A Tutorial on Machine Learning and Data Science Tools with Python

Marcus D. Bloice(✉) and Andreas Holzinger

Holzinger Group HCI-KDD, Institute for Medical Informatics,
Statistics and Documentation, Medical University of Graz, Graz, Austria
{marcus.bloice,andreas.holzinger}@medunigraz.at

Abstract. In this tutorial, we will provide an introduction to the main Python software tools used for applying machine learning techniques to medical data. The focus will be on open-source software that is freely available and is cross platform. To aid the learning experience, a companion GitHub repository is available so that you can follow the examples contained in this paper interactively using Jupyter notebooks. The notebooks will be more exhaustive than what is contained in this chapter, and will focus on medical datasets and healthcare problems. Briefly, this tutorial will first introduce Python as a language, and then describe some of the lower level, general matrix and data structure packages that are popular in the machine learning and data science communities, such as NumPy and Pandas. From there, we will move to dedicated machine learning software, such as SciKit-Learn. Finally we will introduce the Keras deep learning and neural networks library. The emphasis of this paper is readability, with as little jargon used as possible. No previous experience with machine learning is assumed. We will use openly available medical datasets throughout.

Keywords: Machine learning · Deep learning · Neural networks · Tools · Languages · Python

1 Introduction

The target audience for this tutorial paper are those who wish to quickly get started in the area of data science and machine learning. We will provide an overview of the current and most popular libraries with a focus on Python, however we will mention alternatives in other languages where appropriate. All tools presented here are free and open source, and many are licensed under very flexible terms (including, for example, commercial use). Each library will be introduced, code will be shown, and typical use cases will be described. Medical datasets will be used to demonstrate several of the algorithms.

Machine learning itself is a fast growing technical field [1] and is highly relevant topic in both academia and in the industry. It is therefore a relevant skill to have in both academia and in the private sector. It is a field at the intersection of informatics and statistics, tightly connected with data science and knowledge

A. Holzinger (Ed.): ML for Health Informatics, LNAI 9605, pp. 435–480, 2016.
DOI: 10.1007/978-3-319-50478-0_22

discovery [2,3]. The prerequisites for this tutorial are therefore a basic understanding of statistics, as well as some experience in any C-style language. Some knowledge of Python is useful but not a must.

An accompanying GitHub repository is provided to aid the tutorial:

https://github.com/mdbloice/MLDS

It contains a number of notebooks, one for each main section. The notebooks will be referred to where relevant.

2 Glossary and Key Terms

This section provides a quick reference for several algorithms that are not explicity mentioned in this chapter, but may be of interest to the reader. This should provide the reader with some keywords or useful points of reference for other similar libraries to those discussed in this chapter.

BIDMach GPU accelerated machine learning library for algorithms that are not necessarily neural network based.

Caret provides a standardised API for many of the most useful machine learning packages for R. See http://topepo.github.io/caret/index.html. For readers who are more comfortable with R, Caret provides a good substitute for Python's SciKit-Learn.

Mathematica is a commercial symbolic mathematical computation system, developed since 1988 by Wolfram, Inc. It provides powerful machine learning techniques "out of the box" such as image classification [4].

MATLAB is short for MATrix LABoratory, which is a commercial numerical computing environment, and is a proprietary programming language by MathWorks. It is very popular at universities where it is often licensed. It was originally built on the idea that most computing applications in some way rely on storage and manipulations of one fundamental object—the matrix, and this is still a popular approach [5].

R is used extensively by the statistics community. The software package Caret provides a standardised API for many of R's machine learning libraries.

WEKA is short for the Waikato Environment for Knowledge Analysis [6] and has been a very popular open source tool since its inception in 1993. In 2005 Weka received the SIGKDD Data Mining and Knowledge Discovery Service Award: it is easy to learn and simple to use, and provides a GUI to many machine learning algorithms [7].

Vowpal Wabbit Microsoft's machine learning library. Mature and actively developed, with an emphasis on performance.

3 Requirements and Installation

The most convenient way of installing the Python requirements for this tutorial is by using the Anaconda scientific Python distribution. Anaconda is a collection of the most commonly used Python packages preconfigured and ready to use. Approximately 150 scientific packages are included in the Anaconda installation.

To install Anaconda, visit

https://www.continuum.io/downloads

and install the version of Anaconda for your operating system.

All Python software described here is available for Windows, Linux, and Macintosh. All code samples presented in this tutorial were tested under Ubuntu Linux 14.04 using Python 2.7. Some code examples may not work on Windows without slight modification (e.g. file paths in Windows use \ and not / as in UNIX type systems).

The main software used in a typical Python machine learning pipeline can consist of almost any combination of the following tools:

1. NumPy, for matrix and vector manipulation
2. Pandas for time series and R-like DataFrame data structures
3. The 2D plotting library matplotlib
4. SciKit-Learn as a source for many machine learning algorithms and utilities
5. Keras for neural networks and deep learning

Each will be covered in this book chapter.

3.1 Managing Packages

Anaconda comes with its own built in package manager, known as Conda. Using the `conda` command from the terminal, you can download, update, and delete Python packages. Conda takes care of all dependencies and ensures that packages are preconfigured to work with all other packages you may have installed.

First, ensure you have installed Anaconda, as per the instructions under https://www.continuum.io/downloads.

Keeping your Python distribution up to date and well maintained is essential in this fast moving field. However, Anaconda makes it particularly easy to manage and keep your scientific stack up to date. Once Anaconda is installed you can manage your Python distribution, and all the scientific packages installed by Anaconda using the `conda` application from the command line. To list all packages currently installed, use `conda list`. This will output all packages and their version numbers. Updating all Anaconda packages in your system is performed using the `conda update -all` command. Conda itself can be updated using the `conda update conda` command, while Python can be updated using the `conda update python` command. To search for packages, use the `search` parameter, e.g. `conda search stats` where `stats` is the name or partial name of the package you are searching for.

4 Interactive Development Environments

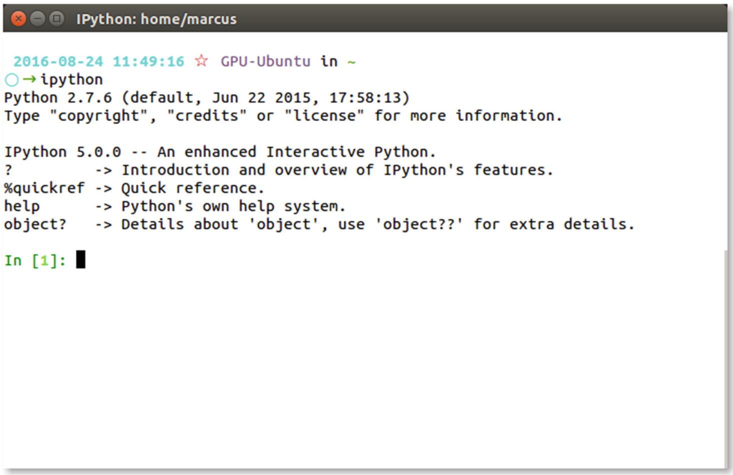
4.1 IPython

IPython is a REPL that is commonly used for Python development. It is included in the Anaconda distribution. To start IPython, run:

```
$ ipython
```

Listing 1. Starting IPython

Some informational data will be displayed, similar to what is seen in Fig. 1, and you will then be presented with a command prompt.



```
2016-08-24 11:49:16 ☆ GPU-Ubuntu in ~
○ → ipython
Python 2.7.6 (default, Jun 22 2015, 17:58:13)
Type "copyright", "credits" or "license" for more information.

IPython 5.0.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: █
```

Fig. 1. The IPython Shell.

IPython is what is known as a REPL: a **R**ead **E**valuate **P**rint **L**oop. The interpreter allows you to type in commands which are evaluated as soon as you press the Enter key. Any returned output is immediately shown in the console. For example, we may type the following:

```
1 In [1]: 1 + 1
2 Out[1]: 2
3 In [2]: import math
4 In [3]: math.radians(90)
5 Out[3]: 1.5707963267948966
6 In [4]:
```

Listing 2. Examining the Read Evaluate Print Loop (REPL)

After pressing return (Line 1 in Listing 2), Python immediately interprets the line and responds with the returned result (Line 2 in Listing 2). The interpreter then awaits the next command, hence Read Evaluate Print Loop.

Using IPython to experiment with code allows you to test ideas without needing to create a file (e.g. `fibonacci.py`) and running this file from the command line (by typing `python fibonacci.py` at the command prompt). Using the IPython REPL, this entire process can be made much easier. Of course, creating permanent files is essential for larger projects.

A useful feature of IPython are the so-called magic functions. These commands are not interpreted as Python code by the REPL, instead they are special commands that IPython understands. For example, to run a Python script you can use the `%run` magic function:

```
>>> %run fibonacci.py 30
Fibonacci number 30 is 832040.
```

Listing 3. Using the `%run` magic function to execute a file.

In the code above, we have executed the Python code contained in the file `fibonacci.py` and passed the value 30 as an argument to the file.

The file is executed as a Python script, and its output is displayed in the shell. Other magic functions include `%timeit` for timing code execution:

```
>>> def fibonacci(n):
...     if n == 0: return 0
...     if n == 1: return 1
...     return fibonacci(n-1) + fibonacci(n-2)
>>> %timeit fibonacci(25)
10 loops, best of 3: 30.9 ms per loop
```

Listing 4. The `%timeit` magic function can be used to check execution times of functions or any other piece of code.

As can be seen, executing the `fibonacci(25)` function takes on average 30.9 ms. The `%timeit` magic function is clever in how many loops it performs to create an average result, this can be as few as 1 loop or as many as 10 million loops.

Other useful magic functions include `%ls` for listing files in the current working directory, `%cd` for printing or changing the current directory, and `%cpaste` for pasting in longer pieces of code that span multiple lines. A full list of magic functions can be displayed using, unsurprisingly, a magic function: type `%magic` to view all magic functions along with documentation for each one. A summary of useful magic functions is shown in Table 1.

Last, you can use the `?` operator to display in-line help at any time. For example, typing

```
>>> abs?
Docstring:
abs(number) -> number

Return the absolute value of the argument.
Type:      builtin_function_or_method
```

Listing 5. Accessing help within the IPython console.

For larger projects, or for projects that you may want to share, IPython may not be ideal. In Sect. 4.2 we discuss the web-based notebook IDE known as Jupyter, which is more suited to larger projects or projects you might want to share.

Table 1. A non-comprehensive list of IPython magic functions.

Magic Command	Description
`%lsmagic`	Lists all the magic functions
`%magic`	Shows descriptive magic function documentation
`%ls`	Lists files in the current directory
`%cd`	Shows or changes the current directory
`%who`	Shows variables in scope
`%whos`	Shows variables in scope along with type information
`%cpaste`	Pastes code that spans several lines
`%reset`	Resets the session, removing all imports and deleting all variables
`%debug`	Starts a debugger *post mortem*

4.2 Jupyter

Jupyter, previously known as IPython Notebook, is a web-based, interactive development environment. Originally developed for Python, it has since expanded to support over 40 other programming languages including Julia and R.

Jupyter allows for *notebooks* to be written that contain text, live code, images, and equations. These notebooks can be shared, and can even be hosted on GitHub for free.

For each section of this tutorial, you can download a Juypter notebook that allows you to edit and experiment with the code and examples for each topic. Jupyter is part of the Anaconda distribution, it can be started from the command line using using the `jupyter` command:

```
$ jupyter notebook
```

Listing 6. Starting Jupyter

Upon typing this command the Jupyter server will start, and you will briefly see some information messages, including, for example, the URL and port at which the server is running (by default `http://localhost:8888/`). Once the server has started, it will then open your default browser and point it to this address. This browser window will display the contents of the directory where you ran the command.

To create a notebook and begin writing, click the New ▼ button and select Python. A new notebook will appear in a new tab in the browser. A Jupyter notebook allows you to run code blocks and immediately see the output of these blocks of code, much like the IPython REPL discussed in Sect. 4.1.

Jupyter has a number of short-cuts to make navigating the notebook and entering code or text quickly and easily. For a list of short-cuts, use the menu Help → Keyboard Shortcuts.

4.3 Spyder

For larger projects, often a fully fledged IDE is more useful than Juypter's notebook-based IDE. For such purposes, the Spyder IDE is often used. Spyder stands for Scientific PYthon Development EnviRonment, and is included in the Anaconda distribution. It can be started by typing `spyder` in the command line.

