plotly

Show Sidebar Fork on Github

Principal Component Analysis in Python

A step by step tutorial to Principal Component Analysis, a simple yet powerful transformation technique.

About the Author: Some of Sebastian Raschka's greatest passions are "Data Science" and machine learning. Sebastian enjoys everything that involves working with data: The discovery of interesting patterns and coming up with insightful conclusions using techniques from the fields of data mining and machine learning for predictive modeling.

Currently, Sebastian is sharpening his analytical skills as a PhD candidate at Michigan State University where he is working on a highly efficient virtual screening software for computer-aided drug-discovery and a novel approach to protein ligand docking (among other projects). Basically, it is about the screening of a database of millions of 3-dimensional structures of chemical compounds in order to identifiy the ones that could potentially bind to specific protein receptors in order to trigger a biological response.

You can follow Sebastian on Twitter (@rasbt) or read more about his favorite projects on his blog.

Principal Component Analysis (PCA) is a simple yet popular and useful linear transformation technique that is used in numerous applications, such as stock market predictions, the analysis of gene expression data, and many more. In this tutorial, we will see that PCA is not just a "black box", and we are going to unravel its internals in 3 basic steps.

Introduction

The sheer size of data in the modern age is not only a challenge for computer hardware but also a main bottleneck for the performance of many machine learning algorithms. The main goal of a PCA analysis is to identify patterns in data; PCA aims to detect the correlation between variables. If a strong correlation between variables exists, the attempt to reduce the dimensionality only makes sense. In a nutshell, this is what PCA is all about: Finding the directions of maximum variance in high-dimensional data and project it onto a smaller dimensional subspace while retaining most of the information.

PCA Vs. LDA

different classes, which can be useful in pattern classification problem (PCA "ignores" class labels).
In other words, PCA projects the entire dataset onto a different feature (sub)space, and LDA tries to determine a suitable feature (sub)space in order to distinguish between patterns that belong to different classes.

PCA and Dimensionality Reduction

Often, the desired goal is to reduce the dimensions of a d-dimensional dataset by projecting it onto a (k)-dimensional subspace (where $k < d$) in order to increase the computational efficiency while retaining most of the information. An important question is "what is the size of k that represents the data 'well'?"

Later, we will compute eigenvectors (the principal components) of a dataset and collect them in a projection matrix. Each of those eigenvectors is associated with an eigenvalue which can be interpreted as the "length" or "magnitude" of the corresponding eigenvector. If some eigenvalues have a significantly larger magnitude than others that the reduction of the dataset via PCA onto a smaller dimensional subspace by dropping the "less informative" eigenpairs is reasonable.

A Summary of the PCA Approach

or perform Singular Vector Decomposition.

- Sort eigenvalues in descending order and choose the k eigenvectors that correspond to the k largest eigenvalues where k is the number of dimensions of the new feature subspace ($k \leq d$)/.
- Construct the projection matrix $\mathbf{W}$ from the selected k eigenvectors.
- Transform the original dataset $\mathbf{X}$ via $\mathbf{W}$ to obtain a k-dimensional feature subspace $\mathbf{Y}$.

Preparing the Iris Dataset

The iris dataset contains measurements for 150 iris flowers from three different species.

The three classes in the Iris dataset are:

1. Iris-setosa (n=50)
2. Iris-versicolor (n=50)
3. Iris-virginica (n=50)

And the four features of in Iris dataset are:

1. sepal length in cm
2. sepal width in cm
3. petal length in cm
4. petal width in cm

Loading the Dataset

In order to load the Iris data directly from the UCI repository, we are going to use the superb pandas library. If you haven't used pandas yet, I want encourage you to check out the pandas tutorials. If I had to name one Python library that makes working with data a wonderfully simple task, this would definitely be pandas!

```
    header=None,
    sep=',')

df.columns=['sepal_len', 'sepal_wid', 'petal_len', 'petal_wid', 'clas
s']
df.dropna(how="all", inplace=True) # drops the empty line at file-end

df.tail()
```

Out[1]:

	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

In [2]:

```
# split data table into data X and class labels y

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

plotly

Show Sidebar Fork on Github

$$\mathbf{x^T} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} \text{sepal length} \\ \text{sepal width} \\ \text{petal length} \\ \text{petal width} \end{pmatrix}$$

Exploratory Visualization

To get a feeling for how the 3 different flower classes are distributes along the 4 different features, let us visualize them via histograms.

In [3]:
```
import plotly.plotly as py
from plotly.graph_objs import *
import plotly.tools as tls
```

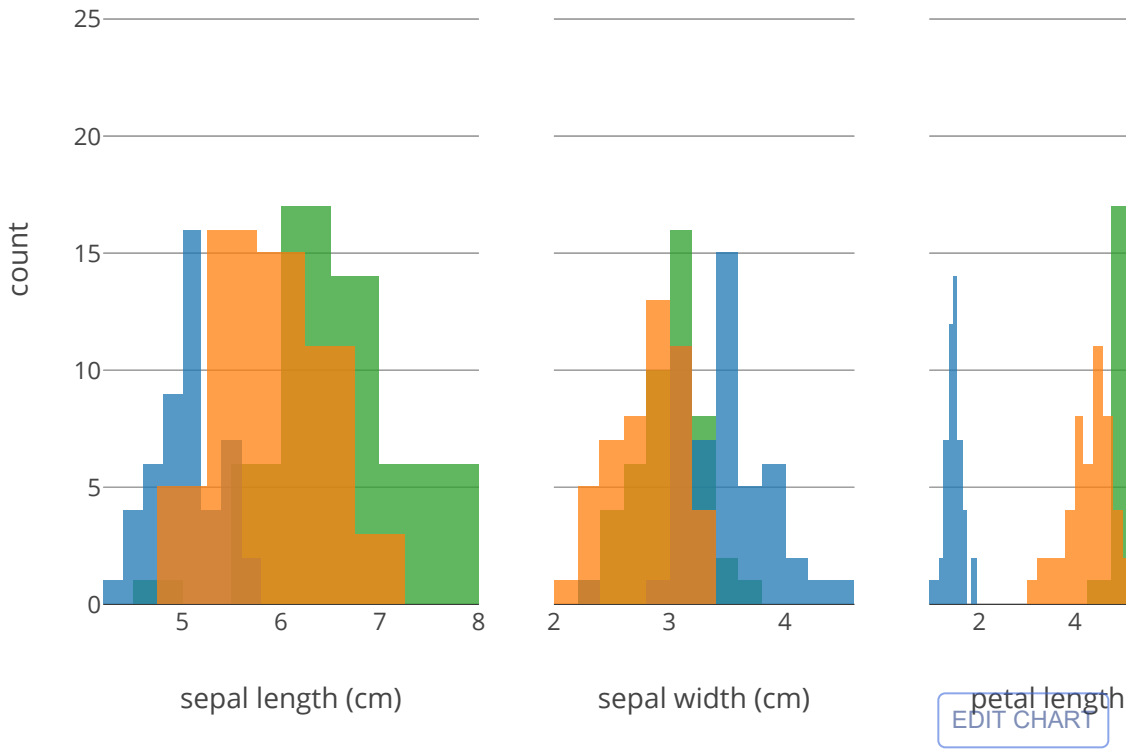
```
colors = {'Iris-setosa': 'rgb(31, 119, 180)',
          'Iris-versicolor': 'rgb(255, 127, 14)',
          'Iris-virginica': 'rgb(44, 160, 44)'}

for col in range(4):
    for key in colors:
        traces.append(Histogram(x=X[y==key, col],
                                opacity=0.75,
                                xaxis='x%s' %(col+1),
                                marker=Marker(color=colors[key]),
                                name=key,
                                showlegend=legend[col]))

data = Data(traces)

