CSCI 4155/6505 (2016): Machine Learning

Thomas P. Trappenberg
Dalhousie University

Acknowledgements

These lecture notes have been inspired by several great sources, which I commend as further readings. In particular, Andrew Ng from Stanford University has several lecture notes on Machine Learning (CS229) and Artificial Intelligence: Principles and Techniques (CS221). His lecture notes and video links to his lectures are available on his web site (`http://robotics.stanford.edu/∼ang`). Excellent book on the theory of machine learning are *Introduction to Machine Learning* by Ethem Alpaydin, 2nd edition, MIT Press 2010, and *Pattern Recognition and Machine Learning* by Christopher Bishop, Springer 2006. The standard book on RL is *Reinforcement Learning: An Introduction* by Richard Sutton and Andrew Barto, MIT press, 1998. The standard book for AI, *Artificial Intelligence: A Modern Approach* by Stuart Russell and Peter Norvig, 2nd edition, Prentice Hall, 2003, does also include some chapters on Machine Learning. Finally, the book by Kevin Murphy, Machine Learning: A Probabilistic Perspective is highly recommended for more in-depth studies.

Several people have contributed considerably to this lecture notes. In particular I would thank my PhD students Paul Hollensen and Patrick Connor.

Contents

1 Introduction

1.1 The basic idea behind supervised ML

In the sense of its words, Machine Learning (ML) is the area that tries to build intelligent machines by defining a machine with specific operational abilities and training it with examples to perform specific tasks. This might sound like a niche area in science and you might wonder why there is now so much interest in this discipline, both academically and in industry. The reason is that ML is really about modeling data that provide the basis of advanced object recognition and data mining and is hence the analytical engine in areas such as data science, big data, data analytics and science in general.

ML has a long history with traces far back in time. One of the first recognized exciting realizations of the promise of learning machines came in the late 1950s and early 1960s with work like Arthur Samuel's self-learning checkers program and Frank Rosenblatt's perceptron. A second wave came in the 1980s and 90s with multilayer perceptrons and recurrent networks. And since 2006 we have a third wave of neural networks, that of deep learning which we will discuss more in this course.

ML is not restricted to neural networks. Indeed, the development of statistical machine learning and Bayesian networks has influenced the field strongly in the last 20 years and has been essential in much of its progress as well as in the deeper understanding of machine learning. This course will hence also introduce these more general ideas.

It is common and somewhat useful to distinguish three areas of machine learning, that of supervised learning, unsupervised learning, and reinforcement learning. Much of what is currently most associated with the success of ML is supervised learning, sometimes also called predictive learning. The basic task of supervised learning is that of taking as input a vector $\mathbf{x}$ of measurements, such as some medical measurements or robotic sensor data, and predicting an output value $\mathbf{y}$ such as the state of a patient's health or the location of obstacles. Note that we follow here a common notation of denoting a vector, matrix or tensor with bold faced letters, whereas we use regular fonds for scalars. We usually call the input vector a feature vector as the components of this are typically a set feature values of an object. The output could be also a multi-dimensional object such as a vector or tensor itself.

Mathematically we can denote the relations between the input and the output as a function

$$y = f(\mathbf{x}). \tag{1.1}$$

We consider the function above as a description of the **true underlying world**, and our main goal is to find out this precise relation. In the above formula we considered a single output value and several input values for illustration purposes, although we see later that we can extend this readily to multiple output values. This would then correspond to a vector function.

The challenge for machine learning to find this function or at least to approximate it sufficiently. Machine learning has several approaches to deal with this. One approach that we will predominantly follow for much of the course is to define a general parameterized function

$$\hat{y} = \hat{f}(\mathbf{x}; \theta). \tag{1.2}$$

This formula describes that we make a parameterized hypothesis in which we specified a function $\hat{f}$ that depend on parameters to approximate the desired input-output relation. This function is called a **model**. The model is generally an approximation of a system to study specific aspects of its behaviour. This often means that not all of the underlying world has to be captured in depth. For example, a building engineer might make a model of a bridge to tests its static without including the ascetic aspects that an architect might emphasize in a model. In our context the word model is synonymous with approximation.

We have indicated that this model is an approximation of the desired relation by using a hat symbol. However, we frequently drop this symbol when the relation is clear from the context, for example when the function contains parameters. The parameters are specified in this function by including the parameter as vector θ behind a semicolon in the function arguments. More appropriately, the formula defines a set of functions in the parameter space. A good solution represented by a point in this parameter space is an approximation that can be used for predicting output values (labels) $\hat{y}$ for specific input values.

