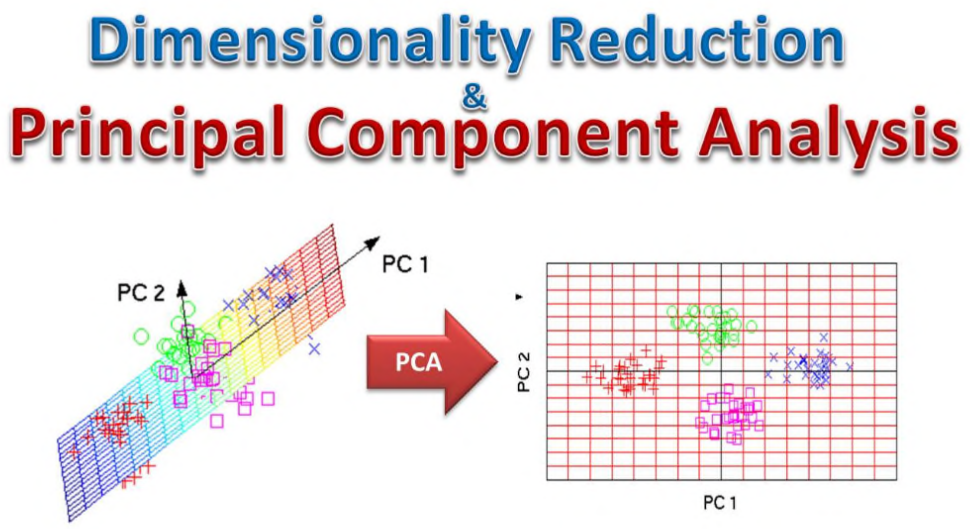
In [1]:

```python
from IPython.display import Image
Image(filename='pca_title.jpg')
```

Out[1]:



Steps Involved in PCA

1. Standardize the data.
2. Obtain the Eigenvectors and Eigenvalues from the covariance matrix.
3. Sort eigenvalues in descending order and choose the k eigenvectors.
4. Construct the projection matrix W from the selected k eigenvectors.
5. Transform the original dataset X via W to obtain a k-dimensional feature subspace Y.

In [50]:

```python
import warnings
warnings.filterwarnings('ignore')
```

In [48]:

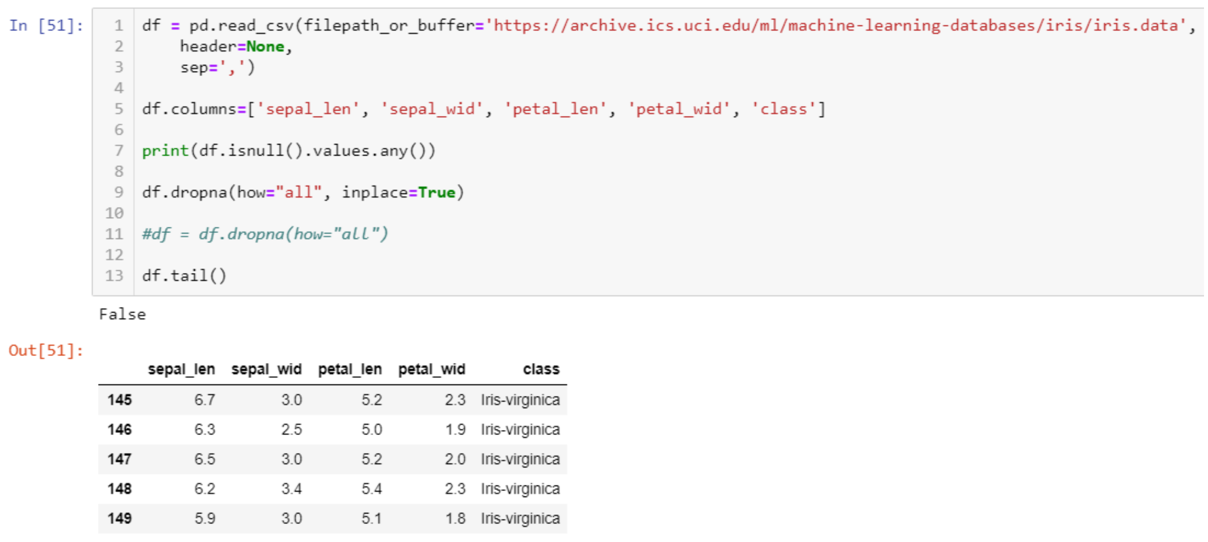
```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA

%matplotlib inline
```

We will use IRIS, Data Set. Data set contains 3 classes of 50 instances each, where each class refers to a type of iris plant.

