CHAPTER 1

Introduction

Machine learning is about extracting knowledge from data. It is a research field at the intersection of statistics, artificial intelligence and computer science, which is also known as predictive analytics or statistical learning. The application of machine learning methods has in recent years become ubiquitous in everyday life. From automatic recommendations of which movies to watch, to what food to order or which products to buy, to personalized online radio and recognizing your friends in your photos, many modern websites and devices have machine learning algorithms at their core.

When you look at at complex websites like Facebook, Amazon or Netflix, it is very likely that every part of the website you are looking at contains multiple machine learning models.

Outside of commercial applications, machine learning has had a tremendous influence on the way data driven research is done today. The tools introduced in this book have been applied to diverse scientific questions such as understanding stars, finding distant planets, analyzing DNA sequences, and providing personalized cancer treatments.

Your application doesn't need to be as large-scale or world-changing as these examples in order to benefit from machine learning. In this chapter, we will explain why machine learning became so popular, and dicuss what kind of problem can be solved using machine learning. Then, we will show you how to build your first machine learning model, introducing important concepts on the way.

Why machine learning?

In the early days of "intelligent" applications, many systems used hand-coded rules of "if" and "else" decisions to process data or adjust to user input. Think of a spam filter

whose job is to move an email to a spam folder. You could make up a black-list of words that would result in an email marked as spam. This would be an example of using an expert designed rule system to design an "intelligent" application. Designing kind of manual design of decision rules is feasible for some applications, in particular for those applications in which humans have a good understanding of how a decision should be made. However, using hand-coded rules to make decisions has two major disadvantages:

1. The logic required to make a decision is specific to a single domain and task. Changing the task even slightly might require a rewrite of the whole system.
2. Designing rules requires a deep understanding of how a decision should be made by a human expert.

One example of where this hand-coded approach will fail is in detecting faces in images. Today every smart phone can detect a face in an image. However, face detection was an unsolved problem until as recent as 2001. The main problem is that the way in which pixels (which make up an image in a computer) are "perceived by" the computer is very different from how humans perceive a face. This difference in representation makes it basically impossible for a human to come up with a good set of rules to describe what constitutes a face in a digital image.

Using machine learning, however, simply presenting a program with a large collection of images of faces is enough for an algorithm to determine what characteristics are needed to identify a face.

Problems that machine learning can solve

The most successful kind of machine learning algorithms are those that automate a decision making processes by generalizing from known examples. In this setting, which is known as a *supervised learning* setting, the user provides the algorithm with pairs of inputs and desired outputs, and the algorithm finds a way to produce the desired output given an input.

In particular, the algorithm is able to create an output for an input it has never seen before without any help from a human.

Going back to our example of spam classification, using machine learning, the user provides the algorithm a large number of emails (which are the input), together with the information about whether any of these emails are spam (which is the desired output). Given a new email, the algorithm will then produce a prediction as to whether or not the new email is spam.

Machine learning algorithms that learn from input-output pairs are called supervised learning algorithms because a "teacher" provides supervision to the algorithm in the form of the desired outputs for each example that they learn from.

While creating a dataset of inputs and outputs is often a laborious manual process, supervised learning algorithms are well-understood and their performance is easy to measure. If your application can be formulated as a supervised learning problem, and you are able to create a dataset that includes the desired outcome, machine learning will likely be able to solve your problem.

Examples of supervised machine learning tasks include:

- **Identifying the ZIP code from handwritten digits on an envelope.** Here the input is a scan of the handwriting, and the desired output is the actual digits in the zip code. To create a data set for building a machine learning model, you need to collect many envelopes. Then you can read the zip codes yourself and store the digits as your desired outcomes.

- **Determining whether or not a tumor is benign based on a medical image.** Here the input is the image, and the output is whether or not the tumor is benign. To create a data set for building a model, you need a database of medical images. You also need an expert opinion, so a doctor needs to look at all of the images and decide which tumors are benign and which are not.

- **Detecting fraudulent activity in credit card transactions.** Here the input is a record of the credit card transaction, and the output is whether it is likely to be fraudulent or not. Assuming that you are the entity distributing the credit cards, collecting a dataset means storing all transactions, and recording if a user reports any transaction as fraudulent.

An interesting thing to note about the three examples above is that although the inputs and outputs look fairly straight-forward, the data collection process for these three tasks is vastly different.

While reading envelopes is laborious, it is easy and cheap. Obtaining medical imaging and expert opinions on the other hand not only requires expensive machinery but also rare and expensive expert knowledge, not to mention ethical concerns and privacy issues. In the example of detecting credit card fraud, data collection is much simpler. Your customers will provide you with the desired output, as they will report fraud. All you have to do to obtain the input output pairs of fraudulent and non-fraudulent activity is wait.

The other type of algorithms that we will cover in this book is unsupervised algorithms. In unsupervised learning, only the input data is known and there is no known output data given to the algorithm. While there are many successful applications of these methods as well, they are usually harder to understand and evaluate.

