LxMLS - Lab Guide

July 19, 2011

Day 0

Basic Tutorials

In this class we will introduce several fundamental concepts needed further ahead. We start with an introduction to Python, the programming language we will use in the lab sessions. Afterwards, we present several notions on probability theory and linear algebra. Finally, we focus on numerical optimization.

The goal of this class is to give you the basic knowledge for you to understand the following lectures. We will not enter in too much detail in any of the topics.

0.1 Python

0.1.1 Running Python

You can access and run Python interactively, simply by running the `python` command. Alternatively, you can save your program to a file and run python on it:

```
python yourfile.py
```

In these lab sessions, we are going to be using Python in interactive mode several times. The standard Python interface is not very friendly, though. IPython, which stands for *interactive Python*, is an improved Python shell. It saves your command history between sessions, has basic auto-complete, and has internal support for interacting with graphs through matplotlib.

IPython is also designed to facilitate running parallel code of clusters of machines, but we will not make use of that functionality.

To run IPython, simply run `IPython` on your command line. For interactive numeric use, the `-pylab` flag imports numpy and matplotlib for you and sets up interactive graphs:

```
IPython -pylab
```

Help and Documentation

There are several ways to get help on IPython:

- Adding a question mark to the end of a function or variable and pressing Enter brings up associated documentation. Unfortunately, not all packages are well documented. Numpy and matplotlib (the two libraries we will extensively use in the lab sessions) are pleasant exceptions;
- `help('print')` gets the online documentation for the `print` keyword;
- `help()`, enters the help system.

When at the help system, type `q` to exit.
For more information on IPython (Pérez and Granger, 2007), check the website: `http://ipython.scipy.org/moin/`

Profiling

If you are interested in checking the performance of your program, you can use the command %prun in IPython (this is an IPython-only feature). For example:

```
def myfunction(x):
    ...

%prun myfunction(22)
```

Exiting

Exit IPython by typing `exit()` or `quit()` (or typing CTRL-D).

0.1.2 Python by Example

The first program of every programmer in every new language prints "Hello, World!". In Python, this simply reads:

```
In[]: print "Hello, World!"
Out[]: Hello, World!
```

Data Structures

In Python, you can create lists of items with the following syntax:

```
countries = ['Portugal','Spain','United Kingdom']
```

A string should be surrounded with apostrophes (’). You can access a list with the following:

- `len(L)`, which returns the number of items in L;
- `L[i]`, which returns the item at index i (the first item has index 0);
- `L[i:j]`, which returns a new list, containing the items between i and j.

Exercise 0.1 *Use L[i:j] to return the countries in the Iberian Peninsula.*

Loops

A loop allows a certain section of code to be repeated a certain number of times. The loops continue until a stop condition is reached. For instance, when a variable has reached a certain value or when the list you are iterating has reached its end. In Python you have `while` and `for` loops.

The following two programs output exactly the same: the even numbers from 2 to 8.

```
1 i = 2
while i < 10:
3   print i
    i += 2
```

```
for i in range(2,10,2):
2     print i
```

The `range` function is built int to Python and it creates lists containing arithmetic progressions.

Exercise 0.2 *David, John, Allysson and Anne are four of your colleagues in the Summer Course. Create a python program to greet all of them. The output should be*

```
Hello, David!
Hello, John!
Hello, Allysson!
Hello, Anne!
```

Note that you have around 100 colleagues. You should use the data structures you have just learned to minimize the lines of code you are using in this exercise.

Control Flow

The `if` statement allows to control the flow of your program. The next program makes a greeting that depends on the time of the day.

```
if hour < 12:
    print 'Good morning!'
elif hour >= 12 and hour < 20:
    print 'Good afternoon!'
else:
    print 'Good evening!'
```

Functions

A function is a block of code that can be reused to perform a similar action. The following is a function in Python.

```
def greet(hour):
    if hour < 12:
        print 'Good morning!'
    elif hour >= 12 and hour < 20:
        print 'Good afternoon!'
    else:
        print 'Good evening!'
```

If you call the function `greet` with different hours of the day, the program will greet you accordingly.