5 Requirements and Conventions

This tutorial makes use of a number of packages which are used extensively in the Python machine learning community. In this chapter, the NumPy, Pandas, and Matplotlib are used throughout. Therefore, for the Python code samples shown in each section, we will presume that the following packages are available and have been loaded before each script is run:

```
>>> import numpy as np
>>> import pandas as pd
>>> import matplotlib.pyplot as plt
```

Listing 7. Standard libraries used throughout this chapter. Throughout this chapter we will assume these libraries have been imported before each script.

Any further packages will be explicitly loaded in each code sample. However, in general you should probably follow each section's Jupyter notebook as you are reading.

In Python code blocks, lines that begin with >>> represent Python code that should be entered into a Python interpreter (See Listing 7 for an example). Output from any Python code is shown **without** any preceding >>> characters.

Commands which need to be entered into the terminal (e.g. bash or the MS-DOS command prompt) begin with $, such as:

```
$ ls -lAh
total 299K
-rw-rw-r-- 1 bloice admin  73K Sep  1 14:11 Clustering.ipynb
-rw-rw-r-- 1 bloice admin  57K Aug 25 16:04 Pandas.ipynb
...
```

Listing 8. Commands for the terminal are preceded by a `$` sign.

Output from the console is shown **without** a preceding $ sign. Some of the commands in this chapter may only work under Linux (such as the example usage of the `ls` command in the code listing above, the equivalent in Windows is the `dir` command). Most commands will, however, work under Linux, Macintosh, and Windows—if this is not the case, we will explicitly say so.

5.1 Data

For the Introduction to Python, NumPy, and Pandas sections we will work with either generated data or with a toy dataset. Later in the chapter, we will move on to medical examples, including a breast cancer dataset, a diabetes dataset, and a high-dimensional gene expression dataset. All medical datasets used in this chapter are freely available and we will describe how to get the data in each relevant section. In earlier sections, generated data will suffice in order to demonstrate example usage, while later we will see that analysing more involved medical data using the same open-source tools is equally possible.

6 Introduction to Python

Python is a general purpose programming language that is used for anything from web-development to deep learning. According to several metrics, it is ranked as one of the top three most popular languages. It is now the most frequently taught introductory language at top U.S. universities according to a recent ACM blog article [8]. Due to its popularity, Python has a thriving open source community, and there are over 80,000 free software packages available for the language on the official Python Package Index (PyPI).

In this section we will give a very short crash course on using Python. These code samples will work best with a Python REPL interpreter, such as IPython or Jupyter (Sects. 4.1 and 4.2 respectively). In the code below we introduce the some simple arithmetic syntax:

```
>>> 2 + 6 + (8 * 9)
80
>>> 3 / 2
1
>>> 3.0 / 2
1.5
>>> 4 ** 4 # To the power of
256
```

Listing 9. Simple arithmetic with Python in the IPython shell.

Python is a dynamically typed language, so you do not define the type of variable you are creating, it is inferred:

```
>>> n = 5
>>> f = 5.5
>>> s ="5"
>>> type(s)
str
>>> type(f)
float
>>> "5" * 5
"55555"
>>> int("5") * 5
25
```

Listing 10. Demonstrating types in Python.

You can check types using the built-in `type` function. Python does away with much of the verbosity of languages such as Java, you do not even need to surround code blocks with brackets or braces:

```
>>> if "5" == 5:
...     print("Will not get here")
>>> elif int("5") == 5:
...     print("Got here")
Got here
```

Listing 11. Statement blocks in Python are indicated using indentation.

As you can see, we use indentation to define our statement blocks. This is the number one source of confusion among those new to Python, so it is important you are aware of it. Also, whereas assignment uses `=`, we check equality using `==` (and inversely `!=`). Control of flow is handled by `if`, `elif`, `while`, `for`, and so on.

While there are several basic data structures, here we will concentrate on lists and dictionaries (we will cover much more on data structures in Sect. 7.1). Other types of data structures are, for example, tuples, which are immutable—their contents cannot be changed after they are created—and sets, where repetition is not permitted. We will not cover tuples or sets in this tutorial chapter, however.

Below we first define a list and then perform a number of operations on this list:

```
 1 >>> powers = [1, 2, 4, 8, 16, 32]
 2 >>> powers
 3 [1, 2, 4, 8, 16, 32]
 4 >>> powers[0]
 5 1
 6 >>> powers.append(64)
 7 >>> powers
 8 [1, 2, 4, 8, 16, 32, 64]
 9 >>> powers.insert(0, 0)
10 >>> powers
11 [0, 1, 2, 4, 8, 16, 32, 64]
12 >>> del powers[0]
13 >>> powers
14 [1, 2, 4, 8, 16, 32, 64]
15 >>> 1 in powers
16 True
17 >>> 100 not in powers
18 True
```

Listing 12. Operations on lists.

Lists are defined using square `[]` brackets. You can index a list using its numerical, zero-based index, as seen on Line 4. Adding values is performed using the `append` and `insert` functions. The `insert` function allows you to define in which position you would like the item to be inserted—on Line 9 of Listing 12 we insert the number 0 at position 0. On Lines 15 and 17, you can see how we can use the `in` keyword to check for membership.

You just saw that lists are indexed using zero-based numbering, we will now introduce dictionaries which are key-based. Data in dictionaries are stored using key-value pairs, and are indexed by the keys that you define:

```
1  >>> numbers = {"bingo": 3458080, "tuppy": 3459090}
2  >>> numbers
3  {"bingo": 3458080, "tuppy": 3459090}
4  >>> numbers["bingo"]
5  3458080
6  >>> numbers["monty"] = 3456060
7  >>> numbers
8  {"bingo": 3458080, "monty": 3456060, "tuppy": 3459090}
9  >>> "tuppy" in numbers
10 True
```

Listing 13. Dictionaries in Python.

We use curly {} braces to define dictionaries, and we must define both their values and their indices (Line 1). We can access elements of a dictionary using their keys, as in Line 4. On Line 6 we insert a new key-value pair. Notice that dictionaries are not ordered. On Line 9 we can also use the `in` keyword to check for membership.

To traverse through a dictionary, we use a `for` statement in conjunction with a function depending on what data we wish to access from the dictionary:

```
1  >>> for name, number in numbers.iteritems():
2  ...     print("Name:" + name + ", number:" + str(number))
3  Name: bingo, number: 3458080
4  Name: monty, number: 3456060
5  Name: tuppy, number: 3459090
6
7  >>> for key in numbers.keys():
8  ...    print(key)
9  bingo
10 monty
11 tuppy
12
13 >>> for val in numbers.values():
14 ...    print(val)
15 3458080
16 3456060
17 3459090
```

Listing 14. Iterating through dictionaries.

First, the code above traverses through each key-value pair using `iteritems()` (Line 1). When doing so, you can specify a variable name for each key and value (in that order). In other words, on Line 1, we have stated that we wish to store each key in the variable `name` and each value in the variable `number` as we go through the `for` loop. You can also access only the keys or values using the `keys` and `values` functions respectively (Lines 7 and 13).

As mentioned previously, many packages are available for Python. These need to be loaded into the current environment before they are used. For example, the code below uses the `os` module, which we must first import before using:

```
1  >>> import os
2  >>> os.listdir("./")
3  ["BookChapter.ipynb",
4   "NumPy.ipynb",
5   "Pandas.ipynb",
6   "fibonacci.py",
7   "LinearRegression.ipynb",
8   "Clustering.ipynb"]
9  >>> from os import listdir # Alternatively
10 >>> listdir("./")
11 ["BookChapter.ipynb",
12  "NumPy.ipynb",
13 ...
```

Listing 15. Importing packages using the `import` keyword.

Two ways of importing are shown here. On Line 1 we are importing the entire `os` name space. This means we need to call functions using the `os.listdir()` syntax. If you know that you only need one function or submodule you can import it individually using the method shown on Line 9. This is often the preferred method of importing in Python.

Lastly, we will briefly see how functions are defined using the `def` keyword:

```
>>> def addNumbers(x, y):
...     return x + y
>>> addNumbers(4, 2)
6
```

Listing 16. Functions are defined using the `def` keyword.

Notice that you do not need to define the return type, or the arguments' types. Classes are equally easy to define, and this is done using the `class` keyword. We will not cover classes in this tutorial. Classes are generally arranged into modules, and further into packages. Now that we have covered some of the basics of Python, we will move on to more advanced data structures such as 2-dimensional arrays and data frames.

7 Handling Data

7.1 Data Structures and Notation

In machine learning, more often than not the data that you analyse will be stored in matrices and vectors. Generally speaking, your data that you wish to analyse will be stored in the form of a matrix, often denoted using a bold upper case symbol, generally $\mathbf{X}$, and your label data will be stored in a vector, denoted with a lower case bold symbol, often $\mathbf{y}$.

A data matrix $\mathbf{X}$ with n samples and m features is denoted as follows:

$$\mathbf{X} \in \mathbb{R}^{n \times m} = \begin{bmatrix} x_{1,1} & x_{1,2} & x_{1,3} & \dots & x_{1,m} \\ x_{2,1} & x_{2,2} & x_{2,3} & \dots & x_{2,m} \\ x_{3,1} & x_{3,2} & x_{3,3} & \dots & x_{3,m} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_{n,1} & x_{n,2} & x_{n,3} & \dots & x_{n,m} \end{bmatrix}$$

Each column, m, of this matrix contains the features of your data and each row, n, is a sample of your data. A single sample of your data is denoted by its subscript, e.g. $\mathbf{x}_i = [x_{i,1} \; x_{i,2} \; x_{i,3} \; \cdots \; x_{i,m}]$

In supervised learning, your labels or targets are stored in a vector:

$$\mathbf{y} \in \mathbb{R}^{n \times 1} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{bmatrix}$$

Note that number of elements in the vector $\mathbf{y}$ is equal to the number of samples n in your data matrix $\mathbf{X}$, hence $\mathbf{y} \in \mathbb{R}^{n \times 1}$.

For a concrete example, let us look at the famous Iris dataset. The Iris flower dataset is a small toy dataset consisting of $n = 150$ *samples* or *observations* of three species of Iris flower (Iris setosa, Iris virginica, and Iris versicolor). Each sample, or row, has $m = 4$ *features*, which are measurements relating to that sample, such as the petal length and petal width. Therefore, the features of the Iris dataset correspond to the columns in Table 2, namely sepal length, sepal width, petal length, and petal width. Each observation or sample corresponds to one row in the table. Table 2 shows a few rows of the Iris dataset so that you can become acquainted with how it is structured. As we will be using this dataset in several sections of this chapter, take a few moments to examine it.

Table 2. The Iris flower dataset.

	Sepal length	Sepal width	Petal length	Petal width	Class
1	5.1	3.5	1.4	0.2	setosa
2	4.9	3.0	1.4	0.2	setosa
3	4.7	3.2	1.3	0.2	setosa
⋮	⋮	⋮	⋮	⋮	⋮
150	5.9	3.0	5.1	1.8	virginica

In a machine learning task, you would store this table in a matrix $\mathbf{X}$, where $\mathbf{X} \in \mathbb{R}^{150 \times 4}$. In Python $\mathbf{X}$ would therefore be stored in a 2-dimensional array with 150 rows and 4 columns (generally we will store such data in a variable named `X`). The 1st row in Table 2 corresponds to 1st row of $\mathbf{X}$, namely $\mathbf{x}_1 = [5.1\ 3.5\ 1.4\ 0.2]$. See Listing 17 for how to represent a vector as an array and a matrix as a two-dimensional array in Python. While the data is stored in a matrix $\mathbf{X}$, the Class column in Table 2, which represents the species of plant, is stored separately in a target vector $\mathbf{y}$. This vector contains what are known as the *targets* or *labels* of your dataset. In the Iris dataset, $\mathbf{y} = [y_1\ y_2\ \cdots\ y_{150}], \quad y_i \in \{setosa, versicolor, virginica\}$. The labels can either be nominal, as is the case in the Iris dataset, or continuous. In a supervised machine learning problem, the principle aim is to **predict the label for a given sample**. If the targets are nominal, this is a classification problem. If the targets are continuous this is a regression problem. In an unsupervised machine learning task you do not have the target vector $\mathbf{y}$, and you only have access to the dataset $\mathbf{X}$. In such a scenario, the aim is to find patterns in the dataset $\mathbf{X}$ and cluster observations together, for example.

We will see examples of both classification algorithms and regression algorithms in this chapter as well as supervised and unsupervised problems.

```
>>> v1 = [5.1, 3.5, 1.4, 0.2]
>>> v2 = [
...      [5.1, 3.5, 1.4, 0.2],
...      [4.9, 3.0, 1.3, 0.2]
..       ]
```

Listing 17. Creating 1-dimensional (`v1`) and 2-dimensional data structures (`v2`) in Python (Note that in Python these are called lists).

