

Implementing PCA in Python with Scikit-Learn

 stackabuse.com/implementing-pca-in-python-with-scikit-learn

With the availability of high performance CPUs and GPUs, it is pretty much possible to solve every regression, classification, clustering and other related problems using machine learning and deep learning models. However, there are still various factors that cause performance bottlenecks while developing such models. Large number of features in the dataset is one of the factors that affect both the training time as well as accuracy of machine learning models. You have different options to deal with huge number of features in a dataset.

1. Try to train the models on original number of features, which take days or weeks if the number of features is too high.
2. Reduce the number of variables by merging correlated variables.
3. Extract the most important features from the dataset that are responsible for maximum variance in the output. Different statistical techniques are used for this purpose e.g. linear discriminant analysis, factor analysis, and principal component analysis.

Principal Component Analysis

Principal component analysis, or **PCA**, is a statistical technique to convert high dimensional data to low dimensional data by selecting the most important features that capture maximum information about the dataset. The features are selected on the basis of variance that they cause in the output. The feature that causes highest variance is the first principal component. The feature that is responsible for second highest variance is considered the second principal component, and so on. It is important to mention that principal components do not have any correlation with each other.

Advantages of PCA

There are two main advantages of dimensionality reduction with PCA.

1. The training time of the algorithms reduces significantly with less number of features.
2. It is not always possible to analyze data in high dimensions. For instance if there are 100 features in a dataset. Total number of scatter plots required to visualize the data would be $100(100-1)2 = 4950$. Practically it is not possible to analyze data this way.

Normalization of Features

It is imperative to mention that a feature set must be normalized before applying PCA. For instance if a feature set has data expressed in units of Kilograms, Light years, or Millions, the variance scale is huge in the training set. If PCA is applied on such a feature set, the resultant loadings for features with high variance will also be large. Hence, principal components will be biased towards features with high variance, leading to false results.

Finally, the last point to remember before we start coding is that PCA is a statistical technique and can only be applied to numeric data. Therefore, categorical features are required to be converted into numerical features before PCA can be applied.

Implementing PCA with Scikit-Learn

In this section we will implement PCA with the help of Python's Scikit-Learn library. We will follow the classic machine learning pipeline where we will first import libraries and dataset, perform exploratory data analysis and preprocessing, and finally train our models, make predictions and evaluate accuracies. The only additional step will be to perform PCA to find out optimal number of features before we train our models. These steps have been implemented as follows:

Importing Libraries

```
import numpy as np
import pandas as pd
```

Importing Dataset

The dataset we are going to use in this article is the famous Iris data set. Some additional information about the Iris dataset is available at:

<https://archive.ics.uci.edu/ml/datasets/iris>

The dataset consists of 150 records of Iris plant with four features: 'sepal-length', 'sepal-width', 'petal-length', and 'petal-width'. All of the features are numeric. The records have been classified into one of the three classes i.e. 'Iris-setosa', 'Iris-versicolor', or 'Iris-verginica'.

Execute the following script to download the dataset using `pandas` :

```
url = "https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data"
names = ['sepal-length', 'sepal-width', 'petal-length', 'petal-width', 'Class']
dataset = pd.read_csv(url, names=names)
```

Let's take a look at what our dataset looks like:

```
dataset.head()
```

Executing the above command will display the first five rows of our dataset as shown below:

	sepal-length	sepal-width	petal-length	petal-width	Class
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

Preprocessing

The first preprocessing step is to divide the dataset into a feature set and corresponding labels. The following script performs this task:

```
X = dataset.drop('Class', 1)
y = dataset['Class']
```

The script above stores the feature sets into the `X` variable and the series of corresponding labels in to the `y` variable.