In [49]:

```python
Image(filename='iris.png')
```

Out[49]:



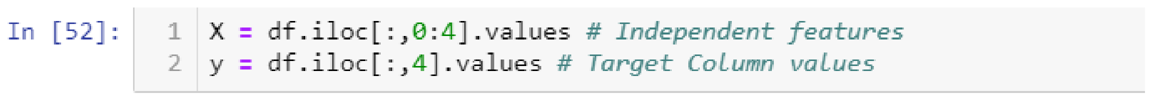
```
In [51]:
df = pd.read_csv(filepath_or_buffer='https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data',
    header=None,
    sep=',')

df.columns=['sepal_len', 'sepal_wid', 'petal_len', 'petal_wid', 'class']

print(df.isnull().values.any())

df.dropna(how="all", inplace=True)

#df = df.dropna(how="all")

df.tail()
```

```
False
```

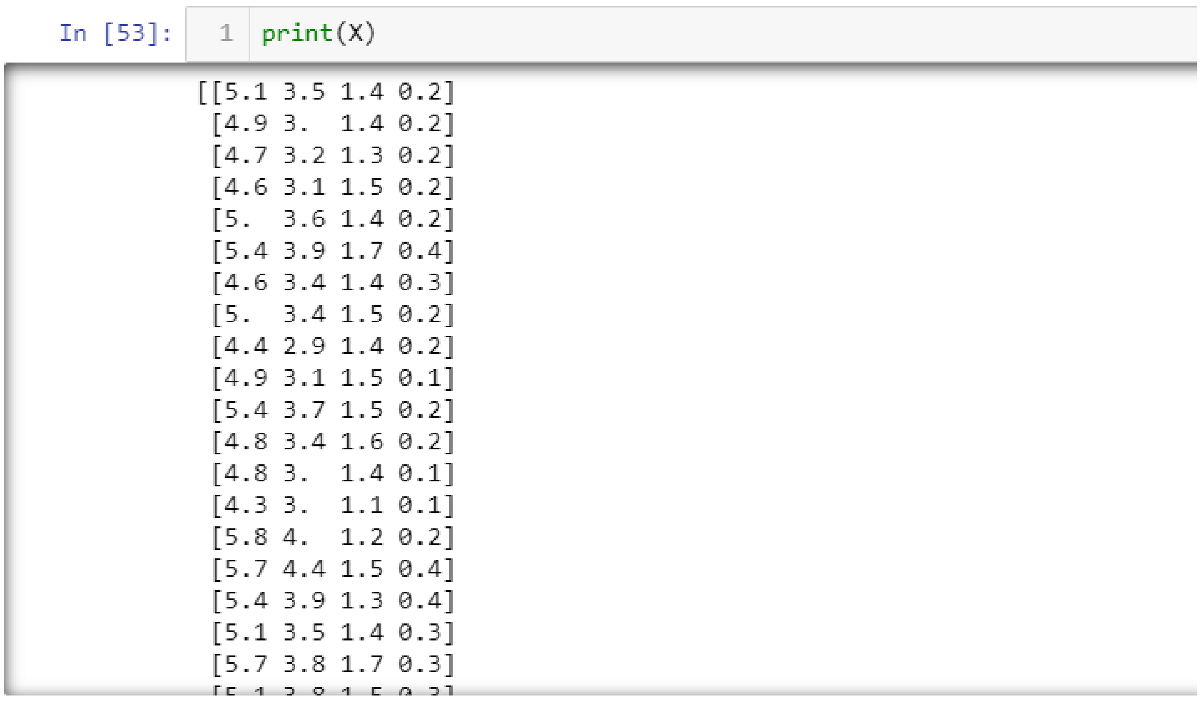
Out[51]:

	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.