In situations where your data is split into subsets, such as a training set and a test set, you will see notation such as $\mathbf{X}_{\text{train}}$ and $\mathbf{X}_{\text{test}}$. Datasets are often split into a training set and a test set, where the training set is used to learn a model, and the test set is used to check how well the model fits to unseen data.

In a machine learning task, you will almost always be using a library known as NumPy to handle vectors and matrices. NumPy provides very useful matrix manipulation and data structure functionality and is optimised for speed. NumPy is the *de facto* standard for data input, storage, and output in the Python machine learning and data science community[1]. Another important library which is frequently used is the Pandas library for time series and tabular data structures. These packages compliment each other, and are often used side by side in a typical data science stack. We will learn the basics of NumPy and Pandas in this chapter, starting with NumPy in Sect. 7.2.

[1] To speed up certain numerical operations, the `numexpr` and `bottleneck` optimised libraries for Python can be installed. These are included in the Anaconda distribution, readers who are not using Anaconda are recommended to install them both.

7.2 NumPy

NumPy is a general data structures, linear algebra, and matrix manipulation library for Python. Its syntax, and how it handles data structures and matrices is comparable to that of MATLAB[2].

To use NumPy, first import it (the convention is to import it as `np`, to avoid having to type out `numpy` each time):

```
>>> import numpy as np
```

Listing 18. Importing NumPy. It is convention to import NumPy as `np`.

Rather than repeat this line for each code listing, we will assume you have imported NumPy, as per the instructions in Sect. 3. Any further imports that may be required for a code sample will be explicitly mentioned.

Listing 19 describes some basic usage of NumPy by first creating a NumPy array and then retrieving some of the elements of this array using a technique called *array slicing*:

```
 1 >>> vector = np.arange(10) # Make an array from 0 - 9
 2 >>> vector
 3 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
 4 >>> vector[1]
 5 1
 6 >>> vector[0:3]
 7 [0, 1, 2]
 8 >>> vector[0:-3] # Element 0 to the 3rd last element
 9 [0, 1, 2, 3, 4, 5, 6]
10 >>> vector[3:7] # From index 3 but not including 7
11 [3, 4, 5, 6]
```

Listing 19. Array slicing in NumPy.

In Listing 19, Line 1 we have created a vector (actually a NumPy array) with 10 elements from 0–9. On Line 2 we simply print the contents of the vector, the contents of which are shown on Line 3. Arrays in Python are 0-indexed, that means to retrieve the first element you must use the number 0. On Line 4 we retrieve the 2$^{\text{nd}}$ element which is 1, using the square bracket indexing syntax: `array[i]`, where `i` is the index of the value you wish to retrieve from `array`. To retrieve subsets of arrays we use a method known as *array slicing*, a powerful technique that you will use constantly, so it is worthwhile to study its usage carefully! For example, on Line 9 we are retrieving all elements beginning with element 0 to the 3rd last element. Slicing 1D arrays takes the form `array[<startpos>:<endpos>]`, where the start position `<startpos>` and end position `<endpos>` are separated with a `:` character. Line 11 shows another example of array slicing. Array slicing **includes** the element indexed by the `<startpos>` up to but **not including** the element indexed by `<endpos>`.

[2] Users of MATLAB may want to view this excellent guide to NumPy for MATLAB users: http://mathesaurus.sourceforge.net/matlab-numpy.html.

A very similar syntax is used for indexing 2-dimensional arrays, but now we must index first the rows we wish to retrieve followed by the columns we wish to retrieve. Some examples are shown below:

```
>>> m = np.arange(9).reshape(3,3)
>>> m
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
>>> m[0] # Row 0
array([0, 1, 2])
>>> m[0, 1] # Row 0, column 1
1
>>> m[:, 0] # All rows, 0th column
array([0, 3, 6])
>>> m[:,:] # All rows, all columns
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
>>> m[-2:, -2:] # Lower right corner of matrix
array([[4, 5],
       [7, 8]])
>>> m[:2, :2] # Upper left corner of matrix
array([[0, 1],
       [3, 4]])
```

Listing 20. 2D array slicing.

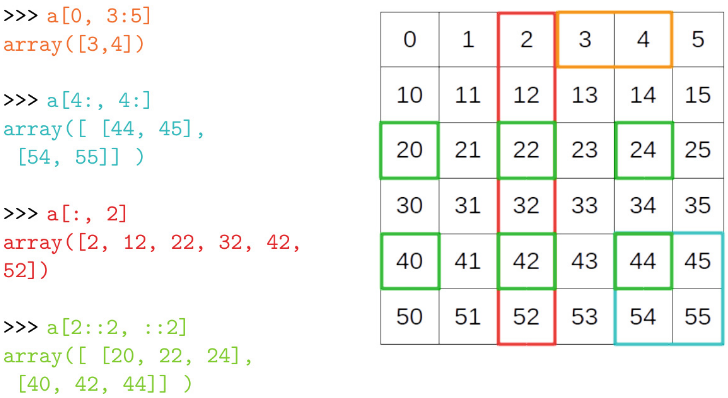
As can be seen, indexing 2-dimensional arrays is very similar to the 1-dimensional arrays shown previously. In the case of 2-dimensional arrays, you first specify your row indices, follow this with a comma (,) and then specify your column indices. These indices can be ranges, as with 1-dimensional arrays. See Fig. 2 for a graphical representation of a number of array slicing operations in NumPy.

With element wise operations, you can apply an operation (very efficiently) to every element of an n-dimensional array:

```
>>> m + 1
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
>>> m**2 # Square every element
array([[ 0,  1,  4],
       [ 9, 16, 25],
       [36, 49, 64]])
>>> v * 10
array([ 10,  20,  30,  40,  50,  60,  70,  80,  90, 100])
>>> v = np.array([1, 2, 3])
>>> v
array([1, 2, 3])
>>> m + v
```

```
15 array([[ 1,  3,  5],
16        [ 4,  6,  8],
17        [ 7,  9, 11]])
18 >>> m * v
19 array([[ 0,  2,  6],
20        [ 3,  8, 15],
21        [ 6, 14, 24]])
```

Listing 21. Element wise operations and array broadcasting.



```
>>> a[0, 3:5]
array([3,4])

>>> a[4:, 4:]
array([ [44, 45],
 [54, 55]] )

>>> a[:, 2]
array([2, 12, 22, 32, 42,
52])

>>> a[2::2, ::2]
array([ [20, 22, 24],
 [40, 42, 44]] )
```

0	1	2	3	4	5
10	11	12	13	14	15
20	21	22	23	24	25
30	31	32	33	34	35
40	41	42	43	44	45
50	51	52	53	54	55

Fig. 2. Array slicing and indexing with NumPy. Image has been redrawn from the original at http://www.scipy-lectures.org/_images/numpy_indexing.png.

There are a number of things happening in Listing 21 that you should be aware of. First, in Line 1, you will see that if you apply an operation on a matrix or array, the operation will be applied *element wise* to each item in the n-dimensional array. What happens when you try to apply an operation using, let's say a vector and a matrix? On lines 14 and 18 we do exactly this. This is known as *array broadcasting*, and works when two data structures share at least one dimension size. In this case we have a 3×3 matrix and are performing an operation using a vector with 3 elements.

7.3 Pandas

Pandas is a software library for data analysis of tabular and time series data. In many ways it reproduces the functionality of R's DataFrame object. Also, many common features of Microsoft Excel can be performed using Pandas, such as "group by", table pivots, and easy column deletion and insertion.

Pandas' DataFrame objects are label-based (as opposed to index-based as is the case with NumPy), so that each column is typically given a name which can be called to perform operations. DataFrame objects are more similar to spreadsheets, and each column of a DataFrame can have a different type, such as boolean, numeric, or text. Often, it should be stressed, you will use NumPy and Pandas in conjunction. The two libraries complement each other and are not competing frameworks, although there is overlap in functionality between the two. First, we must get some data. The Python package SciKit-Learn provides some sample data that we can use (we will learn more about SciKit-Learn later). SciKit-Learn is part of the standard Anaconda distribution.

In this example, we will load some sample data into a Pandas DataFrame object, then rename the DataFrame object's columns, and lastly take a look at the first three rows contained in the DataFrame:

```
>>> import pandas as pd # Convention
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
>>> df = pd.DataFrame(iris.data)
>>> df.columns = ["sepal_l", "sepal_w", "petal_l",
    "petal_w"]
>>> df.head(3)
   sepal_l  sepal_w  petal_l  petal_w
0      5.1      3.5      1.4      0.2
1      4.9      3.0      1.4      0.2
2      4.7      3.2      1.3      0.2
```

Listing 22. Reading data into a Pandas DataFrame.

Selecting columns can performed using square brackets or dot notation:

```
>>> df["sepal_l"]
0     5.1
1     4.9
2     4.7
...
>>> df.sepal_l # Alternatively
0     5.1
1     4.9
2     4.7
...
```

Listing 23. Accessing columns using Pandas' syntax.

You can use square brackets to access individual cells of a column:

```
>>> df["sepal_l"][0]
5.1
>>> df.sepal_l[0] # Alternatively
5.1
```

Listing 24. Accessing individual cells of a DataFrame.

To insert a column, for example the species of the plant, we can use the following syntax:

```
1 >>> df["name"] = iris.target
2 >>> df.loc[df.name == 0, "name"] = "setosa"
3 >>> df.loc[df.name == 1, "name"] = "versicolor"
4 >>> df.loc[df.name == 2, "name"] = "virginica"
5 # Alternatively
6 >>> df["name"] = [iris.target_names[x] for x in iris.
      target]
```

Listing 25. Inserting a new column into a DataFrame and replacing its numerical values with text.

In Listing 25 above, we created a new column in our DataFrame called `name`. This is a numerical class label, where 0 corresponds to setosa, 1 corresponds to versicolor, and 2 corresponds to virginica. First, we add the new column on Line 1, and we then replace these numerical values with text, shown in Lines 2–4. Alternatively, we could have just done this in one go, as shown on Line 6 (this uses a more advanced technique called a list comprehension).

We use the `loc` and `iloc` keywords for selecting rows and columns, in this case the 0$^{\text{th}}$ row:

```
>>> df.iloc[0]
sepal_l        5.1
sepal_w        3.5
petal_l        1.4
petal_w        0.2
name        setosa
Name: 0, dtype: object
```

Listing 26. The `iloc` function is used to access items within a DataFrame by their index rather than their label.

You use `loc` for selecting with labels, and `iloc` for selecting with indices. Using `loc`, we first specify the rows, then the columns, in this case we want the first three rows of the `sepal_l` column:

```
>>> df.loc[:3, "sepal_l"]
0    5.1
1    4.9
2    4.7
3    4.6
Name: sepal_l, dtype: float64
```

Listing 27. Using the `loc` function also allows for more advanced commands.

Because we are selecting a column using a label, we use the `loc` keyword above. Here we select the first 5 rows of the DataFrame using `iloc`:

```
>>> df.iloc[:5]
   sepal_l  sepal_w  petal_l  petal_w    name
0      5.1      3.5      1.4      0.2  setosa
1      4.9      3.0      1.4      0.2  setosa
2      4.7      3.2      1.3      0.2  setosa
3      4.6      3.1      1.5      0.2  setosa
4      5.0      3.6      1.4      0.2  setosa
5      5.4      3.9      1.7      0.4  setosa
```

Listing 28. Selecting the first 5 rows of the DataFrame using the `iloc` function. To select items using text labels you must use the `loc` keyword.

Or rows 15 to 20 and columns 2 to 4:

```
>>> df.iloc[15:21, 2:5]
    petal_l  petal_w    name
15      1.5      0.4  setosa
16      1.3      0.4  setosa
17      1.4      0.3  setosa
18      1.7      0.3  setosa
19      1.5      0.3  setosa
20      1.7      0.2  setosa
```

Listing 29. Selecting specific rows and columns. This is done in much the same way as NumPy.

Now, we may want to quickly examine the DataFrame to view some of its properties:

```
>>> df.describe()
          sepal_l     sepal_w     petal_l     petal_w
count  150.000000  150.000000  150.000000  150.000000
mean     5.843333    3.054000    3.758667    1.198667
std      0.828066    0.433594    1.764420    0.763161
min      4.300000    2.000000    1.000000    0.100000
25%      5.100000    2.800000    1.600000    0.300000
50%      5.800000    3.000000    4.350000    1.300000
75%      6.400000    3.300000    5.100000    1.800000
max      7.900000    4.400000    6.900000    2.500000
```

Listing 30. The `describe` function prints some commonly required statistics regarding the DataFrame.

