A Complete Guide to Principal Component Analysis — PCA in Machine Learning

Ashutosh Tripathi
Jul 11, 2019 · 10 min read ★

Principal Component Analysis or PCA is a widely used technique for dimensionality reduction of the large data set. Reducing the number of components or features costs some accuracy and on the other hand, it makes the large data set simpler, easy to explore and visualize. Also, it reduces the computational complexity of the model which makes machine learning algorithms run faster. It is always a question and debatable how much accuracy it is sacrificing to get less complex and reduced dimensions data set. we don't have a fixed answer for this however we try to keep most of the variance while choosing the final set of components.

In this article, we will be discussing the step by step approach to achieve dimensionality reduction using PCA and then I will also show how can we do all this using python library.

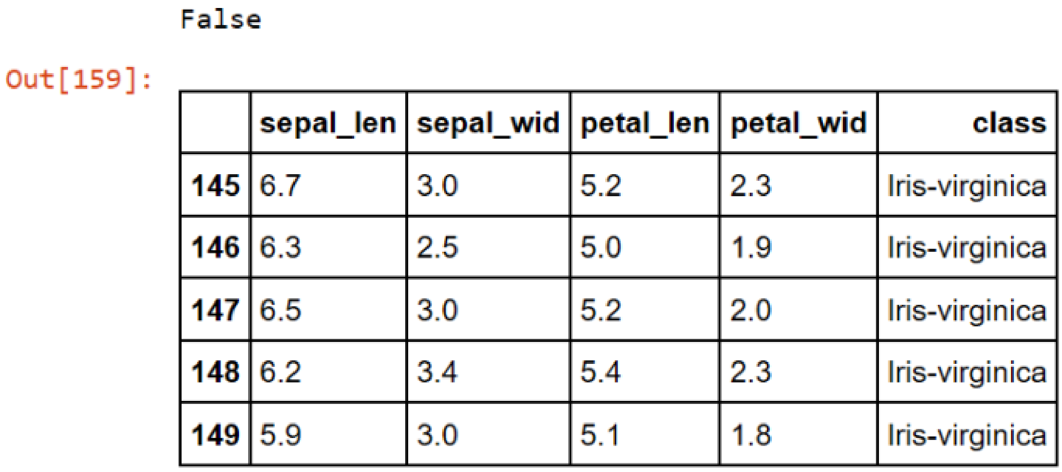
Steps Involved in PCA

1. Standardize the data. (with mean =0 and variance = 1)
2. Compute the Covariance matrix of dimensions.
3. Obtain the Eigenvectors and Eigenvalues from the covariance matrix (we can also use correlation matrix or even Single value decomposition, however in this post will focus on covariance matrix).
4. Sort eigenvalues in descending order and choose the top k Eigenvectors that correspond to the k largest eigenvalues (k will become the number of dimensions of the new feature subspace k≤d, d is the number of original dimensions).
5. Construct the projection matrix W from the selected k Eigenvectors.

6. Transform the original data set X via W to obtain the new k-dimensional feature subspace Y.

Let’s import some of the required libraries and also the Iris data set which I will use to explain each of the points in details.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from sklearn.preprocessing import standardScaler
%matplotlib inline

df = pd.read_csv(
filepath_or_buffer='https://archive.ics.uci.edu/ml/machine-learning-
databases/iris/iris.data', header=None, sep=',') df.columns=
['sepal_len', 'sepal_wid', 'petal_len', 'petal_wid', 'class']
print(df.isnull().values.any()) df.dropna(how="all", inplace=True) #
drops the empty line at file-end #if inplace = False then we have to
assign back to dataframe as it is a copy #df =
df.some_operation(inplace=False) #No need to assign back to
dataframe when inplace = True #df.some_operation(inplace=True)
#Print Last five rows. df.tail()
```

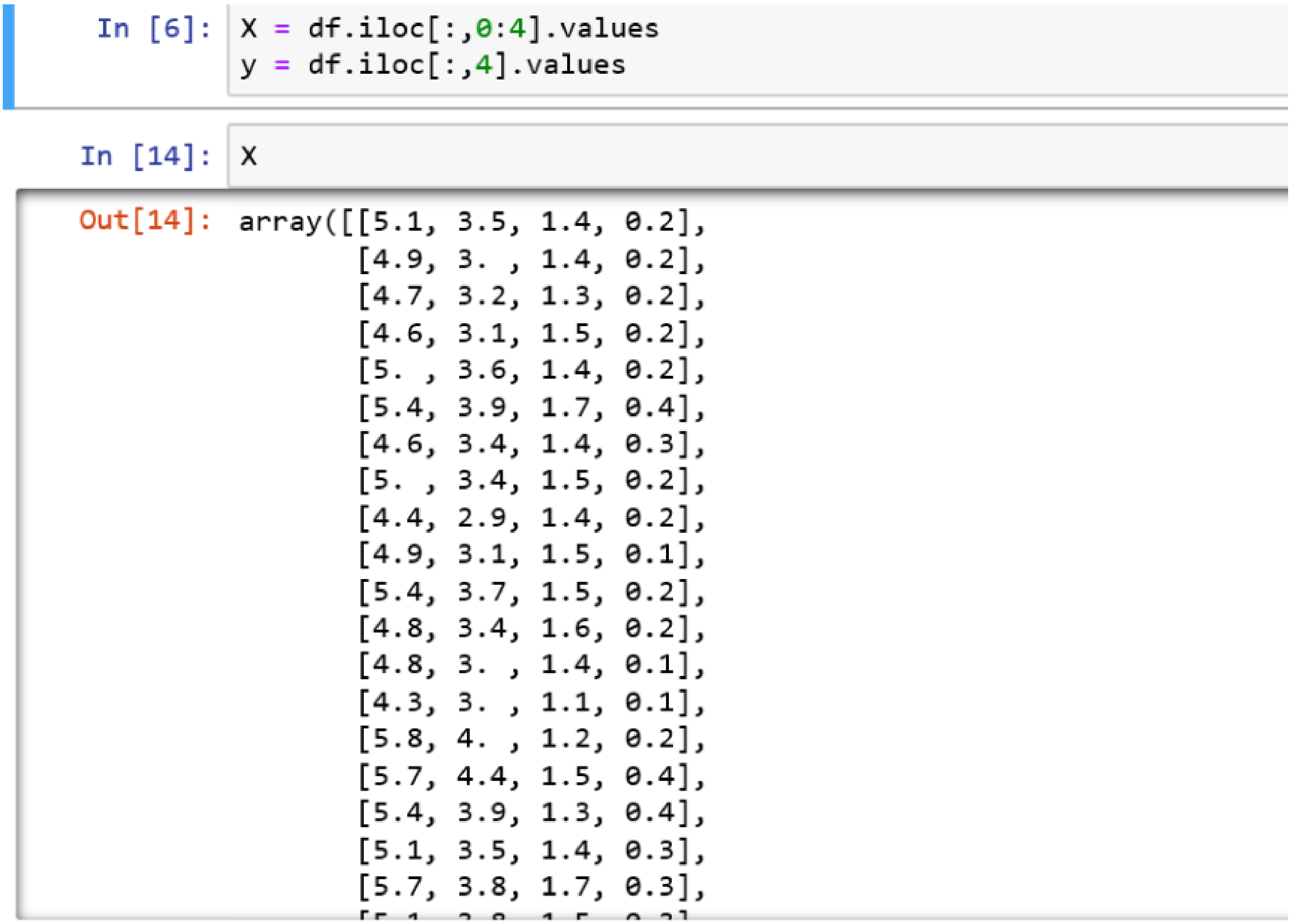