Exercise 0.3 *Note that the previous code allows the hour to be less than 0 or more than 24. Change the code in order to indicate that the hour given as input is invalid. Your output should be something like:*

```
greet(50)
Invalid hour: it should be between 0 and 24.
```

```
greet(-5)
Invalid hour: it should be between 0 and 24.
```

Indentation

In Python, indentation is important. This is how Python differentiates between nested and non-nested blocks of commands. For instance, consider the following code and its output:

```
a=1
while a <= 3:
    print a
    a += 1
```

```
1
2
3
```

Exercise 0.4 *Can you predict the output of the following code:*

```
a=1
while a <= 3:
    print a
a += 1
```

0.1.3 Plotting in Python - Matplolib

Matplotlib is a plotting library for Python. It supports 2D and 3D plots of various forms. It can show them interactively or save them to a file (several output formats are supported).

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

X = np.linspace(-4, 4, 1000)

plt.plot(X, X**2*np.cos(X**2))
plt.savefig("simple.pdf")
```

Exercise 0.5 *Try running the following on IPython, which will introduce you to some of the basic numeric and plotting operations.*

```
# This will import the numpy library
# and give it the np abbreviation
import numpy as np

# This will import the plotting library
import matplotlib.pyplot as plt

# Linspace will return 1000 points,
# evenly spaced between -4 and +4
X = np.linspace(-4, 4, 1000)

# Y[i] = X[i]**2
Y = X**2

# Plot using a red line ('r')
plt.plot(X, Y, 'r')

# arange returns points ranging from -4 to +4
# (the upper argument is excluded!)
Ints = np.arange(-4,5)

# We plot these on top of the previous plot
# using blue circles (o means a little circle)
plt.plot(Ints, Ints**2, 'bo')

# You may notice that the plot is tight around the line
# Set the display limits to see better
plt.xlim(-4.5,4.5)
plt.ylim(-1,17)
```

0.1.4 Numpy

Numpy is a library needed for scientific computing with Python.

Multidimensional Arrays

The main object of numpy is the multidimensional array. A multidimensional array is a table with all elements of the same type and can have several dimensions.

Numpy provides various functions to access and manipulate multidimensional arrays. In one dimensional arrays, you can index, slice, and iterate as you can with lists. In a two dimensional array M, you can use perform these operations along several dimensions.

- M[i,j], to access the item in the ith row and j-th column;
- M[i:j,:], to get the all the rows between the i-th and j-th;
- M[:,i], to get the i-th column of M.

Again, as it happened with the lists, the first item of every column and every row has index 0.

```
import numpy as np
A = np.array([
    [1,2,3],
    [2,3,4],
    [4,5,6]])

A[0,:] # This is [1,2,3]
A[0] # This is [1,2,3] as well

A[:,0] # this is [1,2,4]

A[1:,0] # This is [ [2], [4] ]. Why?
        # Because it is the same as A[1:n,0] where n is the
            size of the array.
```

Mathematical Operations

There are many helpful functions in numpy. For basic mathematical operations, we have `np.log`, `np.exp`, `np.cos`,... with the expected meaning. These operate both on single arguments and on arrays (where they will behave element wise).

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

X = np.linspace(0, 4 * np.pi, 1000)
C = np.cos(X)
S = np.sin(X)

plt.plot(X, C)
plt.plot(X, S)
```

Other functions take a whole array and compute a single value from it. For example, `np.sum`, `np.mean`,... These are available as both free functions and as methods on arrays.

```
import numpy as np

A = np.arange(100)
print np.mean(A)
print A.mean()

C = np.cos(A)
print C.ptp()
```

Exercise 0.6 *Run the above example and lookup the* `ptp` *function/method (use the* `?` *functionality in ipython).*

Exercise 0.7 *Consider the following approximation to compute an integral*

$$\int_0^1 f(x)dx \approx \sum_{i=0}^{999} \frac{f(i/1000)}{1000}.$$

Use numpy to implement this for $f(x) = x^2$. *You should not need to use any loops. The exact value is* $1/3$. *How close is the approximation?*

0.2 Probability Theory

Probability is the mathematical language for quantifying uncertainty. The **sample space** $\mathcal{X}$ is the set of possible outcomes of an experiment. **Events** are subsets of $\mathcal{X}$.

