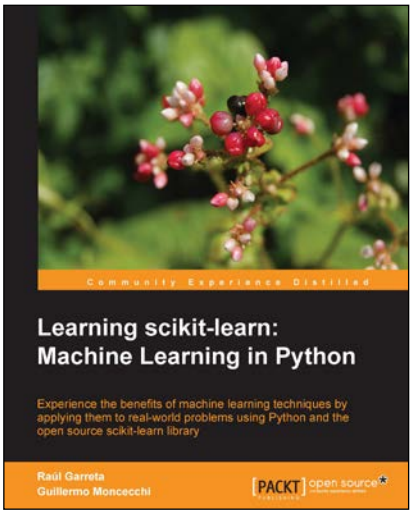
[PACKT] open source
community experience distilled
PUBLISHING

Learning scikit-learn: Machine Learning in Python

Raúl Garreta

Guillermo Moncecchi



Chapter No. 1

"Machine Learning – A Gentle Introduction"

In this package, you will find:

A Biography of the authors of the book

A preview chapter from the book, Chapter NO.1 "Machine Learning – A Gentle Introduction"

A synopsis of the book's content

Information on where to buy this book

About the Authors

Raúl Garreta is a Computer Engineer with much experience in the theory and application of Artificial Intelligence (AI), where he specialized in Machine Learning and Natural Language Processing (NLP).

He has an entrepreneur profile with much interest in the application of science, technology, and innovation to the Internet industry an d startups. He has worked in many software companies, handling everything from video games to implantable medical devices.

In 2009, he co-founded Tryolabs with the objective to apply AI to the development of intelligent software products, where he performs as the CTO and Product Manager of the company. Besides the application of Machine Learning and NLP, Tryolabs' expertise lies in the Python programming language and has been catering to many clients in Silicon Valley. Raul has also worked in the development of the Python community in Uruguay, co-organizing local PyDay and PyCon conferences.

He is also an assistant professor at the Computer Science Institute of Universidad de la República in Uruguay since 2007, where he has been working on the courses of Machine Learning, NLP, as well as Automata Theory and Formal Languages. Besides this, he is finishing his Masters degree in Machine Learning and NLP. He is also very interested in the research and application of Robotics, Quantum Computing, and Cognitive Modeling. Not only is he a technology enthusiast and science fiction lover (geek) but also a big fan of arts, such as cinema, photography, and painting.

I would like to thank my girlfriend for putting up with my long working sessions and always supporting me. Thanks to my parents, grandma, and aunt Pinky for their unconditional love and for always support my projects. Thanks to my friends and teammates at Tryolabs for always pushing me forward. Thanks Guillermo for joining me in writing this book. Thanks Diego Garat for introducing me to the amazing world of Machine Learning back in 2005.

Also, I would like to have a special mention to the open source Python and scikit-learn community for their dedication and professionalism in developing these beautiful tools.

Guillermo Moncecchi is a Natural Language Processing researcher at the Universidad de la República of Uruguay. He received a PhD in Informatics from the Universidad de la República, Uruguay and a Ph.D in Language Sciences from the Université Paris Ouest, France. He has participated in several international projects on NLP. He has almost 15 years of teaching experience on Automata Theory, Natural Language Processing, and Machine Learning.

He also works as Head Developer at the Montevideo Council and has lead the development of several public services for the council, particularly in the Geographical Information Systems area. He is one of the Montevideo Open Data movement leaders, promoting the publication and exploitation of the city's data.

I would like to thank my wife and kids for putting up with my late night writing sessions, and my family, for being there. You are the best I have.

Thanks to Javier Couto for his invaluable advice. Thanks to Raúl for inviting me to write this book. Thanks to all the people of the Natural Language Group and the Instituto de Computación at the Universidad de la República. I am proud of the great job we do every day building the uruguayan NLP and ML community.

Learning scikit-learn: Machine Learning in Python

Suppose you want to predict whether tomorrow will be a sunny or rainy day. You can develop an algorithm that is based on the current weather and your meteorological knowledge using a rather complicated set of rules to return the desired prediction. Now suppose that you have a record of the day-by-day weather conditions for the last five years, and you find that every time you had two sunny days in a row, the following day also happened to be a sunny one. Your algorithm could generalize this and predict that tomorrow will be a sunny day since the sun reigned today and yesterday. This algorithm is a pretty simple example of learning from experience. This is what **Machine Learning** is all about: algorithms that learn from the available data.

In this book, you will learn several methods for building Machine Learning applications that solve different real-world tasks, from document classification to image recognition.

We will use **Python**, a simple, popular, and widely used programming language, and **scikit-learn** an open source Machine Learning library.

In each chapter, we will present a different Machine Learning setting and a couple of well-studied methods as well as show step-by-step examples that use Python and scikit-learn to solve concrete tasks. We will also show you tips and tricks to improve algorithm performance, both from the accuracy and computational cost point of views.

