2 Scientific programming with Python

This chapter is a brief introduction to scientific programming with the Python programming environment and more specific examples of using ML libraries. Python is a high level programming language similar to Matlab and R that gains increasing popularity in the machine learning community. The main reason to use Python is that it is freely available and that it now provides considerable support for machine learning with packages such as sklearn and tensorflow that will be discussed in this book. We assume some familiarity with programming concepts and concentrate on a quick introduction to the specific environment and supporting libraries used in this book. The programs in this book are based on Python 3, and we assume that all relevant packages are installed. At this point we need the NumPy and the matplotlib libraries as well as the Spyder and Jupyter programming environments. Comprehensive documentation and tutorials for Python are available at https://www.python.org.

2.1 programming environment

We will be using a notebook environment called Jupyter that allows us to run a Python script and can show outputs and text comments. Jupyter can be accessed through the browser and contains form fields in which code and comments can be added. These fields can then be executed and the feedback from print commands or figure plots can be displayed after each block within the same document. This makes it very useful in documenting code and exercises. An example program is shown in Fig. ??

A more traditional programming environment that mimics the Matlab and R environments is called Spyder. Spyder provides a graphical user interface to basic tools such as an editor and an Python interpreter as well as support for viewing variables and debugging. An example view of Spyder is shown in Figure 2.2. On the left is the editor window that contains a syntax sensitive display to write the programs, and on the right is the console to launch line commands such as executing and interpreting the code. As an interpreted language is also possible to work with the programs in an interactive way, such as running a simulation and than plotting results in various ways.

2.2 Basic constructs

As a general purpose programming language, Python contains of course basic programming concepts such as basic data types, loops, conditional statements, and subroutines. We will briefly review the associated syntax below. In addition to such basic programming constructs, all major programming languages such as Python are supported by a
Basic constructs

First Python examples

In [1]:
   from numpy import *
   from matplotlib.pyplot import *

In [2]:
   # basic data types
   aScalar=i
   print(aScalar)

   4

In [3]:
   aList = [1,2,3]
   print(aList)

   [1, 2, 3]

Fig. 2.1 An example of a Python program within the Jupyter notebook. The example code is discussed further below.

Fig. 2.2 The Spyder programming environment for Python.

large number of libraries that enable a wide area of programming styles and specialized functions. We are here mainly interested in basic scientific computing in contrast to system programming, and we need multidimensional arrays. We therefore base practically all programs in this book on the NumPy library. The NumPy library provides basic support of common scientific constructs and functions, and most of all provide support for N-dimensional arrays. We will use this well established constructs to implement vectors, matrices and higher dimensional arrays. Note that there is a separate matrix class which is limited to a two dimensional structure.

An established way to import these libraries in our programs is to map them to the name space “np” with the command import numpy as np. In this way, the specific methods or functions of NumPy are accessed with the prefix np. In addition to
importing NumPy, we always import a plotting library as plotting results will be very useful and the common way to communicate results. We specifically use the popular PyPlot package of the Matplotlib library. Hence, we nearly always start our program with the two lines

```python
import numpy as np
import matplotlib.pyplot as plt
```

In the following we walk through some program in the Jupyter environment called FirstProgram. These lines of code are intended to show the syntax of the basic programming constructs that we need in this book. We start by demonstrating the basic data data types that we will be using frequently. We are mainly concerned with numerical data of which a scalar is the simplest example,

```python
# basic data types
aScalar = 4
print(aScalar)
```

We here showed the code as well as the response of running the program. Note that we can include comment lines with the hash-tag symbol. Also, we included a print function that will report the value of the variable `scalar` defined above. The type of the variables are dynamically assigned in Python. That is, a variable name and corresponding memory space is allocated the first time a variable with this name is used on the left hand side of an assignment operator `=`.

Most of the time we need to work on a large collection of data so that we need a concept to access the data collection. In Python there are several forms of lists. For example, a basic one dimensional list is given in the basic Python stack by enclosing a semicolon-separated list in square brackets such as

```python
aList = [1, 2, 3]
print(aList)
```

Such lists are already and often used. However, since we need to perform well defined mathematical operations on lists of data it is useful to introduce a more versatile constructs of such data collections in forms of NumPy arrays.

Before proceeding it might be good to review some of the naming conventions. A basic data structure for a collection of data is usually called an array in computer science. In contrast to these simple data structure concepts, the mathematical concepts of a vector or matrix are different in that they include well defined mathematical operations on these data structures. Thus, the mathematical concept of a vector is a one-dimensional array on which some operations are defined such as adding two vectors with the same dimension by adding their components, or multiplying a vector with a scalar by multiplying each component of the vector with a scalar. Similar, a matrix is a two-dimensional construct with correspondingly defined operation. We can even generalize this to higher dimensions, and such mathematical constructs are called
a tensor. Indeed, it is very convenient to view a vector or matrix operation just as a special case of the general tensor operations.