Example 0.1 (*discrete space*) *Let* H *denote "heads" and* T *denote "tails." If we toss a coin twice, then* $\mathcal{X} = \{HH, HT, TH, TT\}$. *The event that the first toss is heads is* $A = \{HH, HT\}$.

Sample space can also be *continuous* (eg., $\mathcal{X} = \mathbb{R}$). The union of events A and B is defined as $A \bigcup B = \{\omega \in \mathcal{X} \mid \omega \in A \vee \omega \in B\}$. If $A_1, \ldots, A_n$ is a sequence of sets then $\bigcup_{i=1}^{n} A_i = \{\omega \in \mathcal{X} \mid \omega \in A_i \text{ for at least one i}\}$. We say that $A_1, \ldots, A_n$ are **disjoint** or **mutually exclusive** if $A_i \cap A_j = \varnothing$ whenever $i \neq j$.

We want to assign a real number $P(A)$ to every event A, called the **probability** of A. We also call P a **probability distribution** or **probability measure**.

Definition 0.1 *A function P that assigns a real number $P(A)$ to each event A is a* **probability distribution** *or a* **probability measure** *if it satisfies the three following axioms:*

Axiom 1: $P(A) \geq 0$ for every A
Axiom 2: $P(\mathcal{X}) = 1$
Axiom 3: If $A_1, \ldots, A_n$ *are disjoint then*

$$P\left(\bigcup_{i=1}^{n} A_i\right) = \sum_{i=1}^{n} P(A_i).$$

One can derive many properties of P from these axioms:

$$\begin{array}{rcll} P(\varnothing) & = & 0 & \\ A \subseteq B & \Rightarrow & P(A) \leq P(B) & \\ 0 \leq & P(A) & \leq 1 & \\ P(A') & = & 1 - P(A) & \text{(A' is the complement of A)} \\ P(A \cup B) & = & P(A) + P(B) - P(A \cap B) & \\ A \cap B = \phi & \Rightarrow & P(A \cup B) = P(A) + P(B). & \end{array}$$

An important case is when events are **independent**, this is also a usual approximation which lends several practical advantages to the computation of joint probability.

Definition 0.2 *Two events A and B are* **independent** *if*

$$P(AB) = P(A)P(B) \tag{1}$$

often denoted as $A \perp\!\!\!\perp B$. *A set of events* $\{A_i : i \in I\}$ *is independent if*

$$P\left(\bigcap_{i \in J} A_i\right) = \prod_{i \in J} P(A_i)$$

for every finite subset J of I.

For events A and B, where $P(B) > 0$, **conditional probability** of A given B has occurred is defined as

$$P(A|B) = \frac{P(AB)}{P(B)}. \tag{2}$$

Events A and B are independent if and only if $P(A|B) = P(A)$. This follows from the definitions of independence and conditional probability.

A preliminary result that forms the basis for the famous Bayes' theorem is the law of total probability which states that if $A_1, \ldots, A_k$ is a partition of $\mathcal{X}$, then for any event B,

$$P(B) = \sum_{i=1}^{k} P(B|A_i)P(A_i). \tag{3}$$

Using Equations 2 and 3, one can derive the famous Bayes' theorem.

Theorem 0.1 (Bayes' Theorem) *Let $A_1, \ldots A_k$ be a partition of $\mathcal{X}$ such that $P(A_i) > 0$ for each i. If $P(B) > 0$ then, for each $i = 1, \ldots, k$,*

$$P(A_i|B) = \frac{P(B|A_i)P(A_i)}{\sum_j P(B|A_j)P(A_j)}. \tag{4}$$

Remark 0.1 *$P(A_i)$ is called the* **prior probability** *of A and $P(A_i|B)$ is the* **posterior probability** *of A.*

Remark 0.2 *In Bayesian Statistical Inference, Bayes' theorem is used to compute the estimates of parameters for distributions from data; Where, prior is the initial* belief *about the parameters, likelihood is the distribution function of the parameter (usually trained from data) and posterior is the* updated belief *about the parameters.*