layout = Layout(barmode='overlay',
                xaxis=XAxis(domain=[0, 0.25], title='sepal length (c
m)'),
                xaxis2=XAxis(domain=[0.3, 0.5], title='sepal width (c
m)'),
                xaxis3=XAxis(domain=[0.55, 0.75], title='petal length
 (cm)'),
                xaxis4=XAxis(domain=[0.8, 1], title='petal width (c
m)'),
                yaxis=YAxis(title='count'),
                title='Distribution of the different Iris flower feat
ures')

fig = Figure(data=data, layout=layout)
py.iplot(fig)
```



Standardizing

Whether to standardize the data prior to a PCA on the covariance matrix depends on the measurement scales of the original features. Since PCA yields a feature subspace that maximizes the variance along the axes, it makes sense to standardize the data, especially, if it was measured on different scales. Although, all features in the Iris dataset were measured in centimeters, let us continue with the transformation of the data onto unit scale (mean=0 and variance=1), which is a requirement for the optimal performance of many machine learning algorithms.

1 - Eigendecomposition - 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.

Covariance Matrix

$\sigma_{jk} = \frac{1}{n-1} \sum_{i=1}^{N} \left(x_{ij} - \bar{x}_j \right) \left(x_{ik} - \bar{x}_k \right).$

We can summarize the calculation of the covariance matrix via the following matrix equation:
$\Sigma = \frac{1}{n-1} \left((\mathbf{X} - \bar{\mathbf{x}})^T \; (\mathbf{X} - \bar{\mathbf{x}}) \right)$
where $\bar{\mathbf{x}}$ is the mean vector $\bar{\mathbf{x}} = \sum_{k=1}^{n} x_i$.
The mean vector is a d-dimensional vector where each value in this vector represents the sample mean of a feature column in the dataset.

In [6]:
```
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)
```

```
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]]
```

The more verbose way above was simply used for demonstration purposes, equivalently, we could have used the numpy `cov` function:

```
 [-0.11010327  1.00671141 -0.42333835 -0.358937  ]
 [ 0.87760486 -0.42333835  1.00671141  0.96921855]
 [ 0.82344326 -0.358937    0.96921855  1.00671141]]
```

Next, we perform an eigendecomposition on the covariance matrix:

In [8]:
```
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]
```

Especially, in the field of "Finance," the correlation matrix typically used instead of the covariance matrix. However, the eigendecomposition of the covariance matrix (if the input data was standardized) yields the same results as a eigendecomposition on the correlation matrix, since the correlation matrix can be understood as the normalized covariance matrix.
Eigendecomposition of the standardized data based on the correlation matrix:

In [9]:
```
cor_mat1 = np.corrcoef(X_std.T)

eig_vals, eig_vecs = np.linalg.eig(cor_mat1)

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
[ 2.91081808  0.92122093  0.14735328  0.02060771]
```

Eigendecomposition of the raw data based on the correlation matrix:

```
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
[ 2.91081808  0.92122093  0.14735328  0.02060771]
```

We can clearly see that all three approaches yield the same eigenvectors and eigenvalue pairs:

- Eigendecomposition of the covariance matrix after standardizing the data.
- Eigendecomposition of the correlation matrix.
- Eigendecomposition of the correlation matrix after standardizing the data.

While the eigendecomposition of the covariance or correlation matrix may be more intuitiuve, most PCA implementations perform a Singular Vector Decomposition (SVD) to improve the computational efficiency. So, let us perform an SVD to confirm that the result are indeed the same:

In [11]:
```
u,s,v = np.linalg.svd(X_std.T)
u
```

Out[11]:
```
array([[-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]])
```

2 - Selecting 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, which can confirmed by the following two lines of code:

```
Everything ok!
```

In order to decide which eigenvector(s) can 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 choose the top k eigenvectors.

In [13]:
```
# Make a 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 high to low
eig_pairs.sort()
eig_pairs.reverse()

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

```
Eigenvalues in descending order:
2.91081808375
0.921220930707
0.147353278305
0.0206077072356
```

much information (variance) can be attributed to each of the principal components.