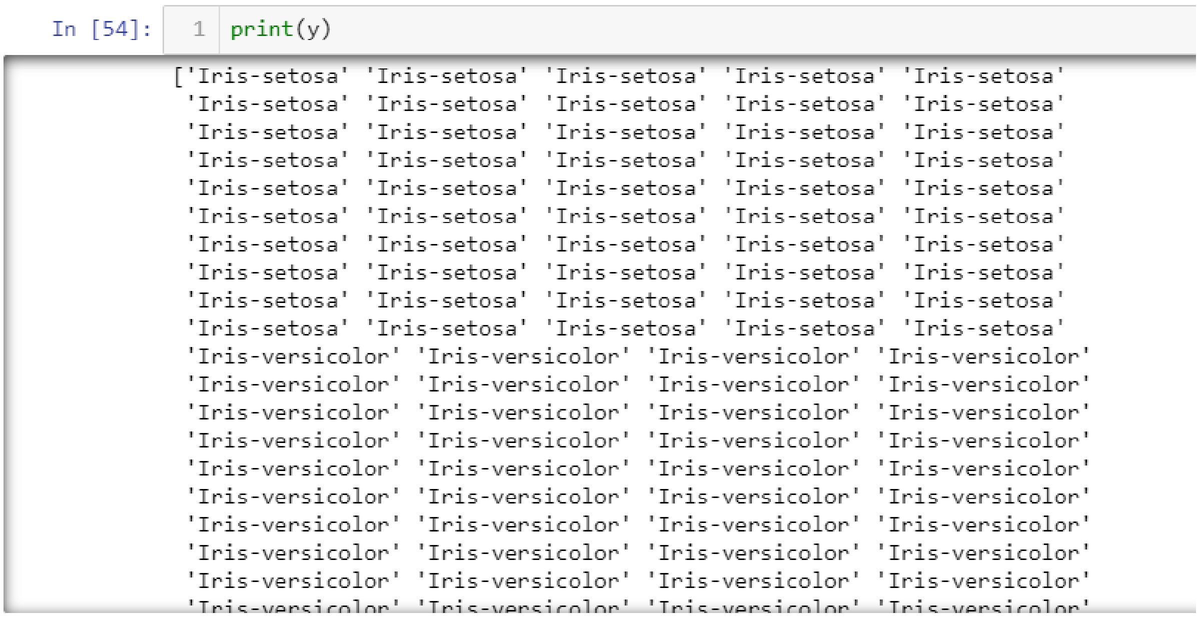


```
In [52]:
X = df.iloc[:,0:4].values # Independent features
y = df.iloc[:,4].values # Target Column values
```



```
In [53]:
print(X)
```

```
[[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 [54]:  print(y)
```

```
['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-versicolor' 'Iris-versicolor' 'Iris-versicolor'
 'Iris-versicolor' 'Iris-versicolor' 'Iris-versicolor' 'Iris-versicolor'
 'Iris-versicolor' 'Iris-versicolor' 'Iris-versicolor' 'Iris-versicolor'
 'Iris-versicolor' 'Iris-versicolor' 'Iris-versicolor' 'Iris-versicolor'
 'Iris-versicolor' 'Iris-versicolor' 'Iris-versicolor' 'Iris-versicolor'