0.2.1 Probability distribution functions

A **random variable** is a mapping $X : \mathcal{X} \to \mathbb{R}$ that assigns a real number $X(\omega)$ to each outcome ω. Given a random variable X, an important function called the **cumulative distributive function** (or **distribution function**) is defined as:

Definition 0.3 *The* **cumulative distribution function** *CDF $F_X : \mathbb{R} \to [0, 1]$ of a random variable X is defined by $F_X(x) = P(X \leq x)$.*

The CDF is important because it captures the complete information about the random variable. The CDF is right-continuous, non-decreasing and is normalized ($\lim_{x\to-\infty} F(x) = 0$ and $\lim_{x\to\infty} F(x) = 1$).

Example 0.2 (*discrete CDF*) *Flip a fair coin twice and let X be the random variable indicating the number of heads. Then $P(X = 0) = P(X = 2) = 1/4$ and $P(X = 1) = 1/2$. The distribution function is*

$$F_X(x) = \begin{cases} 0 & x < 0 \\ 1/4 & 0 \leq x < 1 \\ 3/4 & 1 \leq x < 2 \\ 1 & x \geq 2. \end{cases}$$

Definition 0.4 *X is discrete if it takes countable many values* $\{x_1, x_2, \ldots\}$. *We define the* **probability function** *or* **probability mass function** *for X by*

$$f_X(x) = P(X = x).$$

Definition 0.5 *A random variable X is* **continuous** *if there exists a function* f_X *such that* $f_X \geq 0$ *for all* x, $\int_{-\infty}^{\infty} f_X(x)dx = 1$ *and for every* $a \leq b$

$$P(a < X < b) = \int_a^b f_X(x)dx. \tag{5}$$

The function f_X *is called the* **probability density function** *(PDF). We have that*

$$F_X(x) = \int_{-\infty}^{x} f_X(t)dt$$

and $f_X(x) = F_X'(x)$ *at all points x at which* F_X *is differentiable.*

A discussion of a few important distributions and related properties:

0.2.2 Bernoulli

The **Bernoulli distribution** is a discrete probability distribution that takes 1 with the success probability p and 0 with the failure probability $q = 1 - p$. A single bernoulli trial is parametrized with the success probability p, and the input $k \in \{0, 1\}$ (1=success, 0=failure), and can be expressed as

$$f(k; p) = p^k q^{1-k} = p^k (1 - p)^{1-k}$$

0.2.3 Binomial

The probability distribution for the number of successes in n Bernoulli trials is called a **Binomial distribution**, which is also a discrete distribution. The Binomial distribution can be expressed as exactly j successes is

$$f(j, n; p) = \binom{n}{j} p^j q^{n-j} = \binom{n}{j} p^j (1 - p)^{n-j}$$

where n is the number of Bernoulli trails with probability p of success on each trial.

0.2.4 Categorical

The Categorical distribution (often conflated with the Multinomial distribution, in fields such as Natural Language Processing), is another generalization of the Bernoulli distribution, allowing the definition of a set of possible outcomes, rather than simply the events "success" and "failure" defined in the Bernoulli distribution. Considering a set of outcomes indexed from 1 to n, the distribution takes the form of

$$f(x_i; p_1, ..., p_n) = p_i.$$

Where parameters $p_1, ..., p_n$ is the set with the occurrence probability of each outcome. Note that we must ensure that $\sum_{i=1}^{n} p_i = 1$, so we can set $p_n = \sum_{i=1}^{n-1} p_i = 1$.

0.2.5 Multinomial

The Multinomial distribution is a generalization of the Binomial distribution and the Categorical distribution, since it considers multiple outcomes, as the Categorial distribution, and multiple trials, as in the Binomial distribution. Considering a set of outcomes indexed from 1 to n, the vector $x_1, ..., x_n$, where x_i indicates the number of times the event with index i occurs, follows the Multinomial distribution

$$f(x_1, ..., x_n; p_1, ..., p_n) = \frac{n!}{x_1!...x_n!} p_1^{x_1} ... p_n^{x_n}.$$

Where parameters $p_1, ..., p_n$ is the occurrence probability of the respective outcome.

