# Going Further with Scikit-Learn

## Overview

In this lab you’ll dig a bit deeper into the scikit-learn API, to see an example of unsupervised learning. This will complement the discussion of supervised learning in the chapter, where we showed how to use Naïve Bayes classifiers to detect and predict classifications for a dataset.

## Source folders

Student folder : PythonML\Student\04-ScikitLearnGoingFurther

Solution folder: PythonML\Solutions\04-ScikitLearnGoingFurther

## Roadmap

1. Getting started with sample data
2. Performing principal component analysis on the data
3. Plotting the components as vectors on top of the datapoints
4. (If time permits) Reducing the dimensionality of the data

## Introduction to dimensionality reduction

Machine learning is often applied to very large and complex datasets, containing millions of samples and maybe hundreds of features per sample. Faced with this quantity and richness of data, it can be difficult to detect relationships between features and to understand which features are more important than others.

In this kind of scenario, you will typically use **dimensionality reduction** to reduce the number of dimensions in your feature space. This can make it easier to spot underlying relationships between features, easier to visualize the data in a graph, and faster to process.

There are two ways to do dimensionality reduction:

* *Feature elimination –* Eliminate features that you think are superfluous or less important. For example, if you have 200 measured features, you might eliminate 190 of them, so that you can focus on just 10 of them. The advantage of feature elimination is that it simplifies your dataset, but the disadvantage is that you lose all information about the features you’ve eliminated.
* *Feature extraction* – This is a better approach. It uses all the existing features to create a new set of independent features, ranked according to their influence on the dataset. For example, imagine you have 200 features; you can extract these features into up to 200 independent new features, where each new feature is a combination of all of the 200 “old” features.

The cool thing about feature extraction is that you can decide to keep as many of the new independent features as you want, but drop the least important ones (with feature extraction, the features are ranked in order of their influence on the dataset, i.e. in how much a new feature contributes to the dataset values). For example, you might decide to keep the 10 most important new features, and drop the 190 least important ones. The critical thing is that all the new features are combinations of the old features, so you’re still keeping the most valuable parts of your old variables.

A common technique for performing feature extraction is **principal component analysis** **(PCA)**. PCA is a fast and flexible unsupervised learning algorithm and is well supported in scikit-learn. In a nutshell, it takes existing data and identifies components (i.e. axes) along which the data shows greatest variance. These components are all orthogonal to each other, i.e. independent of each other (in mathematical terminology, the components are mutually exclusive eigenvectors).

The components are ranked in order of importance/influence. The principal component is the line along which the data has greatest variance, i.e. this component exhibits the greatest influence on the dataset values; the next highest component exhibits the next highest variance, so it has less influence on the dataset values; and so on.

You decide how many components you want to retain – you can pick just the top few, and you’ll know these have the greatest influence on the data. For example, you could take a 200-feature dataset and use PCA to reduce it to the 10 principal components – these 10 principal components are like an x-y-z graph in 10-dimensionsal space.

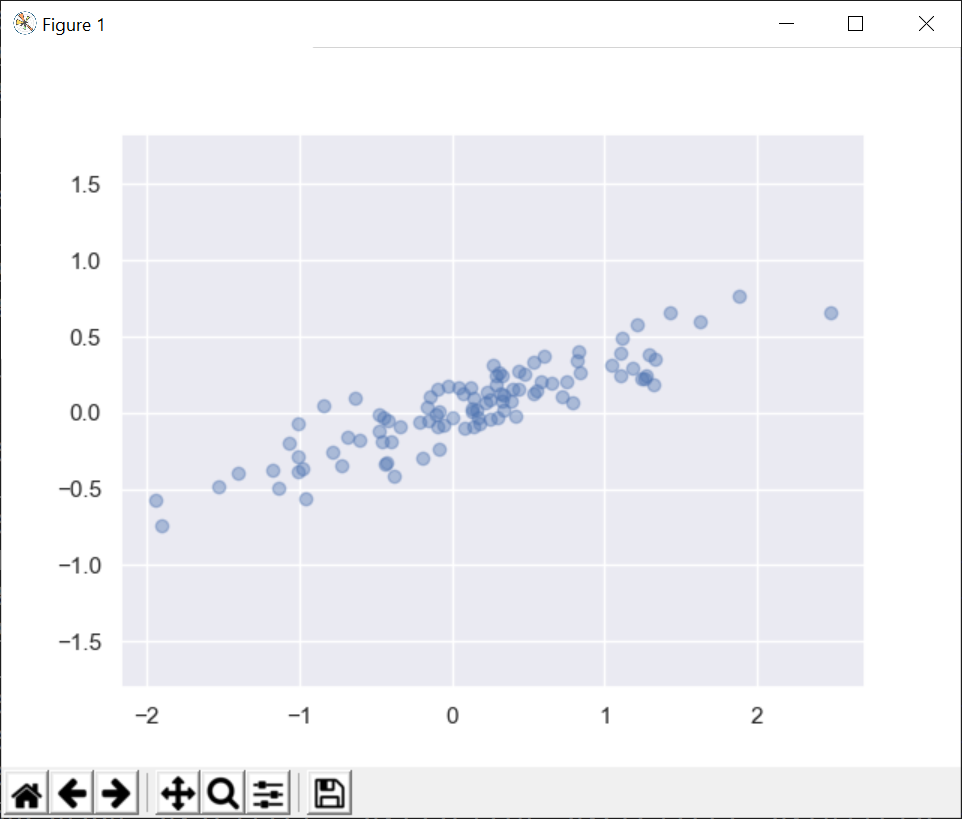
## Exercise 1: Getting started with sample data