What This Book Covers

Chapter 1, Machine Learning – A Gentle Introduction, presents the main concepts behind Machine Learning while solving a simple classification problem: discriminating flower species based on its characteristics.

Chapter 2, Supervised Learning, introduces four classification methods: Support Vector Machines, Naive Bayes, decision trees, and Random Forests. These methods are used to recognize faces, classify texts, and explain the causes for surviving from the Titanic accident. It also presents Linear Models and revisits Support Vector Machines and Random Forests, using them to predict house prices in Boston.

Chapter 3, Unsupervised Learning, describes methods for dimensionality reduction with Principal Component Analysis to visualize high dimensional data in just two dimensions. It also introduces clustering techniques to group instances of handwritten digits according to a similarity measure using the k-means algorithm.

Chapter 4, Advanced Features, shows how to preprocess the data and select the best features for learning, a task called Feature Selection. It also introduces Model Selection: selecting the best method parameters using the available data and parallel computation.

1
Machine Learning – A Gentle Introduction

"I was into data before it was big" – @ml_hipster

You have probably heard recently about big data. The Internet, the explosion of electronic devices with tremendous computational power, and the fact that almost every process in our world uses some kind of software, are giving us huge amounts of data every minute.

Think about social networks, where we store information about people, their interests, and their interactions. Think about process-control devices, ranging from web servers to cars and pacemakers, which permanently leave logs of data about their performance. Think about scientific research initiatives, such as the genome project, which have to analyze huge amounts of data about our DNA.

There are many things you can do with this data: examine it, summarize it, and even visualize it in several beautiful ways. However, this book deals with another use for data: as a source of experience to improve our algorithms' performance. These algorithms, which can learn from previous data, conform to the field of Machine Learning, a subfield of Artificial Intelligence.

Any machine learning problem can be represented with the following three concepts:

- We will have to learn to solve a task *T*. For example, build a spam filter that learns to classify e-mails as spam or ham.
- We will need some experience *E* to learn to perform the task. Usually, experience is represented through a dataset. For the spam filter, experience comes as a set of e-mails, manually classified by a human as spam or ham.
- We will need a measure of performance *P* to know how well we are solving the task and also to know whether after doing some modifications, our results are improving or getting worse. The percentage of e-mails that our spam filtering is correctly classifying as spam or ham could be P for our spam-filtering task.

Scikit-learn is an open source Python library of popular machine learning algorithms that will allow us to build these types of systems. The project was started in 2007 as a *Google Summer of Code* project by *David Cournapeau*. Later that year, *Matthieu Brucher* started working on this project as part of his thesis. In 2010, *Fabian Pedregosa*, *Gael Varoquaux*, *Alexandre Gramfort*, and *Vincent Michel* of INRIA took the project leadership and produced the first public release. Nowadays, the project is being developed very actively by an enthusiastic community of contributors. It is built upon NumPy (`http://www.numpy.org/`) and SciPy (`http://scipy.org/`), the standard Python libraries for scientific computation. Through this book, we will use it to show you how the incorporation of previous data as a source of experience could serve to solve several common programming tasks in an efficient and probably more effective way.

In the following sections of this chapter, we will start viewing how to install scikit-learn and prepare your working environment. After that, we will have a brief introduction to machine learning in a practical way, trying to introduce key machine learning concepts while solving a simple practical task.

Installing scikit-learn

Installation instructions for scikit-learn are available at `http://scikit-learn.org/stable/install.html`. Several examples in this book include visualizations, so you should also install the `matplotlib` package from `http://matplotlib.org/`. We also recommend installing IPython Notebook, a very useful tool that includes a web-based console to edit and run code snippets, and render the results. The source code that comes with this book is provided through IPython notebooks.

An easy way to install all packages is to download and install the Anaconda distribution for scientific computing from `https://store.continuum.io/`, which provides all the necessary packages for Linux, Mac, and Windows platforms. Or, if you prefer, the following sections gives some suggestions on how to install every package on each particular platform.

Linux

Probably the easiest way to install our environment is through the operating system packages. In the case of Debian-based operating systems, such as Ubuntu, you can install the packages by running the following commands:

- Firstly, to install the package we enter the following command:

```
# sudo apt-get install build-essential python-dev python-numpy
python-setuptools python-scipy libatlas-dev
```

- Then, to install matplotlib, run the following command:

```
# sudo apt-get install python-matplotlib
```

- After that, we should be ready to install scikit-learn by issuing this command:

```
# sudo pip install scikit-learn
```

- To install IPython Notebook, run the following command:

```
# sudo apt-get install ipython-notebook
```

- If you want to install from source, let's say to install all the libraries within a virtual environment, you should issue the following commands:

```
# pip install numpy
# pip install scipy
# pip install scikit-learn
```

- To install Matplotlib, you should run the following commands:

```
# pip install libpng-dev libjpeg8-dev libfreetype6-dev
# pip install matplotlib
```

- To install IPython Notebook, you should run the following commands:

```
# pip install ipython
# pip install tornado
# pip install pyzmq
```

Mac

You can similarly use tools such as MacPorts and HomeBrew that contain precompiled versions of these packages.

