All About ML

All About ML — Part 8: Understanding Principal Component Analysis — PCA

Dharani J Follow
Mar 24 · 7 min read

In Machine Learning, while working on real world problems we come across data sets that have more than 20 features or high dimensional data. To check the correlation between them, we might have to visualize 20C2 = 190 2D scatter plots! That's a lot to visualize. On top of that, most of them will not be informative. Clearly if we have many features it gets clumsy to analyze the features and understand their relations. Rather than analyzing each pair from many, if we can try to reduce the dimension to a small range by capturing all the information then we can effortlessly get insights from data.

There are two ways of dimensionality reduction:

1. **Feature Elimination:** In this method if we have 100 features we try to eliminate features that have least importance by various methods like finding p-value, observing co-relation matrix etc.,
2. **Feature Extraction:** In this method if we have 100 features, we generate 100 new features where each new feature is a combination that preserves the information of these 100 old features. we keep as many of the new independent variables as we want, but we drop the "least important ones".

> ***Principal component analysis*** *is a technique of* feature extraction *it combines input features in a specific way and later drop the "least important" variables while still retaining the most explainable information of all of the variables!* As an added benefit, each of the "new" variables after PCA are all independent of one another. *PCA — Reduce Dimensions, Preserve Information*

Principal Component Analysis:

Principal Component analysis has paved a perfect path for dimension reduction. The goal of PCA is to identify patterns in a data set, and then filter out the variables to their crucial features so that the data is simplified with preserving as much as information possible. Let us understand how does PCA do it here in this blog.

If X1, X2,…,Xp are original p number of features, let Z1, Z2,…,ZM represent M<p linear combinations of original p predictors.

$$Z_m = \sum_{j=1}^{p} \phi_{jm} X_j$$

For some constants φ1m, φ2m …,φpm, m = 1,…,M. We can then fit the linear regression model using lease squares as shown below:

$$y_i = \theta_0 + \sum_{m=1}^{M} \theta_m z_{im} + \epsilon_i, \quad i = 1, \ldots, n,$$

The regression coefficients are given by θ0, θ1,...,θM

The term dimension reduction comes from the fact that this approach reduces the problem of estimating the p+1 coefficients β0, β1,…,βp to the simpler problem of estimating the M+1 coefficients θ0, θ1,…,θM , where M<p.

$$\sum_{m=1}^{M} \theta_m z_{im} = \sum_{m=1}^{M} \theta_m \sum_{j=1}^{p} \phi_{jm} x_{ij} = \sum_{j=1}^{p} \sum_{m=1}^{M} \theta_m \phi_{jm} x_{ij} = \sum_{j=1}^{p} \beta_j x_{ij} \text{ Where, } \beta_j = \sum_{m=1}^{M} \theta_m \phi_{jm}$$

We assume that the directions in which X1,…,Xp show the most variation are the directions that are associated with Y. Most of the times this assumption is valid to obtain good results. This helps in fitting a least squares model to

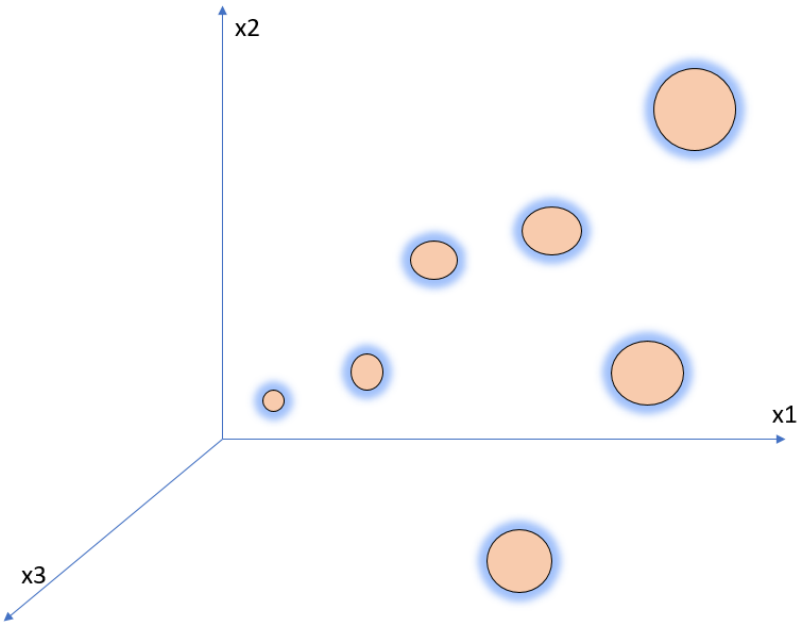
Z1,…,ZM will lead to better results than fitting a least squares model to X1,…,Xp, since most or all of the information in the data that relates to the response is contained in Z1,…,ZM, and by estimating only M <<p coefficients we can **mitigate overfitting**.