To create a NumPy array we use the NumPy function `array()`. For example, a one-dimensional Python list can be turned into a NumPy vector like,

```
Vector = array([1, 2, 3])
print(Vector)
print(Vector[1], Vector[-1])
```

```
[1 2 3]
2 3
```

As shown in the last line, we can access an element of the array with indices in square brackets. The first element in an array has the index 0. Hence, the print command returns the second element in the vector. It is useful to think about this index as the offset from the first element. The index -1 accesses the last element in the vector. Unfortunately there is no distinction between a row vector and a column vector in NumPy, so that this need some more careful considerations when a distinction is necessary. We return to this point in a moment.

Similar to defining a vector with NumPy, a two dimensional array with the appropriate definition of mathematical operations is called a matrix and can be defined and accessed with NumPy like

```
print(aMatrix)
print(aMatrix[1][2])
```

```
[[1 2 3]
 [4 5 6]]
6
```

The notation indicates that a two-dimensional array is considered in the python syntax as a one-dimensional list of a one-dimensional list. Note how individual array elements are accessed; the first index specifies the position in the column, and the second index specifies the position in the row. This is equivalent to the common mathematical notation for matrices. With this we can revise the notation for the vectors above by defining a row vector as

```
Vector = np.array([[1, 2, 3]])
```

This can then be converted into a column vector with the help of the transpose operation

```
print(Vector.T)
```

```
[[1]
 [2]
 [3]]
```

After defining such NumPy arrays we can apply mathematical function on these NumPy array. For example, some element-wise operations on matrices are

```
matrix2 = np.array([[5, 5, 6], [7, 8, 9]])
```
result1 = aMatrix * matrix2  # element-wise
result2 = aMatrix ** 3  # element-wise exponentiation:
result3 = aMatrix > 3  # find the indices where (matrix > 3)
print(result1, result2, result3)

[[ 5 10 18]
[ 28 40 54]] [[ 1 8 27]
[ 64 125 216]] [[False False False]
[ True  True  True]]

A basic matrix multiplication, also called a dot product or inner product, is implemented as function np.dot(a, b) and in Python 3 also as operator @.

result = aMatrix @ matrix2.T
print(result)

[[ 33  50]
[ 81 122]]

We have thereby included the transpose operation through the operator specification "texttt.T". Such operator specification are common in object-oriented programming constructs.

We are commonly in need of accessing subsets of data in arrays and also to merge arrays. To access a subset of an array we can first generate an index vector called idx below which specifies the indices we want to process such as the first and second element in the second row of the matrix called aMatrix defined earlier

idx = [[1], [0, 2]]
print(aMatrix[idx])

[4  6]

Another useful example is to make a vector with a list,

x = np.arange(10)

which is the same as array(range(10)), and to extract every second element of a vector,

print(x[::2])

[0  2  4  6  8]

The array indexing is the same as x[0:-1:2] because the default boundaries for the first and second limits is the first and last element. Merging two arrays is done with the NumPy concatenate() method.

print(np.concatenate((aMatrix, matrix2).axis=1))

[[1 2 3 5 5 6]
[4 5 6 7 8 9]]
So far we have discussed the basic numerical data types that we need. Besides these numerical data types, there are of course others such as characters. Text data a simply enclosed in parenthesis like.

```python
text = 'Hello World!
print(text)
Hello World!
```

In the following we show three fundamental programming constructs, that of loops, conditional statements, and functions. To loop through some code one can use the following construct,

```python
for i in range(4):
    print(i)
```

0
1
2
3

which starts at i=0 and goes in steps of one until i=3. Note that Python is sensitive to the code position; the indented code represents the block of statements executed inside the loop. A conditional statement takes the form

```python
if scalar < 1:
    print("true")
else:
    print("false")
```

```
false
```

Again note the indentation to specify the block of code for each condition.

To structure code better, specifically to define some code that can be reused, we have the option to define functions like

```python
def func(arg1, arg2=10):
    arg = arg1 + arg2
    return arg;
```

```python
a=1
print(func(2), func(a, 2))
```

```
12 3
```

Simple variables are passed by value in Python, but more complex objects might be referred by reference. It is hence good to be careful when changing the content of calling variables in the functions.