Examples of unsupervised learning include:

- **Identifying topics in a set of blog posts.** If you have a large collection of text data, you might want to summarize it and find prevalent themes in it. You might not know beforehand what these topics are, or how many topics there might be. Therefore, there are no known outputs.

- **Segmenting customers into groups with similar preferences.** Given a set of customer records, you might want to identify which customers are similar, and whether there are groups of customers with similar preferences. For a shopping site these might be "parents", "bookworms" or "gamers". Since you don't know in advanced what these groups might be, or even how many there are, you have no known outputs.

- **Detecting abnormal access patterns to a website.** To identify abuse or bugs, it is often helpful to find access patterns that are different from the norm. Each abnormal pattern might be very different, and you might not have any recorded instances of abnormal behavior. Since in this example you only observe traffic, and you don't know what constitutes normal and abnormal behavior, this is an unsupervised problem.

For both supervised and unsupervised learning tasks, it is important to have a representation of your input data that a computer can understand. Often it is helpful to think of your data as a table. Each data point that you want to reason about (each email, each customer, each transaction) is a row, and each property that describes that data point (say the age of a customer, the amount or location of a transaction) is a column.

You might describe users by their age, their gender, when they created an account and how often they bought from your online shop. You might describe the image of a tumor by the gray-scale values of each pixel, or maybe by using the size, shape and color of the tumor to describe it.

Each entity or row here is known as data point or *sample* in machine learning, while the columns, the properties that describe these entities, are called *features*.

We will later go into more detail on the topic of building a good representation of your data, which is called feature extraction or feature engineering. You should keep in mind however that no machine learning algorithm will be able to make a prediction on data for which it has no information. For example, if the only feature that you have for a patient is their last name, no algorithm will be able to predict their gender. This information is simply not contained in your data. If you add another feature that contains their first name, you will have much better luck, as it is often possible to tell the gender by a person's first name.

Knowing your data

Quite possibly the most important part in the machine learning process is understanding the data you are working with. It will not be effective to randomly choose an algorithm and throw your data at it. It is necessary to understand what is going on in your dataset before you begin building a model. Each algorithm is different in terms of what data it works best for, what kinds data it can handle, what kind of data it is optimized for, and so on. Before you start building a model, it is important to know the answers to most of, if not all of, the following questions:

- How much data do I have? Do I need more?
- How many features do I have? Do I have too many? Do I have too few?
- Is there missing data? Should I discard the rows with missing data or handle them differently?
- What question(s) am I trying to answer? Do I think the data collected can answer that question?

The last bullet point is the most important question, and certainly is not easy to answer. Thinking about these questions will help drive your analysis.

Keeping these basics in mind as we move through the book will prove helpful, because while scikit-learn is a fairly easy tool to use, it is geared more towards those with domain knowledge in machine learning.

Why Python?

Python has become the lingua franca for many data science applications. It combines the powers of general purpose programming languages with the ease of use of domain specific scripting languages like matlab or R.

Python has libraries for data loading, visualization, statistics, natural language processing, image processing, and more. This vast toolbox provides data scientists with a large array of general and special purpose functionality.

As a general purpose programming language, Python also allows for the creation of complex graphic user interfaces (GUIs), web services and for integration into existing systems.

What this book will cover

In this book, we will focus on applying machine learning algorithms for the purpose of solving practical problems. We will focus on how to write applications using the machine learning library scikit-learn for the Python programming language. Impor-

tant aspects that we will cover include formulating tasks as machine learning problems, preprocessing data for use in machine learning algorithms, and choosing appropriate algorithms and algorithmic parameters.

We will focus mostly on supervised learning techniques and algorithms, as these are often the most useful ones in practice, and they are easy for beginners to use and understand.

We will also discuss several common types of input, including text data.

What this book will not cover

This book will not cover the mathematical details of machine learning algorithms, and we will keep the number of formulas that we include to a minimum. In particular, we will not assume any familiarity with linear algebra or probability theory. As mathematics, in particular probability theory, is the foundation upon which machine learning is build, we will not be able to go into the analysis of the algorithms in great detail. If you are interested in the mathematics of machine learning algorithms, we recommend the text book "Elements of Statistical Learning" by Hastie, Tibshirani and Friedman, which is available for free at the authors website[footnote: http://statweb.stanford.edu/~tibs/ElemStatLearn/]. We will also not describe how to write machine learning algorithms from scratch, and will instead focus on how to use the large array of models already implemented in scikit-learn and other libraries.

We will not discuss reinforcement learning, which is about an agent learning from its interaction with an environment, and we will only briefly touch upon deep learning.

Some of the algorithms that are implemented in scikit-learn but are outside the scope of this book include Gaussian Processes, which are complex probabilistic models, and semi-supervised models, which work with supervised information on only some of the samples.

We will not also explicitly talk about how to work with time-series data, although many of techniques we discuss are applicable to this kind of data as well. Finally, we will not discuss how to do machine learning on natural images, as this is beyond the scope of this book.