```
False
Out[159]:
```

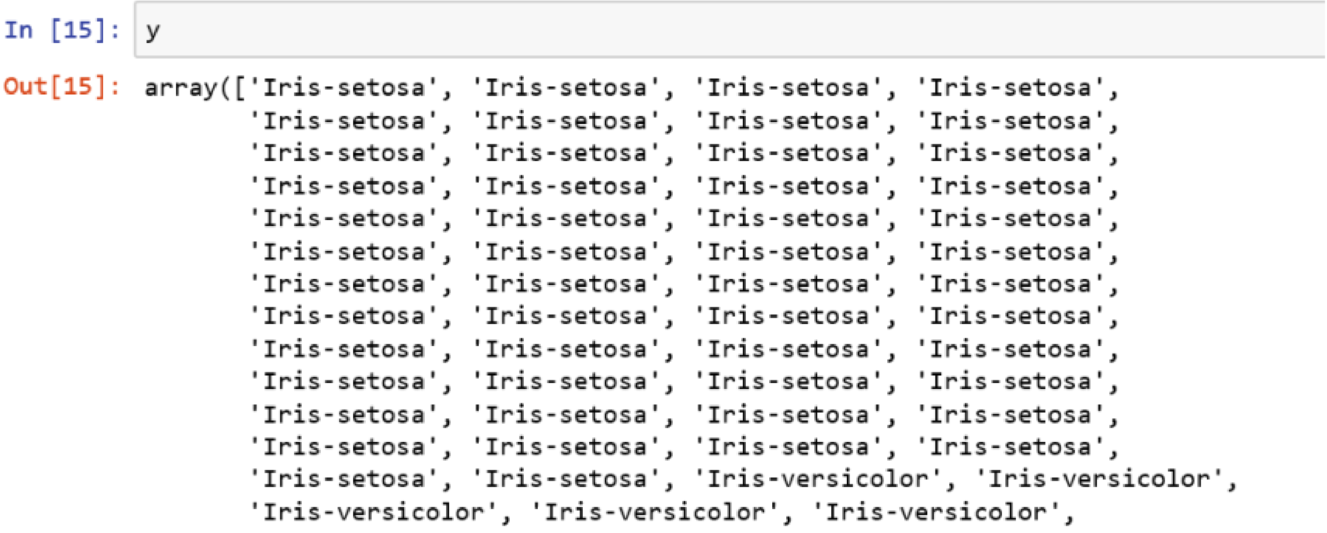
	sepal_len	sepal_wid	petal_len	petal_wid	class
145	6.7	3.0	5.2	2.3	Iris-virginica
146	6.3	2.5	5.0	1.9	Iris-virginica
147	6.5	3.0	5.2	2.0	Iris-virginica
148	6.2	3.4	5.4	2.3	Iris-virginica
149	5.9	3.0	5.1	1.8	Iris-virginica

Separate the Target column that is the class column values in y array and rest of the values of the independent features in X array variables as below.

```
X = df.iloc[:,0:4].values
y = df.iloc[:,4].values
```



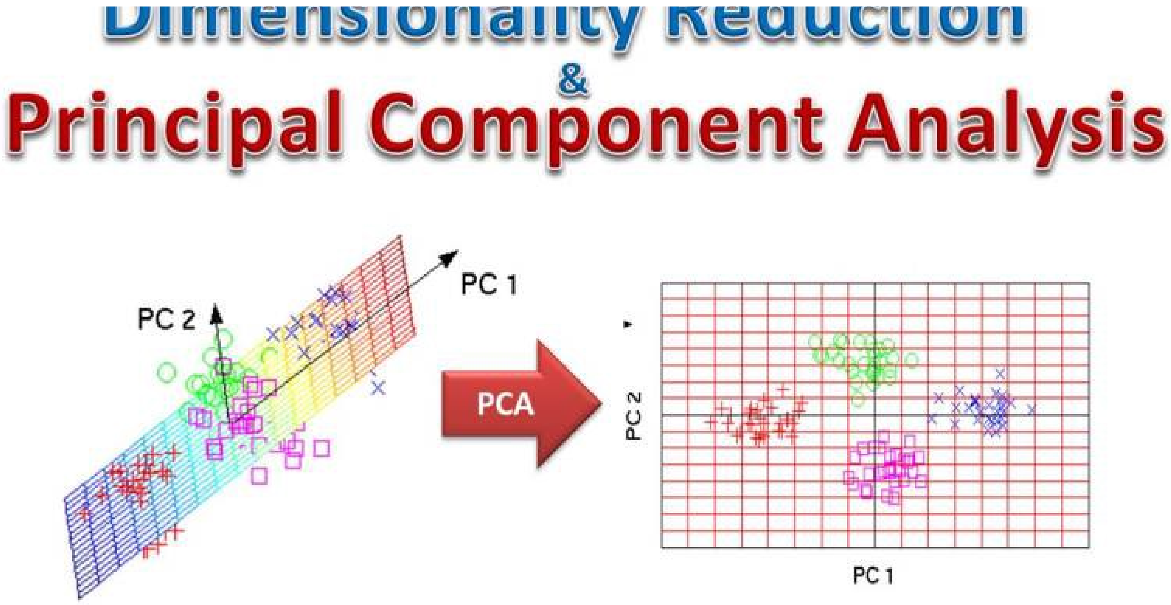
```
In [6]:  X = df.iloc[:,0:4].values
         y = df.iloc[:,4].values