We will later go one step further by considering the more general case when we might not be able to predict an exact value but at least the probability that a certain value will occur. Indeed, it is quite common that the process under investigation includes stochastic (random) or unknown factors. True underlying world model is thus better described by a probability density function

$$P(Y = y|\mathbf{x}). \tag{1.3}$$

Formulating ML learning in a probabilistic context has been most useful and provides us with the formalization that created the most insight into this field. In the stochastic framework we are then modelling a density function

$$p(\hat{Y} = \hat{y}|\mathbf{x}; \theta). \tag{1.4}$$

For now we will follow the function approximation formalization, but we return to the probabilistic framework later.

1.2 Training, validating and testing

Coming up with the right parameterized approximation function is the hard problem in machine learning, and we will later discuss several choices. There are also methods to systematically develop the approximation function from the data, generally called non-parametric methods. However, we assume for now that we have a parameterized approximation function. The question is then how we determine good parameters. This is where the learning process comes in.

In supervised learning we must be given some examples of input-output relation from which we learn. We can think about these examples as given by a teacher. The teacher data called the **training set** are used to directly determine the parameters of the model. We can denote this training set as

$$\{\mathbf{x}(i), y(i)\}, \tag{1.5}$$

where the superscript i labels the specific training example. These indices are enclose in brackets to not confuse them with exponents.

There are different ways – usually called a training algorithm – to determine the parameters of a model. Let us illustrate this with an example where we assume we have a single input feature, x, and we hypotheses that the output y is linearly related to x. Mathematically we write this linear model as

$$\hat{y} = ax + b, \tag{1.6}$$

where a the slope of the linear function and b is the y-axis intercept or bias of this function. Using the training data to determine good parameters is also called regression in this context.

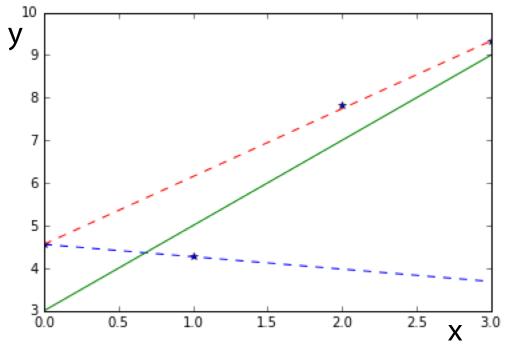


Fig. 1.1 A form of linear regression of data and cross-validation.

Let us illustrate an example regression procedure (there are different possible regression procedures) with the help of an example. For this we choose the true underlying model

$$y = 2x + 3 + \eta, \tag{1.7}$$

where η is a normal distributed random variable. We added this random addition to the perfect linear model to include right away a typical challenge in ML, that of having imprecise measurements and hence noisy training data. The other way to interpret the model is actually to accept the 'world' as stochastic and hence we are considering a stochastic model. In any case, from this model we chose some data points by sampling,

$$(0, 4.56)(1, 4.27), (2, 7.81), (3, 9.33) \tag{1.8}$$

We now make the assumption of a parameterized linear function

$$\hat{y} = ax + b, \tag{1.9}$$

with two parameters a and b, and use a simple method to determine the two parameters. As we have a linear equation with only two unknowns, we only need two data points to determine their values. Using the two first data points as emphtraining set we get

$$a = \frac{y^{(2)} - y^{(1)}}{x^{(2)} - x^{(1)}} = -0.29 \tag{1.10}$$

$$b = y^{(1)} - ax^{(1)} = 4.56 \tag{1.11}$$

This is not a very good agreement with the ideal values of $a = 2$ and $b = 3$. How about using the first and last data point as training set. The estimate of the y-intercept stays the same, but the slope estimation becomes $a = 1.59$ which is already much better.

But how would we know what the best solution is when we do not already know the solution? The answer is a procedure called **cross validation** where we use the data not used for training to validate the goodness of prediction on other data. Thus, the data not used directly to estimate the model parameters are called the **validation set**. We chose here to evaluate the goodness of the model with the mean square error (MSE) on the data not used for training

$$MSE = \frac{1}{N_{\text{val}}} \sum_{i=1}^{N_{\text{val}}} (\hat{y} - y)^2, \tag{1.12}$$

where N_{val} is the number of validation data. If we calculate the MSE for all possible data combinations we can choose as final estimation the estimation that leads to the smallest MSE of the unseen validation data points.