In this exercise you’ll start working with some sample data, which will serve as a simple example of how to use PCA to reduce the dimensionality of a dataset. It will also help you understand how “components” work in PCA.

In the *student* folder, open reduceSimpleData.py in a text editor and review the code. The code does the following (see inline comments for details):

* Creates a 100-by-2 array of random numbers, representing 100 (x, y) points. In machine learning terminology, this is a dataset where n\_samples=100 and n\_features=2.
* Prints the entire array, i.e. 100 (x, y) values.
* Splices off column 0 in the array (i.e. the x values), and prints it.
* Splices off column 1 in the array (i.e. the y values) and prints it.
* Draws the 100 (x, y) points on a scatterplot graph.

Open a Command Prompt window in the *student* folder, and run the script file through the Python interpreter. The script displays the array values on the console, and also displays a scatterplot graph such as the following:



**Exercise 2: Performing principal component analysis on the data**

In this exercise you’ll use a **principal component analysis (PCA)** object from scikit-learn to model the relationship between the (x, y) features in the sample dataset.

You’ll configure the PCA object to identify 2 principal components for the data. These principal components will be like orthogonal vectors that show the 2 main axes along which the variance of values is greatest (for a better understanding of where this is heading, take a look at the graph on the next page, which shows these two components plotted as vectors on top of the datapoints).

So, let’s get started. The first thing to do is to import the PCA class from scikit-learn as follows:

from sklearn.decomposition import PCA

Next, create a scikit-learn PCA model object to find 2 principal components:

pca = PCA(n\_components=2)

n\_components can be any value up to n\_features for the dataset. Here, n\_components and n\_features are both 2, so you haven’t actually reduced the dimensionality (you’ll see how to do that later in the lab). You may be wondering if there’s any point in using PCA if you don’t reduce the dimensionality – the answer is “yes”, because PCA calculates the axes along which the data shows greatest variance, and this can help you understand the data better.

Now fit the PCA model object to the data, i.e. compute the 2 principal components for the data:

pca.fit(X)

After you've fitted the PCA model object to the data, the PCA object has two useful attributes:

* components\_  
  Array of shape (n\_components, n\_features), indicating how much each feature influences each principal component. The bigger the number, the bigger the influence that a feature has on a component. (A negative number indicates an inverse correlation between a feature and a component).
* explained\_variance\_  
  Array of shape (n\_components), indicating the amount of variance explained by each component. The first principal component explains the highest variance, the second principal component explains the next highest variance, and so on.

Print the value of these two attributes as follows:

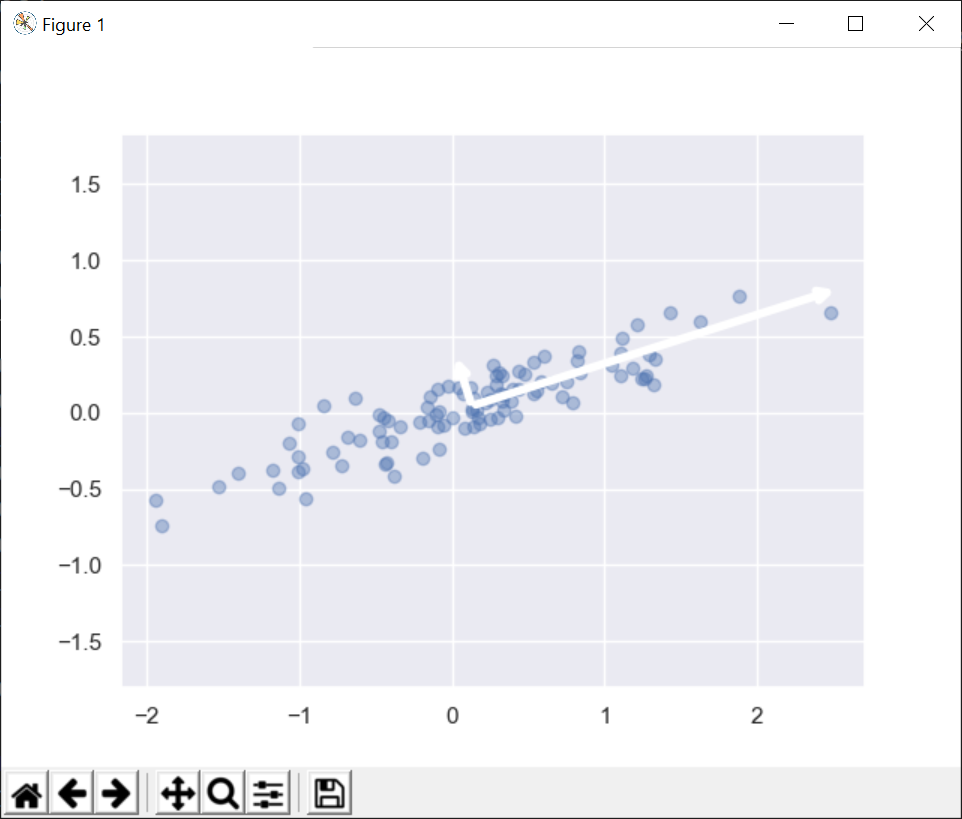
print("PCA components\_\n", pca.components\_)

print("PCA explained\_variance\_\n", pca.explained\_variance\_)

The absolute numbers don’t matter, it’s the relative values that matter (e.g. how much more significant is one feature than another on a given component, and what’s the relative importance of each component on the data).

**Exercise 3: Plotting the components as vectors on top of the datapoints**

In this exercise you’ll draw another scatterplot graph, with the 2 principal components drawn as vectors on top of the datapoints - see the diagram below. Note the white vectors – these represent the two principal components, and indicate the lines along which the data variance is greatest. The longer the vector, the greater the variance of data along this axis.



To help you out, we’ve provided a helper file named helpers.py. It has a function named draw\_vector(), which draws a vector from point p0 to point p1 on a graph. To be able to make use of this function, add the following statement in reduceSimpleData.py:

from helpers import draw\_vector

Now add the following code in reduceSimpleData.py, to draw a scatterplot graph showing the original datapoints again:

plt.scatter(X[:, 0], X[:, 1], alpha=0.4)

Now add the following code, to draw the 2 principal components as vectors on top of the datapoints (copy-and-paste this into your code, and see below for explanations):

for len, vec in zip(pca.explained\_variance, pca.components):

v = vec \* 3 \* np.sqrt(len)

draw\_vector(pca.mean\_, pca.mean\_ + v)

* pca.explained\_variance\_ is an array of shape (n\_components,). Each row contains a single value that indicates the variance (i.e. importance) of a component.
* pca.components\_ is an array of shape (n\_components, n\_features). Each row specifies the axis for a component, expressed in terms of the importance of the (x, y) features on that component. For example, if a row in pca.components\_ contains (5, 1) then the x feature is 5 times more important than the y feature for this component. We’ll therefore draw this component as a vector where the x length is 5 times longer than the y length.
* zip() zips these two arrays together into a collection of tuples that looks like this:

[

( comp0\_expVar, [comp0\_feat0, comp0\_feat1] ),

( comp1\_expVar, [comp1\_feat0, comp1\_feat1] )

]

* We loop through this collection of tuples, and assign an explained\_variance\_ to len and a component\_ to vec.
* Inside the loop, we do a bit of (empirically nice) math to scale the vector by its length (components with a larger variance will be drawn longer). We call draw\_vector() to draw the vector, starting at pca.mean (the computed mean point for the feature data).
* You can now show the graph (containing the datapoints and component vectors) as follows:

plt.axis('equal')