For easy understanding lets take a data set with 3 dimensions and lets understand how can we reduce it to 2 dimensions visually:

1. Plot data

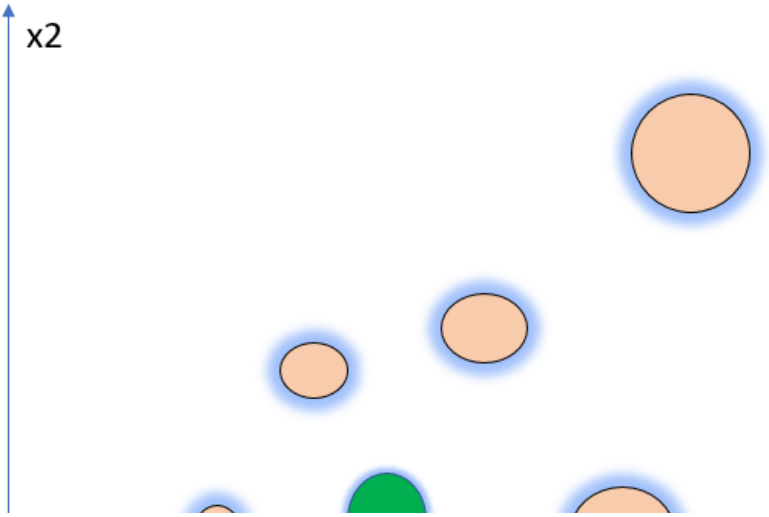
Let's assume our data looks like below. On the left, are features x1, x2, x3. On the right, those points are plotted.

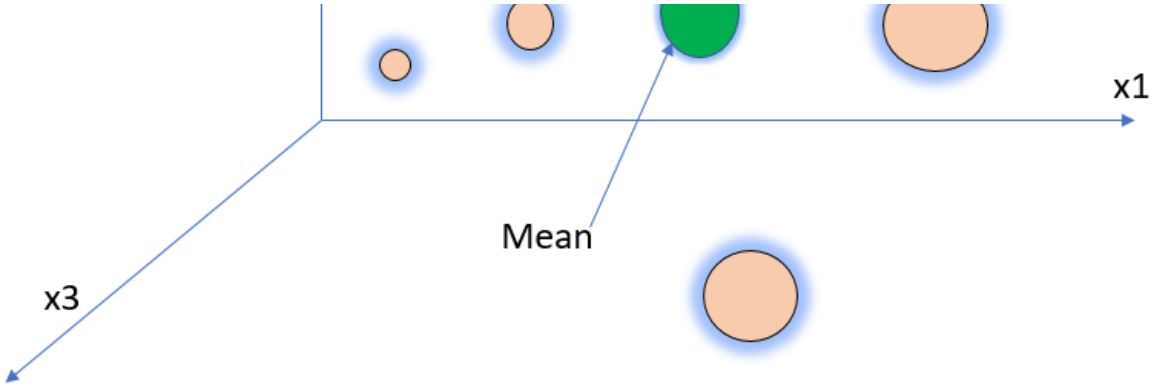
x1	x2	x3
1	1	1
2	1.5	1.5
3	3	3
3.75	1	4
4	3.5	4
5	2	3
6	5	7