0.2.6 Gaussian Distribution

A very important theorem in probability theory called the **Central Limit Theorem** states that, under very general conditions, if we sum a very large number of mutually independent random variables, then the distribution of the sum can be closely approximated by a certain specific continuous density called the normal (or Gaussian) density. The normal density function with parameters μ and σ is defined as follows:

$$f_X(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-(x-\mu)^2/2\sigma^2}, \ -\infty < x < \infty.$$

Fig. 1, compares a plot of normal density for the cases $\mu = 0$ and $\sigma = 1$, and $\mu = 0$ and $\sigma = 2$.

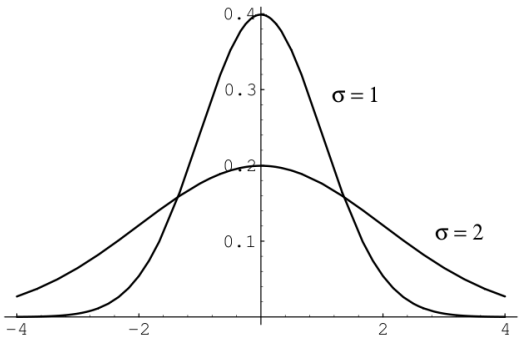

Figure 1: Normal density for two sets of parameter values.

0.2.7 Conjugate Priors

Definition 0.6 *let $\mathcal{F} = \{f_X(x|s), s \in \mathcal{X}\}$ be a class of likelihood functions; let $\mathcal{P}$ be a class of probability (density or mass) functions; if, for any x, any $p_S(s) \in \mathcal{P}$, and any $f_X(x|s) \in \mathcal{F}$, the resulting a posteriori probability function $p_S(s|x) = f_X(x|s)p_S(s)$ is still in $\mathcal{P}$, then $\mathcal{P}$ is called a conjugate family, or a family of* **conjugate priors**, *for $\mathcal{F}$.*

0.2.8 Maximum Likelihood Estimation

Until now, we assumed that, for every distribution, the parameters θ are known and are used we calculate $p(x|\theta)$. There are some cases where the values of the parameters are easy to infer, such as the probability p getting a head using a fair coin, used on a Bernoulli or Binomial distribution. However, in many problems, these values are complex to define, and it is more viable to estimate the parameters using the data x. For instance, in the example above with the coin toss, if the coin is somehow tampered to have a biased behavior, rather than examining the dynamics of the structure of the coin to infer a parameter for p, a person could simply throw the coin n times and count the number of heads h and set $p = \frac{h}{n}$. In doing this, the person is using the data x to estimate θ.

With this in mind, we will know generalize this process. We define the probability $p(\theta|x)$, which is probability of the parameter θ, given the data x. This probability is called **likelihood** $\mathcal{L}(\theta|x)$ and measures how well the parameter θ models the data x. This can be defined in terms of the distribution f as

$$\mathcal{L}(\theta|x_1, ..., x_n) = \prod_{i=1}^{n} f(x_i|\theta)$$

where $x_1, ..., x_n$ are iid samples.

To understand this concept better, we go back to the tampered coin example again. Suppose that we throw the coin 5 times and get the sequence [1,1,1,1,1] (1=head, 0=tail). Using the Bernoulli distribution 0.2.2 f to model this problem, we get the following likelihood values:

- $\mathcal{L}(0, x) = f(1, 0)^5 = 0^5 = 0$
- $\mathcal{L}(0.2, x) = f(1, 0.2)^5 = 0.2^5 = 0.00032$
- $\mathcal{L}(0.4, x) = f(1, 0.4)^5 = 0.4^5 = 0.01024$
- $\mathcal{L}(0.6, x) = f(1, 0.6)^5 = 0.6^5 = 0.07776$
- $\mathcal{L}(0.8, x) = f(1, 0.8)^5 = 0.8^5 = 0.32768$
- $\mathcal{L}(1, x) = f(1, 1)^5 = 1^5 = 1$

If we get the sequence [1,0,1,1,0] instead, the likelihood values would be:

- $\mathcal{L}(0, x) = f(1, 0)^3 f(0, 0)^2 = 0^3 \times 1^2 = 0$
- $\mathcal{L}(0.2, x) = f(1, 0.2)^3 f(0, 0.2)^2 = 0.2^3 \times 0.8^2 = 0.00512$
- $\mathcal{L}(0.4, x) = f(1, 0.4)^3 f(0, 0.4)^2 = 0.4^3 \times 0.6^2 = 0.02304$
- $\mathcal{L}(0.6, x) = f(1, 0.6)^3 f(0, 0.6)^2 = 0.6^3 \times 0.4^2 = 0.03456$
- $\mathcal{L}(0.8, x) = f(1, 0.8)^3 f(0, 0.8)^2 = 0.8^3 \times 0.2^2 = 0.02048$
- $\mathcal{L}(1, x) = f(1, 1)^5 = 1^3 \times 0^2 = 0$

We can that likelihood is highest when the distribution f with parameter p is the best fit for the observed samples. Thus, the best estimate for p according to x would the value where $\mathcal{L}(p, x)$ is highest.

The value of the parameter θ with the highest likelihood is called **maximum likelihood estimate(MLE)** and is defined as

$$\hat{\theta}_{mle} = argmax_\theta \mathcal{L}(\theta|x)$$

Finding this for our example is relatively easy, since we can simply derivate the likelihood function to find the absolute maximum. For the sequence [1,0,1,1,0], the likelihood would be given as

$$\mathcal{L}(p, x) = f(1, p)^3 f(0, p)^2 = p^3(1 - p)^2$$

And the MLE estimate would be given by:

$$\frac{\delta \mathcal{L}(p, x)}{\delta p} = 0$$

which resolves into

$$p_{mle} = 0.6$$

Exercise 0.8 *Over the next couple of exercises we will make use of the Galton dataset, a dataset of heights of fathers and sons from the 1877 paper that first discussed the "regression to the mean" phenomenon.*

- *Use the* `load()` *function in the* `galton.py` *file to load the dataset.*
- *What are the mean height and standard deviation of all the people in the sample? What is the mean height of the fathers and of the sons?*
- *Plot a histogram of all the heights (you might want to use the* `plt.hist` *function and the* `ravel` *method on arrays).*
- *Plot the height of the father versus the height of the son.*
- *You should notice that there are several points that are exactly the same (e.g., there are 21 pairs with the values 68.5 and 70.2). Use the* `?` *command in ipython to read the documentation for the* `numpy.random.rand` *function and add random jitter (i.e., move the point a little bit) to the points before displaying them. Does your impression of the data change?*

0.3 Essential Linear Algebra

Linear Algebra provides a compact way of representing and representing on sets of linear equations.

$$\begin{aligned} 4x_1 \quad -5x_2 \quad &= -13 \\ -2x_1 \quad +3x_2 \quad &= 9 \end{aligned}$$

This is a system of linear equations in 2 variables. In matrix notation we can write the system more compactly as

$$Ax = b$$

with

$$A = \begin{bmatrix} 4 & 5 \\ -2 & 3 \end{bmatrix}, \; b = \begin{bmatrix} -13 \\ 9 \end{bmatrix}$$

0.3.1 Notation

We use the following notation:

- By $A \in \mathbb{R}^{m \times n}$, we denote a **matrix** with m rows and n columns, where the entries of A are real numbers.

- By $x \in \mathbb{R}^n$, we denote a **vector** with n entries. A vector can also be thought of as a matrix with n rows and 1 column, known as a **column vector**. A **row vector** — a matrix with 1 row and n columns is denoted as x^T, the transpose of x.
- The ith element of a vector x is denoted x_i

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}.$$

Exercise 0.9 *In the rest of the school we will represent both matrices and vectors as numpy arrays. You can create arrays in different ways, one possible way is to create an array of zeros.*

```
import numpy as np
m = 3
n = 2
a = np.zeros([m,n])
print a
[[ 0.  0.]
 [ 0.  0.]
 [ 0.  0.]]
```