Scikit-learn

Scikit-learn is an open-source project, meaning that scikit-learn is free to use and distribute, and anyone can easily obtain the source code to see what is going on behind the scenes. The scikit-learn project is constantly being developed and improved, and has a very active user community. It contains a number of state-of-the-art machine learning algorithms, as well as comprehensive documentation about each algorithm on the website [footnote http://scikit-learn.org/stable/documentation]. Scikit-learn is

a very popular tool, and the most prominent Python library for machine learning. It is widely used in industry and academia, and there is a wealth of tutorials and code snippets about scikit-learn available online. Scikit-learn works well with a number of other scientific Python tools, which we will discuss later in this chapter.

While studying the book, we recommend that you also browse the scikit-learn user guide and API documentation for additional details, and many more options to each algorithm. The online documentation is very thorough, and this book will provide you with all the prerequisites in machine learning to understand it in detail.

Installing Scikit-learn

Scikit-learn depends on two other Python packages, NumPy and SciPy. For plotting and interactive development, you should also install matplotlib, IPython and the Jupyter notebook. We recommend using one of the following pre-packaged Python distributions, which will provide the necessary packages:

- Anaconda (*https://store.continuum.io/cshop/anaconda/*): a Python distribution made for large-scale data processing, predictive analytics, and scientific computing. Anaconda comes with NumPy, SciPy, matplotlib, IPython, Jupyter notebooks, and scikit-learn. Anaconda is available on Mac OS X, Windows, and Linux.

- Enthought Canopy (*https://www.enthought.com/products/canopy/*): another Python distribution for scientific computing. This comes with NumPy, SciPy, matplotlib, and IPython, but the free version does not come with scikit-learn. If you are part of an academic, degree-granting institution, you can request an academic license and get free access to the paid subscription version of Enthought Canopy. Enthought Canopy is available for Python 2.7.x, and works on Mac, Windows, and Linux.

- Python(x,y) (*https://code.google.com/p/pythonxy/*): a free Python distribution for scientific computing, specifically for Windows. Python(x,y) comes with NumPy, SciPy, matplotlib, IPython, and scikit-learn.

If you already have a python installation set up, you can use pip to install any of these packages.

```
$ pip install numpy scipy matplotlib ipython scikit-learn
```

We do not recommended using pip to install NumPy and SciPy on Linux, as it involves compiling the packages from source. See the scikit-learn website for more detailed installation.

Essential Libraries and Tools

Understanding what scikit-learn is and how to use it is important, but there are a few other libraries that will enhance your experience. Scikit-learn is built on top of the NumPy and SciPy scientific Python libraries. In addition to knowing about NumPy and SciPy, we will be using Pandas and matplotlib. We will also introduce the Jupyter Notebook, which is an browser-based interactive programming environment. Briefly, here is what you should know about these tools in order to get the most out of scikit-learn.

If you are unfamiliar with numpy or matplotlib, we recommend reading the first chapter of the scipy lecture notes[footnote: http://www.scipy-lectures.org/].

Jupyter Notebook

The Jupyter Notebook is an interactive environment for running code in the browser. It is a great tool for exploratory data analysis and is widely used by data scientists. While Jupyter Notebook supports many programming languages, we only need the Python support. The Jypyter Notebook makes it easy to incorporate code, text, and images, and all of this book was in fact written as an IPython notebook.

All of the code examples we include can be downloaded from github [FIXME add github footnote].

NumPy

NumPy is one of the fundamental packages for scientific computing in Python. It contains functionality for multidimensional arrays, high-level mathematical functions such as linear algebra operations and the Fourier transform, and pseudo random number generators.

The NumPy array is the fundamental data structure in scikit-learn. Scikit-learn takes in data in the form of NumPy arrays. Any data you're using will have to be converted to a NumPy array. The core functionality of NumPy is this "ndarray", meaning it has *n* dimensions, and all elements of the array must be the same type. A NumPy array looks like this:

```
import numpy as np

x = np.array([[1, 2, 3], [4, 5, 6]])
x

array([[1, 2, 3],

       [4, 5, 6]])
```

SciPy

SciPy is both a collection of functions for scientific computing in python. It provides, among other functionality, advanced linear algebra routines, mathematical function optimization, signal processing, special mathematical functions and statistical distributions. Scikit-learn draws from SciPy's collection of functions for implementing its algorithms.

The most important part of scipy for us is `scipy.sparse` with provides *sparse matrices*, which is another representation that is used for data in scikit-learn. Sparse matrices are used whenever we want to store a 2d array that contains mostly zeros:

```
from scipy import sparse

# create a 2d numpy array with a diagonal of ones, and zeros everywhere else
eye = np.eye(4)
print("Numpy array:\n%s" % eye)

# convert the numpy array to a scipy sparse matrix in CSR format
# only the non-zero entries are stored
sparse_matrix = sparse.csr_matrix(eye)
print("\nScipy sparse CSR matrix:\n%s" % sparse_matrix)
```

```
Numpy array:

[[ 1.  0.  0.  0.]

 [ 0.  1.  0.  0.]

 [ 0.  0.  1.  0.]

 [ 0.  0.  0.  1.]]



Scipy sparse CSR matrix:

  (0, 0)    1.0

  (1, 1)    1.0

  (2, 2)    1.0

  (3, 3)    1.0
```

More details on scipy sparse matrices can be found in the scipy lecture notes.