Cross validation is generally used to tune the **hyper-parameters**. Hyper-parameters are parameters of the learning algorithms. In our specific examples these are the choice of which data points to choose. Later we will discuss other learning algorithms that have hyper-parameter like a learning rate or the number of learning steps.

Finally, the ultimate test is using data that where not given to us during a learning phase to test the prediction of the model with the parameters that we estimated during the learning and cross-validation procedure. These data that are usually not given to the researchers during a training phase is recalled the **test set**. It is essential to be very clear about the differences between the training set, the validation set and the test set.

1.3 Unsupervised and reinforcement learning

Up to now we have discussed supervised learning where a teacher provides detailed examples of what the output of a machine should be. In **reinforcement learning** there is still some feedback from a teacher or the environment in the form of indicators that show how desirable the outputs of the learner is. This is often described as reward or punishment. However, the learning process requires some exploration as the learner must find the required action to attain a rewarding position. Such a learning requirement represents a common task in robotics and human learning, although a supervised learning system could be in itself a component of such a learning agenda, for example in the required vision system.

The last basic category of **unsupervised learning** is another important type of learning. This the of learning is more directly related to supervised learning except that the training data sets has no labels, That is, we are for example given a large data set of images but without detailed descriptions what the photographs represent. So one might ask how this can be of any use. However, there is a lot of information in the pictures itself as they give examples of how natural images look like or that there are usually edges and a certain distribution of colors. In other words, we can learn a lot about the statistics of the feature values from a large collection of unlabelled examples.

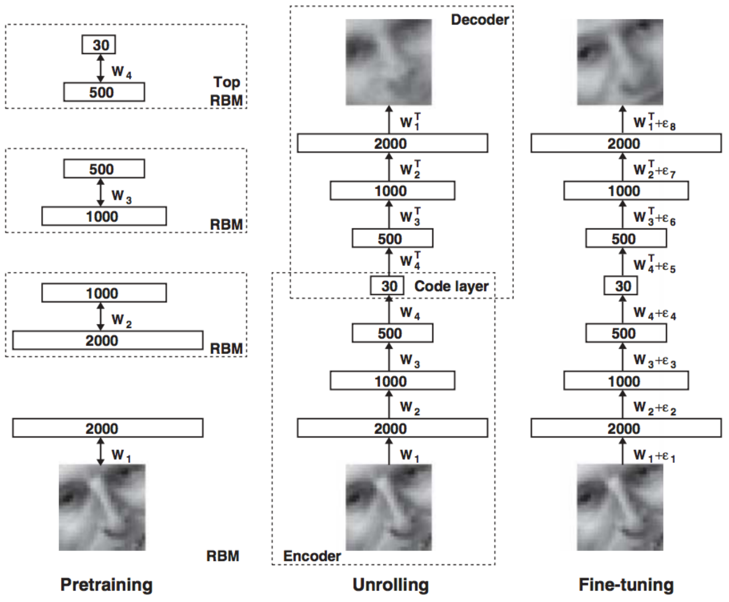


Fig. 1.2 A form of linear regression of data and cross-validation.

Some knowledge of the statistics of data is important to guide good representations of the data. We will se later that a good representation of data is a key element of solving mapping problems such as object recognition. Indeed, we can view the success of deep learning as the result of learning good layers of representations between the raw sensory input and the desired semantic space. It is therefore **representational learning** based on unsupervised and supervised learning that had build the foundation of much progress. For example, Figure 1.2 shows a famous work of Geoff Hinton and Ruslan Salakhutdinov published in Science 2006 of a neural network with several layers that transforms an input image to itself. At first it uses some unsupervised learning techniques that learns to reconstruct each input layer with a hidden layer. Such a network is called a **restricted Boltzmann machine**. Next we stack these layers on top of each other and train it like a supervised learning problem where the desired output (label) is the image itself. This is called an **autoencoder**. While the ability of translating an input image to itself might be questionable, the interesting part of such an autoencoder is that it has a number of representational layers in-between and that it has a bottleneck in the middle. This layer provides us with a **compressed representation** that is also interesting as it has been shown that it provides an interesting similarity measure between different inputs that can represent some semantic components. It is this kind of learning that provides a stream to more 'intelligent' processing.