You will notice that the `name` column is not included as Pandas quietly ignores this column due to the fact that the column's data cannot be analysed in the same way.

Sorting is performed using the `sort_values` function: here we sort by the sepal length, named `sepal_l` in the DataFrame:

```
>>> df.sort_values(by="sepal_l", ascending=True).head(5)
    sepal_l  sepal_w  petal_l  petal_w    name
13      4.3      3.0      1.1      0.1  setosa
42      4.4      3.2      1.3      0.2  setosa
38      4.4      3.0      1.3      0.2  setosa
8       4.4      2.9      1.4      0.2  setosa
41      4.5      2.3      1.3      0.3  setosa
```

Listing 31. Sorting a DataFrame using the `sort_values` function.

A very powerful feature of Pandas is the ability to write conditions within the square brackets:

```
>>> df[df.sepal_l > 7]
     sepal_l  sepal_w  petal_l  petal_w       name
102      7.1      3.0      5.9      2.1  virginica
105      7.6      3.0      6.6      2.1  virginica
107      7.3      2.9      6.3      1.8  virginica
109      7.2      3.6      6.1      2.5  virginica
117      7.7      3.8      6.7      2.2  virginica
118      7.7      2.6      6.9      2.3  virginica
122      7.7      2.8      6.7      2.0  virginica
125      7.2      3.2      6.0      1.8  virginica
129      7.2      3.0      5.8      1.6  virginica
130      7.4      2.8      6.1      1.9  virginica
131      7.9      3.8      6.4      2.0  virginica
135      7.7      3.0      6.1      2.3  virginica
```

Listing 32. Using a condition to select a subset of the data can be performed quickly using Pandas.

New columns can be easily inserted or removed (we saw an example of a column being inserted in Listing 25, above):

```
1 >>> sepal_l_w = df.sepal_l + df.sepal_w
2 >>> df["sepal_l_w"] = sepal_l_w # Creates a new column
3 >>> df.head(5)
4    sepal_l  sepal_w  petal_l  petal_w    name  sepal_l_w
5 0      5.1      3.5      1.4      0.2  setosa        8.6
6 1      4.9      3.0      1.4      0.2  setosa        7.9
7 2      4.7      3.2      1.3      0.2  setosa        7.9
8 3      4.6      3.1      1.5      0.2  setosa        7.7
9 4      5.0      3.6      1.4      0.2  setosa        8.6
```

Listing 33. Adding and removing columns

There are few things to note here. On Line 1 of Listing 33 we use the dot notation to access the DataFrame's columns, however we could also have said `sepal_l_w = df["sepal_l"] + df["sepal_w"]` to access the data in each column. The next important thing to notice is that you can insert a new column easily by specifying a label that is new, as in Line 2 of Listing 33. You can delete a column using the `del` keyword, as in `del df["sepal_l_w"]`.

Missing data is often a problem in real world datasets. Here we will remove all cells where the value is greater than 7, replacing them with NaN (Not a Number):

```
1 >>> import numpy as np
2 >>> len(df)
3 150
4 >>> df[df > 7] = np.NaN
5 >>> df = df.dropna(how="any")
6 >>> len(df)
7 138
```

Listing 34. Dropping rows that contain missing data.

After replacing all values greater than 7 with NaN (Line 4), we used the `dropna` function (Line 5) to remove the 12 rows with missing values. Alternatively you may want to replace NaN values with a value with the `fillna` function, for example the mean value for that column:

```
1 >>> for col in df.columns:
2 ...     df[col] = df[col].fillna(value=df[col].mean())
```

Listing 35. Replacing missing data with mean values.

As if often the case with Pandas, there are several ways to do everything, and we could have used either of the following:

```
1 >>> df.fillna(lambda x: x.fillna(value=x.mean())
2 >>> df.fillna(df.mean())
```

Listing 36. Demonstrating several ways to handle missing data.

Line 1 demonstrates the use of a lambda function: these are functions which are not declared and are a powerful feature of Python. Either of the above examples in Listing 36 are preferred to the loop shown in Listing 35. Pandas offers a number of methods for handling missing data, including advanced interpolation techniques[3].

Plotting in Pandas uses matplotlib (more on which later), where publication quality prints can be created, for example you can quickly create a scatter matrix, a frequently used plot in data exploration to find correlations:

```
1 >>> from pandas.tools.plotting import scatter_matrix
2 >>> scatter_matrix(df, diagonal="kde")
```

Listing 37. Several plotting features are built in to Pandas including scatter matrix functionality as shown here.

Which results in the scatter matrix seen in Fig. 3. You can see that Pandas is intelligent enough not to attempt to print the `name` column—these are known as nuisance columns and are silently, and temporarily, dropped for certain operations. The `kde` parameter specifies that you would like density plots

[3] See http://pandas.pydata.org/pandas-docs/stable/missing_data.html for more methods on handling missing data.

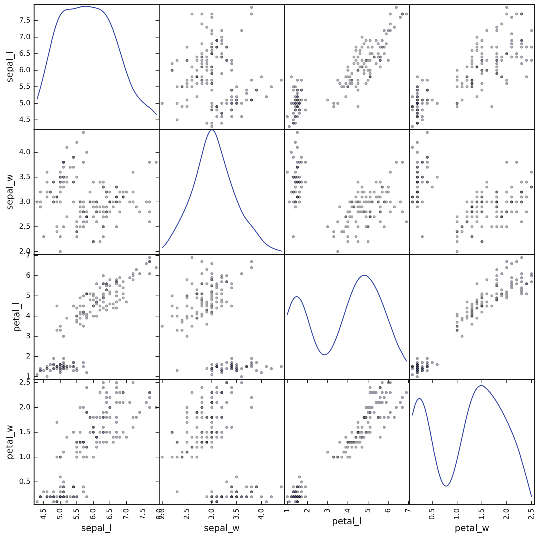


Fig. 3. Scatter matrix visualisation for the Iris dataset.

for the diagonal axis of the matrix, alternatively you can specify `hist` to plot histograms.

For many examples on how to use Pandas, refer to the SciPy Lectures website under http://www.scipy-lectures.org/. Much more on plotting and visualisation in Pandas can be found in the Pandas documentation, under: http://pandas.pydata.org/pandas-docs/stable/visualization.html. Last, a highly recommendable book on the topic of Pandas and NumPy is *Python for Data Analysis* by Wes McKinney [9].

Now that we have covered an introduction into Pandas, we move on to visualisation and plotting using matplotlib and Seaborn.

8 Data Visualisation and Plotting

In Python, a commonly used 2D plotting library is matplotlib. It produces publication quality plots, an example of which can be seen in Fig. 4, which is created as follows:

```
>>> import matplotlib.pyplot as plt # Convention
>>> x = np.random.randint(100, size=25)
>>> y = x*x
>>> plt.scatter(x, y); plt.show()
```

Listing 38. Plotting with matplotlib

This tutorial will not cover matplotlib in detail. We will, however, mention the Seaborn project, which is a high level abstraction of matplotlib, and has the added advantage that is creates better looking plots by default. Often all that is

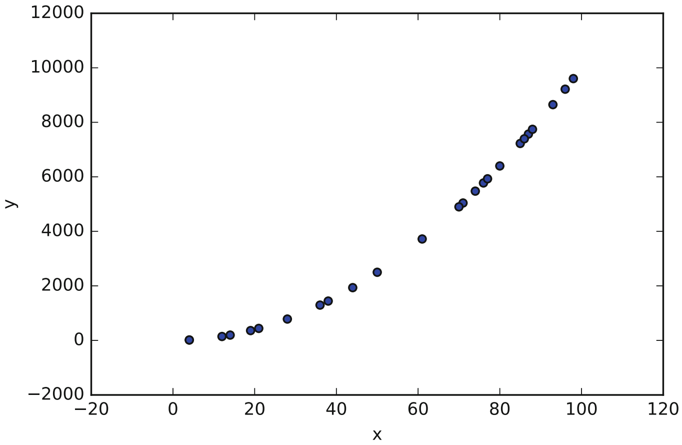


Fig. 4. A scatter plot using matplotlib.

necessary is to import Seaborn, and plot as normal using matplotlib in order to profit from these superior looking plots. As well as better looking default plots, Seaborn has a number of very useful APIs to aid commonly performed tasks, such as `factorplot`, `pairplot`, and `jointgrid`.

Seaborn can also perform quick analyses on the data itself. Listing 39 shows the same data being plotted, where a linear regression model is also fit by default:

```
>>> import seaborn as sns # Convention
>>> sns.set() # Set defaults
>>> x = np.random.randint(100, size=25)
>>> y = x*x
>>> df = pd.DataFrame({"x": x, "y": y})
>>> sns.lmplot(x="x", y="y", data=df); plt.show()
```

Listing 39. Plotting with Seaborn.

This will output a scatter plot but also will fit a linear regression model to the data, as seen in Fig. 5.

For plotting and data exploration, Seaborn is a useful addition to the data scientist's toolbox. However, matplotlib is more often than not the library you will encounter in tutorials, books, and blogs, and is the basis for libraries such as Seaborn. Therefore, knowing how to use both is recommended.

9 Machine Learning

We will now move on to the task of machine learning itself. In the following sections we will describe how to use some basic algorithms, and perform regression, classification, and clustering on some freely available medical datasets concerning breast cancer and diabetes, and we will also take a look at a DNA microarrray dataset.

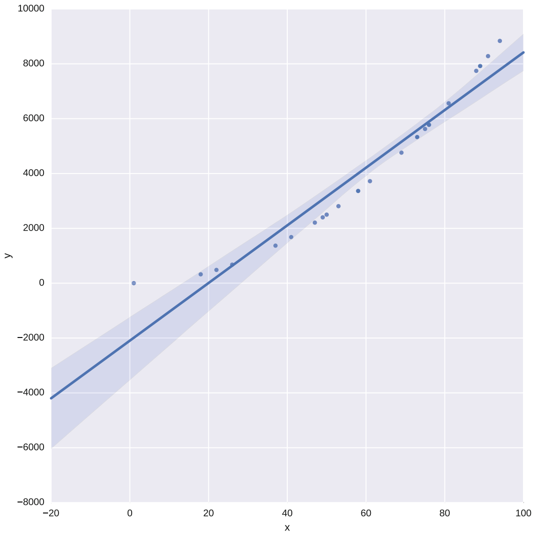


Fig. 5. Seaborn's `lmplot` function will fit a line to your data, which is useful for quick data exploration.

9.1 SciKit-Learn

SciKit-Learn provides a standardised interface to many of the most commonly used machine learning algorithms, and is the most popular and frequently used library for machine learning for Python. As well as providing many learning algorithms, SciKit-Learn has a large number of convenience functions for common preprocessing tasks (for example, normalisation or k-fold cross validation). SciKit-Learn is a very large software library. For tutorials covering nearly all aspects of its usage see http://scikit-learn.org/stable/documentation.html. Several tutorials in this chapter followed the structure of examples found on the SciKit-Learn documentation website [10].

9.2 Linear Regression

In this example we will use a diabetes dataset that is available from SciKit-Learn's `datasets` package.

The diabetes dataset consists of 442 samples (the patients) each with 10 features. The features are the patient's age, sex, body mass index (BMI), average blood pressure, and six blood serum values. The target is the disease progression. We wish to investigate if we can fit a linear model that can accurately predict the disease progression of a new patient given their data.

For visualisation purposes, however, we shall only take one of the features of the dataset, namely the Body Mass Index (BMI). So we shall investigate if there is correlation between BMI and disease progression (bmi and prog in Table 3).

First, we will load the data and prepare it for analysis:

Table 3. Diabetes dataset

	age	sex	bmi	map	tc	ldl	hdl	tch	ltg	glu	prog
1	0.038	0.050	0.061	0.021	−0.044	−0.034	−0.043	−0.002	0.019	−0.017	151
2	−0.001	−0.044	−0.051	−0.026	−0.008	−0.019	0.074	−0.039	−0.068	−0.092	75
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
442	−0.045	−0.044	−0.073	−0.081	0.083	0.027	0.173	−0.039	−0.004	0.003	57

```
1  >>> from sklearn import datasets, linear_model
2  >>> d = datasets.load_diabetes()
3  >>> X = d.data
4  >>> y = d.target
5  >>> np.shape(X)
6  (442, 10)
7  >>> X = X[:,2] # Take only the BMI column (index 2)
8  >>> X = X.reshape(-1, 1)
9  >>> y = y.reshape(-1, 1)
10 >>> np.shape(X)
11 (442, 1)
12 >>> X_train = X[:-80] # We will use 80 samples for testing
13 >>> y_train = y[:-80]
14 >>> X_test = X[-80:]
15 >>> y_test = y[-80:]
```

Listing 40. Loading a diabetes dataset and preparing it for analysis.