In [14]: X

Out[14]: array([[5.1, 3.5, 1.4, 0.2],
                [4.9, 3. , 1.4, 0.2],
                [4.7, 3.2, 1.3, 0.2],
                [4.6, 3.1, 1.5, 0.2],
                [5. , 3.6, 1.4, 0.2],
                [5.4, 3.9, 1.7, 0.4],
                [4.6, 3.4, 1.4, 0.3],
                [5. , 3.4, 1.5, 0.2],
                [4.4, 2.9, 1.4, 0.2],
                [4.9, 3.1, 1.5, 0.1],
                [5.4, 3.7, 1.5, 0.2],
                [4.8, 3.4, 1.6, 0.2],
                [4.8, 3. , 1.4, 0.1],
                [4.3, 3. , 1.1, 0.1],
                [5.8, 4. , 1.2, 0.2],
                [5.7, 4.4, 1.5, 0.4],
                [5.4, 3.9, 1.3, 0.4],
                [5.1, 3.5, 1.4, 0.3],
                [5.7, 3.8, 1.7, 0.3],
```



```
In [15]: y

Out[15]: array(['Iris-setosa', 'Iris-setosa', 'Iris-setosa', 'Iris-setosa',
                'Iris-setosa', 'Iris-setosa', 'Iris-setosa', 'Iris-setosa',
                'Iris-setosa', 'Iris-setosa', 'Iris-setosa', 'Iris-setosa',
                'Iris-setosa', 'Iris-setosa', 'Iris-setosa', 'Iris-setosa',
                'Iris-setosa', 'Iris-setosa', 'Iris-setosa', 'Iris-setosa',
                'Iris-setosa', 'Iris-setosa', 'Iris-setosa', 'Iris-setosa',
                'Iris-setosa', 'Iris-setosa', 'Iris-setosa', 'Iris-setosa',
                'Iris-setosa', 'Iris-setosa', 'Iris-setosa', 'Iris-setosa',
                'Iris-setosa', 'Iris-setosa', 'Iris-setosa', 'Iris-setosa',
                'Iris-setosa', 'Iris-setosa', 'Iris-setosa', 'Iris-setosa',
                'Iris-setosa', 'Iris-setosa', 'Iris-setosa', 'Iris-setosa',
                'Iris-setosa', 'Iris-setosa', 'Iris-setosa', 'Iris-setosa',
                'Iris-setosa', 'Iris-setosa', 'Iris-versicolor', 'Iris-versicolor',
                'Iris-versicolor', 'Iris-versicolor', 'Iris-versicolor',
```

Iris data set is now stored in the form of a 150×4 matrix where the columns are the different features, and every row represents a separate flower sample. Each sample row x can be pictured as a 4-dimensional vector as we can see in the above screenshot of x output values.

Now let's understand each of the points in detail.



1. Standardization

When there are different scales used for the measurement of the values of the features, then it is advisable to do the standardization to bring all the feature spaces with mean = 0 and variance = 1.

The reason why standardization is very much needed before performing PCA is that PCA is very sensitive to variances. Meaning, if there are large differences between the scales (ranges) of the features, then those with larger scales will dominate over those with the small scales.

For example, a feature that ranges between 0 to 100 will dominate over a feature that ranges between 0 to 1 and it will lead to biased results. So, transforming the data to the same scales will prevent this problem. That is where we use standardization to bring the features with mean value 0 and variance 1.

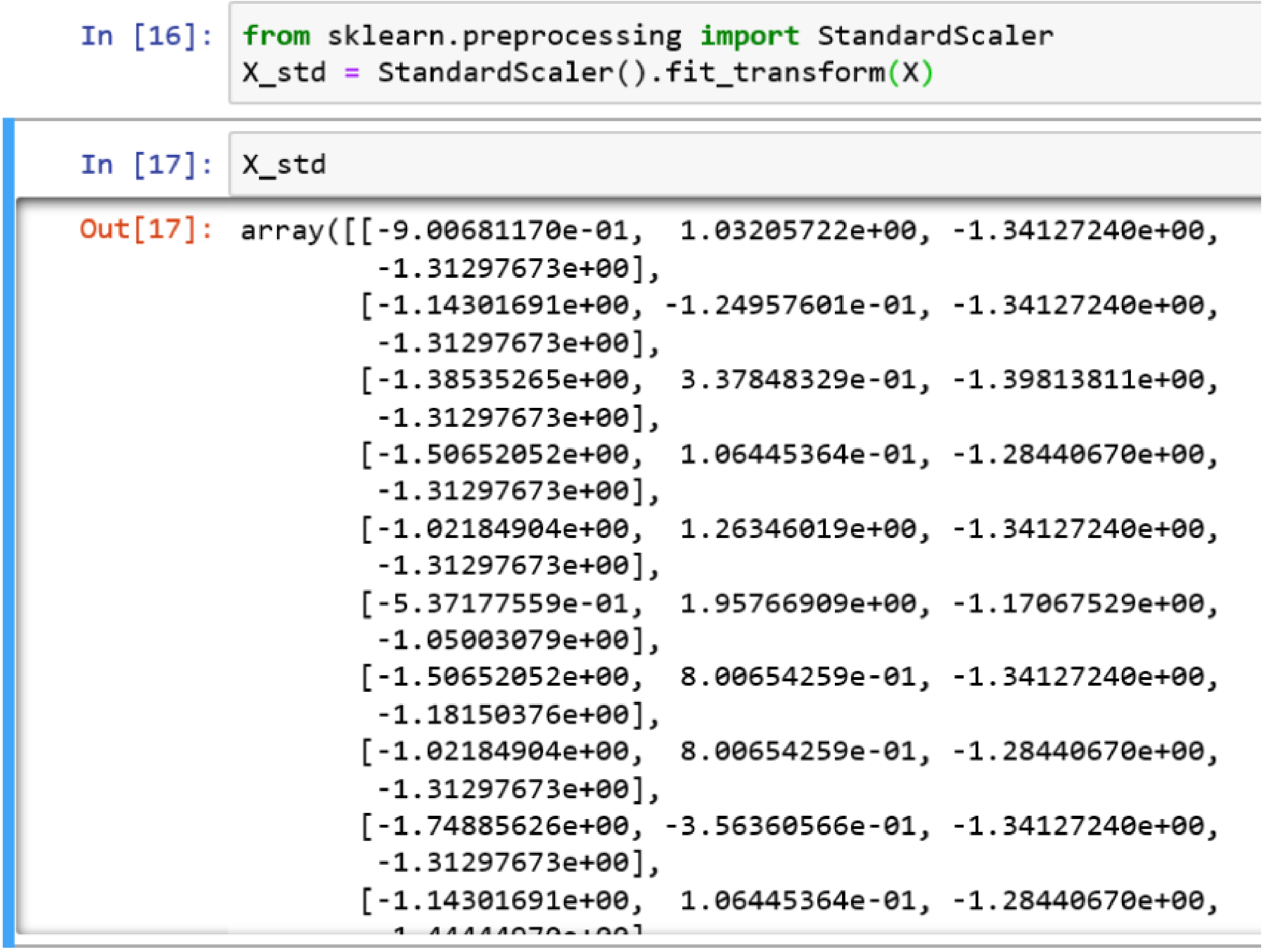
So here is the formula to calculate the standardized value of features:

$$\textit{Standardized value of } x_i = \frac{x_i - \textit{mean of } x}{\textit{std Deviation of } x}$$

In this article, I am using the Iris data set. Although all features in the Iris data set are measured in centimetres, Still I will continue with the transformation of the data onto the unit scale (mean=0 and variance=1), which is a requirement for the optimal performance of many machine learning algorithms. Also, it will help us to understand how this process works.