1.4 Causal learning, overfitting and regression

I would like to close the introductory overview with on more concept that is central in machine learning. For this we go back to the supervised learning of repression. As mentioned previously finding the right parameterized function is somewhat difficult. There is an important area of **causal learning** that tries to provide specific probabilistic models of the components that provide the necessary foundations of the inference engine. Inference here means a system that can 'argue' about a solution in a probabilistic sense. Such systems fall generally in the domain of Bayesian learning, and we will include some introduction to this important systems in this course. Figure 1.3 shows an famous example form Judea Pearl, one of the inventors of this important modelling framework.

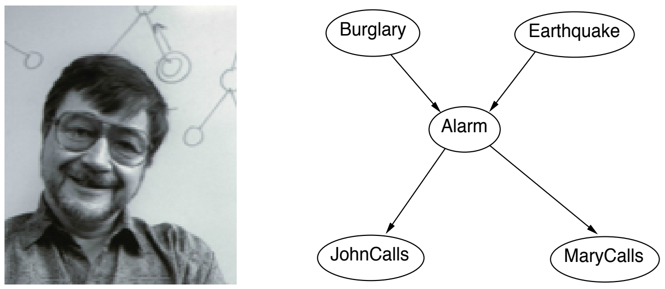


Fig. 1.3 Judea Pearl and causal modeling

Above we have discussed a case where we assumed a linear function, but regression with more general non-linear functions brings another level of challenges. An example of data that do not follow a linear trend is shown in Fig.1.4A. There, the number of transistors of microprocessors is plotted against the year each processor was introduced. This plot includes a line showing a linear regression, which is of course not very good. It is however interesting to note that this linear approximation shows some systematic deviation in some regional under and over estimation of the data. This systematic deviation or **bias** suggest that we have to take more complex functions into account. Finding the right function is one of the most difficult tasks, and there is not a simple algorithm that can give us the answer. This task is therefore an important area where experience, a good understanding of the problem domain, and a good understanding of scientific methods are required.

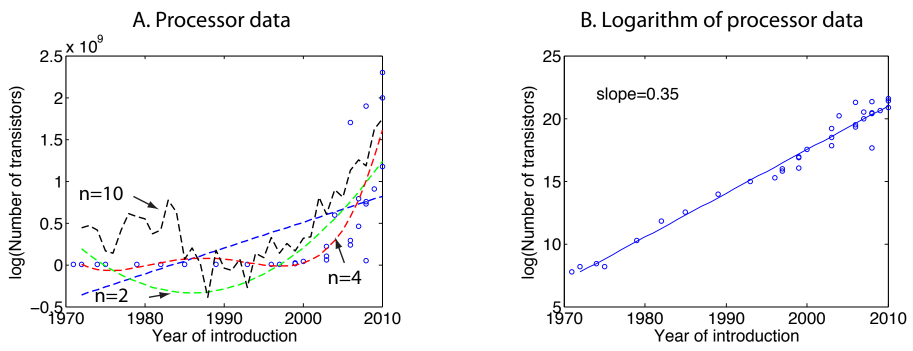


Fig. 1.4 Data showing the number of transistors in microprocessors plotted against the year they were introduced. (A) Data and some linear and polynomial fits of the data. (B) Logarithm of the data and linear fit of these data.

It is often a good idea to visualize data an various ways since the human mind is often quite advanced in 'seeing' trends and patterns. Domain-knowledge thereby very valuable as specialists in the area from which the data are collected can give important advice of or they might have specific hypothesis that can be investigated. It is also good to know common mechanisms that might influence processes. For example, the rate of change in basic growth processes is often proportional to the size of the system itself. Such an situation lead to exponential growth. (Think about why this is the case). Such situations can be revealed by plotting the functions on a logarithmic scale or the logarithm of the function as shown in Fig.1.4B. A linear fit of the logarithmic values is also shown, confirming that the average growth of the number of transistors in microprocessors is exponential.

But how about more general functions. For example, we can consider a polynomial of order n, that can be written as

$$y = \theta_0 x^0 + \theta_1 x^1 + \theta_2 x^2 + ... + \theta_n x^n \tag{1.13}$$

We can again use a regression method to determine the parameters from the data by minimizing the LMS error function between the hypothesis and the data. The LMS regression of the transistor data to a polynomials for orders $n = 2, 4, 10$ are shown in Fig.**??**A as dashed lines.

A major question when fitting data with fairly general non-linear functions is the order that we should consider. The polynomial of order $n = 4$ seem to somewhat fit the data. However, notice there are systematic deviations between the curve and the data points. For example, all the data between years 1980 and 1995 are below the fitted curve, while earlier data are all above the fitted curve. Such a systematic **bias** is typical when the order of the model is too low. However when we increase the order, then we usually get large fluctuations, or **variance**, in the curves. This fact is also called **overfitting** the data since we have typically too many parameters compared to the number of data points so that our model starts describing individual data points with their fluctuations that we earlier assumed to be due to some noise in the system.

