2 Practical ML 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. The basic idea behind this chapter is to jump right away into some examples. So we will intentionally only cover some essential basics to keep us going. We will continue to refine programming issues throughout the course and will talk about the science behind the algorithms later.

2.1 General scientific programming in Python

2.1.1 Resources and installation

Python is a high level programming language that gains increasing popularity in the machine learning community (Matlab has been dominating before). We assume some familiarity with programming concepts and concentrate on the specific environment and supporting libraries for this class. A comprehensive documentation and tutorials are available at https://www.python.org. Some good resources for scientific computing with Python are:

- numpy-useful for itsN-dimensional array objects
  `http://www.numpy.org/`
- matplotlib- 2D plotting library producing publication quality figures
  `http://matplotlib.org/`
- scikit-learn - collection of machine learning functions and tools
  `http://scikit-learn.org/stable/`

The heart of numpy is support for specific data types, in particular for N-dimensional arrays on which most of our code will be based. Together with scipy, which contains useful scientific routines, and plotting packages such as Matplotlib, this defines a useful scientific high-level numerical programming environment similar to Matlab and R. The main reason to use Python is that it is freely available and that it also provides a good base language for packages such as tensorflow which we will be using for deep learning. There are a variety of good documentations on the associated web pages. We will be using the Ubuntu operating system with Python 3 and supporting programs. An image with these components is provided so that you can install this on your own computer or use computers in our faculty. Please see the helps desk for any problems with the installation.

2.1.2 The Spyder programming environment

We will be using a programming environment called Spyder that provides a graphical user interface to basic tools such as an editor and an python interpreter. Start Spyder and you should see the programing environment similar to the one shown in Figure 2.1. On the left is a editor window in which we can write the program. On the right is the console that executes and interpreted the code.

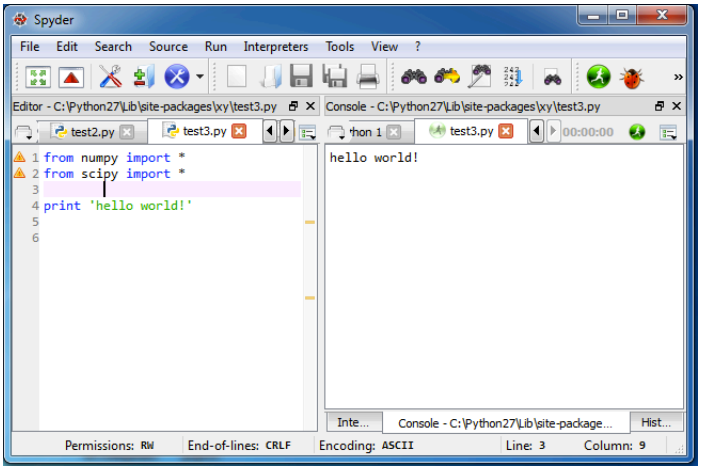

Fig. 2.1 The Spyder programming environment for Python.

2.1.3 Main programming constructs

The following lines of course are intended to show the syntax of the basic programming constructs that we need in this course. We will be using Python as a scientific programming language, and we will always import the pylab library that includes a lot of useful functions.

```
from pylab import *
```

Next we consider the basic data types that we are using. We are mainly concerned here with numerical data of which a scalar is the simplest example,

```
#basic data types
scalar=4
print(scalar)
```

Note that we can include comment lines with the hashtag symbol. We also included a print function that will report the value of the variable `scalar` that we defined here. Note that the type of the variables are dynamically assigned in python.

Most of the time we need to work on a large collection of data so that we need a contract to access the data collection. For we use some form of lists. There are slightly different concepts of such constructs in python. The basic one dimensional list if given by enclosing a semicolon-separated list in square brackets such as

```
list = [1,2,3]
```

However, basically need to perform well defined mathematical operations with these list, which makes these one dimensional list formally a vector. The basic data structure for a collection of data is usually called an array in computer science. Thus, the mathematical concept of a **vector** is a one-dimensional array with some operations defined to it. To confuse this matter a bit more, we will use the numpy constrict of an array to implement a vector. bumpy is a collection of numerical functions that is included in pylab. The bumpy function `array()` turns a Python list into a vector,

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

As shown in the last line, we can access an element of the array with indices in square brackets, and the first element in an array has the index 0. The index -1 accesses the last element in the vector.