2. Mark center of the data

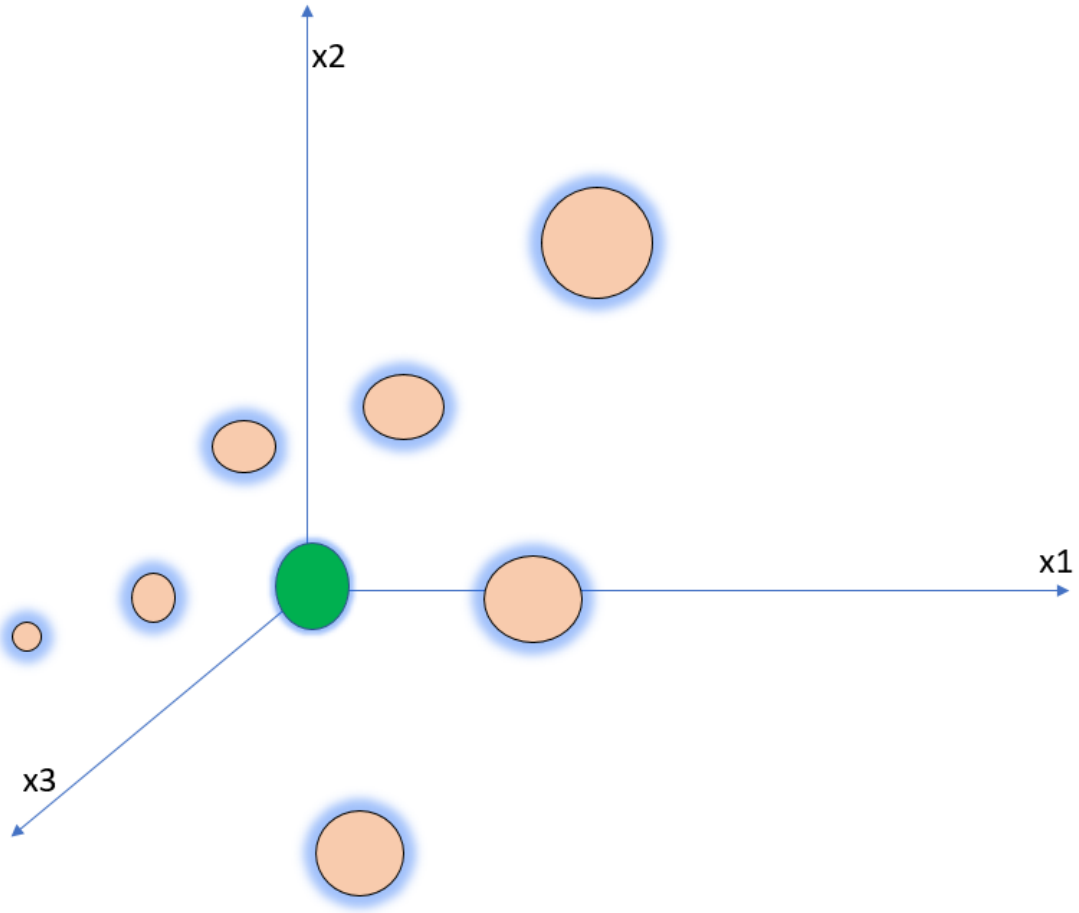
Green circle is the mean of features: x1, x2, x3.





3. Center shifting

Shift the center of axes to the mean of the data points.