You can check the shape and the data type of your array using the following commands:

```
print a.shape
(3, 2)
print a.dtype.name
float64
```

*This shows you that "a" is an 3*2 arrays of type float64. By default, arrays contain 64 bit floating point numbers. You can specify the particular array type by using the keyword dtype.*

```
a = np.zeros([m,n],dtype=int)
print a.dtype
int64
```

(On your computer, particularly if you have an older computer, `int` *might denote 32 bits integers).*

There are many other ways to create arrays, you can create arrays from lists of numbers:

```
a = np.array([[2,3],[3,4]])
print a
[[2 3]
 [3 4]]
```

There are many more ways to create arrays in numpy and we will get to see them as we progress in the classes.

0.3.2 Some Matrix Operations and Properties

- Product of two matrices $A \in \mathbb{R}^{m\times n}$ and $B \in \mathbb{R}^{n\times p}$ is the matrix $C = AB \in \mathbb{R}^{m\times p}$, where

$$C_{ij} = \sum_{k=1}^{n} A_{ik}B_{kj}.$$

Exercise 0.10 *You can multiply two matrix by looping over both indexes and multiplying the individual entries.*

```
a = np.array([[2,3],[3,4]])
b = np.array([[1,1],[1,1]])
a_dim1, a_dim2 = a.shape
b_dim1, b_dim2 = b.shape
c = np.zeros([a_dim1,b_dim2])
for i in xrange(a_dim1):
    for j in xrange(b_dim2):
        for k in xrange(a_dim2):
            c[i,j] += a[i,k]*b[j,k]
print c
```

This is, however, cumbersome and inefficient. Numpy supports matrix multiplication with the dot function:

```
d = np.dot(a,b)
print d
```

Important note: *with numpy, you must use* `dot` *to get matrix multiplication, the expression a * b denotes element-wise multiplication.*

- Matrix multiplication is associative: $(AB)C = A(BC)$.

- Matrix multiplication is distributive: $A(B+C) = AB + AC$.
- Matrix multiplication is (generally) not commutative : $AB \neq BA$.
- Given two vectors $x, y \in \mathbb{R}^n$ the product $x^T y$, called **inner product** or **dot product**

$$x^T y \in \mathbb{R} = \begin{bmatrix} x_1 & x_2 & \dots & x_n \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \sum_{i=1}^{n} x_i y_i.$$

```
a = np.array([1,2])
b = np.array([1,1])
np.dot(a,b)
```

- Given vectors $x \in \mathbb{R}^m$ and $y \in \mathbb{R}^n$, the **outer product** $xy^T \in \mathbb{R}^{m \times n}$ is a matrix whose entries are given by $(xy^T)_{ij} = x_i y_j$,

$$xy^T \in \mathbb{R}^{m \times n} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \begin{bmatrix} y_1 & y_2 & \dots & y_n \end{bmatrix} = \begin{bmatrix} x_1 y_1 & x_1 y_2 & \dots & x_1 y_n \\ x_2 y_1 & x_2 y_2 & \dots & x_2 y_n \\ \vdots & \vdots & \ddots & \vdots \\ x_m y_1 & x_m y_2 & \dots & x_m y_n \end{bmatrix}.$$

```
np.outer(a,b)
array([[1, 1],
       [2, 2]])
```

- The **identity matrix**, denoted $I \in \mathbb{R}^{n \times n}$, is a square matrix with ones on the diagonal and zeros everywhere else. That is,

$$I_{ij} = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases}$$

It has the property that for all $A \in \mathbb{R}^{m \times n}$, $AI = A = IA$.

```
I = np.eye(2)
x = np.array([2.3, 3.4])

print I
```

```
print np.dot(I,x)

[[ 1.,  0.],
 [ 0.,  1.]]
[2.3, 3.4]
```

- A **diagonal matrix** is a matrix where all non-diagonal elements are 0.
- The **transpose** of a matrix results from "'flipping" the rows and columns. Given a matrix $A \in \mathbb{R}^{m \times n}$, the transpose $A^T \in \mathbb{R}^{n \times m}$ is the $n \times m$ matrix whose entries are given by $(A^T)_{ij} = A_{ji}$.