Windows

To install scikit-learn on Windows, you can download a Windows installer from the downloads section of the project web page: `http://sourceforge.net/projects/scikit-learn/files/`

Checking your installation

To check that everything is ready to run, just open your Python (or probably better, IPython) console and type the following:

```
>>> import sklearn as sk
>>> import numpy as np
>>> import matplotlib.pyplot as plt
```

We have decided to precede Python code with `>>>` to separate it from the sentence results. Python will silently import the scikit-learn, NumPy, and matplotlib packages, which we will use through the rest of this book's examples.

If you want to execute the code presented in this book, you should run IPython Notebook:

```
# ipython notebook
```

This will allow you to open the corresponding notebooks right in your browser.

Datasets

As we have said, machine learning methods rely on previous experience, usually represented by a dataset. Every method implemented on scikit-learn assumes that data comes in a dataset, a certain form of input data representation that makes it easier for the programmer to try different methods on the same data. Scikit-learn includes a few well-known datasets. In this chapter, we will use one of them, the Iris flower dataset, introduced in 1936 by *Sir Ronald Fisher* to show how a statistical method (discriminant analysis) worked (yes, they were into data before it was big). You can find a description of this dataset on its own Wikipedia page, but, essentially, it includes information about 150 elements (or, in machine learning terminology, instances) from three different Iris flower species, including sepal and petal length and width. The natural task to solve using this dataset is to learn to guess the Iris species knowing the sepal and petal measures. It has been widely used on machine learning tasks because it is a very easy dataset in a sense that we will see later. Let's import the dataset and show the values for the first instance:

```
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
>>> X_iris, y_iris = iris.data, iris.target
>>> print X_iris.shape, y_iris.shape
  (150, 4) (150,)
>>> print X_iris[0], y_iris[0]
  [ 5.1  3.5  1.4  0.2] 0
```

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

We can see that the `iris` dataset is an object (similar to a dictionary) that has two main components:

- A `data` array, where, for each instance, we have the real values for sepal length, sepal width, petal length, and petal width, in that order (note that for efficiency reasons, scikit-learn methods work on NumPy `ndarrays` instead of the more descriptive but much less efficient Python dictionaries or lists). The shape of this array is `(150, 4)`, meaning that we have 150 rows (one for each instance) and four columns (one for each feature).
- A `target` array, with values in the range of 0 to 2, corresponding to each instance of Iris species (0: setosa, 1: versicolor, and 2: virginica), as you can verify by printing the `iris.target.target_names` value.

While it's not necessary for every dataset we want to use with scikit-learn to have this exact structure, we will see that every method will require this data array, where each instance is represented as a list of features or attributes, and another target array representing a certain value we want our learning method to learn to predict. In our example, the petal and sepal measures are our real-valued attributes, while the flower species is the one-of-a-list class we want to predict.

Our first machine learning method – linear classification

To get a grip on the problem of machine learning in scikit-learn, we will start with a very simple machine learning problem: we will try to predict the Iris flower species using only two attributes: sepal width and sepal length. This is an instance of a classification problem, where we want to assign a label (a value taken from a discrete set) to an item according to its features.

Let's first build our training dataset—a subset of the original sample, represented by the two attributes we selected and their respective target values. After importing the dataset, we will randomly select about 75 percent of the instances, and reserve the remaining ones (the evaluation dataset) for evaluation purposes (we will see later why we should always do that):

```
>>> from sklearn.cross_validation import train_test_split
>>> from sklearn import preprocessing
>>> # Get dataset with only the first two attributes
>>> X, y = X_iris[:, :2], y_iris
>>> # Split the dataset into a training and a testing set
>>> # Test set will be the 25% taken randomly
>>> X_train, X_test, y_train, y_test = train_test_split(X, y,
  test_size=0.25, random_state=33)
>>> print X_train.shape, y_train.shape
  (112, 2) (112,)
>>> # Standardize the features
>>> scaler = preprocessing.StandardScaler().fit(X_train)
>>> X_train = scaler.transform(X_train)
>>> X_test = scaler.transform(X_test)
```