Of course, we can generalize such data collections to higher dimension arrangements. For example, a two dimensional array with the appropriate definition of mathematical operations is called a **matrix** and can be defined and accessed in python like

```
matrix=array([[1,2,3],[4,5,6]])
print(matrix)
print(matrix[1][2])
```

Corresponding mathematical constructs in higher dimensions are called a **tensor** that we will talk about later. Some element-wise operations on matrices are

```
matrix2=array([[5,5,6],[7,8,9]])
result2=matrix * matrix2  #element-wise
result3 = matrix **3  #element-wise exponentiation:
result4 = matrix >3  #find the indices where (matrix > 3)
```

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

```
result5=matrix @ matrix2.T
print(result2,result3,result4,result5)
```

So far we have discussed the basic data types that we need. Besides these numerical data ties there are of course others such as logical or characters. Please consult Python documentation for these data types when needed. We now mention three more basic programming constructs, that of loops, logical statements, and functions.

To loop through some code one can use the following construct,

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

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

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

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

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

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

Variables are passed by reference.

One final example of basic programming we need is that of plotting graphs. Plotting graphs is a useful scientific tool, and an example of a basic line plot can is given in the following code.

```
#plotting
x=arange(100)  #same as array(range(10))
y=sin(0.1*x)
plot(x,y)
```

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

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

2.2 Further useful functions

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

ones, zeros, size, ndim, shape, nonzero, reshape, shape, max, min, mean, std, sum, sqrt, exp, floor, ceil, single, int, rand, randn, seed, savefig, savetxt, csv, ...

show multiple plots, barplots, scatter plot,...)

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 training set consisting of 4 data points that are derived from a line $y = 2x + 3$ with added Gaussian noise,

```
from pylab import *
# training set
```

```
n=4
x=array(range(4)); y=2*x+3+randn(n)
plot(x,y,'*')
```

For the learning tasks we chose a linear model $\hat{y} = ax + b$, see it as a wise choice, with two parameters, the slope a the the intercept b. 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,

```
# one example
a=(y[1]-y[0])/(x[1]-x[0])
b=y[0]-a*x[0]
yhat=a*x+b
plot(x,yhat,'b--')
ytrue=2*x+3
plot(x,ytrue,'g-')
```

We plotted here this specific solution in black and well as the best possible solution in green which we know as we know what the parameters were that have been used to generate the data and also since the Gaussian noise is unbiased (symmetric around zero)

Of course, this solution is only one possible solution since we could have used any other pair to determine the parameters. Indeed, we should try out all and use all the remaining points to see how good one specific solution will predict the reminder. This is exactly the essence of cross validation.

To determine all the possible combination we use a preferred function from the `itertools` collection,

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

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

```
#try out all possible pairs
error=[]
for i in range(len(c)):
  #train fold
  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,

```
#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=a*x+b
plot(x,yhat,'r--')
```

2.4 Classification with support vector machine using scikit-learn

We will now show an explicit example of classification using a support vector machine from the scikit-learn collection of machine learning methods at http://scikit-learn.org/. This library started as a Google Summer of Code project by David Cournapeau and developed into an open source library. We will later have a look of what kind of algorithms are implemented, but for now we are just using one of the methods for classification called support vector machine. The SVM in scikit-learn is actually a wrapper to the very popular SVMLIB implementation by Chih-Chung Chang and Chih-Jen Lin. We will go through the code here with some explanations.

We begin as usual by importing libraries we need and to create the training set.