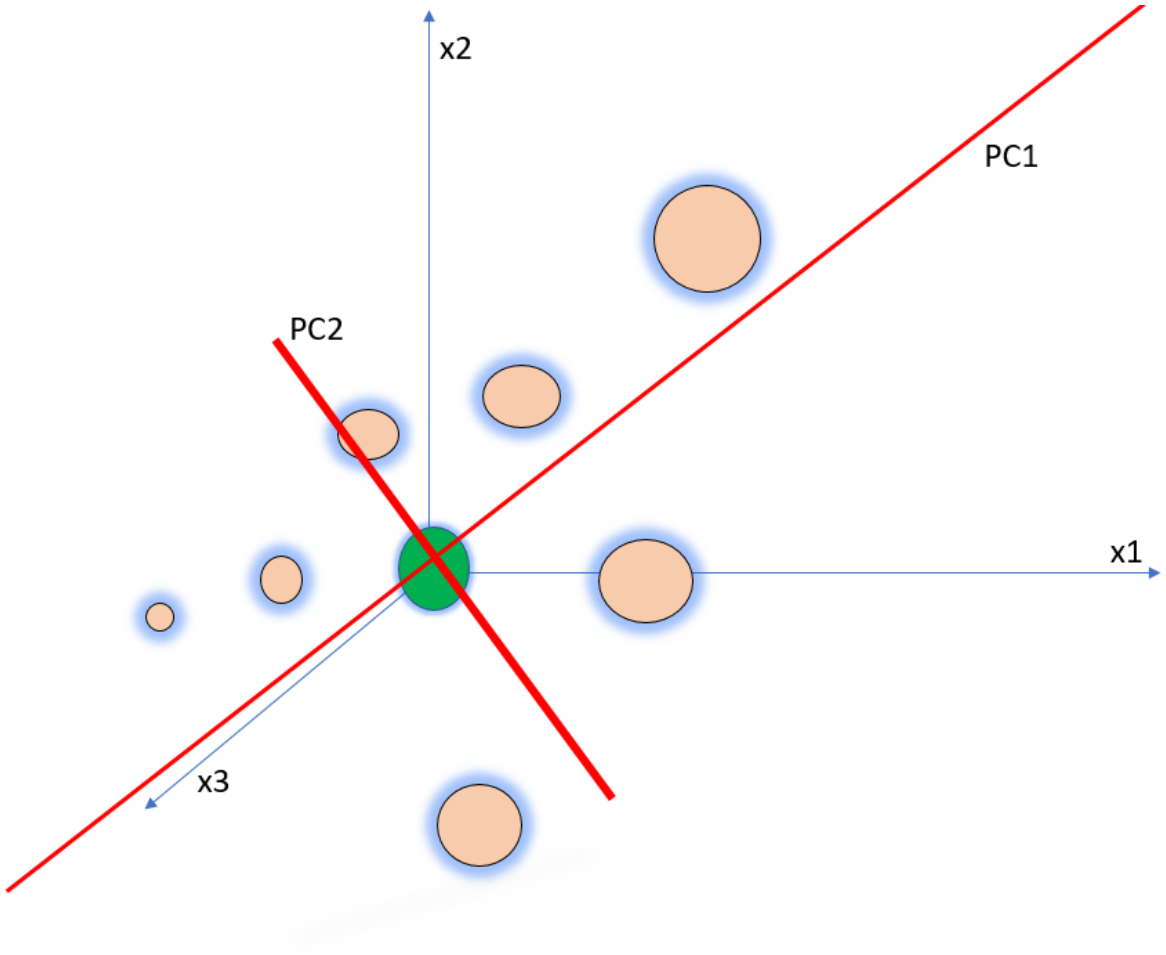


Note data points relative position doesn't change

4. Line of best fit

The line of best fit is called PC1 (principal component 1). PC1 **maximizes** the sum of squared distances from where points meet the line of best fit at a right angle. **Principal Component's direction of the data is that along which the observations vary the most.**

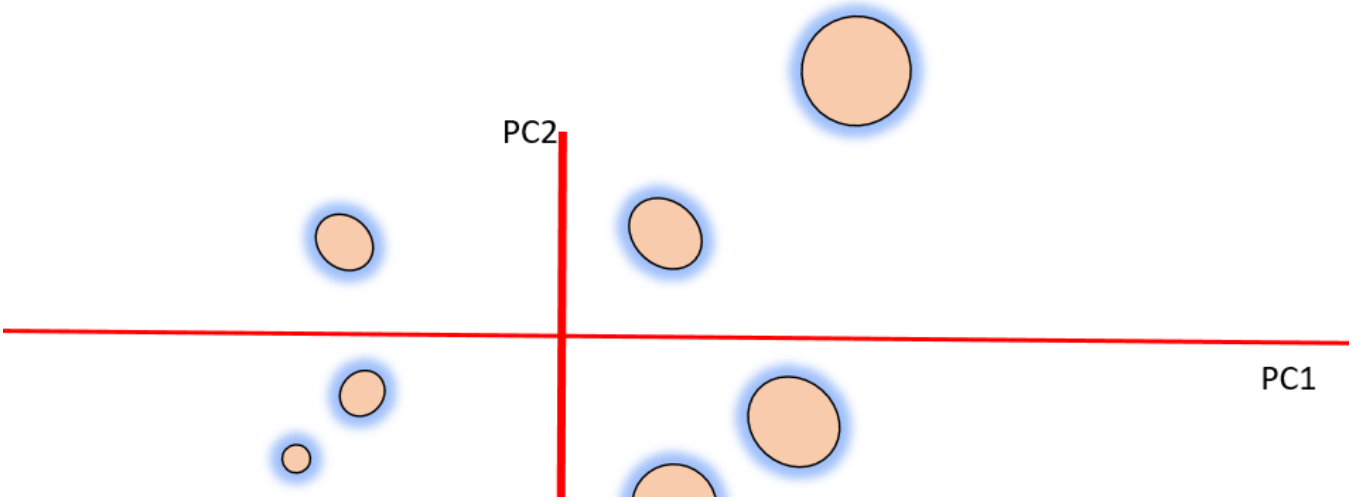
PC1 is a linear combination of x1, x2 and x3, meaning it contains parts of each x1, x2 and x3. PC2 is also a linear combination of each x1, x2 and x3 but exactly in perpendicular direction to PC1. PC1 and PC2 now both explain most of the variance in our features.