The next preprocessing step is to divide data into training and test sets. Execute the following script to do so:

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=0)
```

As mentioned earlier, PCA performs best with a normalized feature set. We will perform standard scalar normalization to normalize our feature set. To do this, execute the following code:

```
from sklearn.preprocessing import StandardScaler

sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)
```

Applying PCA

It is only a matter of three lines of code to perform PCA using Python's Scikit-Learn library. The `PCA` class is used for this purpose. PCA depends only upon the feature set and not the label data. Therefore, PCA can be considered as an unsupervised machine learning technique.

Performing PCA using Scikit-Learn is a two-step process:

1. Initialize the `PCA` class by passing the number of components to the constructor.
2. Call the `fit` and then `transform` methods by passing the feature set to these methods. The `transform` method returns the specified number of principal components.

Take a look at the following code:

```
from sklearn.decomposition import PCA

pca = PCA()
X_train = pca.fit_transform(X_train)
X_test = pca.transform(X_test)
```

In the code above, we create a `PCA` object named `pca`. We did not specify the number of components in the constructor. Hence, all four of the features in the feature set will be returned for both the training and test sets.

The `PCA` class contains `explained_variance_ratio_` which returns the variance caused by each of the principal components. Execute the following line of code to find the "explained variance ratio".

```
explained_variance = pca.explained_variance_ratio_
```

The `explained_variance` variable is now a float type array which contains variance ratios for each principal component. The values for the `explained_variance` variable looks like this:

```
0.722265
0.239748
0.0333812
0.0046056
```

It can be seen that first principal component is responsible for 72.22% variance. Similarly, the second principal component causes 23.9% variance in the dataset. Collectively we can say that $(72.22 + 23.9)$ 96.21% percent of the classification information contained in the feature set is captured by the first two principal components.

Let's first try to use 1 principal component to train our algorithm. To do so, execute the following code:

```
from sklearn.decomposition import PCA

pca = PCA(n_components=1)
X_train = pca.fit_transform(X_train)
X_test = pca.transform(X_test)
```

The rest of the process is straight forward.

Training and Making Predictions

In this case we'll use random forest classification for making the predictions.

```
from sklearn.ensemble import RandomForestClassifier

classifier = RandomForestClassifier(max_depth=2, random_state=0)
classifier.fit(X_train, y_train)

y_pred = classifier.predict(X_test)
```

Performance Evaluation

```
from sklearn.metrics import confusion_matrix
from sklearn.metrics import accuracy_score

cm = confusion_matrix(y_test, y_pred)
print(cm)
print('Accuracy' + accuracy_score(y_test, y_pred))
```

The output of the script above looks like this:

```
[[11  0  0]
 [ 0 12  1]
 [ 0  1  5]]
0.933333333333
```

It can be seen from the output that with only one feature, the random forest algorithm is able to correctly predict 28 out of 30 instances, resulting in 93.33% accuracy.

Results with 2 and 3 Principal Components

Now let's try to evaluate classification performance of the random forest algorithm with 2 principal components. Update this piece of code:

```
from sklearn.decomposition import PCA

pca = PCA(n_components=2)
X_train = pca.fit_transform(X_train)
X_test = pca.transform(X_test)
```

Here the number of components for PCA has been set to 2. The classification results with 2 components are as follows:

```
[[11  0  0]
 [ 0 10  3]
 [ 0  2  4]]
0.833333333333
```

With two principal components the classification accuracy decreases to 83.33% compared to 93.33% for 1 component.

With three principal components, the result looks like this:

```
[[11  0  0]
 [ 0 12  1]
 [ 0  1  5]]
0.933333333333
```

With three principal components the classification accuracy again increases to 93.33%

Results with Full Feature Set

Let's try to find the results with full feature set. To do so, simply remove the PCA part from the script that we wrote above. The results with full feature set, without applying PCA looks like this:

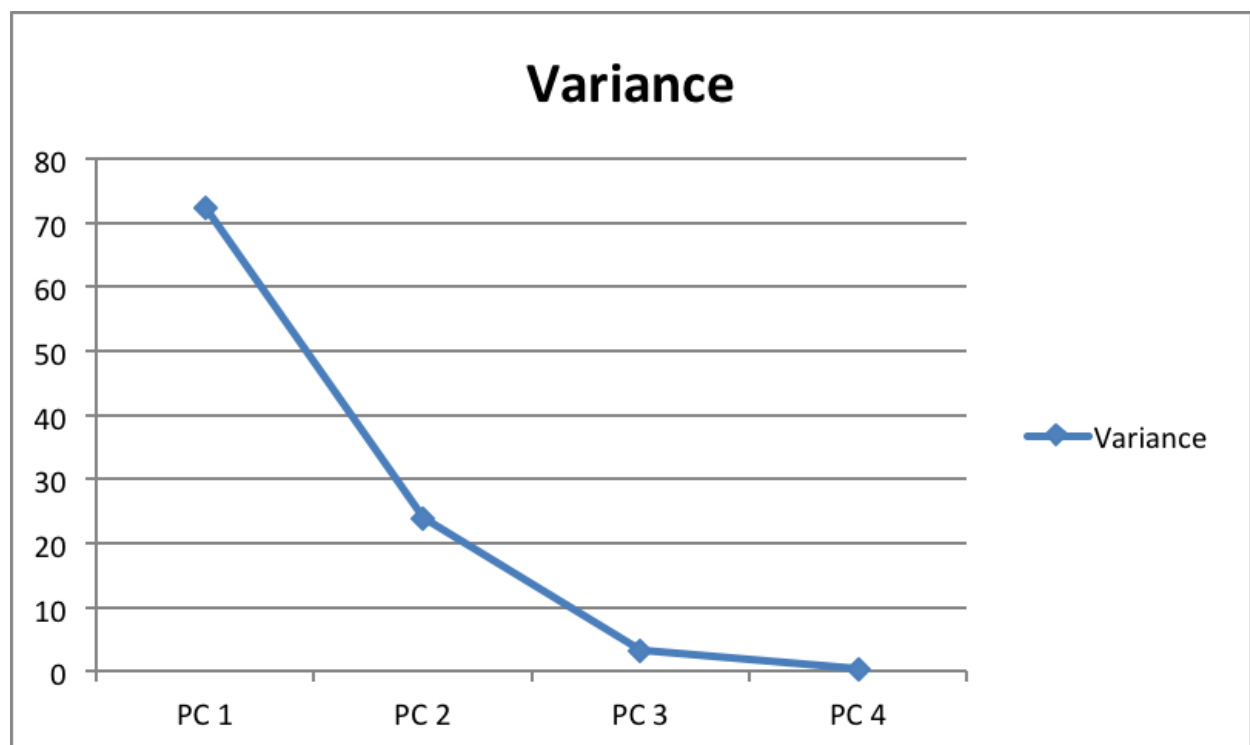
```
[[11  0  0]
 [ 0 13  0]
 [ 0  2  4]]
0.933333333333
```

The accuracy received with full feature set is for random forest algorithm is also 93.33%.

Discussion

From the above experimentation we achieved optimal level of accuracy while significantly reducing the number of features in the dataset. We saw that accuracy achieved with only 1 principal component is equal to the accuracy achieved with full feature set i.e. 93.33%. It is also pertinent to mention that the accuracy of a classifier doesn't necessarily improve with increased number of principal components. From the results we can see that the accuracy achieved with one principal component (93.33%) was greater than the one achieved with two principal components (83.33%).

The number of principal components to retain in a feature set depends on several conditions such as storage capacity, training time, performance, etc. In some dataset all the features are contributing equally to the overall variance, therefore all the principal components are crucial to the predictions and none can be ignored. A general rule of thumb is to take number of principal components that contribute to significant variance and ignore those with diminishing variance returns. A good way is to plot the variance against principal components and ignore the principal components with diminishing values as shown in the following graph:



For instance, in the chart above, we can see that after the third principal component the change in variance almost diminishes. Therefore, first three components can be selected.