This difficulty to find the right balance between these two effects is also called the **bias-variance tradeoff**.

The bias-variance tradeoff is quite important in practical applications of machine learning because the complexity of the underlying problem is often not know. It then becomes quite important to study the performance of the learned solutions in some detail. A schematic figure showing the bias-variance tradeoff is shown in Fig.1.5. The plot shows the error rate as evaluated by the training data (dashed line) and validation curve (solid line) when considering models with different complexities. When the model complexity is lower than the true complexity of the problem, then it is common to have a large error both in the training set and in the evaluation due to some systematic bias. In the case when the complexity of the model is larger than the generative model of the data, then it is common to have a small error on the training data but a large error on the generalization data since the predictions are becoming too much focused on the individual examples. Thus varying the complexity of the data, and performing experiments such as training the system on different number of data sets or for different training parameters or iterations can reveal some of the problems of the models.

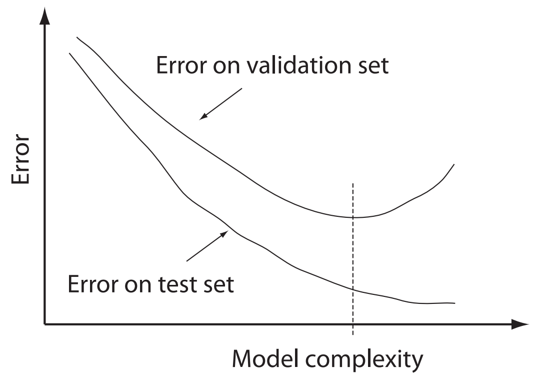


Fig. 1.5 Illustration of bias-variance tradeoff.

Deep neural networks are a form of high dimensional non-linear fitting function, and preventing overfitting is therefore a very important component in deep learning. Deep networks have many free parameters, and large data sets (big data) has therefor been important for the recent progress in this area in combination with other techniques to prevent overfitting such as a technique called dropout that we will discuss later. In general on can think about techniques to prevent overfitting as restricting the possible range of the parameters. Indeed, learning from data already represent providing information about the values of the parameters, and restricting such ranges further is a key element in machine learning. This are is generally discussed under the heading of **regularization**.

Machine learning methods are often easy to apply through application programs that implement these techniques. This is good news. However, a deeper understanding of the methods is necessary to make these applications and their conclusion appropriate. The machine learning algorithms will come up with some predictions, but if these predictions are sensible is important to comprehend and evaluate. Machine learning education needs therefore to go beyond learning how to run an application program, and this course thrives to find a balance between practical applications and their theoretical foundation.

2 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. Also, more specific resources for scientific computing with Python are:

```
http://docs.scipy.org/doc/numpy-dev/user/quickstart.html
http://docs.scipy.org/doc/numpy-dev/user/numpy-for-matlab-users.html
http://www.scipy-lectures.org/index.html
http://matplotlib.org/users/beginner.html
```

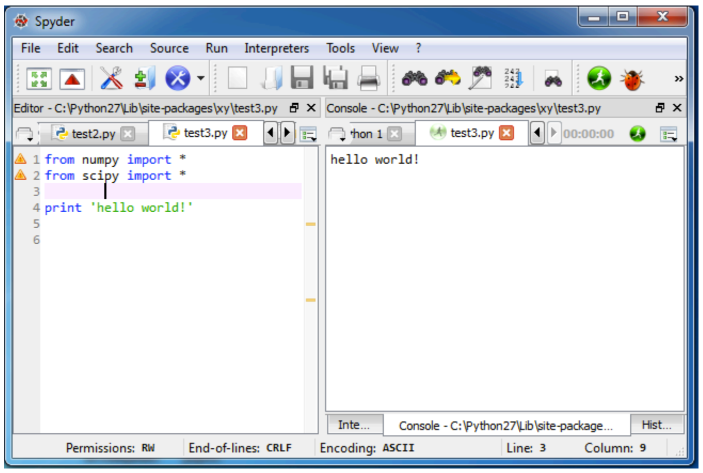
We will be using the Ubuntu operating system with Python 3 and supporting programs. An image with these components is provided so that you can install this on your own computer or use computers in our faculty. Please see the helps desk for any problems with the installation.