Note once again that it is convention to store your target in a variable called $\mathbf{y}$ and your data in a matrix called $\mathbf{X}$ (see Sect. 7.1 for more details). In the example above, we first load the data in Lines 2–4, we then extract only the 3$^{\text{rd}}$ column, discarding the remaining 9 columns. Also, we split the data into a training set, $\mathbf{X}_{\text{train}}$, shown in the code as `X_train` and a test set, $\mathbf{X}_{\text{test}}$, shown in the code as `X_test`. We did the same for the target vector $\mathbf{y}$. Now that the data is prepared, we can train a linear regression model on the training data:

```
1 >>> linear_reg = linear_model.LinearRegression()
2 >>> linear_reg.fit(X_train, y_train)
3 LinearRegression(copy_X=True, fit_intercept=True, n_jobs
      =1, normalize=False)
4 >>> linear_reg.score(X_test, y_test)
5 0.36469910696163765
```

Listing 41. Fitting a linear regression model to the data.

As we can see, after fitting the model to the training data (Lines 1–2), we test the trained model on the test set (Line 4). Plotting the data is done as follows:

```
>>> plt.scatter(X_test, y_test)
>>> plt.plot(X_test, linear_reg.predict(X_test))
>>> plt.show()
```

Listing 42. Plotting the results of the trained model.

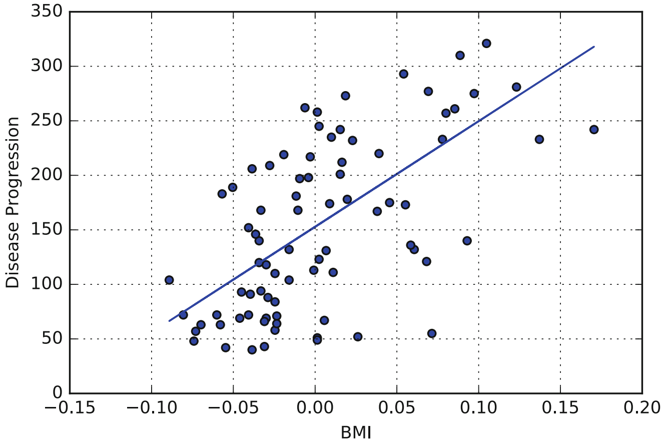


Fig. 6. A model generated by linear regression showing a possible correlation between Body Mass Index and diabetes disease progression.

A similar output to that shown in Fig. 6 will appear.

Because we wished to visualise the correlation in 2D, we extracted only one feature from the dataset, namely the Body Mass Index feature. However, there is no reason why we need to remove features in order to plot possible correlations. In the next example we will use Ridge regression on the diabetes dataset maintaining 9 from 10 of its features (we will discard the gender feature for simplicity as it is a nominal value). First let us split the data and apply it to a Ridge regression algorithm:

```
>>> from sklearn import cross_validation
>>> from sklearn.preprocessing import normalize
>>> X = datasets.load_diabetes().data
>>> y = datasets.load_diabetes().target
>>> y = np.reshape(y, (-1,1))
>>> X = np.delete(X, 1, 1) # remove col 1, axis 1
>>> X_train, X_test, y_train, y_test = cross_validation.
    train_test_split(X, y, test_size=0.2)
```

Listing 43. Preparing a cross validation dataset.

We now have a shuffled train and test split using the `cross_validation` function (previously we simply used the last 80 observations in **X** as a test set, which can be problematic—proper shuffling of your dataset before creating a train/test split is almost always a good idea).

Now that we have preprocessed the data correctly, and have our train/test splits, we can train a model on the training set `X_train`:

```
>>> ridge = linear_model.Ridge(alpha=0.0001)
>>> ridge.fit(X_train, y_train)
Ridge(alpha=0.0001, copy_X=True, fit_intercept=True,
    max_iter=None, normalize=False, random_state=None,
    solver="auto", tol=0.001)
>>> ridge.score(X_test, y_test)
0.52111236634294411
>>> y_pred = ridge.predict(X_test)
```

Listing 44. Training a ridge regression model on the diabetes dataset.

We have made out predictions, but how do we plot our results? The linear regression model was built on 9-dimensional data set, so what exactly should we plot? The answer is to plot the predicted outcome versus the actual outcome for the test set, and see if this follows any kind of a linear trend. We do this as follows:

```
>>> plt.scatter(y_test, y_pred)
>>> plt.plot([y.min(), y.max()], [y.min(), y.max()])
```

Listing 45. Plotting the predicted versus the actual values.

The resulting plot can be see in Fig. 7.

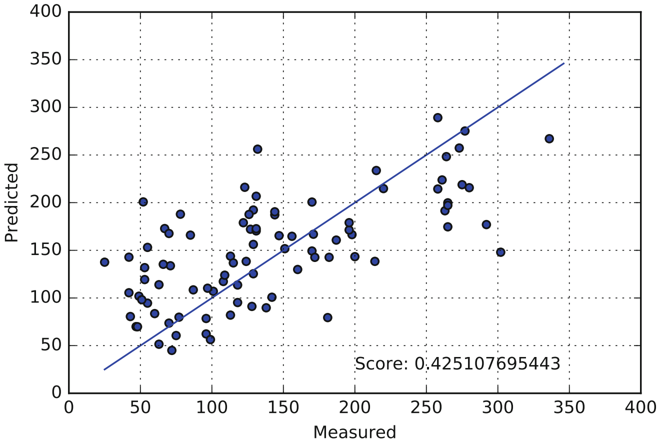


Fig. 7. Plotting the predicted versus the actual values in the test set, using a model trained on a separate training set.

However, you may have noticed a slight problem here: if we had taken a different test/train split, we would have gotten different results. Hence it is common to perform a 10-fold cross validation:

```
>>> from sklearn.cross_validation import cross_val_score,
    cross_val_predict
>>> ridge_cv = linear_model.Ridge(alpha=0.1)
>>> score_cv = cross_val_score(ridge_cv, X, y, cv=10)
>>> score_cv.mean()
0.45358728032634499
>>> y_cv = cross_val_predict(ridge_cv, X, y, cv=10)
>>> plt.scatter(y, y_cv)
>>> plt.plot([y.min(), y.max()], [y.min(), y.max()]);
```

Listing 46. Computing the cross validated score.

The results of the 10-fold cross validated scored can see in Fig. 8.

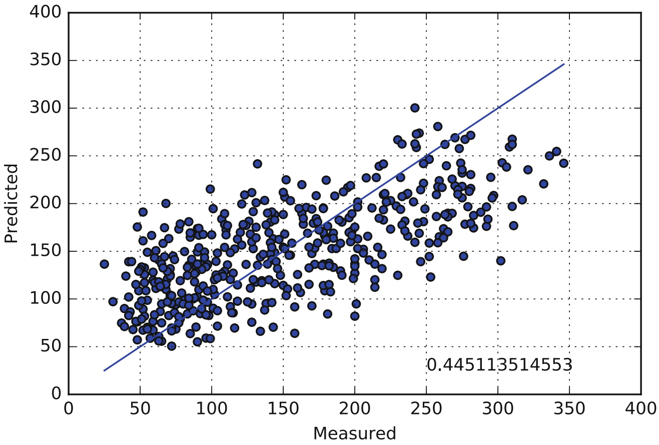


Fig. 8. Plotting predictions versus the actual values using cross validation.

9.3 Non-linear Regression and Model Complexity

Many relationships between two variables are not linear, and SciKit-Learn has several algorithms for non-linear regression. One such algorithm is the Support Vector Regression algorithm, or SVR. SVR allows you to learn several types of models using different kernels. Linear models can be learned with a linear kernel, while non-linear curves can be learned using a polynomial kernel (where you can specify the degree) for example. As well as this, SVR in SciKit Learn can use a Radial Basis Function, Sigmoid function, or your own custom kernel.

For example, the code below will produce similar data to the examples shown in Sect. 8.

```
1 >>> x = np.random.rand(100)
2 >>> y = x*x
3 >>> y[::5] += 0.4 # add 0.4 to every 5th item
4 >>> x.sort()
5 >>> y.sort()
6 >>> plt.scatter(x, y); plot.show();
```

Listing 47. Creating a non-smooth curve dataset for demonstration of various regression techniques.

This will produce data similar to what is seen in Fig. 9. The data describes an almost linear relationship between x and y. We added some noise to this in Line 3 of Listing 47.

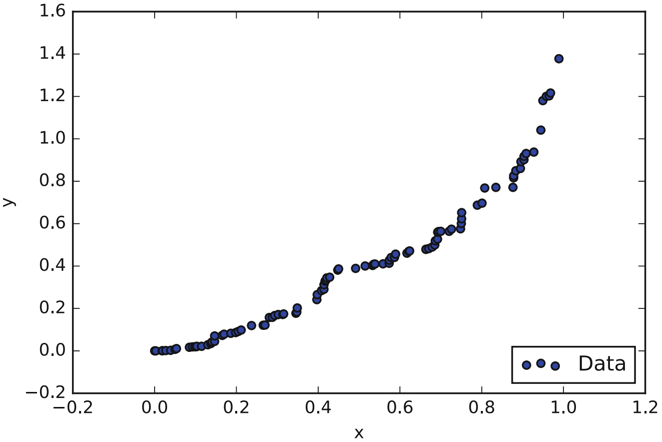


Fig. 9. The generated dataset which we will fit our regression models to.

Now we will fit a function to this data using an SVR with a linear kernel. The code for this is as follows:

```
>>> lin = linear_model.LinearRegression()
>>> x = x.reshape(-1, 1)
>>> y = y.reshape(-1, 1)
>>> lin.fit(x,y)
LinearRegression(copy_X=True, fit_intercept=True, n_jobs
    =1, normalize=False)
>>> lin.fit(x, y)
>>> lin.score(x, y)
0.9222025374710559
```

Listing 48. Training a linear regression model on the generated non-linear data.

We will now plot the result of the fitted model over the data, to see for ourselves how well the line fits the data:

```
>>> plt.scatter(x,y, label="Data")
>>> plt.plot(x, lin.predict(x))
>>> plot.show()
```

Listing 49. Plotting the linear model's predictions.

The result of this code can be seen in Fig. 10.

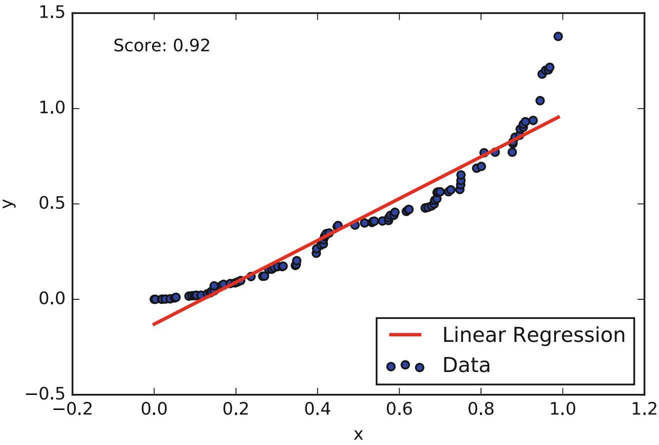


Fig. 10. Linear regression on a demonstration dataset.

While this does fit the data quite well, we can do better—but not with a linear function. To achieve a better fit, we will now use a non-linear regression model, an SVR with a polynomial kernel of degree 3. The code to fit a polynomial SVR is as follows:

```
>>> from sklearn.svm import SVR
>>> poly_svm = SVR(kernel="poly", C=1000)
>>> poly_svm.fit(x, y)
SVR(C=1000, cache_size=200, coef0=0.0, degree=3, epsilon
    =0.1,gamma="auto", kernel="poly", max_iter=-1,
    shrinking=True,tol=0.001, verbose=False)
>>> poly_svm.score(x, y)
0.94273329580447318
```

Listing 50. Training a polynomial Support Vector Regression model.

Notice that the SciKit-Learn API exposes common interfaces irregardless of the model—both the linear regression algorithm and the support vector regression algorithm are trained in exactly the same way, i.e.: they take the same basic parameters and expect the same data types and formats (`X` and `y`) as input. This makes experimentation with many different algorithms easy. Also, notice that once you have called the `fit()` function in both cases (Listings 48 and 50), a summary of the model's parameters are returned. These do, of course, vary from algorithm to algorithm.