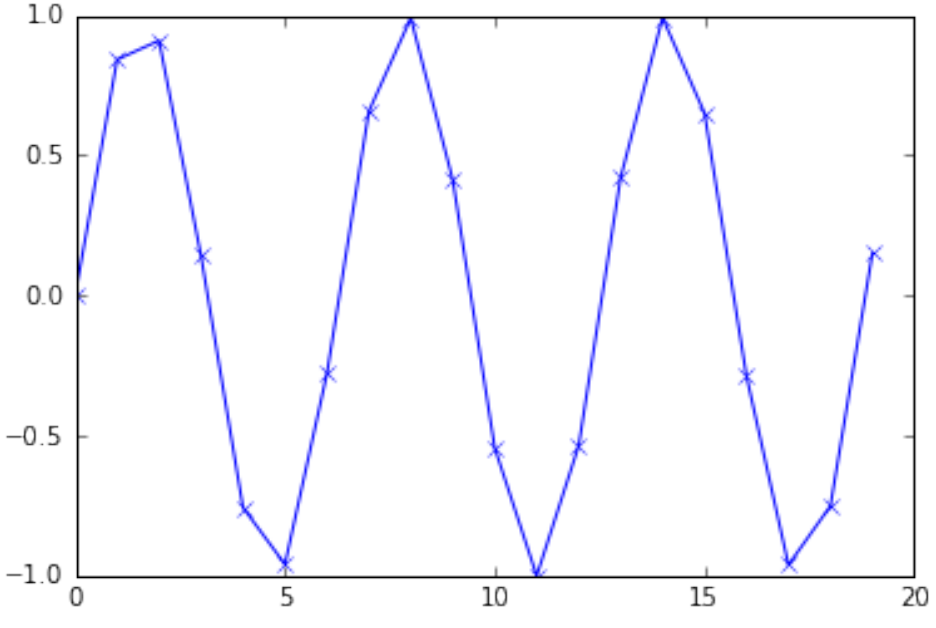
matplotlib

Matplotlib is the primary scientific plotting library in Python. It provides function for making publication-quality visualizations such as line charts, histograms, scatter

plots, and so on. Visualizing your data and any aspects of your analysis can give you important insights, and we will be using matplotlib for all our visualizations.

```
%matplotlib inline
import matplotlib.pyplot as plt

# Generate a sequence of integers
x = np.arange(20)
# create a second array using sinus
y = np.sin(x)
# The plot function makes a line chart of one array against another
plt.plot(x, y, marker="x")
```



Pandas

Pandas is a Python library for data wrangling and analysis. It is built around a data structure called DataFrame, that is modeled after the R DataFrame. Simply put, a Pandas Pandas DataFrame is a table, similar to an Excel Spreadsheet. Pandas provides a great range of methods to modify and operate on this table, in particular it allows SQL-like queries and joins of tables. Another valuable tool provided by Pandas is its ability to ingest from a great variety of file formats and databases, like SQL, Excel files and comma separated value (CSV) files. Going into details about the functionality of Pandas is out of the scope of this book. However, "Python for Data Analysis" by Wes McKinney provides a great guide.

Here is a small example of creating a DataFrame using a dictionary:

```
import pandas as pd

# create a simple dataset of people
data = {'Name': ["John", "Anna", "Peter", "Linda"],
        'Location' : ["New York", "Paris", "Berlin", "London"],
        'Age' : [24, 13, 53, 33]
       }

data_pandas = pd.DataFrame(data)
data_pandas
```

	Age	Location	Name
0	24	New York	John
1	13	Paris	Anna
2	53	Berlin	Peter
3	33	London	Linda

Python2 versus Python3

There are two major versions of Python that are widely used at the moment: Python2 (more precisely 2.7) and Python3 (with the latest release being 3.5 at the time of writing), which sometimes leads to some confusion. Python2 is no longer actively developed, but because Python3 contains major changes, Python2 code does usually not run without changes on Python3. If you are new to Python, or are starting a new project from scratch, we highly recommend using the latests version of Python3.

If you have a large code-base that you rely on that is written for Python2, you are excused from upgrading for now. However, you should try to migrate to Python3 as soon as possible. Writing any new code, it is for the most part quite easy to write code that runs under Python2 and Python3 [Footnote: The `six` package can be very handy for that].

All the code in this book is written in a way that works for both versions. However, the exact output might differ slightly under Python2.