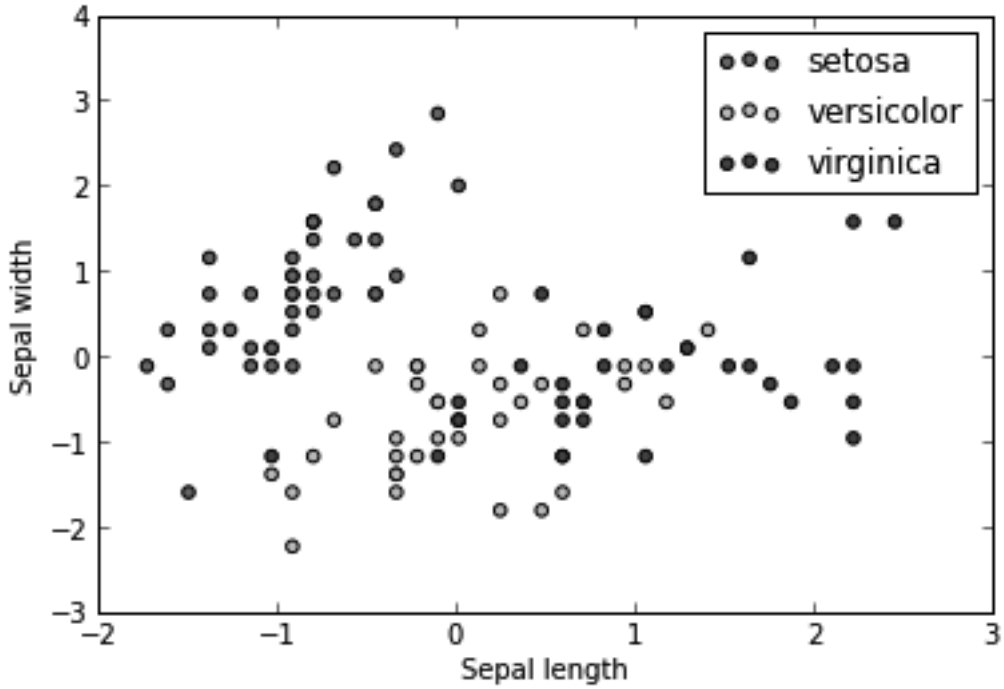
The `train_test_split` function automatically builds the training and evaluation datasets, randomly selecting the samples. Why not just select the first 112 examples? This is because it could happen that the instance ordering within the sample could matter and that the first instances could be different to the last ones. In fact, if you look at the Iris datasets, the instances are ordered by their target class, and this implies that the proportion of `0` and `1` classes will be higher in the new training set, compared with that of the original dataset. We always want our training data to be a representative sample of the population they represent.

The last three lines of the previous code modify the training set in a process usually called feature scaling. For each feature, calculate the average, subtract the mean value from the feature value, and divide the result by their standard deviation. After scaling, each feature will have a zero average, with a standard deviation of one. This standardization of values (which does not change their distribution, as you could verify by plotting the `X` values before and after scaling) is a common requirement of machine learning methods, to avoid that features with large values may weight too much on the final results.

Now, let's take a look at how our training instances are distributed in the two-dimensional space generated by the learning feature. `pyplot`, from the matplotlib library, will help us with this:

```
>>> import matplotlib.pyplot as plt
>>> colors = ['red', 'greenyellow', 'blue']
>>> for i in xrange(len(colors)):
>>>     xs = X_train[:, 0][y_train == i]
>>>     ys = X_train[:, 1][y_train == i]
>>>     plt.scatter(xs, ys, c=colors[i])
>>> plt.legend(iris.target_names)
>>> plt.xlabel('Sepal length')
>>> plt.ylabel('Sepal width')
```

The `scatter` function simply plots the first feature value (sepal width) for each instance versus its second feature value (sepal length) and uses the target class values to assign a different color for each class. This way, we can have a pretty good idea of how these attributes contribute to determine the target class. The following screenshot shows the resulting plot:



Looking at the preceding screenshot, we can see that the separation between the red dots (corresponding to the Iris setosa) and green and blue dots (corresponding to the two other Iris species) is quite clear, while separating green from blue dots seems a very difficult task, given the two features available. This is a very common scenario: one of the first questions we want to answer in a machine learning task is if the feature set we are using is actually useful for the task we are solving, or if we need to add new attributes or change our method.

Given the available data, let's, for a moment, redefine our learning task: suppose we aim, given an Iris flower instance, to predict if it is a setosa or not. We have converted our problem into a binary classification task (that is, we only have two possible target classes).

If we look at the picture, it seems that we could draw a straight line that correctly separates both the sets (perhaps with the exception of one or two dots, which could lie in the incorrect side of the line). This is exactly what our first classification method, linear classification models, tries to do: build a line (or, more generally, a hyperplane in the feature space) that best separates both the target classes, and use it as a decision boundary (that is, the class membership depends on what side of the hyperplane the instance is).

To implement linear classification, we will use the SGDClassifier from scikit-learn. **SGD** stands for **Stochastic Gradient Descent**, a very popular numerical procedure to find the local minimum of a function (in this case, the loss function, which measures how far every instance is from our boundary). The algorithm will learn the coefficients of the hyperplane by minimizing the loss function.