```
from pylab import *
from sklearn import svm

# training
n=100
x1=array([randn(n)+1,randn(n)+1]); y1=zeros(n)
x2=array([randn(n)+3,randn(n)+3]); y2=zeros(n)+1
x = hstack((x1,x2)).T
y = hstack((y1,y2))
```

In real world application the data set is of course usually supplied by a third party often through a data file. Here we simulate an example that consists of two 2-dimensional Gaussian classes each with a unit covariance matrix and different means. The mean of the first class is $\mu_1 = (1,1)$ and the second class has $\mu_2 = (3,3)$. The distributions of these two classes are shown in Fig. 2.2.

We now define a classifier model. We are using a Support Vector Classifier, as specific support vector machine for classification, with two parameters that we will discuss only later, namely we are using a linear kernel and regularization parameter $C = 1$,

```
SVC = svm.SVC(kernel='linear', C=1)
SVC.fit(x, y)
```

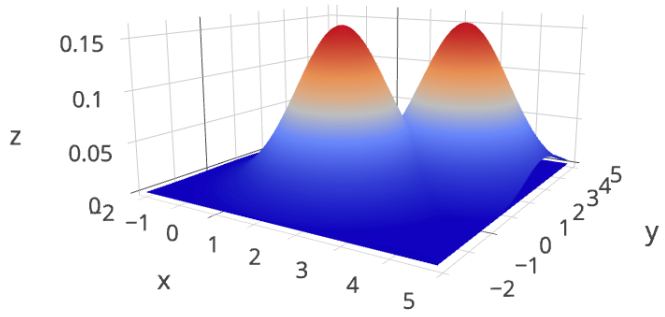

Fig. 2.2 Two 2d-Gaussian curves with unit covariance and different means.

The second line implements the learning, that is it tales the examples in arrays `x` and `y` and fit the model to it. At this point we have a trained model `SVC` that we can use to predict data. We will test its performance with some new sample data,

```
# testing
x1=array([randn(n)+1,randn(n)+1]); y1=zeros(n)
x2=array([randn(n)+3,randn(n)+3]); y2=zeros(n)+1
x = hstack((x1,x2)).T
y = hstack((y1,y2))
```

that we also plot with different symbols and color. We use the model for predicting the labels for the class with the command

```
a=SVC.predict(x)
```

and calculate the percentage of correct labels with

```
print("Percentage Correct:", (n-sum(abs(y-a)))/n)
```

Finally, we also like to plot the results

```
plot(x1[0,:],x1[1,:],'xr')
plot(x2[0,:],x2[1,:],'ob')
show()
```

2.5 Other classification methods including MLP with Tensorflow

The final example here is using two more classifiers in addition to the SVM on the same two-Gaussian example, namely a random forrest (RF) classifier and a multilayer perceptron MLP). The RF is also implemented in sklearn and is hence verst similar. We are only changing the name of the model. For the MLP we use Googles Tensorflow implementation which is also quite similar to the sklearn notation. The only difference is that the model has of course different parameters and hyper-parameters.

```
from pylab import *
from sklearn import svm
from sklearn.ensemble import RandomForestClassifier
import tensorflow as tf
from tensorflow.contrib import learn
tf.logging.set_verbosity(tf.logging.ERROR)

# data (generation of training data)
n =100
x1= array([randn(n)+1,randn(n)+1]);y1=zeros(n,int)
x2= array([randn(n)+3,randn(n)+3]);y2=zeros(n,int)+1
x = hstack((x1,x2)).T
y = hstack((y1,y2))
plot(x1[0,:] , x1[1,:] , 'xr' )
plot(x2[0,:] , x2[1,:] , 'ob' )
show()

# making model and training (fitting) them
SVC = svm.SVC(kernel ='linear', C=1)
SVC.fit(x,y)

RF = RandomForestClassifier(n_estimators=10)
RF.fit(x,y)

feature_columns = [tf.contrib.layers.real_valued_column("", dimension=4)]
MLP = learn.DNNClassifier(
        feature_columns=feature_columns,
        hidden_units = [10, 20, 10],
        n_classes = 2)
MLP.fit(x,y,steps = 500, batch_size = 128)

# generating testing data
x1= array([randn(n)+1,randn(n)+1]);y1= zeros(n)
x2= array([randn(n)+3,randn(n)+3]);y2= zeros(n)+1
x = hstack((x1 ,x2)).T
y = hstack((y1 ,y2))