```
from sklearn.preprocessing import StandardScaler
X_std = StandardScaler().fit_transform(X)
```

In the output screenshot below you see that all X_std values are standardized in the range of -1 to +1.



```
In [16]: from sklearn.preprocessing import StandardScaler
         X_std = StandardScaler().fit_transform(X)

In [17]: X_std

Out[17]: array([[-9.00681170e-01,  1.03205722e+00, -1.34127240e+00,
                 -1.31297673e+00],
                [-1.14301691e+00, -1.24957601e-01, -1.34127240e+00,
                 -1.31297673e+00],
                [-1.38535265e+00,  3.37848329e-01, -1.39813811e+00,
                 -1.31297673e+00],
                [-1.50652052e+00,  1.06445364e-01, -1.28440670e+00,
                 -1.31297673e+00],
                [-1.02184904e+00,  1.26346019e+00, -1.34127240e+00,
                 -1.31297673e+00],
                [-5.37177559e-01,  1.95766909e+00, -1.17067529e+00,
                 -1.05003079e+00],
                [-1.50652052e+00,  8.00654259e-01, -1.34127240e+00,
                 -1.18150376e+00],
                [-1.02184904e+00,  8.00654259e-01, -1.28440670e+00,
                 -1.31297673e+00],
                [-1.74885626e+00, -3.56360566e-01, -1.34127240e+00,
                 -1.31297673e+00],
                [-1.14301691e+00,  1.06445364e-01, -1.28440670e+00,
                 -1.44441979e+00]
```

2. Eigen decomposition — Computing Eigenvectors and Eigenvalues

The eigenvectors and eigenvalues of a covariance (or correlation) matrix represent the “core” of a PCA:

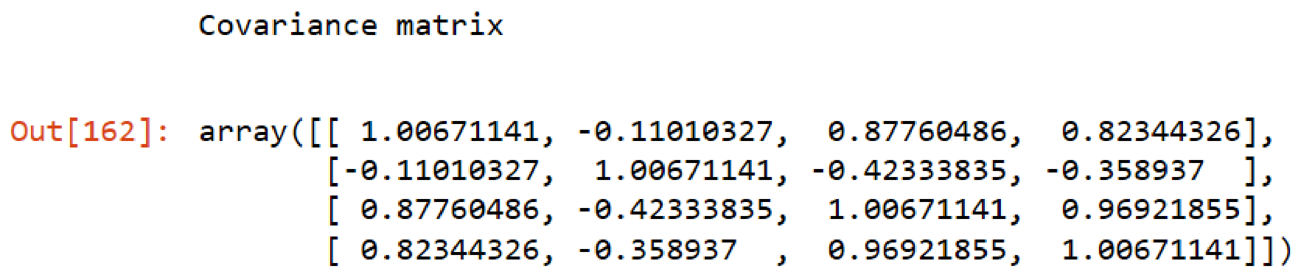
- The Eigenvectors (principal components) determine the directions of the new feature space, and the eigenvalues determine their magnitude.
- In other words, the eigenvalues explain the variance of the data along the new feature axes. It means the corresponding eigenvalue tells us that how much variance is included in that new transformed feature.

- To get eigenvalues and Eigenvectors we need to compute the covariance matrix. So in the next step let's compute it.

2.1 Covariance Matrix

The classic approach to PCA is to perform the Eigen decomposition on the covariance matrix Σ, which is a d×d matrix where each element represents the covariance between two features. Note, d is the number of original dimensions of the data set. In Iris data set we have 4 features hence covariance matrix will be of order 4×4.

```
#mean_vec = np.mean(X_std, axis=0)
#cov_mat = (X_std - mean_vec).T.dot((X_std - mean_vec)) /
(X_std.shape[0]-1)
#print('Covariance matrix \n%s' %cov_mat)
print('Covariance matrix \n')
cov_mat= np.cov(X_std, rowvar=False)
cov_mat
```



```
           Covariance matrix

Out[162]: array([[ 1.00671141, -0.11010327,  0.87760486,  0.82344326],
                 [-0.11010327,  1.00671141, -0.42333835, -0.358937  ],
                 [ 0.87760486, -0.42333835,  1.00671141,  0.96921855],
                 [ 0.82344326, -0.358937  ,  0.96921855,  1.00671141]])
```

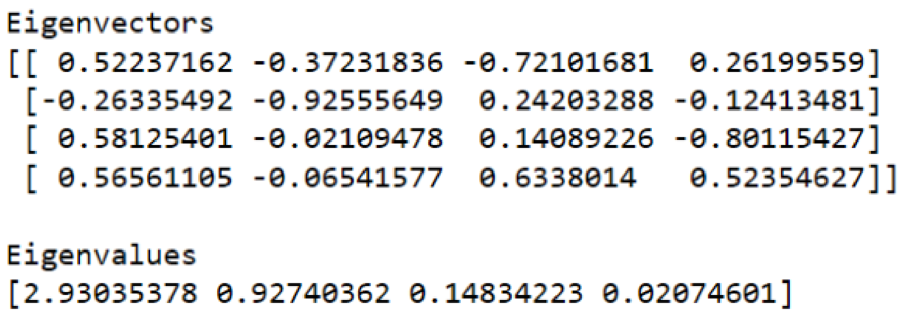
2.2 Eigenvectors and Eigenvalues computation from the covariance matrix

Here if we know concepts of Linear Algebra and how to calculate Eigenvectors and Eigenvalues of the matrix then this is going to be very helpful in understanding the below concepts. So it would be advisable to go through some of the basic concepts of Linear Algebra to have a deeper understanding of how everything works.

Here I am using numpy array to calculate Eigenvectors and Eigenvalues of the standardized feature space values as following:

```
cov_mat = np.cov(X_std.T)
eig_vals, eig_vecs = np.linalg.eig(cov_mat)
```

```
print('Eigenvectors \n%s' %eig_vecs)
print('\nEigenvalues \n%s' %eig_vals)
```

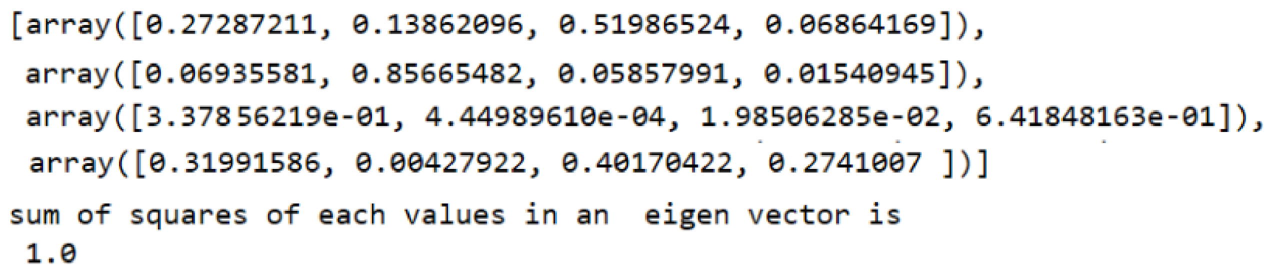


```
Eigenvectors
[[ 0.52237162 -0.37231836 -0.72101681  0.26199559]
 [-0.26335492 -0.92555649  0.24203288 -0.12413481]
 [ 0.58125401 -0.02109478  0.14089226 -0.80115427]
 [ 0.56561105 -0.06541577  0.6338014   0.52354627]]

Eigenvalues
[2.93035378 0.92740362 0.14834223 0.02074601]
```

2.3 Eigen Vectors verification

As we know that the sum of the square of each value in an Eigenvector is 1. So let’s see if it holds which mean we have computed Eigenvectors correctly.

```
sq_eig=[] for i in eig_vecs: sq_eig.append(i**2)
    print(sq_eig)
sum(sq_eig)
print("sum of squares of each values in an eigen vector is \n",
0.27287211+ 0.13862096+0.51986524+ 0.06864169)
for ev in eig_vecs: np.testing.assert_array_almost_equal(1.0,
np.linalg.norm(ev))
```



```
[array([0.27287211, 0.13862096, 0.51986524, 0.06864169]),
 array([0.06935581, 0.85665482, 0.05857991, 0.01540945]),
 array([3.37856219e-01, 4.44989610e-04, 1.98506285e-02, 6.41848163e-01]),
 array([0.31991586, 0.00427922, 0.40170422, 0.2741007 ])]
sum of squares of each values in an  eigen vector is
 1.0
```

3. Selecting The Principal Components

- 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.

- However, the eigenvectors only define the directions of the new axis, since they have all the same unit length 1.

So now the question comes that how to select the new set of Principal components. The rule behind is that we sort the Eigenvalues in descending order and then choose the top k features concerning top k Eigenvalues.

The idea here is that by choosing top k we have decided that the variance which corresponds to those k feature space is enough to describe the data set. And by losing the remaining variance of those not selected features, won't cost the accuracy much or we are OK to loose that much accuracy that costs because of neglected variance.

So this is the decision which we have to make based on the problem set given and also based on the business case. There is no perfect rule to decide it.

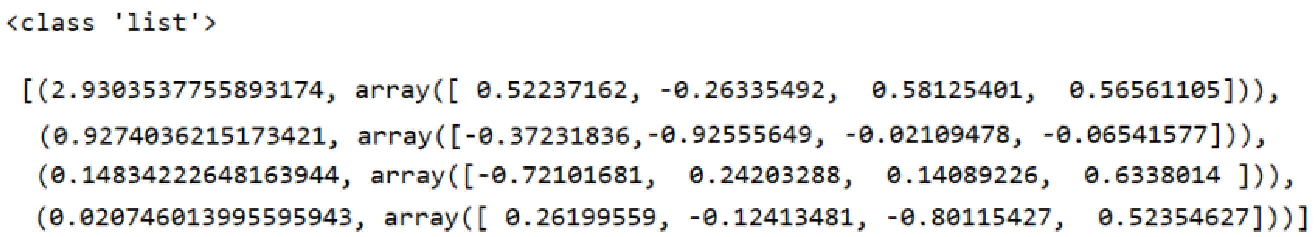
Now let's find out the Principal components using the following steps:

3.1 Sorting eigenvalues

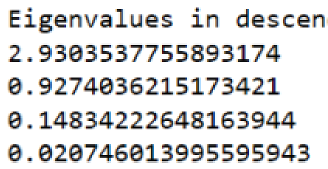
To decide which Eigenvector(s) can be dropped without losing too much information for the construction of lower-dimensional subspace, we need to inspect the corresponding eigenvalues:

- The Eigenvectors with the lowest eigenvalues bear the least information about the distribution of the data; those are the ones can be dropped.
- To do so, the common approach is to rank the eigenvalues from highest to lowest to choose the top k Eigenvectors.

```
#Make a list of (eigenvalue, eigenvector) tuples
eig_pairs = [(np.abs(eig_vals[i]), eig_vecs[:,i])
for i in range(len(eig_vals))]
    print(type(eig_pairs))
#Sort the (eigenvalue, eigenvector) tuples from high to low
eig_pairs.sort()
eig_pairs.reverse()
print("\n",eig_pairs)
#Visually confirm that the list is correctly sorted by decreasing
eigenvalues
print('\n\n\nEigenvalues in descending order:')
for i in eig_pairs:
    print(i[0])
```



```
<class 'list'>

 [(2.9303537755893174, array([ 0.52237162, -0.26335492,  0.58125401,  0.56561105])),
  (0.9274036215173421, array([-0.37231836,-0.92555649, -0.02109478, -0.06541577])),
  (0.14834222648163944, array([-0.72101681,  0.24203288,  0.14089226,  0.6338014 ])),
  (0.020746013995595943, array([ 0.26199559, -0.12413481, -0.80115427,  0.52354627]))]
```

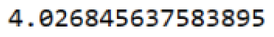