To use any method in scikit-learn, we must first create the corresponding classifier object, initialize its parameters, and train the model that better fits the training data. You will see while you advance in this book that this procedure will be pretty much the same for what initially seemed very different tasks.

```
>>> from sklearn.linear_modelsklearn._model import SGDClassifier
>>> clf = SGDClassifier()
>>> clf.fit(X_train, y_train)
```

The `SGDClassifier` initialization function allows several parameters. For the moment, we will use the default values, but keep in mind that these parameters could be very important, especially when you face more real-world tasks, where the number of instances (or even the number of attributes) could be very large. The `fit` function is probably the most important one in scikit-learn. It receives the training data and the training classes, and builds the classifier. Every supervised learning method in scikit-learn implements this function.

What does the classifier look like in our linear model method? As we have already said, every future classification decision depends just on a hyperplane. That hyperplane is, then, our model. The `coef_` attribute of the `clf` object (consider, for the moment, only the first row of the matrices), now has the coefficients of the linear boundary and the `intercept_` attribute, the point of intersection of the line with the y axis. Let's print them:

```
>>> print clf.coef_
[[-28.53692691  15.05517618]
  [ -8.93789454  -8.13185613]
  [ 14.02830747 -12.80739966]]
>>> print clf.intercept_
[-17.62477802  -2.35658325  -9.7570213 ]
```

Indeed in the real plane, with these three values, we can draw a line, represented by the following equation:

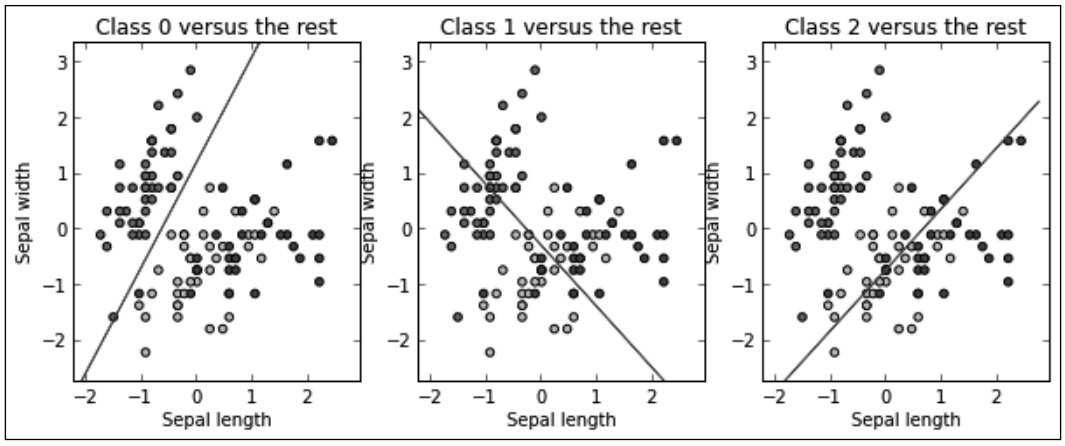
-17.62477802 - 28.53692691 * x1 + 15.05517618 * x2 = 0

Now, given *x1* and *x2* (our real-valued features), we just have to compute the value of the left-side of the equation: if its value is greater than zero, then the point is above the decision boundary (the red side), otherwise it will be beneath the line (the green or blue side). Our prediction algorithm will simply check this and predict the corresponding class for any new iris flower.

But, why does our coefficient matrix have three rows? Because we did not tell the method that we have changed our problem definition (how could we have done this?), and it is facing a three-class problem, not a binary decision problem. What, in this case, the classifier does is the same we did—it converts the problem into three binary classification problems in a one-versus-all setting (it proposes three lines that separate a class from the rest).

The following code draws the three decision boundaries and lets us know if they worked as expected:

```
>>> x_min, x_max = X_train[:, 0].min() - .5, X_train[:, 0].max() +
    .5
>>> y_min, y_max = X_train[:, 1].min() - .5, X_train[:, 1].max() +
    .5
>>> xs = np.arange(x_min, x_max, 0.5)
>>> fig, axes = plt.subplots(1, 3)
>>> fig.set_size_inches(10, 6)
>>> for i in [0, 1, 2]:
>>>     axes[i].set_aspect('equal')
>>>     axes[i].set_title('Class '+ str(i) + ' versus the rest')
>>>     axes[i].set_xlabel('Sepal length')
>>>     axes[i].set_ylabel('Sepal width')
>>>     axes[i].set_xlim(x_min, x_max)
>>>     axes[i].set_ylim(y_min, y_max)
>>>     sca(axes[i])
>>>     plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train,
        cmap=plt.cm.prism)
>>>     ys = (-clf.intercept_[i] -
        Xs * clf.coef_[i, 0]) / clf.coef_[i, 1]
>>>     plt.plot(xs, ys, hold=True)
```



The first plot shows the model built for our original binary problem. It looks like the line separates quite well the Iris setosa from the rest. For the other two tasks, as we expected, there are several points that lie on the wrong side of the hyperplane.