# prediction
a=SVC.predict(x)
b=RF.predict(x)
c=list(MLP.predict(x, as_iterable=True))

#evaluation
print('PercentageCorrect␣SVM:␣', (n-sum(abs(y-a)))/n )
print('PercentageCorrect␣RF␣:␣', (n-sum(abs(y-b)))/n )
print('PercentageCorrect␣MLP␣:␣', (n-sum(abs(y-c)))/n )
```

The result of running the program is shown in Fig. 2.3. All three classifiers give the same result in this run which is close to the optimal result in this example. When running this program repeatedly there will be slight differences in the answers. We will later discuss the stochastic nature of machine learning.

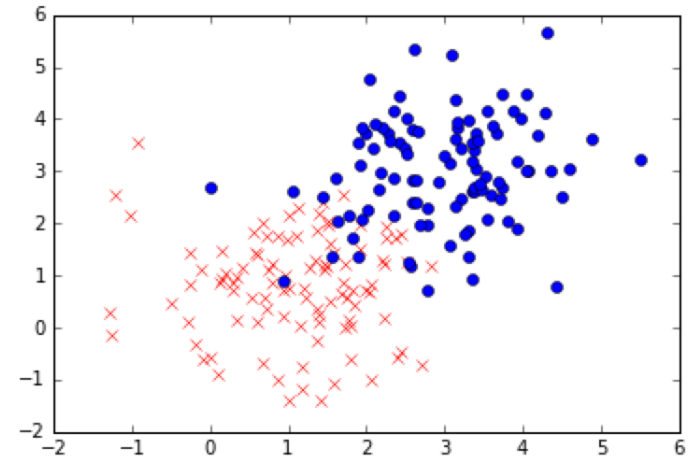

Fig. 2.3 The plot shows the data points in the two-Gaussian example that consists of two Gaussian classes with the same unit co-variance but different mean values. The problem is that these classes overlap. Below the figure is the percentage correct of three machine learning classifiers, that of a Support Vector Machine (SVM), a Random Forrest (RF) classifier), and a multilayer perceptron (MLP).

2.6 Applying ML methods to specific problems

As we have seen in the previous section, writing a program that applies ML algorithms to data is usually not too difficult. It is common that new algorithms will find its way to graphical data mining tools, which makes them available to an even larger application community. However, applying such algorithms correctly in different application domains can be challenging and it is well known that some experience is required. We therefore concentrate in the following in explaining what is behind these algorithms and how different theoretical concepts are explored by the different algorithms. Some understanding of the algorithms is absolutely necessary to avoid pitfalls in their application.

Each application is different, and David Wolpert coined the so called "No free lunch" theorem which basically states that there is not a single algorithms that covers all applications better than some other algorithms. This is further complicated by the fact that many machine learning algorithms suffered from the need of carefully

choosing parameters of the algorithms, which again requires experience as well as experimentation. The first substantial breakthrough in the application of machine learning methods outside a small research community came with the SVM classifiers as they were somewhat easier to apply than earlier neural networks or Bayesian methods. SVMs are often a good starting place when exploring new applications. However, we see now their limitations and know that deep networks have been able to advance applications that are difficult for basic SVMs. We will later explore the reason for this.

The basic first step for the application of ML methods is how to represent the data. We discussed already in the first chapter how to convert different type of inputs to numerical vectors or tensors. However, there are usually many different possibilities to represent the data, such as a fine grain representation or using some summary statistics. In the past it has been crucial to work out an appropriate high level data representation. However, the recent progress in deep learning has enabled to treat this representation itself as part of the learning problem. Representational learning has thus become an important part of machine learning.

Once the problem has been defined by representing the data and possible goals in an appropriate way, and once the appropriate ML algorithm has been chosen, it is then the main challenge to chose good parameters of the algorithms such as the number of neurons or layers of neurons in neural networks, which kernel to use in support vector machines, how many training steps to take in gradient descent learning, or how many data to use for learning versus validation. We call these parameters of the algorithms the **hyperparameters**. Choosing the right hyperparameters is commonly a major question and to make it clear from the start, there is no simple answer. Thinking about how to approach this question with appropriate experiments and to understand the options and possible approaches is hence a mayor part of machine learning applications.