```
Eigenvalues in descending order:
2.9303537755893174
0.9274036215173421
0.14834222648163944
0.020746013995595943
```

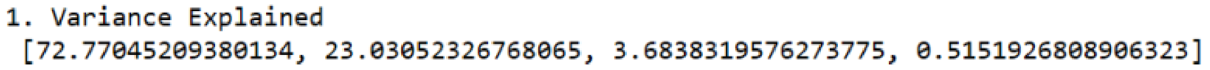
3.2 Explained Variance

- After sorting the eigenpairs, the next question is “how many principal components are we going to choose for our new feature subspace?”
- A useful measure is the so-called “explained variance,” which can be calculated from the eigenvalues.
- The explained variance tells us how much information (variance) can be attributed to each of the principal components.

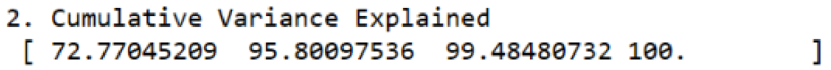
```
tot = sum(eig_vals)
print("\n",tot)
var_exp = [(i / tot)*100 for i in sorted(eig_vals, reverse=True)]
print("\n\n1. Variance Explained\n",var_exp)
cum_var_exp = np.cumsum(var_exp)
print("\n\n2. Cumulative Variance Explained\n",cum_var_exp)
print("\n\n3. Percentage of variance the first two principal
components each contain\n ",var_exp[0:2])
print("\n\n4. Percentage of variance the first two principal
components together contain\n",sum(var_exp[0:2]))
```



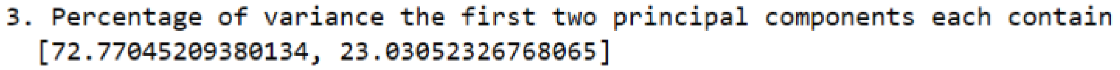
```
 4.026845637583895
```



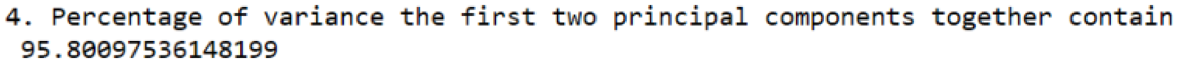
```
1. Variance Explained
 [72.77045209380134, 23.03052326768065, 3.6838319576273775, 0.5151926808906323]
```



```
2. Cumulative Variance Explained
 [ 72.77045209  95.80097536  99.48480732 100.        ]
```



```
3. Percentage of variance the first two principal components each contain
  [72.77045209380134, 23.03052326768065]
```



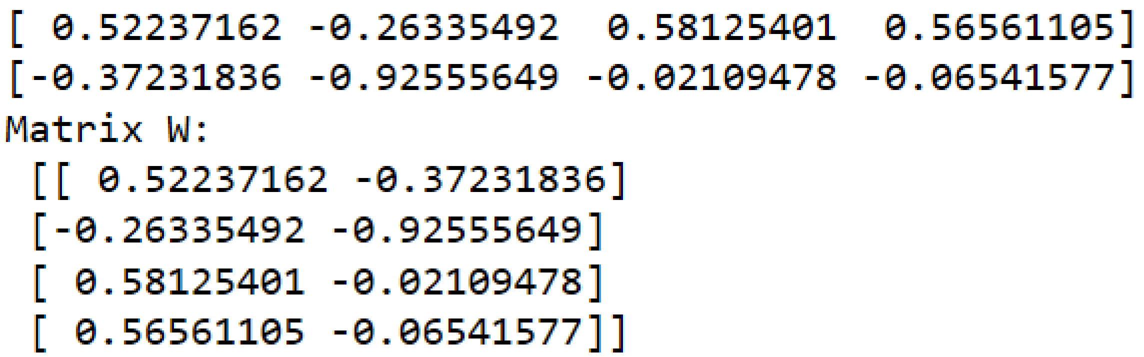
```
4. Percentage of variance the first two principal components together contain
 95.80097536148199
```

4. Construct the projection matrix W from the selected k eigenvectors

- Projection matrix will be used to transform the Iris data onto the new feature subspace or we say newly transformed data set with reduced dimensions.
- It is a matrix of our concatenated top k Eigenvectors.

Here, we are reducing the 4-dimensional feature space to a 2-dimensional feature subspace, by choosing the "top 2" Eigenvectors with the highest Eigenvalues to construct our d×k-dimensional Eigenvector matrix W.

```
print(eig_pairs[0][1])
print(eig_pairs[1][1])
matrix_w = np.hstack((eig_pairs[0][1].reshape(4,1), eig_pairs[1]
[1].reshape(4,1)))
#hstack: Stacks arrays in sequence horizontally (column wise).
print('Matrix W:\n', matrix_w)
```

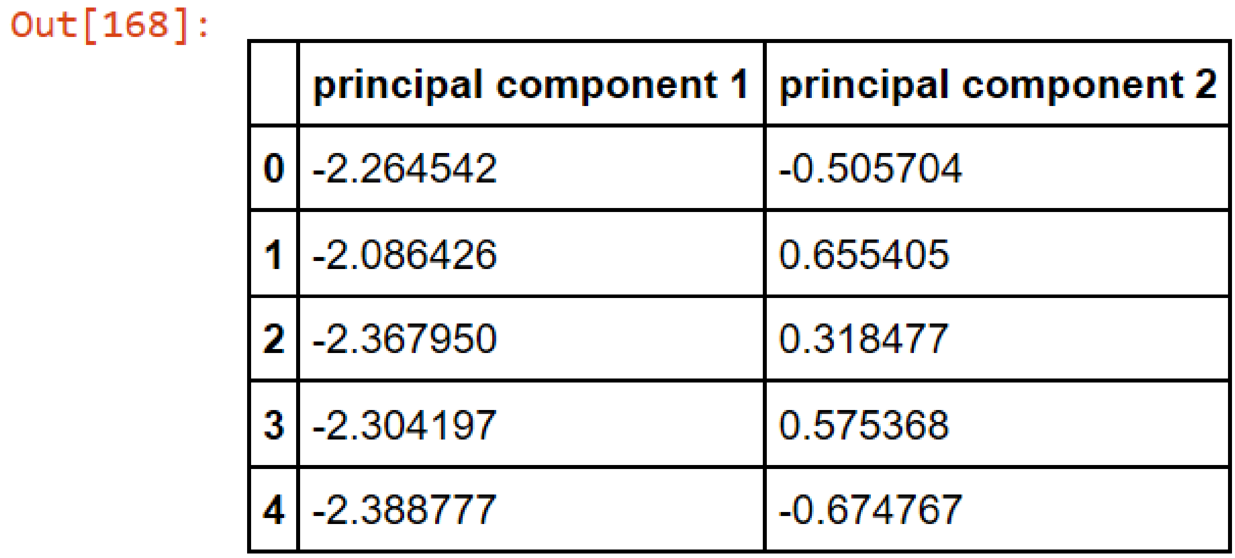


```
[ 0.52237162 -0.26335492  0.58125401  0.56561105]
[-0.37231836 -0.92555649 -0.02109478 -0.06541577]
Matrix W:
 [[ 0.52237162 -0.37231836]
 [-0.26335492 -0.92555649]
 [ 0.58125401 -0.02109478]
 [ 0.56561105 -0.06541577]]
```

5. Projection Onto the New Feature Space

In this last step, we will use the 4×2-dimensional projection matrix W to transform our samples onto the new subspace via the equation Y=X×W, where the output matrix Y will be a 150×2 matrix of our transformed samples.