Now, the end of the story: suppose that we have a new flower with a sepal width of 4.7 and a sepal length of 3.1, and we want to predict its class. We just have to apply our brand new classifier to it (after normalizing!). The predict method takes an array of instances (in this case, with just one element) and returns a list of predicted classes:

```
>>>print clf.predict(scaler.transform([[4.7, 3.1]]))
[0]
```

If our classifier is right, this Iris flower is a setosa. Probably, you have noticed that we are predicting a class from the possible three classes but that linear models are essentially binary: something is missing. You are right. Our prediction procedure combines the result of the three binary classifiers and selects the class in which it is more confident. In this case, we will select the boundary line whose distance to the instance is longer. We can check that using the classifier `decision_function` method:

```
>>>print clf.decision_function(scaler.transform([[4.7, 3.1]]))
[[ 19.73905808   8.13288449 -28.63499119]]
```

Evaluating our results

We want to be a little more formal when we talk about a good classifier. What does that mean? The performance of a classifier is a measure of its effectiveness. The simplest performance measure is accuracy: given a classifier and an evaluation dataset, it measures the proportion of instances correctly classified by the classifier. First, let's test the accuracy on the training set:

```
>>> from sklearn import metrics
>>> y_train_pred = clf.predict(X_train)
>>> print metrics.accuracy_score(y_train, y_train_pred)
0.821428571429
```

This figure tells us that 82 percent of the training set instances are correctly classified by our classifier.

Probably, the most important thing you should learn from this chapter is that measuring accuracy on the training set is really a bad idea. You have built your model using this data, and it is possible that your model adjusts well to them but performs poorly in future (previously unseen data), which is its purpose. This phenomenon is called **overfitting**, and you will see it now and again while you read this book. If you measure based on your training data, you will never detect overfitting. So, never measure based on your training data.

This is why we have reserved part of the original dataset (the testing partition)—we want to evaluate performance on previously unseen data. Let's check the accuracy again, now on the evaluation set (recall that it was already scaled):

```
>>> y_pred = clf.predict(X_test)
>>> print metrics.accuracy_score(y_test, y_pred)
0.684210526316
```

We obtained an accuracy of 68 percent in our testing set. Usually, accuracy on the testing set is lower than the accuracy on the training set, since the model is actually modeling the training set, not the testing set. Our goal will always be to produce models that avoid overfitting when trained over a training set, so they have enough generalization power to also correctly model the unseen data.

Accuracy on the test set is a good performance measure when the number of instances of each class is similar, that is, we have a uniform distribution of classes. But if you have a skewed distribution (say, 99 percent of the instances belong to one class), a classifier that always predicts the majority class could have an excellent performance in terms of accuracy despite the fact that it is an extremely naive method.

Within scikit-learn, there are several evaluation functions; we will show three popular ones: precision, recall, and F1-score (or f-measure). They assume a binary classification problem and two classes—a positive one and a negative one. In our example, the positive class could be Iris setosa, while the other two will be combined into one negative class.

- **Precision**: This computes the proportion of instances predicted as positives that were correctly evaluated (it measures how right our classifier is when it says that an instance is positive).
- **Recall**: This counts the proportion of positive instances that were correctly evaluated (measuring how right our classifier is when faced with a positive instance).
- **F1-score**: This is the harmonic mean of precision and recall, and tries to combine both in a single number.

The harmonic mean is used instead of the arithmetic mean because the latter compensates low values for precision and with high values for recall (and vice versa). On the other hand, with harmonic mean we will always have low values if either precision or recall is low. For an interesting description of this issue refer to the paper `http://www.cs.odu.edu/~mukka/cs795sum12dm/Lecturenotes/Day3/F-measure-YS-26Oct07.pdf`

We can define these measures in terms of True and False, and Positives and Negatives:

	Prediction: Positive	**Prediction: Negative**
Target cass: Positive	True Positive (TP)	False Negative (FN)
Target cass: Negative	False Positive (FP)	True Negative (TN)

With m being the sample size (that is, TP + TN + FP + FN), we have the following formulae:

- Accuracy = (TP + TN) / m
- Precision = TP / (TP + FP)
- Recall = TP / (TP + FN)
- F1-score = 2 * Precision * Recall / (Precision + Recall)

Let's see it in practice:

```
>>> print metrics.classification_report(y_test, y_pred,
  target_names=iris.target_names)
              precision    recall  f1-score   support

setosa           1.00          1.00      1.00         8
versicolor       0.43          0.27      0.33        11
virginica        0.65          0.79      0.71        19

avg / total      0.66          0.68      0.66        38
```

We have computed precision, recall, and f1-score for each class and their average values. What we can see in this table is:

- The classifier obtained 1.0 precision and recall in the `setosa` class. This means that for precision, 100 percent of the instances that are classified as setosa are really setosa instances, and for recall, that 100 percent of the setosa instances were classified as setosa.
- On the other hand, in the `versicolor` class, the results are not as good: we have a precision of 0.43, that is, only 43 percent of the instances that are classified as versicolor are really versicolor instances. Also, for versicolor, we have a recall of 0.27, that is, only 27 percent of the versicolor instances are correctly classified.