 'Iris-versicolor' 'Iris-versicolor' 'Iris-versicolor' 'Iris-versicolor'
 'Iris-versicolor' 'Iris-versicolor' 'Iris-versicolor' 'Iris-versicolor'
 'Iris-versicolor' 'Iris-versicolor' 'Iris-versicolor' '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.

Now let’s understand each step of PCA in detail.

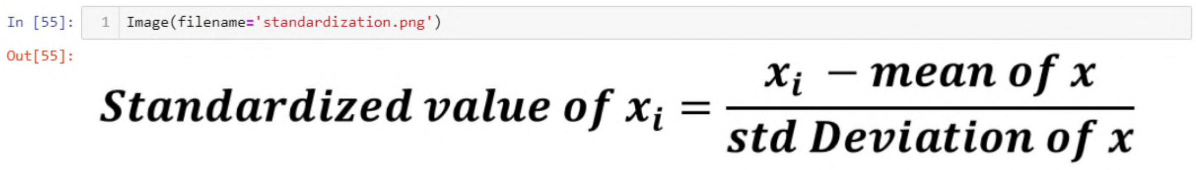
1. Standardization

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 0 to 100 will dominate over a feature that ranges between 0 to 50 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.

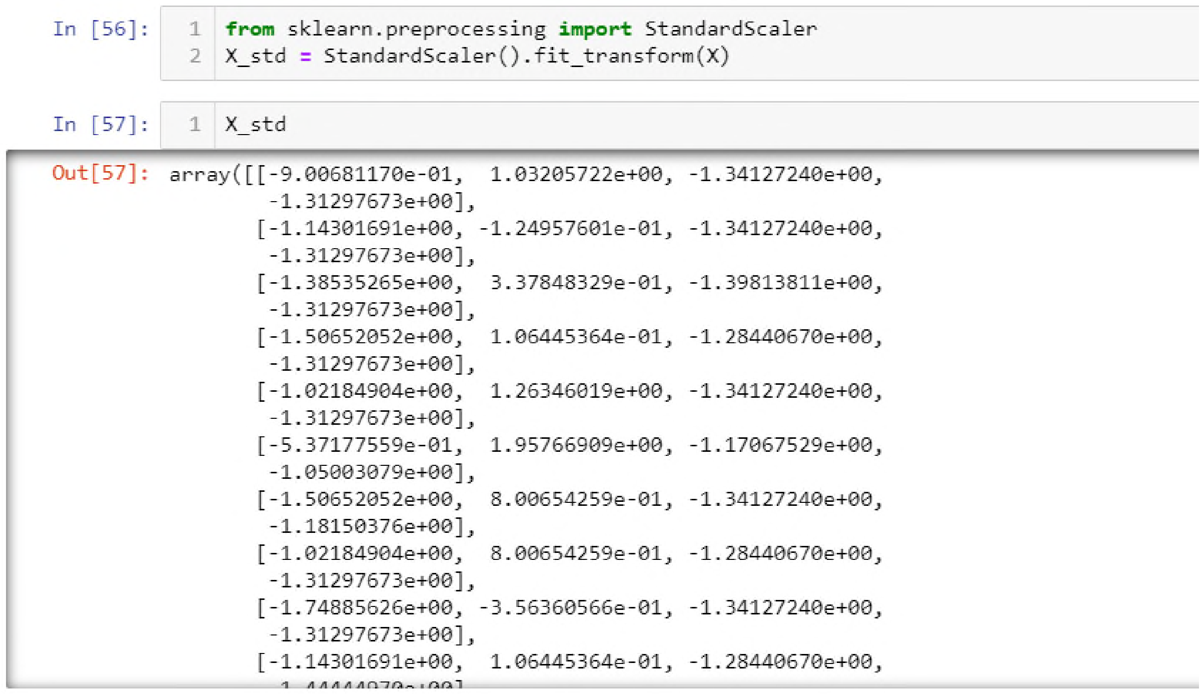
So here is the formula to calculate the standardized value of features:



```
In [55]:  Image(filename='standardization.png')
Out[55]:
```

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

In this article, I am using the Iris data set. Although all features in the Iris data set are measured in centimeters, 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.



```
In [56]:  from sklearn.preprocessing import StandardScaler
          X_std = StandardScaler().fit_transform(X)

In [57]:  X_std

Out[57]: 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,
```

Now all the above values (column wise) will have mean 0 and variance 1

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 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.

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.

```
In [58]:  Image(filename='CovarianceFormula.png')
```

Out[58]:

For Population

$$\text{Cov}(x,y) = \frac{\Sigma\,(x_i - \bar{x}) * (y_i - \bar{y})}{N}$$

For Sample

$$\text{Cov}(x,y) = \frac{\Sigma\,(x_i - \bar{x}) * (y_i - \bar{y})}{(N - 1)}$$

```
In [59]:  import numpy as np
          #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[59]:  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:

In [60]:
```python
cov_mat = np.cov(X_std.T) #Transpose the Covariance matrix

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]
```

3. Selecting the Principal Components

Sort the Eigen Values

Check for the explained variance in each Eigen value

Select the top k Eigen values which corresponds to the top k feature subspace.

3.1 Sorting Eigen values

In order 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.

In order to do so, the common approach is to rank the eigenvalues from highest to lowest in order to choose the top k Eigenvectors.

```
In [62]:
# 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'>

 [(2.930353775589315, array([ 0.52237162, -0.26335492,  0.58125401,  0.56561105])), (0.9274036215173407, array([-0.37231836, -
0.92555649, -0.02109478, -0.06541577])), (0.1483422264816398, array([-0.72101681,  0.24203288,  0.14089226,  0.6338014 ])), (0.
02074601399559622, array([ 0.26199559, -0.12413481, -0.80115427,  0.52354627]))]



Eigenvalues in descending order:
2.930353775589315
0.9274036215173407
0.1483422264816398
0.02074601399559622
```

3.2 Explained Variance

After sorting the Eigen pairs, 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.

```
In [63]:
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]))

```

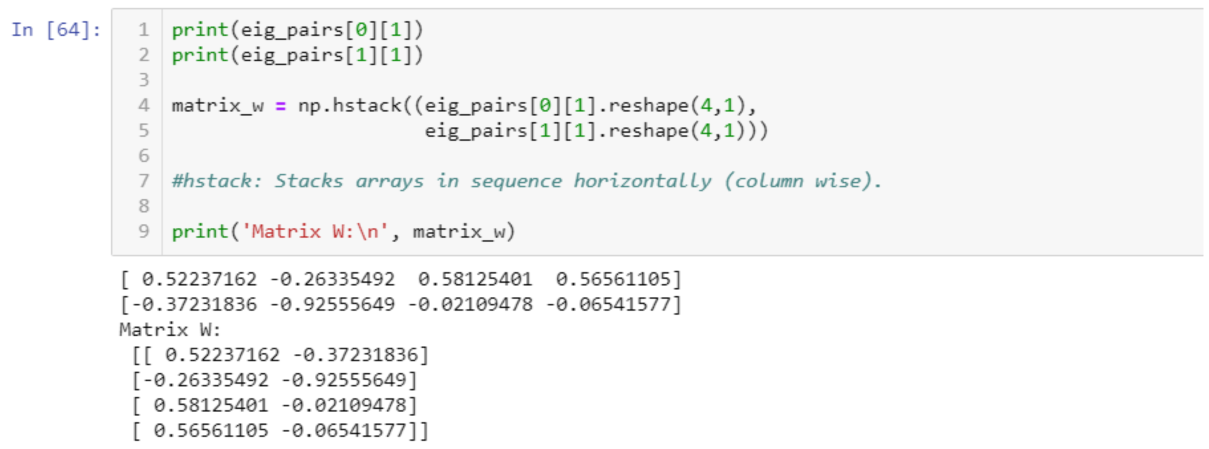
```
1. Variance Explained
 [72.77045209380134, 23.030523267680632, 3.683831957627389, 0.5151926808906395]


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


3. Percentage of variance the first two principal components each contain
  [72.77045209380134, 23.030523267680632]


4. Percentage of variance the first two principal components together contain
 95.80097536148197
```

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 new transformed data set with reduced dimensions.

It is 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.



```
In [64]:
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.

In [65]:
```python
Y = X_std.dot(matrix_w)

principalDf = pd.DataFrame(data = Y
             , columns = ['principal component 1', 'principal component 2'])