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.

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


Spyder
File Edit Search Source Run Interpreters Tools View ?
Editor - C:\Python27\Lib\site-packages\xy\test3.py
Console - C:\Python27\Lib\site-packages\xy\test3.py
test2.py
test3.py
Inte...
Console - C:\Python27\Lib\site-package...
Hist...

```
from numpy import *
from scipy import *

print 'hello world!'
```

```
hello world!
```

Permissions: RW End-of-lines: CRLF Encoding: ASCII Line: 3 Column: 9

Fig. 2.1 The Spyder programming environment for Python.

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])
```

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.

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

```
matrix2=array([[5,5,6],[7,8,9]])
result=matrix @ matrix2.T
print(result)
```

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")
```

Exercises

1. Write a Python function that takes a character string and prints out the character string in reverse order.
2. Write a Python program that uses 3-dimensional plotting routines to plot a two dimensional Gaussian function.
3. more

2.2 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--')
```

2.3 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.

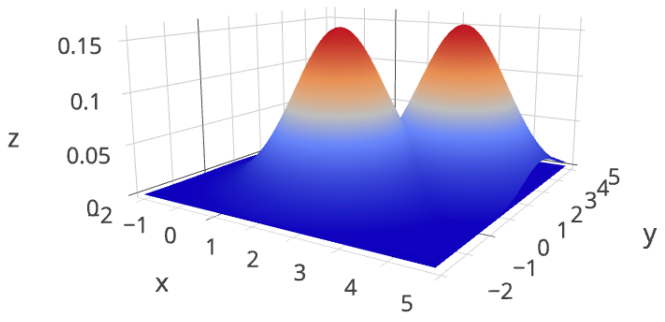


Fig. 2.2 Two 2d-Gaussian curves with unit covariance and different means.

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)
```

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()
```

2.4 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.

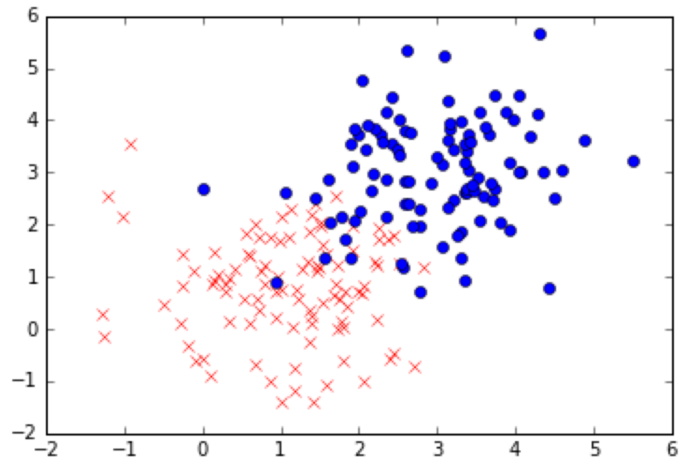
```
 from pylab import *
from sklearn import svm
from sklearn.ensemble import RandomForestClassifier
import tensorflow
from tensorflow.contrib import skflow

# training
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))
```

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

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

MLP = skflow.TensorFlowDNNClassifier(
                          hidden_units=[10, 20, 10],
                          n_classes=2,
                          batch_size=128,
                          steps=500,
                          learning_rate=0.05)
MLP.fit(x, y)



# testing
x1=array([randn(n)+1,randn(n)+1]); y1=zeros(n)
x2=array([randn(n)+3,randn(n)+3]); y2=zeros(n)+1

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

x = hstack((x1,x2)).T
y = hstack((y1,y2))

a=SVC.predict(x)
print("Percentage␣Correct␣SVM:", (n-sum(abs(y-a)))/n)
b=RF.predict(x)
print("Percentage␣Correct␣␣RF:", (n-sum(abs(y-b)))/n)
c=MLP.predict(x)
print("Percentage␣Correct␣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.

Exercise

1. What is the accuracy of the optimal solution of the two-Gaussian problem with the parameters used in the above example? Calculate this bound analytically.
2. This exercise is a little project where you should apply a binary machine learning classification to a problem of your choose. You might have an own data set



```
Percentage Correct SVM: 0.83
Percentage Correct  RF: 0.83
Percentage Correct MLP: 0.83
```

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).

or you could choose a data set from the UCI machine learning repository at http://archive.ics.uci.edu/ml/. Note that many of the problems are not binary, but you could can often change these data sets to a binary problem. For example, a hight value could be changed into two classes of small and large.