Now, we can see that our method (as we expected) is very good at predicting `setosa`, while it suffers when it has to separate the `versicolor` or `virginica` classes. The support value shows how many instances of each class we had in the testing set.

Another useful metric (especially for multi-class problems) is the confusion matrix: in its `(i, j)` cell, it shows the number of class instances `i` that were predicted to be in class `j`. A good classifier will accumulate the values on the confusion matrix diagonal, where correctly classified instances belong.

```
>>> print metrics.confusion_matrix(y_test, y_pred)
[[ 8  0  0]
[ 0  3  8]
[ 0  4 15]]
```

Our classifier is never wrong in our evaluation set when it classifies class `0` (`setosa`) flowers. But, when it faces classes `1` and `2` flowers (`versicolor` and `virginica`), it confuses them. The confusion matrix gives us useful information to know what types of errors the classifier is making.

To finish our evaluation process, we will introduce a very useful method known as cross-validation. As we explained before, we have to partition our dataset into a training set and a testing set. However, partitioning the data, results such that there are fewer instances to train on, and also, depending on the particular partition we make (usually made randomly), we can get either better or worse results. Cross-validation allows us to avoid this particular case, reducing result variance and producing a more realistic score for our models. The usual steps for k-fold cross-validation are the following:

1. Partition the dataset into *k* different subsets.
2. Create *k* different models by training on k-1 subsets and testing on the remaining subset.
3. Measure the performance on each of the *k* models and take the average measure.

Let's do that with our linear classifier. First, we will have to create a composite estimator made by a pipeline of the standardization and linear models. With this technique, we make sure that each iteration will standardize the data and then train/test on the transformed data. The `Pipeline` class is also useful to simplify the construction of more complex models that chain-multiply the transformations. We will chose to have k = 5 folds, so each time we will train on 80 percent of the data and test on the remaining 20 percent. Cross-validation, by default, uses accuracy as its performance measure, but we could select the measurement by passing any scorer function as an argument.

```
>>> from sklearn.cross_validation import cross_val_score, KFold
>>> from sklearn.pipeline import Pipeline
>>> # create a composite estimator made by a pipeline of the
    standarization and the linear model
>>> clf = Pipeline([
        ('scaler', StandardScaler()),
        ('linear_model', SGDClassifier())
])
>>> # create a k-fold cross validation iterator of k=5 folds
>>> cv = KFold(X.shape[0], 5, shuffle=True, random_state=33)
>>> # by default the score used is the one returned by score
    method of the estimator (accuracy)
>>> scores = cross_val_score(clf, X, y, cv=cv)
>>> print scores
[ 0.66666667  0.93333333  0.66666667  0.7         0.6       ]
```

We obtained an array with the *k* scores. We can calculate the mean and the standard error to obtain a final figure:

```
>>> from scipy.stats import sem
>>> def mean_score(scores):
    return ("Mean score: {0:.3f} (+/-
    {1:.3f})").format(np.mean(scores), sem(scores))
>>> print mean_score(scores)
Mean score: 0.713 (+/-0.057)
```

Our model has an average accuracy of 0.71.

Machine learning categories

Classification is only one of the possible machine learning problems that can be addressed with scikit-learn. We can organize them in the following categories:

- In the previous example, we had a set of instances (that is, a set of data collected from a population) represented by certain features and with a particular target attribute. Supervised learning algorithms try to build a model from this data, which lets us predict the target attribute for new instances, knowing only these instance features. When the target class belongs to a discrete set (such as a list of flower species), we are facing a classification problem.
- Sometimes the class we want to predict, instead of belonging to a discrete set, ranges on a continuous set, such as the real number line. In this case, we are trying to solve a **regression** problem (the term was coined by Francis Galton, who observed that the heights of tall ancestors tend to regress down towards a normal value, the average human height). For example, we could try to predict the petal width based on the other three features. We will see that the methods used for regression are quite different from those used for classification.
- Another different type of machine learning problem is that of **unsupervised learning**. In this case, we do not have a target class to predict but instead want to group instances according to some similarity measure based on the available set of features. For example, suppose you have a dataset composed of e-mails and want to group them by their main topic (the task of grouping instances is called **clustering**). We can use it as features, for example, the different words used in each of them.