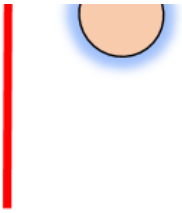


Orthogonal Principal Components preserving most of the information of our data

5. Readjusting the axes

Rotate the axes such that PC1 is X and PC2 is Y axes respectively. Post-rotation, our data is now in just 2 dimensions! And clusters are easy to spot.





Its fairly easy to understand it for 3 dimensions. But to visualize for data with more than 3 dimensions, its not possible. So let's understand what are the steps involved in capturing the Principal Components using python code:

PCA Implementation

Let do the experiment on Iris data set which has 4 predictive features and Y variable as names of different species of flowers. Data can be downloaded from — https://drive.google.com/file/d/1EQD1iE5frCmo5mD8u1H51AU8ldsnuUX7/view?usp=sharing

Step 1 is to standardize the features. Aim of this step is to **standardize** the range of the continuous initial variables so that each one of them contributes equally to the analysis.

$$z = \frac{value - mean}{standard\ deviation}$$

Formula to standardize

Standardization is a crucial step to perform before PCA because PCA is sensitive to variance in original features. If there are large differences between the ranges of initial variables, those variables with larger ranges will dominate over those with small ranges (For example, a variable that ranges between 0 and 100 will dominate over a variable that ranges between 0 and 1), which will lead to biased results. So, transforming the data to comparable scales can prevent this problem.

```
1  import pandas as pd
2  import numpy as np
3
4  from sklearn.preprocessing import StandardScaler
```

```
import matplotlib.pyplot as plt

Iris = pd.read_csv("Iris.csv")
X = Iris[['SepalLengthCm','SepalWidthCm','PetalLengthCm','PetalWidthCm']]
y=Iris.Species

#STEP 1
X = StandardScaler().fit_transform(X) #standardization
```

pca_codepiece1 hosted with ❤ by GitHub view raw

Step2 Covariance Matrix and Eigen values:

No one wants redundant information. Covariance matrix exactly helps us in this point. We calculate the covariance scores of 2 features and based on the values, we can go ahead to finding the principal components.

$$Cov(X,Y) = \frac{\sum(x_i - \bar{x})(y_i - \bar{y})}{N-1}$$

If positive value in covariance matrix, then : the two variables increase or decrease together (correlated) and if negative then : One increases when the other decreases (Inversely correlated)

```
#STEP 2 - Covarience Matrix and EigenDecomposition
```

```
2
3   X_mean = np.mean(X, axis=0)
4   # cov_mat = np.cov(X) #can use cov function if we are directly using numpy
5   cov_mat = (X - X_mean).T.dot((X - X_mean)) / (X.shape[0]-1)
6   print('Covariance matrix \n%s' %cov_mat)
```

pca_codepiece2 hosted with ❤ by GitHub view raw

```
Covariance matrix
[[ 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]]
```

Step 3 — Eigen decomposition: As we have 4 features, we get 4*4 matrix and each value represents the score of all the possible feature combinations. Eigenvectors and eigenvalues are the linear algebra concepts that we need to compute from the covariance matrix in order to determine the ***principal components*** of the data as explained above as PC1, PC2 and so on. The eigenvectors of the Covariance matrix are actually *the* ***directions of the axes where there is the most variance*** (most information) and that we call Principal Components. And eigenvalues are simply the coefficients attached to eigenvectors, which give the ***amount of variance*** *carried in each Principal Component*.

```
1   eig_vals, eig_vecs = np.linalg.eig(cov_mat)
2
3   print('Eigenvectors \n%s' %eig_vecs)
4   print('\nEigenvalues \n%s' %eig_vals)
```

pca_codepiece3 hosted with ❤ by GitHub view raw

```
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]
```

The eigenvectors only define the directions of the new axis and they all have the same unit length 1. In order to decide which eigenvector(s) can be dropped without losing too much information, 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. So we arrange them in descending order and take the top number of components required.

```
# Create list of (eigenvalue, eigenvector) tuples
eig_pairs = [(np.abs(eig_vals[i]), eig_vecs[:,i]) for i in range(len(eig_vals))]

# Sort the (eigenvalue, eigenvector) tuples from in descending order
eig_pairs.sort(key=lambda x: x[0], reverse=True)

# Visually confirm that the list is correctly sorted by decreasing eigenvalues
print('Eigenvalues in descending order:')
for i in eig_pairs:
    print(i[0])
```

```
Eigenvalues in descending order:
2.930353775589317
0.9274036215173419
0.14834222648163944
0.02074601399559593
```

