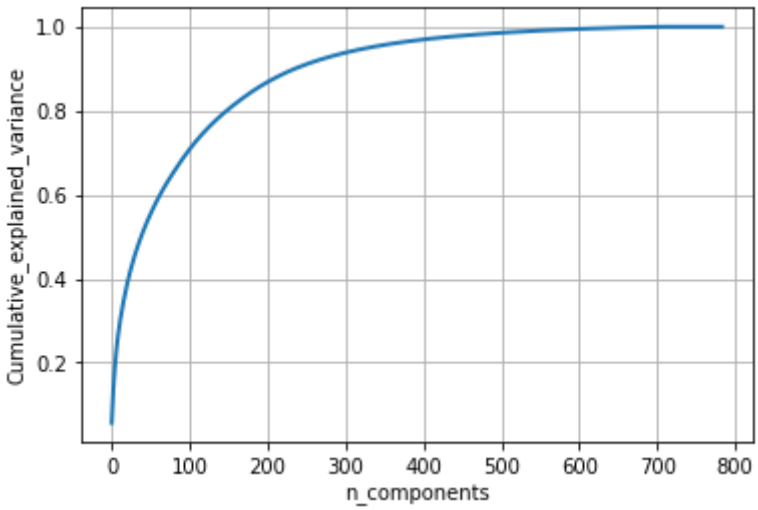


Appending label to the 3d projected data(vertical stack) and plotting the labeled points.

PCA for dimensionality Reduction using Scikit-Learn

Lets see how to use the principle components to reconstruct the original data. We will use the sci-kit learn library and its built-in PCA method.

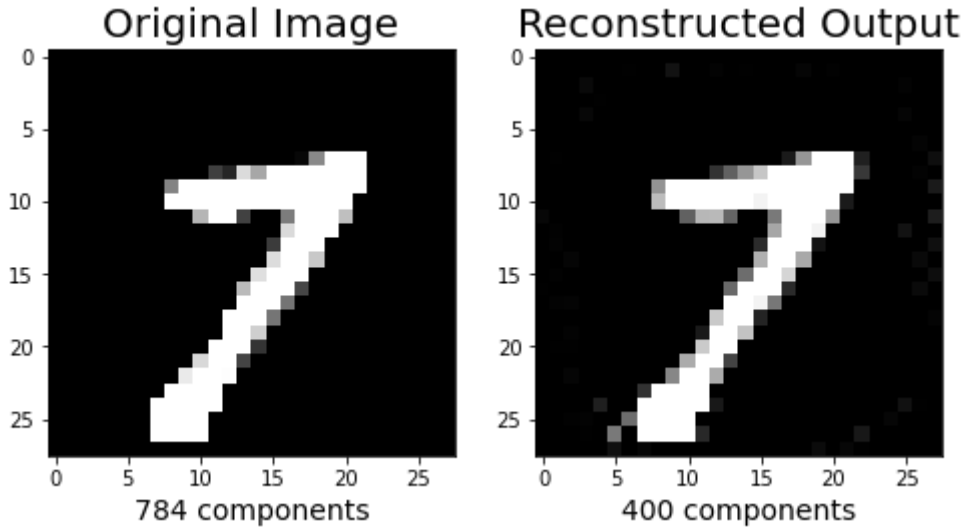
```
# initializing the pca
from sklearn import decomposition
from sklearn.preprocessing import StandardScaler

pca = decomposition.PCA()
scaler = StandardScaler()

#Fit on training set only.
mnist = scaler.fit_transform(sample_data)


# PCA for dimensionality redcution
#We set the maximum number of components
pca.n_components = 784

pca_data = pca.fit_transform(mnist)

#The PCA function internally computes how much of the variance is captured by
#each component.
percentage_var_explained = pca.explained_variance_ / np.sum(pca.explained_variance_

cum_var_explained = np.cumsum(percentage_var_explained)

# Plot the PCA spectrum
plt.figure(1, figsize=(6, 4))

plt.clf()
plt.plot(cum_var_explained, linewidth=2)
plt.axis('tight')
plt.grid()
plt.xlabel('n_components')
plt.ylabel('Cumulative_explained_variance')
plt.show()
```



If we take 200-dimensions, approx. 90% of variance is expalined.

The idea with going from 784 components to 154 is to reduce the running time of a supervised learning algorithm. One of the cool things about PCA is that we can go from a compressed representation (154 components) back to an approximation of the original high dimensional data (784 components).

```
#Select the number of components used to construct the lower dimensional space
#15,200
pca.n_components = 400
# fit the transform and project data to the lower-dimensional space
pca_data = pca.fit_transform(mnist)
#poject back to the original space
approximation = pca.inverse_transform(pca_data)

#Plot the original image and its reconstruction
plt.figure(figsize=(8,4));

# Original Image
plt.subplot(1, 2, 1);
#30000,40000
ind=40000
plt.imshow(sample_data[ind].reshape(28,28)*255,
              cmap = plt.cm.gray, interpolation='nearest',
              clim=(0, 255));
plt.xlabel('784 components', fontsize = 14)
plt.title('Original Image', fontsize = 20);

plt.subplot(1, 2, 2);
plt.imshow(approximation[ind].reshape(28, 28)*255,
              cmap = plt.cm.gray, interpolation='nearest',
              clim=(0, 255));
plt.xlabel(str(pca.n_components)+' components', fontsize = 14)
plt.title('Reconstructed Output', fontsize = 20);
```


Original Image
Reconstructed Output
0
5
10
15
20
25
784 components
400 components