Versions Used in this Book

We are using the following versions of the above libraries in this book:

```
import pandas as pd
print("pandas version: %s" % pd.__version__)

import matplotlib
print("matplotlib version: %s" % matplotlib.__version__)

import numpy as np
print("numpy version: %s" % np.__version__)
```

```
import IPython
print("IPython version: %s" % IPython.__version__)

import sklearn
print("scikit-learn version: %s" % sklearn.__version__)
```

```
pandas version: 0.17.1

matplotlib version: 1.5.1

numpy version: 1.10.4

IPython version: 4.1.2

scikit-learn version: 0.18.dev0
```

While it is not important to match these versions exactly, you should have a version of scikit-learn that is as least as recent as the one we used.

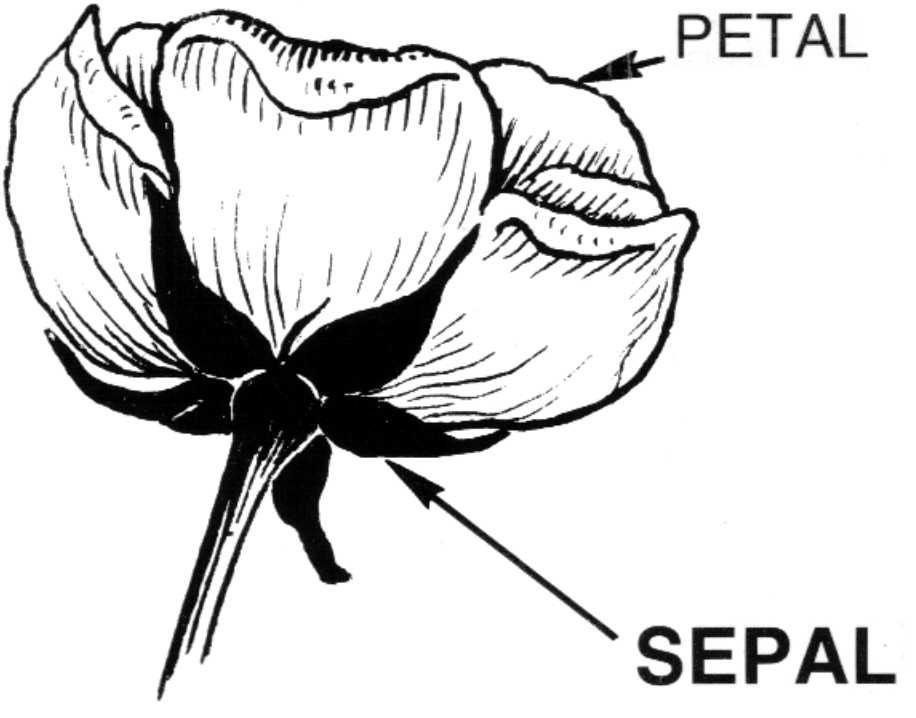
Now that we have everything set up, let's dive into our first appication of machine learning.

A First Application: Classifying iris species

In this section, we will go through a simple machine learning application and create our first model.

In the process, we will introduce some core concepts and nomenclature for machine learning.

Let's assume that a hobby botanist is interested in distinguishing what the species is of some iris flowers that she found. She has collected some measurements associated with the iris: the length and width of the petals, and the length and width of the sepal, all measured in centimeters.



She also has the measurements of some irises that have been previously identified by an expert botanist as belonging to the species Setosa, Versicolor or Virginica. For these measurements, she can be certain of which species each iris belongs to. Let's assume that these are the only species our hobby botanist will encounter in the wild.

Our goal is to build a machine learning model that can learn from the measurements of these irises whose species is known, so that we can predict the species for a new iris.

Since we have measurements for which we know the correct species of iris, this is a supervised learning problem. In this problem, we want to predict one of several options (the species of iris). This is an example of a *classification* problem. The possible outputs (different species of irises) are called *classes*.

Since every iris in the dataset belongs to one of three classes this problem is a three-class classification problem.

The desired output for a single data point (an iris) is the species of this flower. For a particular data point, the species it belongs to is called its *label*.

Meet the data

The data we will use for this example is the iris dataset, a classical dataset in machine learning an statistics.

It is included in scikit-learn in the dataset module. We can load it by calling the `load_iris` function:

```
from sklearn.datasets import load_iris
iris = load_iris()
```

The iris object that is returned by `load_iris` is a *Bunch* object, which is very similar to a dictionary. It contains keys and values:

```
iris.keys()
```

```
dict_keys(['DESCR', 'data', 'target_names', 'feature_names', 'target'])
```

The value to the key `DESCR` is a short description of the dataset. We show the beginning of the description here. Feel free to look up the rest yourself.

```
print(iris['DESCR'][:193] + "\n...")
```

```
Iris Plants Database

====================



Notes

-----

Data Set Characteristics:

    :Number of Instances: 150 (50 in each of three classes)

    :Number of Attributes: 4 numeric, predictive att

...
```

The value with key `target_names` is an array of strings, containing the species of flower that we want to predict:

```
iris['target_names']
```

```
array(['setosa', 'versicolor', 'virginica'],

      dtype='<U10')
```

The `feature_names` are a list of strings, giving the description of each feature:

```
iris['feature_names']
```

```
['sepal length (cm)',
 'sepal width (cm)',
 'petal length (cm)',
 'petal width (cm)']
```

The data itself is contained in the `target` and `data` fields. The `data` contains the numeric measurements of sepal length, sepal width, petal length, and petal width in a numpy array:

```
type(iris['data'])
numpy.ndarray
```

The rows in the data array correspond to flowers, while the columns represent the four measurements that were taken for each flower:

```
iris['data'].shape
(150, 4)
```

We see that the data contains measurements for 150 different flowers.