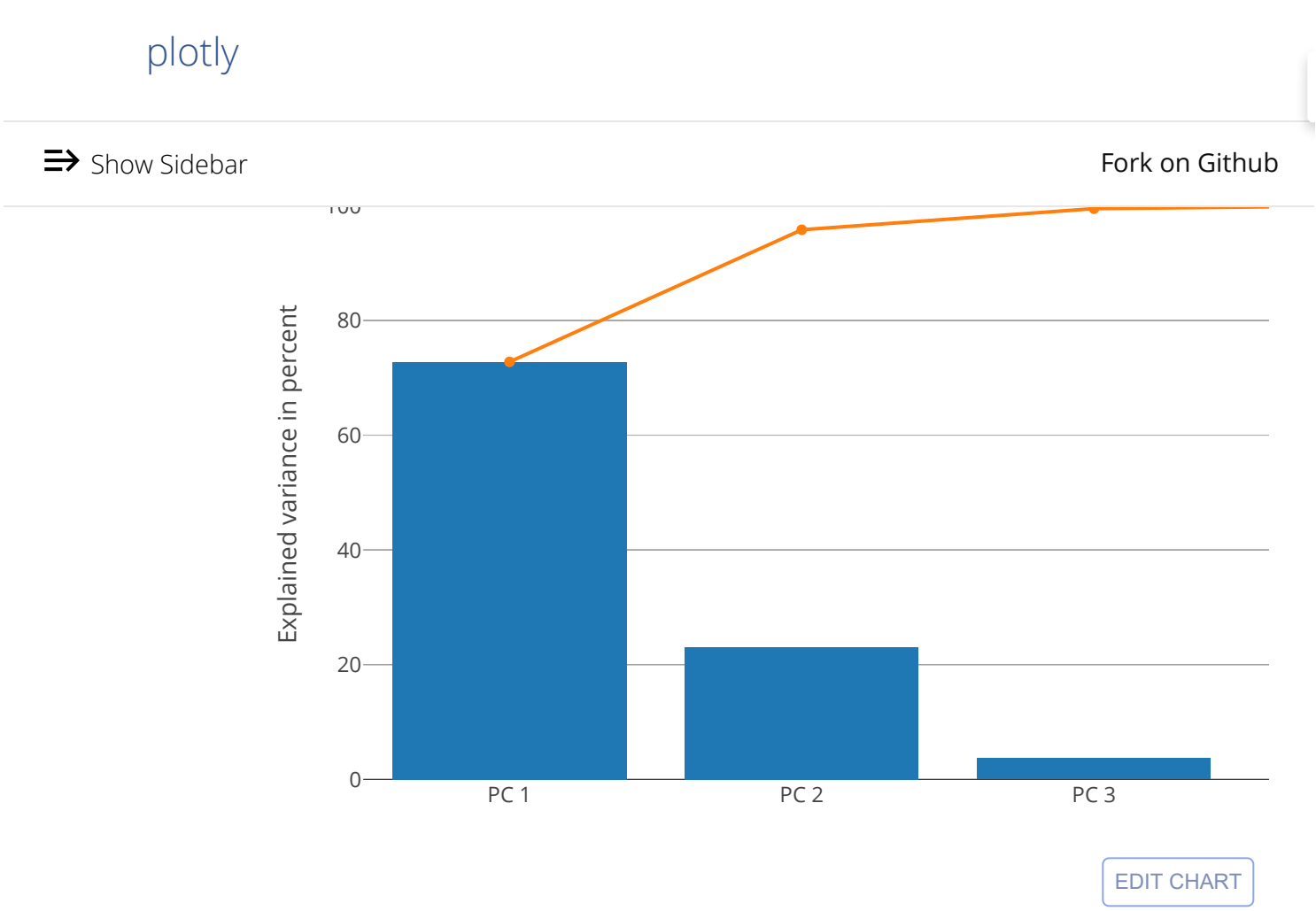
```
        x=['PC %s' %i for i in range(1,5)],
        y=var_exp,
        showlegend=False)

trace2 = Scatter(
        x=['PC %s' %i for i in range(1,5)],
        y=cum_var_exp,
        name='cumulative explained variance')

data = Data([trace1, trace2])

layout=Layout(
        yaxis=YAxis(title='Explained variance in percent'),
        title='Explained variance by different principal components')

fig = Figure(data=data, layout=layout)
py.iplot(fig)
```



The plot above clearly shows that most of the variance (72.77% of the variance to be precise) can be explained by the first principal component alone. The second principal component still bears some information (23.03%) while the third and fourth principal components can safely be dropped without losing to much information. Together, the first two principal components contain 95.8% of the information.