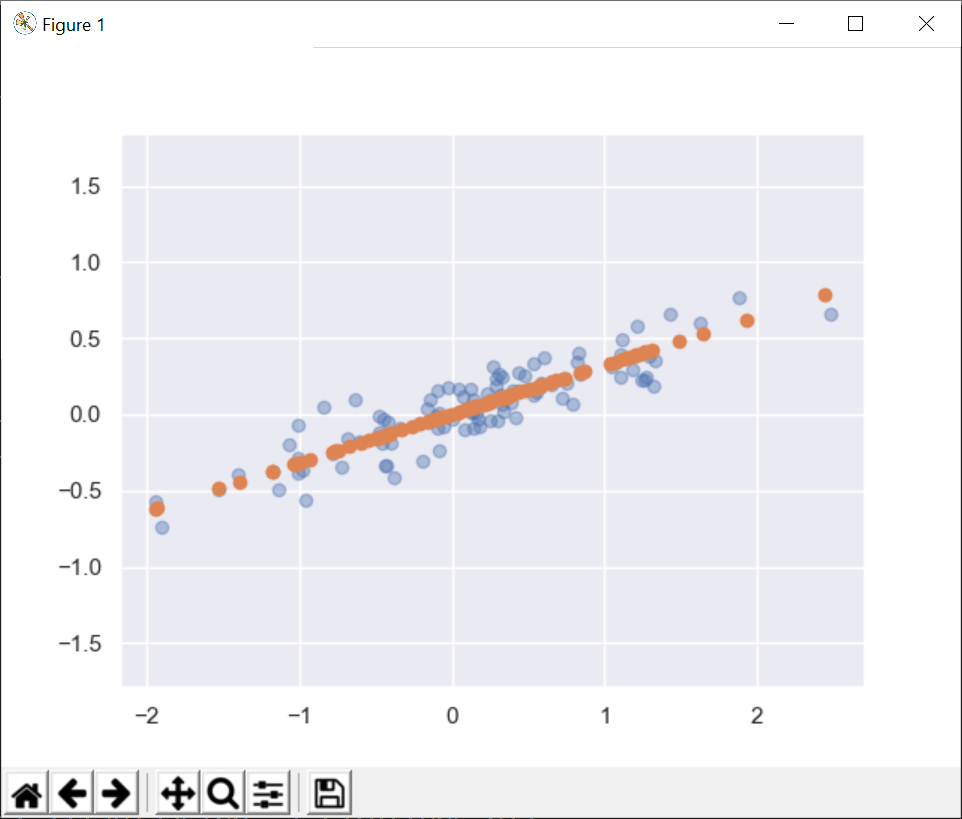
plt.show()

**Exercise 4 (If time permits): Reducing the dimensionality of the data**

Up until now, you’ve been using a PCA model object that computed 2 principal components for 2D data (x, y values).

Now, you’ll create a PCA model object with just 1 principal component – effectively the long vector in the previous graph. This is an example of dimensionality reduction, i.e. you will reduce 2D data (x, y values) into 1D data values along the principal component axis.

By the end of the exercise, you’ll have drawn the following graph (the dark points are the projections of the data points along the principal component axis, i.e. having removed the influence of the less important principal component axis):



Much of the code in this exercise will be similar to before, so we won’t explain everything again in glorious technicolor. We’ll focus on the new bits…

Follow these steps:

* Create a PCA model object that will compute 1 principal component.
* Fit the PCA object to the X dataset.
* Print the components\_ and explained\_variance\_ computed by the PCA object. How is this different from earlier in the lab (when you had 2 principal components)?
* Transform the original 2D dataset into a 1D dataset, by calling the transform() method on the PCA object as follows:

X\_pca = pca.transform(X)

Print X and X\_pca on the console. Note the following:

* X contains 2D values (in the x, y coordinate space).
* X\_pca contains 1D values (along the principal component axis).

This is dimensionality reduction – you’ve reduced 2D x, y values into 1D values. In so doing, you’ve discarded the less important component (axis). This is the whole point about PCA – focus on the most important components and ignore less important ones.

* If you want to plot the X\_pca values, you must convert them back into the 2D coordinate system used by matplotlib. To do this, call the inverse\_transform() method on the PCA object as follows:

X\_new = pca.inverse\_transform(X\_pca)

* Finally, draw two scatterplot graphs showing the values of X and X\_new. X contains the original (x, y) values, whereas X\_new indicates how these points project onto the main principal component (i.e. ignoring other principal components).