Step 4 Principal Components: After sorting the eigenpairs, the next question is "how many principal components are we going to choose for our new feature subspace?" Here comes "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)
var_exp = [(i / tot)*100 for i in sorted(eig_vals, reverse=True)]
cum_var_exp = np.cumsum(var_exp)
cum_var_exp
```

pca_codepiece5 hosted with ❤ by GitHub view raw

```
array([ 72.77045209,  95.80097536,  99.48480732, 100.])
```

The projection matrix is used to transform the Input data(X) onto the new feature subspace. Projection Matrix is a matrix of 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 2-dimensional eigenvector matrix.

```
matrix_w = np.hstack((eig_pairs[0][1].reshape(4,1),
                      eig_pairs[1][1].reshape(4,1)))

```

```
print('Matrix W:\n', matrix_w)
```

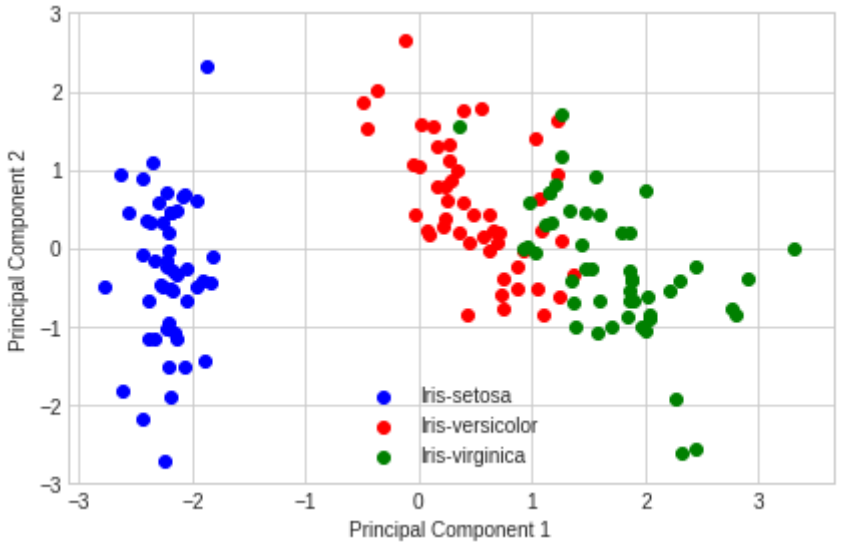
pca_codepiece6 hosted with ❤ by GitHub view raw

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

```
Y = X.dot(matrix_w)


with plt.style.context('seaborn-whitegrid'):
    plt.figure(figsize=(6, 4))
    for lab, col in zip(('Iris-setosa', 'Iris-versicolor', 'Iris-virginica'), ('blue',
        plt.scatter(Y[y==lab, 0], Y[y==lab, 1], label=lab, c=col)
    plt.xlabel('Principal Component 1')
    plt.ylabel('Principal Component 2')
    plt.legend(loc='lower center')
    plt.tight_layout()
    plt.show()
```

pca_codepiece7 hosted with ❤ by GitHub view raw



We can observe that we can identify the clusters by reducing it to 2 dimensions. We have an in built PCA function in sklearn and it does all the above steps for us. Usage of the code:

```
from sklearn.decomposition import PCA as sklearnPCA
sklearn_pca = sklearnPCA(n_components=2)
Y_sklearn = sklearn_pca.fit_transform(X)

sklearn_pca.explained_variance_ratio_

with plt.style.context('seaborn-whitegrid'):
    plt.figure(figsize=(6, 4))
    for lab, col in zip(('Iris-setosa', 'Iris-versicolor', 'Iris-virginica'),
                        ('blue', 'red', 'green')):
        plt.scatter(Y_sklearn[y==lab, 0],
                    Y_sklearn[y==lab, 1],
                    label=lab,
                    c=col)
    plt.xlabel('Principal Component 1')
    plt.ylabel('Principal Component 2')
    plt.legend(loc='lower center')
    plt.tight_layout()
    plt.show()
```

Hope this blog gave you a quick understanding on how PCA works and its implementation for reference. Happy Learning! :)

Reference Book — An Introduction to Statistical Learning: With Applications in R

Pca Principal Component Dimensionality Reduction Machine Learning Feature Engineering