Plotting graphs for data is a useful scientific tool, and we will be using the the popular scientific plotting library matplotlib (http://matplotlib.org/), specifically the pyplot package that provides a slightly simpler interface within the matplotlib
package. We imported this library already at the beginning of the code. Using this library, an example of a basic line plot is given in the following code.

```python
# plotting
x = arange(100)  # same as array(range(100))
y = sin(0.1*x)
plot(x, y)
```

When you submit plots in an assignment or paper, you always need axis labels to know what is plotted. This can be done with

```python
xlabel("x")
ylabel("y")
```

Finally, some of the programs might need some time to run, and it might be necessary to estimate the time of running with some smaller examples and measuring the time. This can be done in the following way.

```python
import time
tic = time.clock()
toc = time.clock()
print(toc - tic)
```

2.2214871933101676e−05

### 2.3 Cross validation example from Intro

To practice Python programming and to deepen our understanding of cross validation, we will now review the program that was used to produce the linear model with cross validation of the example.

In the code below we start by generating the data set consisting of 4 data points that are derived from the equation of a line $y = 2x + 3$ with added Gaussian noise,

```python
import numpy as np
import matplotlib.pyplot as plt

n=4
x=np.array(range(4)); y=2*x+3+np.random.randn(n)
```

For the learning tasks we chose a linear model $\hat{y} = ax + b$ with two parameters, the slope $a$ the the intercept $b$. Of course, this is a wise choice in light of our knowledge how the data were generated, though of course we don’t know this generally. Anyhow, given this parameterized model, our task is now to determine the values for these parameters from the data. Since we have only two unknown we only need two data point to determine, so let us choose the first two data point as training set. From these two data points we can calculate the parameters analytically.

```python
a=(y[1]−y[0])/(x[1]−x[0])
b=y[0]−a*x[0]
what=a*x+b
```
We can also calculate the true underlying noiseless world as

\[ y_{true} = 2x + 3 \]

to plot this later.

Of course, the calculated solution using the first two data points as training set learned solution is only one possible solution since we could have used any other pair to determine the parameters. We could try out all other combinations of two data points for the training set and could use all the remaining data set to validate how good the generalization to these points is. This is the essence of cross validation. In order to calculate all the possible combination of two pairs from the data set we use a predefined function from the `itertools` collection called `combinations()`.

```python
# cross validation
import itertools
c = list(itertools.combinations(x, 2))
```

The list `c` contains all possible pairs. We then loop over all the pairs and determine the parameters for each choice, and then calculate the error for predicting the other data points not used in the determination of the parameters.

```python
import itertools
c = list(itertools.combinations(x, 2))

# try out all possible pairs
error = []
for i in range(len(c)):
    k = c[i][0]
    l = c[i][1]
    a = (y[l] - y[k]) / (x[l] - x[k])
    b = y[k] - a * x[k]
    er = 0
    for j in range(n):
        if j != k and j != l:
            er = er + (y[j] - a * x[j] - b) ** 2
    error.extend([er])
```

This ends the loop. We then take the pair with the minimal cross validation error as our final answer.

```python
# search for best pair with smallest cross validation error
i = error.index(min(error))
k = c[i][0]
l = c[i][1]

# and use this as answer
a = (y[l] - y[k]) / (x[l] - x[k])
b = y[k] - a * x[k]
yhat_best = a * x + b
```
At the end we plot all the solutions.

```python
plt.plot(x, y, '*')
plt.plot(x, yhat, 'b--')
plt.plot(x, ytrue, 'g-')
plt.show()
```

### 2.4 Code efficiency and vectorization

Machine learning is about working with large collection of data. Such data are kept in data bases, spreadsheets, or simply in text files, but to work with them we load them into arrays. Since we define operations on such arrays, it is better to treat these arrays as vectors, matrices, or generally as tensors. Traditional programming languages such as C and Fortran require then to write code that loops over all the indices in order to specify operations that are defined on all the data. For example, let us define two random $n \times n$ matrices with the NumPy random number generator for uniformly distributed numbers.

```python
a = np.random.rand(n, n)
b = np.random.rand(n, n)
```

and a matrix of zeros with the same size,

```python
c = np.zeros((n, n))
```

We can then write the code of adding two numbers with an explicit loop over all indices as

```python
for i in range(n):
    for j in range(n):
        c[i][j] = a[i][j] + b[i][j]
```

In high level programming languages like Python, Matlab and R, it is common to write such operations in a compact form like

```python
c = a + b
```

It is now common to call this style of programming a **vectorized code**. Such a vectorized code is not only much easier to read, but it is also essential to write efficient code. The reason for this is that the system programmers can implement such routines very efficiently, which is difficult to match with the more general but inefficient explicit index operation.