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 $d \times k$-dimensional eigenvector matrix $\mathbf{W}$.

In [15]:
```
matrix_w = np.hstack((eig_pairs[0][1].reshape(4,1),
                      eig_pairs[1][1].reshape(4,1)))

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

```
('Matrix W:\n', array([[ 0.52237162, -0.37231836],
       [-0.26335492, -0.92555649],
       [ 0.58125401, -0.02109478],
       [ 0.56561105, -0.06541577]]))
```

3 - Projection Onto the New Feature Space

In this last step we will use the 4×2-dimensional projection matrix $\mathbf{W}$ to transform our samples onto the new subspace via the equation
$\mathbf{Y} = \mathbf{X} \times \mathbf{W}$, where $\mathbf{Y}$ is a 150×2 matrix of our transformed samples.

plotly

Show Sidebar Fork on Github

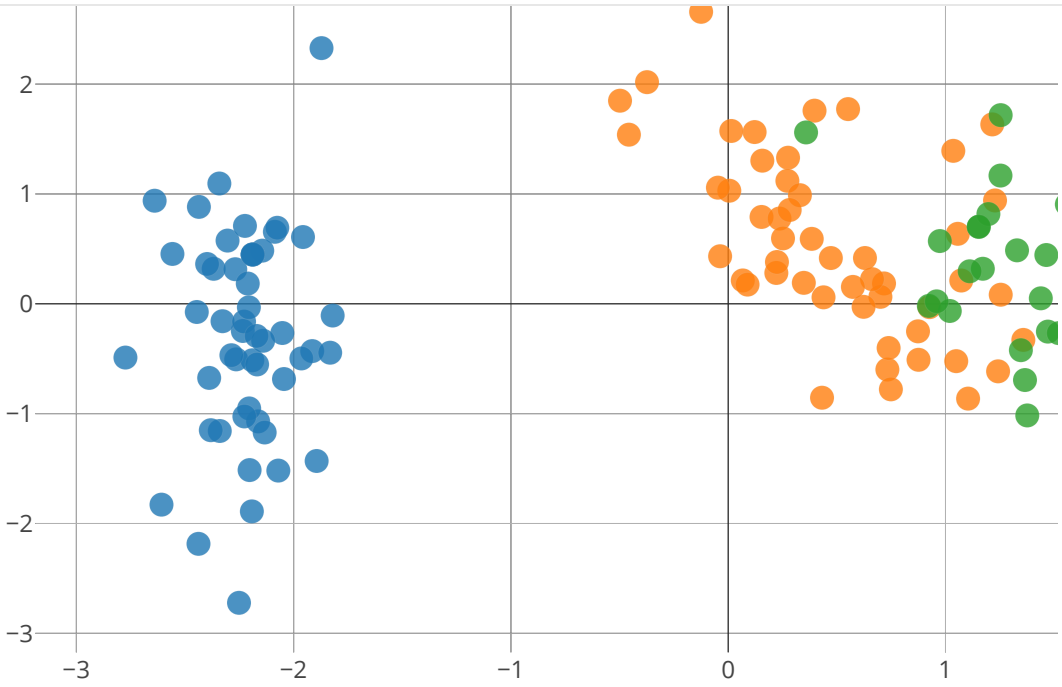
```
            x=Y[y==name,0],
            y=Y[y==name,1],
            mode='markers',
            name=name,
            marker=Marker(
                size=12,
                line=Line(
                    color='rgba(217, 217, 217, 0.14)',
                    width=0.5),
                opacity=0.8))
        traces.append(trace)


data = Data(traces)
layout = Layout(showlegend=True,
                scene=Scene(xaxis=XAxis(title='PC1'),
                yaxis=YAxis(title='PC2'),))