You can see the results of this fit in Fig. 11, the code to produce this plot is as follows:

```
>>> plt.scatter(x,y, label="Data")
>>> plt.plot(x, poly_svm.predict(x))
```

Listing 51. Plotting the results of the Support Vector Regression model with polynomial kernel.

Now we will use an Radial Basis Function Kernel. This should be able to fit the data even better:

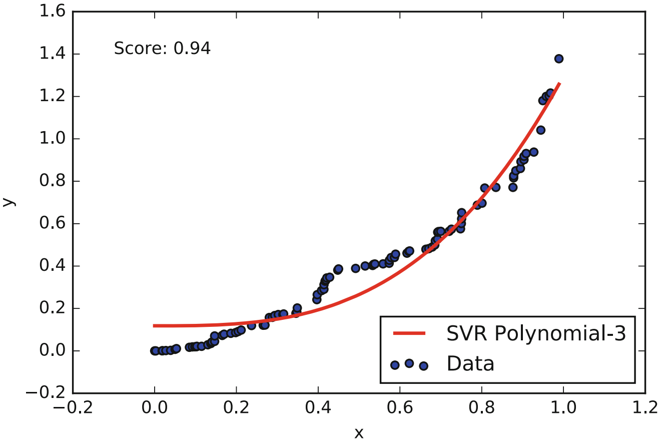


Fig. 11. Fitting a Support Vector Regression algorithm with a polynomial kernel to a sample dataset.

```
>>> rbf_svm = SVR(kernel="rbf", C=1000)
>>> rbf_svm.fit(x,y)
SVR(C=1000, cache_size=200, coef0=0.0, degree=3, epsilon
    =0.1,gamma='auto', kernel='rbf',max_iter=-1, shrinking=
    True,tol=0.001, verbose=False)
>>> rbf_svm.score(x,y)
0.9558322940993608 8
```

Listing 52. Training a Support Vector Regression model with a Radial Basis Function (RBF) kernel.

The result of this fit can be plotted:

```
>>> plt.scatter(x,y, label="Data")
>>> plt.plot(x, rbf_svm.predict(x))
```

Listing 53. Plotting the results of the RBF kernel model.

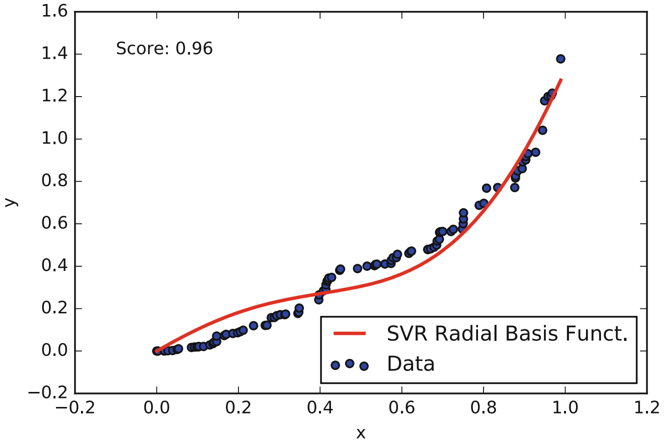


Fig. 12. Non-linear regression using a Support Vector Regression algorithm with a Radial Basis Function kernel.

The plot can be seen in Fig. 12. You will notice that this model probably fits the data best.

A Note on Model Complexity. It should be pointed out that a more complex model will **almost always** fit data better than a simpler model given the same dataset. As you increase the complexity of a polynomial by adding terms, you eventually will have as many terms as data points and you will fit the data perfectly, even fitting to outliers. In other words, a polynomial of, say, degree 4 will nearly always fit the same data better than a polynomial of degree 3—however, this also means that the more complex model could be fitting to noise. Once a model has begun to overfit it is no longer useful as a predictor to new data. There are various methods to spot overfitting, the most commonly used methods are to split your data into a training set and a test set and a method called cross validation. The simplest method is to perform a train/test split: we split the data into a training set and a test set—we then train our model on the training set but we subsequently measure the loss of the model on the held-back test set. This loss can be used to compare different models of different complexity. The best performing model will be that which minimises the loss on the test set.

Cross validation involves splitting the dataset in a way that each data sample is used once for training and for testing. In 2-fold cross validation, the data is shuffled and split into two equal parts: one half of the data is then used for training your model and the other half is used to test your model—this is then reversed, where the original test set is used to train the model and the original training set is used to test the newly created model. The performance is measured averaged across the test set splits. However, more often than not you will find that *k*-fold cross validation is used in machine learning, where, let's say, 10%

of the data is held back for testing, while the algorithm is trained on 90% of the data, and this is repeated 10 times in a stratified manner in order to get the average result (this would be 10-fold cross validation). We saw how SciKit-Learn can perform a 10-fold cross validation simply in Sect. 9.2.

9.4 Clustering

Clustering algorithms focus on ordering data together into groups. In general clustering algorithms are unsupervised—they require no $\mathbf{y}$ response variable as input. That is to say, they attempt to find groups or clusters within data where you do not know the label for each sample. SciKit-Learn has many clustering algorithms, but in this section we will demonstrate hierarchical clustering on a DNA expression microarray dataset using an algorithm from the SciPy library. We will plot a visualisation of the clustering using what is known as a dendrogram, also using the SciPy library.

In this example, we will use a dataset that is described in Sect. 14.3 of the *Elements of Statistical Learning* [11]. The microarray data are available from the book's companion website. The data comprises 64 samples of cancer tumours, where each sample consists of expression values for 6830 genes, hence $\mathbf{X} \in \mathbb{R}^{64\times 6830}$. As this is an **unsupervised** problem, there is no $\mathbf{y}$ target. First let us gather the microarray data (`ma`):

```
>>> from scipy.cluster.hierarchy import dendrogram, linkage
>>> url = "http://statweb.stanford.edu/~tibs/ElemStatLearn/datasets/nci.data"
>>> labels = ["CNS","CNS","CNS","RENAL","BREAST","CNS","CNS","BREAST","NSCLC","NSCLC","RENAL","RENAL","RENAL","RENAL","RENAL","RENAL","RENAL","BREAST","NSCLC","RENAL","UNKNOWN","OVARIAN","MELANOMA","PROSTATE","OVARIAN","OVARIAN","OVARIAN","OVARIAN","OVARIAN","PROSTATE","NSCLC","NSCLC","NSCLC","LEUKEMIA","K562B-repro","K562A-repro","LEUKEMIA","LEUKEMIA","LEUKEMIA","LEUKEMIA","LEUKEMIA","COLON","COLON","COLON","COLON","COLON","COLON","COLON","MCF7A-repro","BREAST","MCF7D-repro","BREAST","NSCLC","NSCLC","NSCLC","MELANOMA","BREAST","BREAST","MELANOMA","MELANOMA","MELANOMA","MELANOMA","MELANOMA","MELANOMA"]
>>> ma =pd.read_csv(url, delimiter="\s*", engine="python", names=labels)
>>> ma = ma.transpose()
>>> X = np.array(ma)
>>> np.shape(X)
(64, 6830)
```

Listing 54. Gathering the gene expression data and formatting it for analysis.

The goal is to cluster the data properly in logical groups, in this case into the cancer types represented by each sample's expression data. We do this using agglomerative hierarchical clustering, using Ward's linkage method:

```
>>> Z = linkage(X, "ward")
>>> dendrogram(Z, labels=labels, truncate_mode="none");
```

Listing 55. Generating a dendrogram using the SciPy package.

This will produce a dendrogram similar to what is shown in Fig. 13.

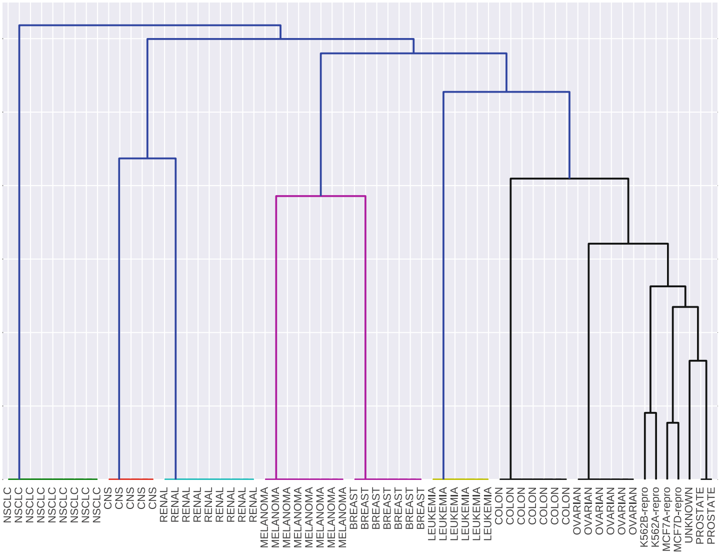


Fig. 13. Dendrogram of the hierarchical clustering of a gene expression dataset relating to cancer tumours.

Note that tumour names shown in Fig. 13 were used only to label the groupings and were not used by the algorithm (such as they might be in a supervised problem).

9.5 Classification

In Sect. 9.4 we analysed data that was **unlabelled**—we did not know to what class a sample belonged (known as unsupervised learning). In contrast to this, a supervised problem deals with **labelled** data where are aware of the discrete classes to which each sample belongs. When we wish to predict which class a sample belongs to, we call this a classification problem. SciKit-Learn has a number of algorithms for classification, in this section we will look at the Support Vector Machine.

We will work on the Wisconsin breast cancer dataset, split it into a training set and a test set, train a Support Vector Machine with a linear kernel, and test the trained model on an unseen dataset. The Support Vector Machine model should be able to predict if a new sample is malignant or benign based on the features of a new, unseen sample:

```
 1 >>> from sklearn import cross_validation
 2 >>> from sklearn import datasets
 3 >>> from sklearn.svm import SVC
 4 >>> from sklearn.metrics import classification_report
 5 >>> X = datasets.load_breast_cancer().data
 6 >>> y = datasets.load_breast_cancer().target
 7 >>> X_train, X_test, y_train, y_test = cross_validation.
        train_test_split(X, y, test_size=0.2)
 8 >>> svm = SVC(kernel="linear")
 9 >>> svm.fit(X_train, y_train)
10 SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
        decision_function_shape=None, degree=3, gamma="auto",
        kernel="linear", max_iter=-1, probability=False,
        random_state=None, shrinking=True,  tol=0.001, verbose=
        False)
11 >>> svm.score(X_test, y_test)
12 0.956140350877192 96
13 >>> y_pred = svm.predict(X_test)
14 >>> classification_report(y_test, y_pred)
15
16              precision    recall  f1-score   support
17
18   malignant       1.00      0.89      0.94        44
19      benign       0.93      1.00      0.97        70
20
21 avg / total       0.96      0.96      0.96       114
```

Listing 56. Training a Support Vector Machine to classify between malignant and benign breast cancer samples.

You will notice that the SVM model performed very well at predicting the malignancy of new, unseen samples from the test set—this can be quantified nicely by printing a number of metrics using the `classification_report` function, shown on Lines 14–21. Here, the precision, recall, and F_1 score ($F_1 = 2 \cdot {}^{\text{precision}\cdot\text{recall}}/_{\text{precision}+\text{recall}}$) for each class is shown. The support column is a count of the number of samples for each class.

Support Vector Machines are a very powerful tool for classification. They work well in high dimensional spaces, even when the number of features is higher than the number of samples. However, their running time is quadratic to the number of samples so large datasets can become difficult to train. Quadratic means that if you increase a dataset in size by 10 times, it will take 100 times longer to train.

Last, you will notice that the breast cancer dataset consisted of 30 features. This makes it difficult to visualise or plot the data. To aid in visualisation of highly dimensional data, we can apply a technique called dimensionality reduction. This is covered in Sect. 9.6, below.

9.6 Dimensionality Reduction

Another important method in machine learning, and data science in general, is dimensionality reduction. For this example, we will look at the Wisconsin breast cancer dataset once again. The dataset consists of over 500 samples, where each sample has 30 features. The features relate to images of a fine needle aspirate of breast tissue, and the features describe the characteristics of the cells present in the images. All features are real values. The target variable is a discrete value (either malignant or benign) and is therefore a classification dataset.