Important concepts related to machine learning

The linear classifier we presented in the previous section could look too simple. What if we use a higher degree polynomial? What if we also take as features not only the sepal length and width, but also the petal length and the petal width? This is perfectly possible, and depending on the sample distribution, it could lead to a better fit to the training data, resulting in higher accuracy. The problem with this approach is that now we must estimate not only the three original parameters (the coefficients for *x1*, *x2*, and the interception point), but also the parameters for the new features *x3* and *x4* (petal length and width) and also the product combinations of the four features.

Intuitively, we would need more training data to adequately estimate these parameters. The number of parameters (and consequently, the amount of training data needed to adequately estimate them) would rapidly grow if we add more features or higher order terms. This phenomenon, present in every machine learning method, is called the idem curse of dimensionality: when the number of parameters of a model grows, the data needed to learn them grows exponentially.

This notion is closely related to the problem of overfitting mentioned earlier. As our training data is not enough, we risk producing a model that could be very good at predicting the target class on the training dataset but fail miserably when faced with new data, that is, our model does not have the generalization power. That is why it is so important to evaluate our methods on previously unseen data.

The general rule is that, in order to avoid overfitting, we should prefer simple (that is, with less parameters) methods, something that could be seen as an instantiation of the philosophical principle of Occam's razor, which states that among competing hypotheses, the hypothesis with the fewest assumptions should be selected.

However, we should also take into account Einstein's words:

> "*Everything should be made as simple as possible, but not simpler.*"

The idem curse of dimensionality may suggest that we keep our models simple, but on the other hand, if our model is too simple we run the risk of suffering from underfitting. Underfitting problems arise when our model has such a low representation power that it cannot model the data even if we had all the training data we want. We clearly have underfitting when our algorithm cannot achieve good performance measures even when measuring on the training set.

As a result, we will have to achieve a balance between overfitting and underfitting. This is one of the most important problems that we will have to address when designing our machine learning models.

Other key concepts to take into account are the idem bias and variance of a machine learning method. Consider an extreme method that, in a binary classification setting, always predicts the positive class for any new instance. Its predictions are, trivially, always the same, or in statistical terms, it has null variance; but it will fail to predict negative examples: it is very biased towards positive results. On the other hand, consider a method that predicts, for a new instance, the class of the nearest instance in the training set (in fact, this method exists, and it is called the 1-nearest neighbor). The generalization assumptions that this method uses are very small: it has a very low bias; but, if we change the training data, results could dramatically change, that is, its variance is very high. These are extreme examples of the **bias-variance tradeoff**. It can be shown that, no matter which method we are using, if we reduce bias, variance will increase, and vice versa.

Linear classifiers have generally low-variance: no matter what subset we select for training, results will be similar. However, if the data distribution (as in the case of the versicolor and virginica species) makes target classes not separable by a hyperplane, these results will be consistently wrong, that is, the method is highly biased.

On the other hand, kNN (a memory-based method we will not address in this book) has very low bias but high variance: the results are generally very good at describing training data but tend to vary greatly when trained on different training instances.

There are other important concepts related to real-world applications where our data will not come naturally as a list of real-valued features. In these cases, we will need to have methods to transform non real-valued features to real-valued ones. Besides, there are other steps related to feature standardization and normalization, which as we saw in our Iris example, are needed to avoid undesired effects regarding the different value ranges. These transformations on the feature space are known as **data preprocessing**.

After having a defined feature set, we will see that not all of the features that come in our original dataset could be useful for resolving our task. So we must also have methods to do feature selection, that is, methods to select the most promising features.

In this book, we will present several problems and in each of them we will show different ways to transform and find the most relevant features to use for learning a task, called **feature engineering**, which is based on our knowledge of the domain of the problem and/or data analysis methods. These methods, often not valued enough, are a fundamental step toward obtaining good results.

Summary

In this chapter, we introduced the main general concepts in machine learning and presented scikit-learn, the Python library we will use in the rest of this book. We included a very simple example of classification, trying to show the main steps for learning, and including the most important evaluation measures we will use. In the rest of this book, we plan to show you different machine learning methods and techniques using different real-world examples for each one. In almost every computational task, the presence of historical data could allow us to improve performance in the sense introduced at the beginning of this chapter.

The next chapter introduces supervised learning methods: we have annotated data (that is, instances where the target class/value is known) and we want to predict the same class/value for future data from the same population. In the case of classification tasks, that is, a discrete-valued target class, several different models exist, ranging from statistical methods, such as the simple **Naïve Bayes** to advanced linear classifiers, such as **Support Vector Machines (SVM)**. Some methods, such as **decision trees**, will allow us to visualize how important a feature is to discriminate between different target classes and have a human interpretation of the decision process. We will also address another type of supervised learning task: regression, that is, methods that try to predict real-valued data.

Where to buy this book

You can buy Learning scikit-learn: Machine Learning in Python from the Packt Publishing website: `http://www.packtpub.com/learning-scikit-learn-machine-in-python/book`.

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

www.PacktPub.com

For More Information:
www.packtpub.com/learning-scikit-learn-machine-in-python/book