fig = Figure(data=data, layout=layout)
py.iplot(fig)
```

plotly

Show Sidebar Fork on Github



EDIT CHART

Shortcut - PCA in scikit-learn

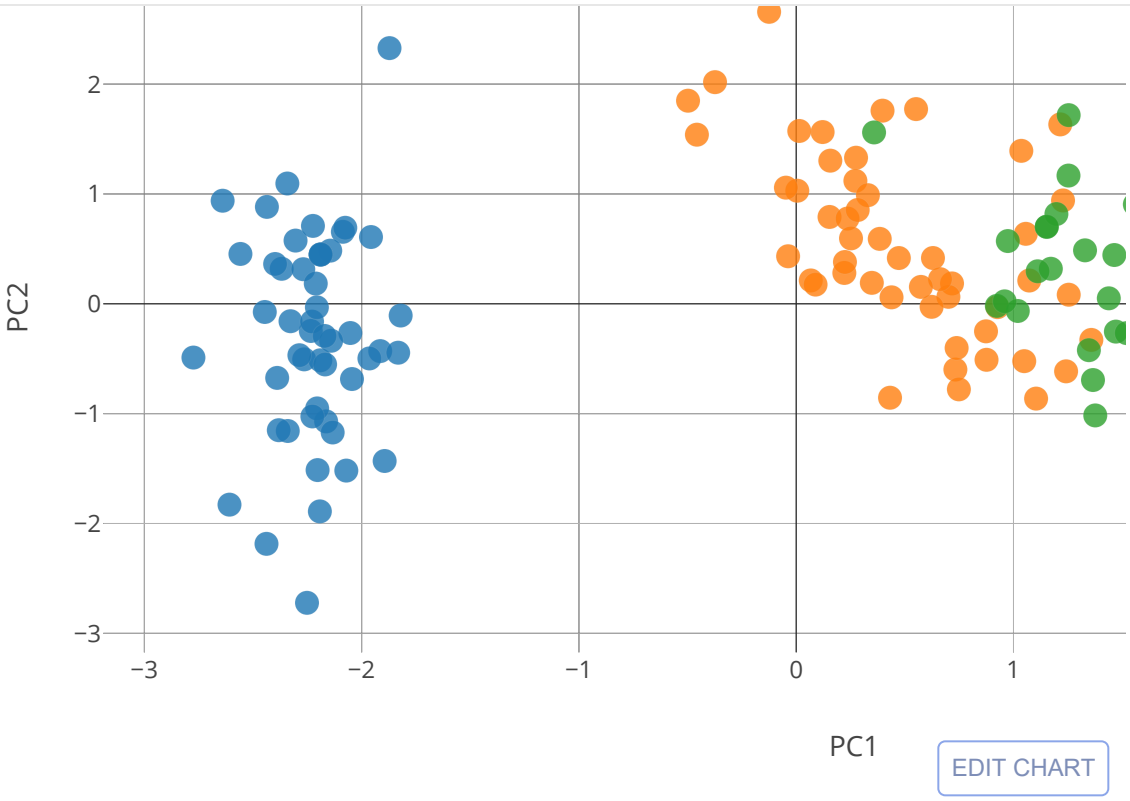
For educational purposes, we went a long way to apply the PCA to the Iris dataset. But luckily, there is already implementation in scikit-learn.

```
            x=Y_sklearn[y==name,0],
            y=Y_sklearn[y==name,1],
            mode='markers',
            name=name,
            marker=Marker(
                size=12,
                line=Line(
                    color='rgba(217, 217, 217, 0.14)',
                    width=0.5),
                opacity=0.8))
        traces.append(trace)


data = Data(traces)
layout = Layout(xaxis=XAxis(title='PC1', showline=False),
                yaxis=YAxis(title='PC2', showline=False))
fig = Figure(data=data, layout=layout)
py.iplot(fig)
```

plotly

Show Sidebar Fork on Github



Still need help?

Contact Us

community.plot.ly

support.plot.ly