principalDf.head()
```

Out[65]:

	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.

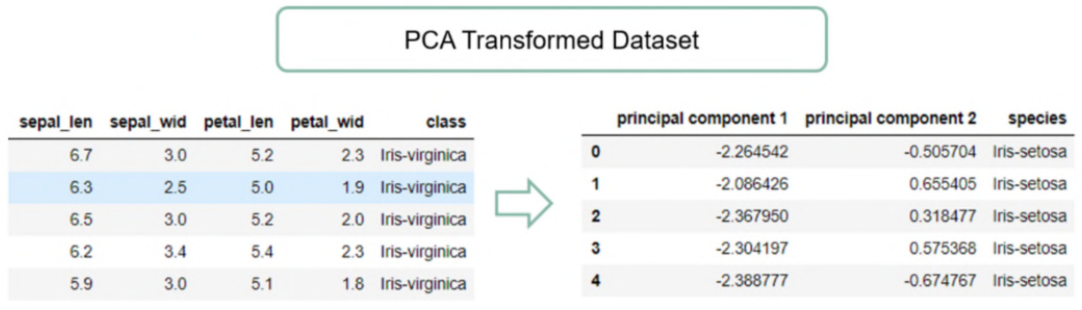
In [66]:
```python
finalDf = pd.concat([principalDf,pd.DataFrame(y,columns = ['species'])], axis = 1)
finalDf.head()
```

Out[66]:

	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

In [2]:
```python
Image(filename='PCA transformed Dataset.png')
```

Out[2]:



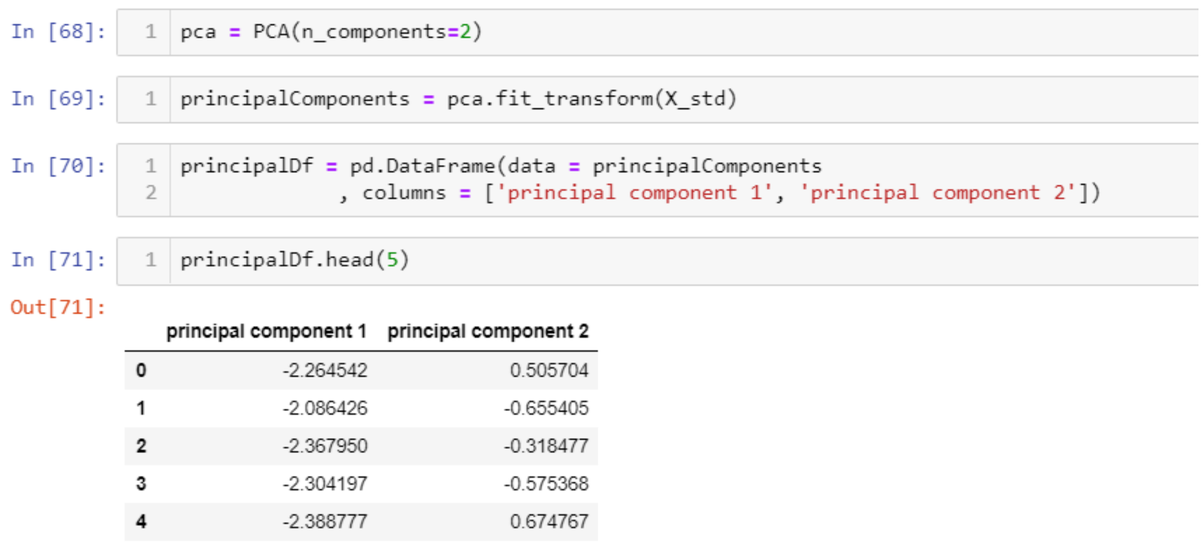
PCA Transformed Dataset

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

	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

Alternatively: Use of Python Libraries to directly compute Principal Components

There are direct libraries in python which computes the principal components directly and no need to do all the above computations. The above mentioned steps were to give you the understanding how everything works.



```
In [68]: pca = PCA(n_components=2)

In [69]: principalComponents = pca.fit_transform(X_std)

In [70]: principalDf = pd.DataFrame(data = principalComponents
                      , columns = ['principal component 1', 'principal component 2'])

In [71]: principalDf.head(5)

Out[71]:
```

	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

Please refer the video link below for the complete explanation.

PCA Video Link: https://youtu.be/XwBhf_BtNPo

Hope you liked the PCA explanation.

Connect with me using:

Blog: https://ashutoshtripathi.com/

YouTube channel: https://www.youtube.com/channel/UC0VU1uXWEYCpVGFSZ0OQD5g/

LinkedIn: https://www.linkedin.com/in/ashutoshtripathi1/

Thank you!