To demonstrate the efficiency issue, let us measure the time of operations for a matrix multiplication. We start as usual by importing the standard NumPy and matplotlib libraries, and we also import a timer routine with

```python
import time
```

We then define a method called `matmul` that implements a matrix multiplication with an explicit iteration over the indices,
def matslow(a, b):
    m = a.shape[1]
    c = np.zeros((m, m))
    for i in range(m):
        for j in range(m):
            for k in range(m):
                c[i, j] = c[i, j] + a[i, k] * b[k, j]
    return c;

and a fast version of this operation in the method matfast which call the NumPy method dot,

def matfast(a, b):
    return np.dot(a, b);

We then evaluate the time these routines take with the following test code,

size = np.array([])
time1 = np.array([])
time2 = np.array([])
for n in range(10, 130, 10):
    size = np.append(size, n)
a = np.random.rand(n, n)
b = np.random.rand(n, n)
c = np.zeros((n, n))

timestart = time.clock()
c = matslow(a, b)
time1 = np.append(time1, time.clock() - timestart)

timestart = time.clock()
c = matfast(a, b)
time2 = np.append(time2, time.clock() - timestart)

The resulting time graph is shown in Fig. ?? This not only shows that the time difference can be substantial for larger arrays, but that the scaling is very different. Some concern that interpreted computer languages are slow come form the inefficient implementations. It is often the most challenging part for experienced programmers of C-like languages to adopt to this vectorized code, but such a programming style is essential to produce efficient code.

2.5 Data handling

Since machine learning requires data, we are commonly faces with importing data from files. There are a variety of tools to handle specific file formats, though we are mostly reading data from text files. In the next chapter we will discuss some common machine learning examples, including a well known classification problem of Iris flowers. The
Scientific programming with Python

Iris dataset was collected from a field on the same day at the Gaspé and were first used by the famous British statistician Ronald Fisher in a 1936 paper. The data consist of 150 samples, 50 samples of each of three species of the Iris flower called Iris Setosa (0), Iris Versicolour (1), and Iris Virginica (2). For our purpose we usually simply give each class a label such as a number as shown in the bracket after the flower names in this example.

The dataset is given here with three text files, named iris.data, feature_names.txt, and target_names.txt. The data file contains both the feature values and the class label, and we can load these data into a NumPy array with the NumPy functions loadtxt. Printing out the shape of the array reveals that there are 150 lines of data, one for each sample, and 5 columns. The first four values are the measured length and width of septals and pedals of the flowers. The last number is the class label. The following code separates this data array into feature matrix and a target vector for all the samples. We also show how text can be handles with the NumPy function genfromtxt

```python
import numpy as np
import matplotlib.pyplot as plt

iris_data = np.loadtxt('iris.data', delimiter=' , ')
print(iris_data.shape)
features = iris_data[:,0:4]
target = iris_data[:,4]

feature_names = np.genfromtxt('feature_names.txt', delimiter=' , ', dtype='str')
feature_names = np.delete(feature_names, -1)
target_names = np.genfromtxt('target_names.txt', delimiter=' , ', dtype='str')
```

With the data in the form of NumPy arrays, it is then easy to apply functions on these arrays to calculate properties of interest. For example, we can calculate the sum of all the septal width and the pedal width, the second and third column respectively, with the command

```python
print(features[:,[1,3]].sum(axis=0))
```

It is also useful to make plots, such as plotting the average of the feature values across the samples in a bar graph where we also indicated the standard deviation with error bars.

```python
plt.bar(np.arange(1,5), features.mean(axis=0))
plt.errorbar(np.arange(1,5), features.mean(axis=0),
    features.std(axis=0), linestyle='None', marker='o', c='r')
plt.show()
```

Also, while such summary statistics is a common way of characterizing data, in the age of advanced computer graphics it is often useful to try and plot the data. For example, a scatter plot of the data points when we would characterize the flowers only
by the pedal length and width can be generated according to the following code. Both 
of these plots are shown in Fig.2.3

```python
plt.scatter(features[:50,0], features[:50,1], s=10, c='r', label=target_names[0])
plt.scatter(features[50:100,0], features[50:100,1], s=10, c='g', label=target_names[1])
plt.scatter(features[100:,0], features[100:,1], s=10, c='b', label=target_names[2])
plt.legend(loc='upper right');
plt.xlabel(feature_names[0])
plt.ylabel(feature_names[1])
plt.show()
```

**Fig. 2.3** Summary statistics and scatter plot of Fisher’s iris data.