You will recall from the Iris example in Sect. 7.3 that we plotted a scatter matrix of the data, where each feature was plotted against every other feature in the dataset to look for potential correlations (Fig. 3). By examining this plot you could probably find features which would separate the dataset into groups. Because the dataset only had 4 features we were able to plot each feature against each other relatively easily. However, as the numbers of features grow, this becomes less and less feasible, especially if you consider the gene expression example in Sect. 9.4 which had over 6000 features.

One method that is used to handle data that is highly dimensional is Principle Component Analysis, or PCA. PCA is an unsupervised algorithm for reducing the number of dimensions of a dataset. For example, for plotting purposes you might want to reduce your data down to 2 or 3 dimensions, and PCA allows you to do this by generating components, which are combinations of the original features, that you can then use to plot your data.

PCA is an unsupervised algorithm. You supply it with your data, **X**, and you specify the number of components you wish to reduce its dimensionality to. This is known as transforming the data:

```
 1 >>> from sklearn.decomposition import PCA
 2 >>> from sklearn import datasets
 3 >>> breast = datasets.load_breast_cancer()
 4 >>> X = breast.data
 5 >>> np.shape(X)
 6 (569, 30)
 7 >>> y = breast.target
 8 >>> pca = PCA(n_components=2)
 9 >>> pca.fit(X)
10 >>> X_reduced = pca.transform(X)
11 >>> np.shape(X_reduced)
12 (569, 2)
13 >>> plt.scatter(X_reduced[:, 0], X_reduced[:, 1], c=y);
```

Listing 57. Performing dimensionality reduction on a breast cancer dataset using Principle Component Analysis.

As you can see, the original dataset had 30 dimensions, $\mathbf{X} \in \mathbb{R}^{569 \times 30}$, and after the PCA fit and transform, we have now a reduced number of dimensions, $\mathbf{X} \in \mathbb{R}^{569 \times 2}$ which we specified using the `n_components=2` parameter in Line 8 of Listing 57 above.

Now we have a reduced dataset, X_reduced, and we can now plot this, the results of which can be seen in Fig. 14. As can be seen in Fig. 14, this data may even be somewhat linearly separable. So let's try to fit a line to the data.

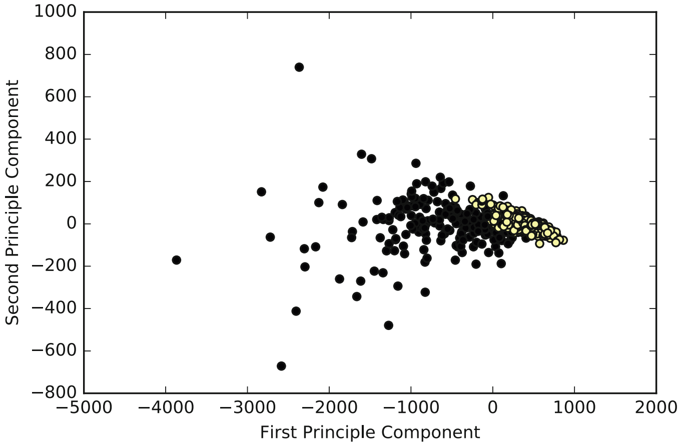


Fig. 14. The data appears somewhat linearly separable after a PCA transformation.

For this, we can use Logistic Regression—which despite its name is actually a classification algorithm:

```
>>> from sklearn.linear_model import LogisticRegression
>>> lr = LogisticRegression()
LogisticRegression(C=1.0, class_weight=None, dual=False,
    fit_intercept=True, intercept_scaling=1, max_iter=100,
    multi_class="ovr", n_jobs=1, penalty="l2", random_state
    =None, solver="liblinear", tol=0.0001, verbose=0,
    warm_start=False)
>>> lr.fit(X_reduced, y)
>>> lr.score(X_reduced, y)
0.93145869947275928
```

Listing 58. Logistic regression on the transformed PCA data.

If we plot this line (for code see the accompanying Jupyter notebook) we will see something similar to that shown in Fig. 15.

Again, you would not use this model for new data—in a real world scenario, you would, for example, perform a 10-fold cross validation on the dataset, choosing the model parameters that perform best on the cross validation. This model would be much more likely to perform well on new data. At the very least, you would randomly select a subset, say 30% of the data, as a test set and train the model on the remaining 70% of the dataset. You would evaluate the model based on the score on the test set and not on the training set.

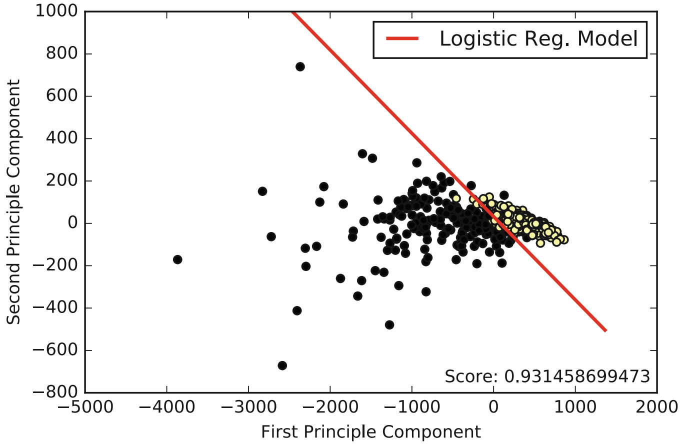


Fig. 15. Logistic Regression algorithm applied to the dimensionally reduced breast cancer dataset.

10 Neural Networks and Deep Learning

While a proper description of neural networks and deep learning is far beyond the scope of this chapter, we will however discuss an example use case of one of the most popular frameworks for deep learning: Keras[4].

In this section we will use Keras to build a simple neural network to classify the Wisconsin breast cancer dataset that was described earlier. Often, deep learning algorithms and neural networks are used to classify images—convolutional neural networks are especially used for image related classification. However, they can of course be used for text or tabular-based data as well. In this chapter we will build a standard feed-forward, densely connected neural network and classify a text-based cancer dataset in order to demonstrate the framework's usage.

In this example we are once again using the Wisconsin breast cancer dataset, which consists of 30 features and 569 individual samples. To make it more challenging for the neural network, we will use a training set consisting of only 50% of the entire dataset, and test our neural network on the remaining 50% of the data.

Note, Keras is not installed as part of the Anaconda distribution, to install it use pip:

```
$sudo pip install keras
```

Listing 59. As Keras is not part of the Anaconda distribution it must be installed separately using pip.

[4] For some metrics, see the Keras author's tweet: https://twitter.com/fchollet/status/765212287531495424.

Keras additionally requires either Theano or TensorFlow to be installed. In the examples in this chapter we are using Theano as a backend, however the code will work identically for either backend. You can install Theano using pip, but it has a number of dependencies that must be installed first. Refer to the Theano and TensorFlow documentation for more information [12].

Keras is a modular API. It allows you to create neural networks by building a stack of modules, from the input of the neural network, to the output of the neural network, piece by piece until you have a complete network. Also, Keras can be configured to use your Graphics Processing Unit, or GPU. This makes training neural networks far faster than if we were to use a CPU. We begin by importing Keras:

```
1 >>> import keras
2 Using Theano backend.
3 Using gpu device 0: GeForce GTX TITAN X (CNMeM is enabled
     with initial size: 90.0 % of memory, cuDNN 4007)
```

Listing 60. Importing Keras will output important information regarding which GPU (if any) the framework has access to.

We import the Keras library on Line 1. This will output a few informational messages (Lines 2–3), which are important and can highlight configuration or driver issues relating to your GPU. If there are errors, you may want to check the Theano configuration `~/.theanorc`, or the Keras configuration file `~/.keras/keras.json`.

Now we will begin to build a network. In this case, we will build a sequential neural network with an input layer, one hidden layer, and an output layer. The input layer is used to read in the data you wish to analyse. The hidden layer will perform some operations on the input data which has been read in by the input layer. The output layer will make a classification or regression prediction based on what it receives from the hidden layer. In Keras we define this network as follows:

```
1 >>> model = Sequential()
2 >>> model.add(Dense(10, input_dim=30, init="uniform",
     activation="relu"))
3 >>> model.add(Dense(6, init="uniform", activation="relu"))
4 >>> model.add(Dense(2, init="uniform", activation="softmax
     "))
5 >>> model.compile(loss="categorical_crossentropy",
     optimizer="adamax", metrics=["accuracy"])
```

Listing 61. Defining a neural network using Keras.

The code in Listing 61, Line 1, first defines that you wish to create a sequential neural network. We then use the `add` function to add layers to the network. Deep Learning algorithms are neural networks with many layers. In this case we are only adding a small number of fully connected layers. Once we have added our input layer, hidden layers, and output layers (Lines 2–4) we can compile the network (Line 5). Compiling the network will allow us to ensure that the network we have created is valid, and it is where you define some parameters such as the type of loss function and the optimiser for this loss function. The type of loss function depends on the type of model you wish to create—for regression you might use the Mean Squared Error loss function, for example.

Once the network has compiled, you can train it using the `fit` function:

```
 1 >>> h = model.fit(X_train, y_train, nb_epoch=20,
       batch_size=10, validation_data=(X_test, y_test))
 2 Train on 284 samples, validate on 285 samples
 3 Epoch 1/20
 4 loss: 0.66 - acc: 0.54 - val_loss: 0.65 - val_acc: 0.56
 5 Epoch 2/20
 6 loss: 0.64 - acc: 0.63 - val_loss: 0.62 - val_acc: 0.71
 7 Epoch 3/20
 8 loss: 0.69 - acc: 0.66 - val_loss: 0.67 - val_acc: 0.78
 9 ...
10 Epoch 20/20
11 loss: 0.21 - acc: 0.91 - val_loss: 0.26 - val_acc: 0.90
```

Listing 62. Keras output when training a neural network.

Listing 62 shows the output of a model while it is learning (Lines 2–11). After you have called the `fit` function, the network starts training, and the accuracy and loss after each epoch (a complete run through the training set) is output for both the training set (`loss` and `acc`) and the test set (`val_loss` and `val_acc`). It important to watch all these metrics during training to ensure that you are not overfitting, for example. The most important metric is the `val_acc` metric, which outputs the current accuracy of the model at a particular epoch on the test data.

Once training is complete, we can make predictions using our trained model on new data, and then evaluate the model:

```
>>> from sklearn.metrics import classification_report
>>> y_pred = model.predict_classes(X_test)
>>> metrics.classification_report(y_test, y_pred)

             precision    recall  f1-score   support

  malignant       0.96      0.78      0.86       110
     benign       0.88      0.98      0.92       175

avg / total       0.91      0.90      0.90       285
```

Listing 63. Printing a classification report of the model's performance.

We may want to view the network's accuracy on the test (or its loss on the training set) over time (measured at each epoch), to get a better idea how well it is learning. An epoch is one complete cycle through the training data. Fortunately, this is quite easy to plot as Keras' `fit` function returns a `history` object which we can use to do exactly this:

```
>>> plt.plot(h.history["val_acc"])
>>> plt.plot(h.history["loss"])
>>> plt.show()
```

Listing 64. Plotting the accuracy and loss of the model over time (per epoch).

This will result in a plot similar to that shown in Fig. 16. Often you will also want to plot the loss on the test set and training set, and the accuracy on the test set and training set. You can see this in Figs. 17 and 18 respectively.

Plotting the loss and accuracy can be used to see if you are overfitting (you experience tiny loss on the training set, but large loss on the test set) and to see when your training has plateaued.

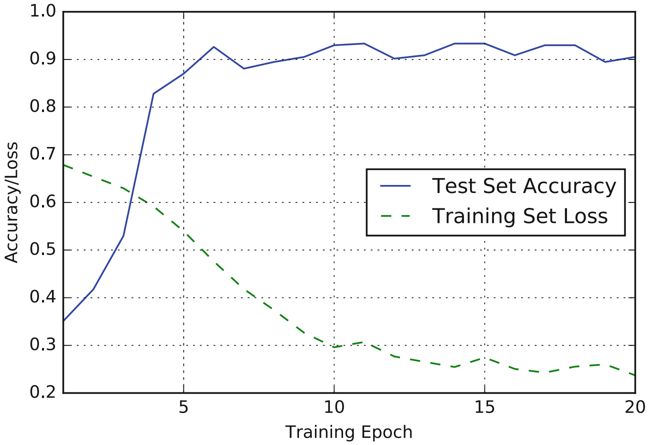


Fig. 16. The accuracy and loss over time for a neural network. In this plot, the loss is reported for the network on the training set, while the accuracy is reported measured against the test set.