Remember that the individual items are called *samples* in machine learning, and their properties are called *features*.

The shape of the data array is the number of samples times the number of features.

This is a convention in scikit-learn, and your data will always be assumed to be in this shape.

Here are the feature values for the first five samples:

```
iris['data'][:5]
array([[ 5.1,  3.5,  1.4,  0.2],
       [ 4.9,  3. ,  1.4,  0.2],
       [ 4.7,  3.2,  1.3,  0.2],
       [ 4.6,  3.1,  1.5,  0.2],
       [ 5. ,  3.6,  1.4,  0.2]])
```

The `target` array contains the species of each of the flowers that were measured, also as a numpy array:

```
type(iris['target'])
numpy.ndarray
```

The target is a one-dimensional array, with one entry per flower:

```
iris['target'].shape
```

```
(150,)
```

The species are encoded as integers from 0 to 2:

```
iris['target']
```

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

The meaning of the numbers are given by the `iris['target_names']` array: 0 means Setosa, 1 means Versicolor and 2 means Virginica.

Measuring Success: Training and testing data

We want to build a machine learning model from this data that can predict the species of iris for a new set of measurements.

Before we can apply our model to new measurements, we need to know whether our model actually works, that is whether we should trust its predictions.

Unfortunately, we can not use the data we use to build the model to evaluate it. This is because our model can always simply remember the whole training set, and will therefore always predict the correct label for any point in the training set. This "remembering" does not indicate to us whether our model will *generalize* well, in other words whether it will also perform well on new data. So before we apply our model to new measurements, we will want to know whether we can trust its predictions.

To assess the models' performance, we show the model new data (that it hasn't seen before) for which we have labels. This is usually done by splitting the labeled data we have collected (here our 150 flower measurements) into two parts.

The part of the data is used to build our machine learning model, and is called the *training data* or *training set*. The rest of the data will be used to access how well the model works and is called *test data*, *test set* or *hold-out set*.

Scikit-learn contains a function that shuffles the dataset and splits it for you, the `train_test_split` function.

This function extracts 75% of the rows in the data as the training set, together with the corresponding labels for this data. The remaining 25% of the data, together with the remaining labels are declared as the test set.

How much data you want to put into the training and the test set respectively is somewhat arbitrary, but using a test-set containing 25% of the data is a good rule of thumb.

In scikit-learn, data is usually denoted with a capital `X`, while labels are denoted by a lower-case `y`.

Let's call `train_test_split` on our data and assign the outputs using this nomenclature:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(iris['data'], iris['target'],
                                                    random_state=0)
```

The `train_test_split` function shuffles the dataset using a pseudo random number generator before making the split. If we would take the last 25% of the data as a test set, all the data point would have the label `2`, as the data points are sorted by the label (see the output for `iris['target']` above). Using a tests set containing only one of the three classes would not tell us much about how well we generalize, so we shuffle our data, to make sure the test data contains data from all classes.

To make sure that we will get the same output if we run the same function several times, we provide the pseudo random number generator with a fixed seed using the `random_state` parameter. This will make the outcome deterministic, so this line will always have the same outcome. We will always fix the `random_state` in this way when using randomized procedures in this book.

The output of the `train_test_split` function are `X_train`, `X_test`, `y_train` and `y_test`, which are all numpy arrays. `X_train` contains 75% of the rows of the dataset, and `X_test` contains the remaining 25%:

```
X_train.shape
```

```
(112, 4)
```

```
X_test.shape
```

```
(38, 4)
```

First things first: Look at your data

Before building a machine learning model, it is often a good idea to inspect the data, to see if the task is easily solvable without machine learning, or if the desired information might not be contained in the data.

Additionally, inspecting your data is a good way to find abnormalities and peculiarities. Maybe some of your irises were measured using inches and not centimeters, for example. In the real world, inconsistencies in the data and unexpected measurements are very common.

One of the best ways to inspect data is to visualize it. One way to do this is by using a scatter plot.

A scatter plot of the data puts one feature along the x-axis, one feature along the y-axis, and draws a dot for each data point.

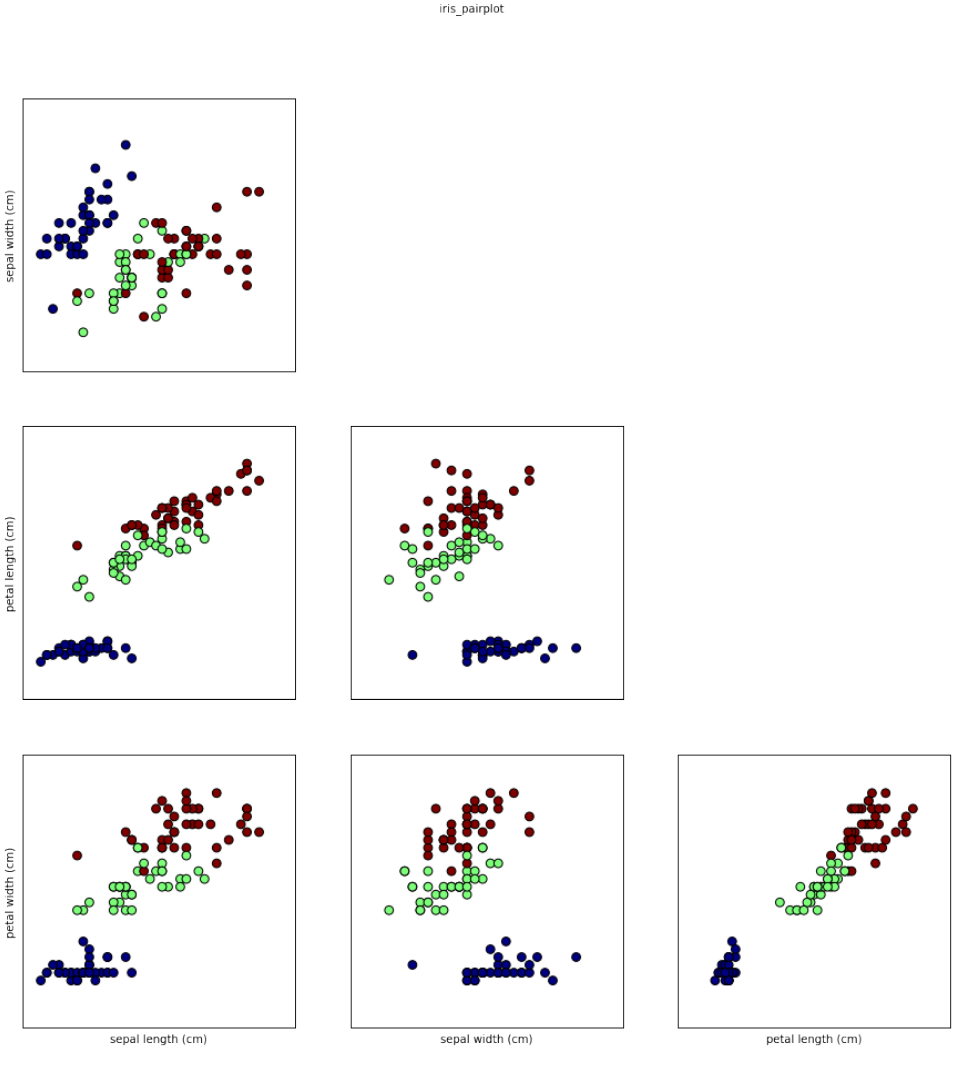
Unfortunately, computer screens have only two dimensions, which allows us to only plot two (or maybe three) features at a time. It is difficult to plot datasets with more than three features this way.

One way around this problem is to do a pair plot, which looks at all pairs of two features. If you have a small number of features, such as the four we have here, this is quite reasonable. You should keep in mind that a pair plot does not show the interaction of all of features at once, so some interesting aspects of the data may not be revealed when visualizing it this way.

Here is a pair plot of the features in the training set. The data points are colored according to the species the iris belongs to:

```
fig, ax = plt.subplots(3, 3, figsize=(15, 15))
plt.suptitle("iris_pairplot")

for i in range(3):
    for j in range(3):
        ax[i, j].scatter(X_train[:, j], X_train[:, i + 1], c=y_train, s=60)
        ax[i, j].set_xticks(())
        ax[i, j].set_yticks(())
        if i == 2:
            ax[i, j].set_xlabel(iris['feature_names'][j])
        if j == 0:
            ax[i, j].set_ylabel(iris['feature_names'][i + 1])
        if j > i:
            ax[i, j].set_visible(False)
```



From the plots, we can see that the three classes seem to be relatively well separated using the sepal and petal measurements. This means that a machine learning model will likely be able to learn to separate them.

Building your first model: k nearest neighbors

Now we can start building the actual machine learning model. There are many classification algorithms in scikit-learn that we could use. Here we will use a k nearest neighbors classifier, which is easy to understand.

Building this model only consists of storing the training set. To make a prediction for a new data point, the algorithm finds the point in the training set that is closest to the new point. Then, it and assigns the label of this closest data training point to the new data point.

The `k` in `k` nearest neighbors stands for the fact that instead of using only the closest neighbor to the new data point, we can consider any fixed number `k` of neighbors in the training (for example, the closest three or five neighbors). Then, we can make a prediction using the majority class among these neighbors. We will go into more details about this later.

Let's use only a single neighbor for now.

All machine learning models in scikit-learn are implemented in their own class, which are called `Estimator` classes. The `k` nearest neighbors classification algorithm is implemented in the `KNeighborsClassifier` class in the `neighbors` module.

Before we can use the model, we need to instantiate the class into an object. This is when we will set any parameters of the model. The single parameter of the `KNeighborsClassifier` is the number of neighbors, which we will set to one:

```
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors=1)
```

The `knn` object encapsulates the algorithm to build the model from the training data, as well the algorithm to make predictions on new data points.

It will also hold the information the algorithm has extracted from the training data. In the case of `KNeighborsClassifier`, it will just store the training set.

To build the model on the training set, we call the `fit` method of the `knn` object, which takes as arguments the numpy array `X_train` containing the training data and the numpy array `y_train` of the corresponding training labels:

```
knn.fit(X_train, y_train)

KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',

           metric_params=None, n_jobs=1, n_neighbors=1, p=2,

           weights='uniform')
```

Making predictions

We can now make predictions using this model on new data, for which we might not know the correct labels.

Imagine we found an iris in the wild with a sepal length of 5cm, a sepal width of 2.9cm, a petal length of 1cm and a petal width of 0.2cm. What species of iris would this be?

We can put this data into a numpy array, again with the shape number of samples (one) times number of features (four):

```
X_new = np.array([[5, 2.9, 1, 0.2]])
X_new.shape
```

```
(1, 4)
```

To make prediction we call the `predict` method of the `knn` object:

```
prediction = knn.predict(X_new)
prediction
```

```
array([0])
```

```
iris['target_names'][prediction]
```

```
array(['setosa'],

      dtype='<U10')
```

Our model predicts that this new iris belongs to the class 0, meaning its species is Setosa.

But how do we know whether we can trust our model? We don't know the correct species of this sample, which is the the whole point of building the model!

Evaluating the model

This is where the test set that we created earlier comes in. This data was not used to build the model, but we do know what the correct species are for each iris in the test set.

We can make a prediction for an iris in the test data, and compare it against its label (the known species). We can measure how well the model works by computing the *accuracy*, which is the fraction of flowers for which the right species was predicted:

```
y_pred = knn.predict(X_test)
np.mean(y_pred == y_test)
```

```
0.97368421052631582
```

We can also use the `score` method of the `knn` object, which will compute the test set accuracy for us:

```
knn.score(X_test, y_test)
```

```
0.97368421052631582
```

For this model, the test set accuracy is about 0.97, which means we made the right prediction for 97% of the irises in the test set. Under some mathematical assumptions, this means that we can expect our model to be correct 97% of the time for new irises.

For our hobby botanist application, this high level of accuracy means that our models may be trustworthy enough to use. In later chapters we will discuss how we can improve performance, and what caveats there are in tuning a model.

Summary

Let's summarize what we learned in this chapter. We started off formulating a task of predicting which species of iris a particular flower belongs to by using physical measurements of the flower. We used a dataset of measurements that was annotated by an expert with the correct species to build our model, making this a supervised learning task. There were three possible species, Setosa, Versicolor or Virginica, which made the task a three-class *classification* problem. The possible species are called *classes* in the classification problem, and the species of a single iris is called its *label*.

The dataset consists of two numpy arrays, one containing the data, which is referred to as `X` in scikit-learn, and one containing the correct or desired outputs, which is called `y`. The array `X` is a two-dimensional array of features, with one row per data point, and one column per feature. The array `y` is a one-dimensional array, which here contained one class label from 0 to 2 for each of the samples.

We split our dataset into a *training set*, to build our model, and a *test set*, to evaluate how well our model will generalize to new, unseen data.

We chose the `k` nearest neighbors classification algorithm, which makes predictions for a new data point by considering its closest neighbor(s) in the training set.

The algorithm is implemented in the `KNeighborsClassifier` class, which contains the algorithm to build the model, as well as the algorithm to make a prediction using the model. We instantiated the class, setting parameters. Then, we built the model by calling the `fit` method, passing the training data `X_train` and training outputs `y_train` as parameters.

We evaluated the model using the `score` method, that computes the *accuracy* of the model. We applied the `score` method to the test set data and the test set labels, and found that our model is about 97% accurate, meaning it is correct 97% of the time on the test set.

This gave us the confidence to apply the model to new data (in our example, new flower measurements), and trust that the model will be correct about 97% of the time.

Here is a summary of the code needed for the whole training and evaluation procedure:

```
X_train, X_test, y_train, y_test = train_test_split(iris['data'], iris['target'],
                                                    random_state=0)

knn = KNeighborsClassifier(n_neighbors=1)
knn.fit(X_train, y_train)

knn.score(X_test, y_test)
```

```
0.97368421052631582
```

This snippet contains the core code for applying any machine learning algorithms using scikit-learn. The `fit`, `predict` and `score` methods are the common interface to supervised models in scikit-learn, and with the concepts introduced in this chapter, you can apply these models to many machine learning tasks.

In the next chapter, we will go into more depth about the different kinds of supervised models in scikit-learn, and how to apply them successfully.