```
Y = X_std.dot(matrix_w)
principalDf = pd.DataFrame(data = Y , columns = ['principal
component 1', 'principal component 2'])
principalDf.head()
```

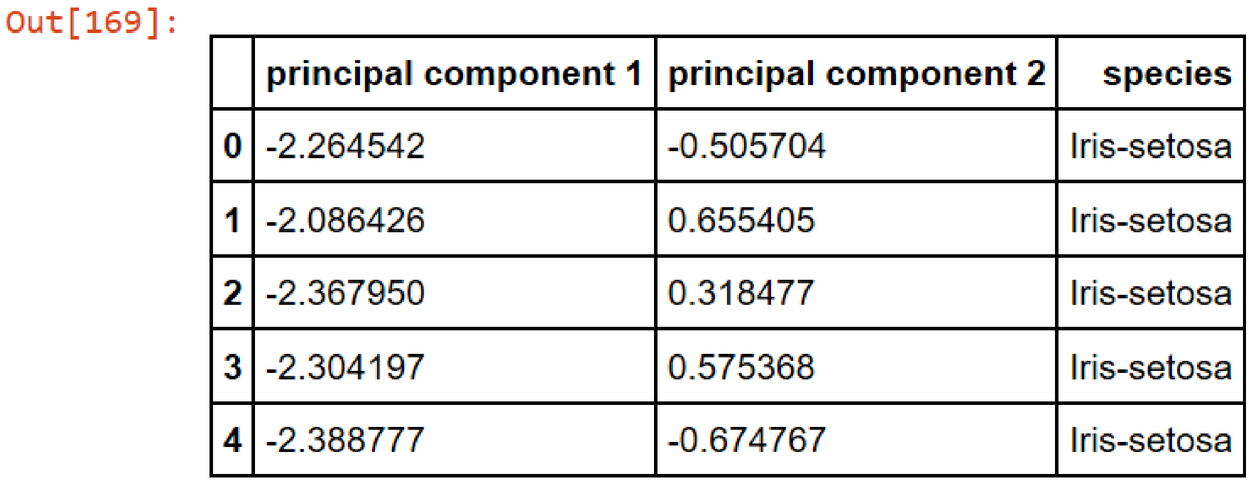


Out[168]:

	principal component 1	principal component 2
0	-2.264542	-0.505704
1	-2.086426	0.655405
2	-2.367950	0.318477
3	-2.304197	0.575368
4	-2.388777	-0.674767

Now let's combine the target class variable which we separated in the very beginning of the post.