We can have seen that using Keras to build a neural network and classify a medical dataset is a relatively straightforward process. Be aware that introspection into the inner workings of neural network models can be difficult to achieve. If introspection is very important, and this can be the case in medicine, then a powerful algorithm is Decision Trees, where the introspection into the workings of the trained model is possible.

11 Future Outlook

While Python has a large number of machine learning and data science tools, there are numerous other mature frameworks for other platforms and languages. In this section we shall highlight a number of other tools that are relatively new and are likely to become more mainstream in the near future.

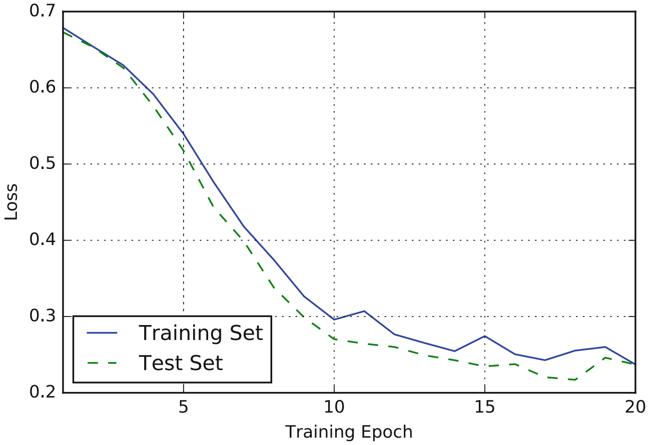


Fig. 17. The loss of the network on the test set and the training set, over time (measured at each epoch).

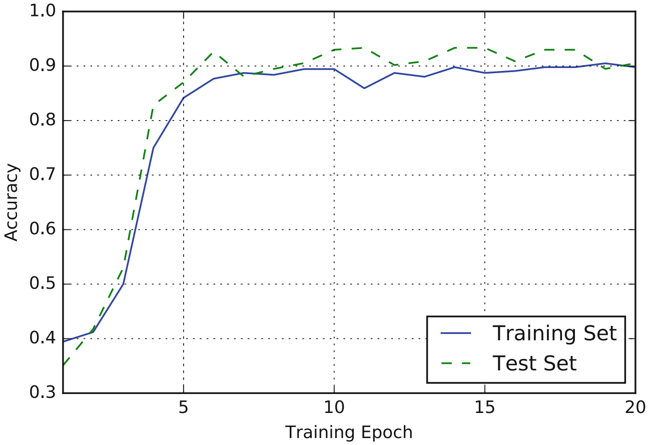


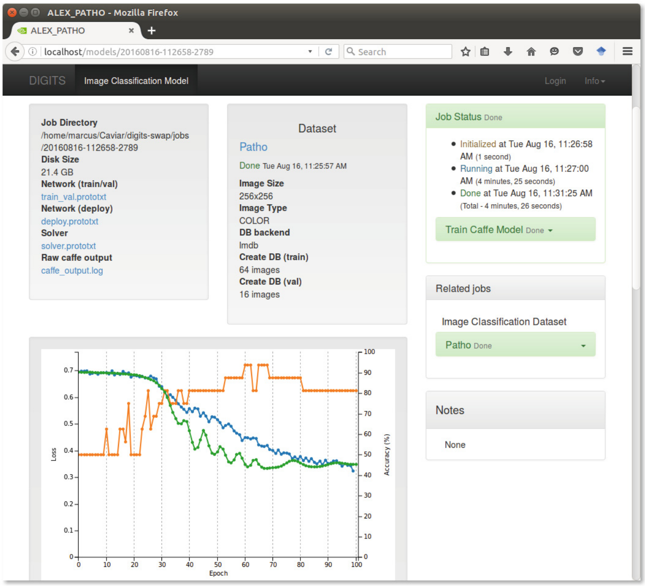
Fig. 18. The accuracy of the network measured against the training set and the test set, over time (measured at each epoch).

11.1 Caffe

Caffe is likely the most used and most comprehensive deep learning platform available. Developed by the Berkeley Vision and Learning Centre, the software provides a modular, schema based approach to defining models, without needing to write much code [13]. Caffe was developed with speed in mind, and has been written in C++ with Python bindings available. Due to its large developer community, Caffe is quickly up to date with new developments in the field. To install Caffe, you must compile it from source, and a detailed description of how to do this is not in this chapter's scope. However, an easier alternative to compiling from source is to use the Caffe version provided by Nvidia's DIGITS software, described in Sect. 11.2.

11.2 DIGITS

Nvidia's DIGITS is a front end for Caffe and Torch, that allows for model training and data set creation via a graphical user interface. Models are defined by the Caffe and Torch model definition schemas respectively. The front end is web-based, a typical example is seen in Fig. 19. The front end provides visual feedback via plots as to the model's accuracy during training. The front end also makes it easier to generate datasets and to organise your models and training runs.



Job Directory
/home/marcus/Caviar/digits-swap/jobs
/20160816-112658-2789
Disk Size
21.4 GB
Network (train/val)
train_val.prototxt
Network (deploy)
deploy.prototxt
Solver
solver.prototxt
Raw caffe output
caffe_output.log

Dataset
Patho
Done Tue Aug 16, 11:25:57 AM
Image Size
256x256
Image Type
COLOR
DB backend
lmdb
Create DB (train)
64 images
Create DB (val)
16 images

Job Status Done

- Initialized at Tue Aug 16, 11:26:58 AM (1 second)
- Running at Tue Aug 16, 11:27:00 AM (4 minutes, 25 seconds)
- Done at Tue Aug 16, 11:31:25 AM (Total - 4 minutes, 26 seconds)

Train Caffe Model Done

Related jobs
Image Classification Dataset
Patho Done

Notes
None

Fig. 19. Nvidia DIGITS in use. Graphs provide visual feedback of the model's accuracy, loss, and other metrics during training.

The advantage to using DIGITS is that is comes with a pre-compiled version of Caffe, saving you the effort of needing to compile Caffe yourself. See https://developer.nvidia.com/digits for information on how to obtain DIGITS.

11.3 Torch

Torch is a popular machine learning library that is contributed to and used by Facebook. It is installed by cloning the latest version from Github and compiling it. See http://torch.ch/docs/getting-started.html for more information.

11.4 TensorFlow

TensorFlow is a deep learning library from Google. For installation details see https://www.tensorflow.org/get_started/os_setup.html. TensorFlow is relatively new compared to other frameworks, but is gaining momentum. Keras can use TensorFlow as a back-end, abstracting away some of the more technical details of TensorFlow and allowing for neural networks to be built in a modular fashion, as we saw in Sect. 10.

11.5 Augmentor

When working with image data, it is often the case that you will not have huge amounts of data for training your algorithms. Deep learning algorithms in particular require large amounts of data, i.e. many samples, in order to be trained effectively. When you have small amounts of data, a technique called *data augmentation* can be applied. Augmentation is the generation of new data through the manipulation of a pre-existing dataset. **Image** augmentation is the generation of new image data through the manipulation of an image dataset.

As an example, say you had a certain number of images, you could quickly double this set of images by flipping each one of them through the horizontal axis, as shown in Fig. 20.

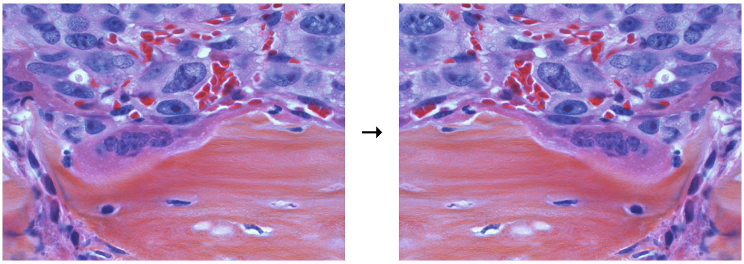


Fig. 20. A histopathology image of cancer cells spreading to bone microenvironment, flipped along its horizontal axis, creating a new image which can also be used for training purposes. Image source: The Web site of the National Cancer Institute (http://www.cancer.gov)/Indiana University Simon Cancer Center. Creators: Khalid Mohammad and Theresa Guise. URL: https://visualsonline.cancer.gov/details.cfm?imageid=10583.

Much work is performed in medicine, in fields such as cell detection or tumour classification, using deep learning. For example in [14] the authors use deep neural networks to detect mitosis in histology images. Augmentation can, in certain cases, aid the analysis of medical image data by artificially generating more training samples.

To aid image augmentation itself, we have created a software tool called *Augmentor*. The Augmentor library is available in Python and Julia versions (in the interests of full disclosure, the first author of this paper is also the author of the Python version of this software package).

You can install Augmentor for Python using pip:

```
$ pip install Augmentor
```

Listing 65. Installing the Augmentor software package using pip. For the Julia version see the package's documentation.

Documentation for the package, including samples and links to the package's source code can be found under http://augmentor.readthedocs.io. For installation instructions on how to install Augmentor for Julia, see the package's documentation at http://augmentorjl.readthedocs.io.

Although larger datasets are important for deep learning, as neural networks consist of many millions of parameters that need to be tuned in order to learn something useful, in the healthcare domain practitioners can be confronted with much smaller datasets or data that consists of very rare events, where traditional approaches suffer due to insufficient training samples. In such cases interactive machine learning (iML) may be of help [15].

12 Conclusion

We hope this tutorial paper makes easier to begin with machine learning in Python, and to begin machine learning using open source software. What we have attempted to show here are the most important data preprocessing tools, the most frequently used Python machine learning frameworks, and have described a broad spectrum of use cases from linear regression to deep learning. For more examples, see the chapter's accompanying Jupyter notebooks, which will be periodically updated.

Acknowledgements. We would like to thank the two reviewers for their suggestions and input which helped improve this tutorial.

References

1. Jordan, M.I., Mitchell, T.M.: Machine learning: trends, perspectives, and prospects. Science **349**(6245), 255–260 (2015)
2. Le Cun, Y., Bengio, Y., Hinton, G.: Deep learning. Nature **521**(7553), 436–444 (2015)
3. Holzinger, A., Dehmer, M., Jurisica, I.: Knowledge discovery and interactive data mining in bioinformatics - state-of-the-art, future challenges and research directions. BMC Bioinform. **15**(S6), I1 (2014)
4. Wolfram, S.: Mathematica: A System for Doing Mathematics by Computer. Addison Wesley Longman Publishing Co., Inc., Boston (1991)
5. Engblom, S., Lukarski, D.: Fast MATLAB compatible sparse assembly on multicore computers. Parallel Comput. **56**, 1–17 (2016)

6. Holmes, G., Donkin, A., Witten, I.H.: Weka: a machine learning workbench. In: Proceedings of the 1994 Second Australian and New Zealand Conference on Intelligent Information Systems, pp. 357–361. IEEE (1994)
7. Read, J., Reutemann, P., Pfahringer, B., Holmes, G.: Meka: a multi-label/multi-target extension to weka. J. Mach. Learn. Res. **17**(21), 1–5 (2016)
8. Guo, P.: Python is Now the Most Popular Introductory Teaching Language at Top U.S. Universities, July 2014. http://cacm.acm.org/blogs/blog-cacm/176450-python-is-now-the-most-popular-introductory-teaching-language-at-top-u-s-universities
9. McKinney, W.: Python for data analysis. O'Reilly (2012)
10. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V.: Scikit-learn: machine learning in python. J. Mach. Learn. Res. (JMLR) **12**(10), 2825–2830 (2011)
11. Hastie, T., Tibshirani, R., Friedman, J.: The Elements of Statistical Learning: Data Mining, Inference and Prediction, 2nd edn. Springer, New York (2009)
12. Bergstra, J., Breuleux, O., Bastien, F., Lamblin, P., Pascanu, R., Desjardins, G., Turian, J., Warde-Farley, D., Bengio, Y.: Theano: A CPU and GPU math compiler in python. In: Procedings of the 9th Python in Science Conference (SCIPY 2010), pp. 1–7 (2010)
13. Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., Darrell, T.: Caffe: convolutional architecture for fast feature embedding. arXiv preprint arXiv:1408.5093 (2014)
14. Cireşan, D.C., Giusti, A., Gambardella, L.M., Schmidhuber, J.: Mitosis detection in breast cancer histology images with deep neural networks. In: Mori, K., Sakuma, I., Sato, Y., Barillot, C., Navab, N. (eds.) MICCAI 2013. LNCS, vol. 8150, pp. 411–418. Springer, Heidelberg (2013). doi:10.1007/978-3-642-40763-5_51
15. Holzinger, A.: Interactive machine learning for health informatics: when do we need the human-in-the-loop? Springer Brain Inform. (BRIN) **3**(2), 119–131 (2016)