```
finalDf = pd.concat([principalDf,pd.DataFrame(y,columns =
['species'])], axis = 1)
finalDf.head()
```



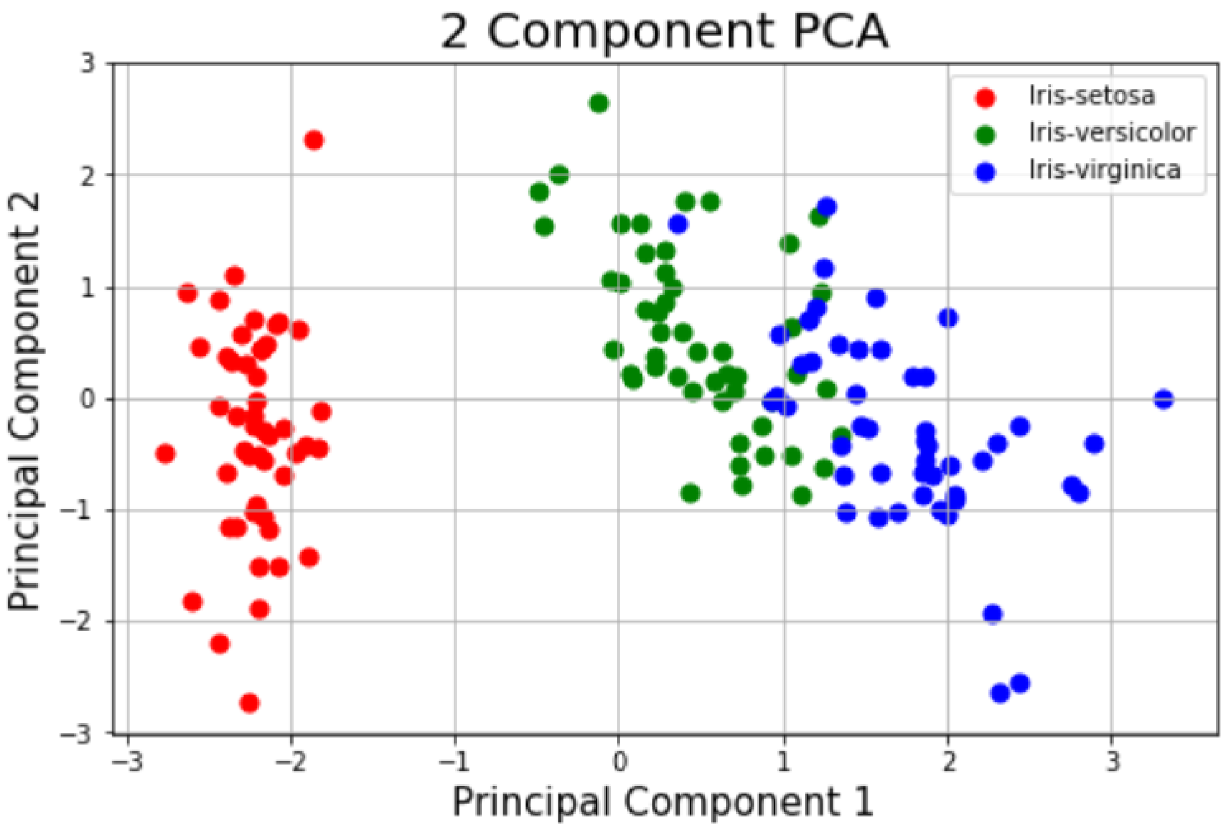
Out[169]:

	principal component 1	principal component 2	species
0	-2.264542	-0.505704	Iris-setosa
1	-2.086426	0.655405	Iris-setosa
2	-2.367950	0.318477	Iris-setosa
3	-2.304197	0.575368	Iris-setosa
4	-2.388777	-0.674767	Iris-setosa

Visualize 2D Projection

Use a PCA projection to 2d to visualize the entire data set. You should plot different classes using different colours or shapes. Classes should be well-separated from each other.

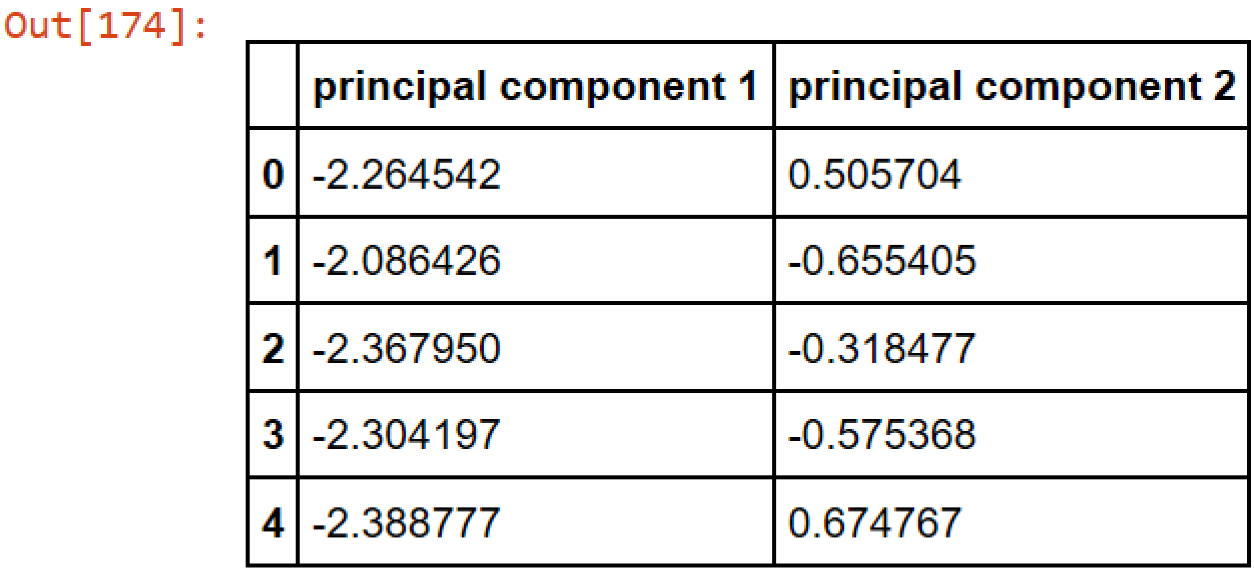
```
fig = plt.figure(figsize = (8,5))
ax = fig.add_subplot(1,1,1)
ax.set_xlabel('Principal Component 1', fontsize = 15)
ax.set_ylabel('Principal Component 2', fontsize = 15)
ax.set_title('2 Component PCA', fontsize = 20)
targets = ['Iris-setosa', 'Iris-versicolor', 'Iris-virginica']
colors = ['r', 'g', 'b']
for target, color in zip(targets,colors):
    indicesToKeep = finalDf['species'] == target
ax.scatter(finalDf.loc[indicesToKeep, 'principal component 1'] ,
finalDf.loc[indicesToKeep, 'principal component 2'] , c = color , s
= 50) ax.legend(targets) ax.grid()
```



Use of Python Libraries to directly compute Principal Components

Alternatively, there are direct libraries in python which compute the principal components directly and no need to do all the above computations. The above-mentioned steps were to give you an understanding of how everything works.

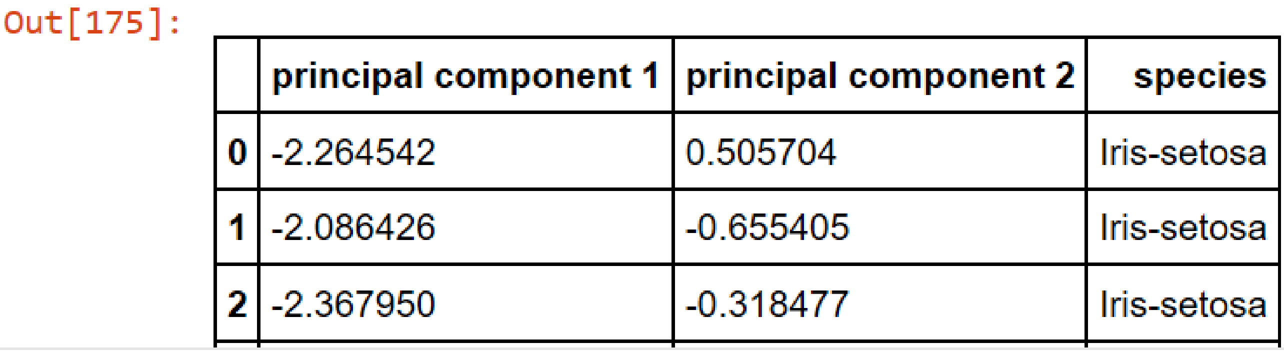
```
pca = PCA(n_components=2)
# Here we can also give the percentage as a paramter to the PCA
function as pca = PCA(.95). .95 means that we want to include 95% of
the variance. Hence PCA will return the no of components which
describe 95% of the variance. However we know from above computation
that 2 components are enough so we have passed the 2 components.

principalComponents = pca.fit_transform(X_std)
principalDf = pd.DataFrame(data = principalComponents , columns =
['principal component 1', 'principal component 2'])

principalDf.head(5) # prints the top 5 rows
```



Out[174]:

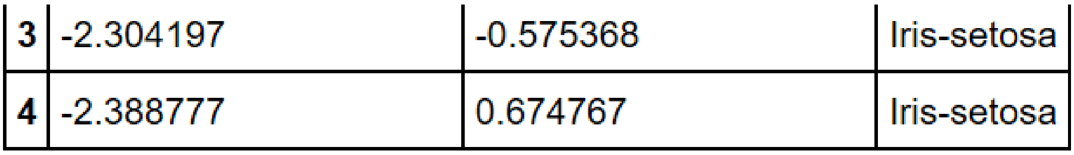
	principal component 1	principal component 2
0	-2.264542	0.505704
1	-2.086426	-0.655405
2	-2.367950	-0.318477
3	-2.304197	-0.575368
4	-2.388777	0.674767

```
finalDf = pd.concat([principalDf, finalDf[['species']]], axis = 1)
finalDf.head(5)
```



Out[175]:

	principal component 1	principal component 2	species
0	-2.264542	0.505704	Iris-setosa
1	-2.086426	-0.655405	Iris-setosa
2	-2.367950	-0.318477	Iris-setosa



3	-2.304197	-0.575368	Iris-setosa
4	-2.388777	0.674767	Iris-setosa

Together, the first two principal components contain 95.80% of the information. The first principal component contains 72.77% of the variance and the second principal component contains 23.03% of the variance. The third and fourth principal component contained the rest of the variance of the data set.

So that’s all about Principal Component Analysis. In my next post, I will write about how PCA speed-up the Machine Learning Algorithms and do the comparison analysis with and without PCA.

Hope it has given you a good understanding of the concept behind PCA. Please share your thoughts/ideas using the comment section below. You may also contact me using the Contact Link.

Originally published at http://ashutoshtripathi.com on July 11, 2019.

Data Science Machine Learning Data Analysis