  Also, $(A^T)^T = A$; $(AB)^T = B^T A^T$; $(A + B)^T = A^T + B^T$

  In numpy, you can access the transpose of a matrix as the `T` attribute:

```
A = np.array([ [1, 2], [3, 4] ])
print A.T
```

- A square matrix $A \in \mathbb{R}^{n \times n}$ is **symmetric** if $A = A^T$.
- The **trace** of a square matrix $A \in \mathbb{R}^{n \times n}$ is the sum of the diagonal elements, $\text{tr}(A) = \sum_{i=1}^{n} A_{ii}$

0.3.3 Norms

The **norm** of a vector is informally the measure of the "length" of the vector. The commonly used Euclidean or ℓ_2 norm is given by

$$\|x\|_2 = \sqrt{\sum_{i=1}^{n} x_i^2}.$$

- More generally, the ℓ_p norm of a vector $x \in \mathbb{R}^n$, where $p \geq 1$ is defined as

$$\|x\|_p = \left(\sum_{i=1}^{n} |x_i|^p \right)^{1/p}.$$

Note: ℓ_1 norm : $\|x\|_1 = \sum_{i=1}^{n} |x_i|$ ℓ_∞ norm : $\|x\|_\infty = \max_i |x_i|$.

0.3.4 Linear Independence and Rank

A set of vectors $\{x_1, x_2, \ldots, x_n\} \subset \mathbb{R}^m$ is said to be **(linearly) independent** if no vector can be represented as a linear combination of the remaining vectors. Conversely, if one vector belonging to the set can be represented as a linear combination of the remaining vectors, then the vectors are said to be **linearly dependent**. That is, if

$$x_n = \sum_{i=1}^{n-1} \alpha_i x_i$$

for some scalar values $\alpha_1, \ldots, \alpha_{n-1} \in \mathbb{R}$.

- The **rank** of a matrix is the number of linearly independent columns.
- For $A \in R^{m \times n}$, $\text{rank}(A) \leq \min(m, n)$. If $\text{rank}(A) = \min(m, n)$, then A is said to be **full rank**.
- For $A \in R^{m \times n}$, $\text{rank}(A) = \text{rank}(A^T)$.
- For $A \in R^{m \times n}$, $B \in R^{n \times p}$, $\text{rank}(AB) \leq \min(\text{rank}(A), \text{rank}(B))$.
- For $A, B \in R^{m \times n}$, $\text{rank}(A + B) \leq \text{rank}(A) + \text{rank}(B)$.
- Two vectors $x, y \in \mathbb{R}^n$ **orthogonal** if $x^T y = 0$. A square matrix $U \in \mathbb{R}^{n \times n}$ is orthogonal if all its columns are orthogonal to each other and are normalized ($\|x\|_2 = 1$), It follows that

$$U^T U = I = U U^T.$$

0.4 Numerical optimization

Most problems in machine learning require minimization/maximization of functions (likelihoods, risk, energy, entropy, etc.,).

$$x^* = \arg\min_x f(x)$$

In a few special cases, we can solve this minimization problem analytically in closed form (solving for optimal x^* in $\nabla_x f(x^*) = 0$), but in most cases such a solutions does not exist. In this section we will cover some basic notions of numerical optimization. The goal is to provide the intuitions behind the methods that will be used in the rest of the school. There are plenty of good textbooks in the subject that you can consult for more information (Nocedal and Wright, 1999; Bertsekas et al., 1995; Boyd and Vandenberghe, 2004).

The most common way to solve the problems when no closed form solution is available is to resort to an iterative algorithm. In this Section, we will see some of these iterative optimization techniques. These iterative algorithms

compute a sequence of points $x^{(0)}, x^{(1)}, \ldots \in \text{domain}(f)$ such that hopefully $x^t = x^*$. Such a sequence is called the **minimizing sequence** for the problem.

0.4.1 Convex Functions

One important concept of the function $f(x)$ is if it is **convex function** (in the shape of a bowl) or a **non-convex-function**. Figures 2 and 3 show an example of a convex and a non-convex function. Convex functions are particular useful since you are guarantee that the minimizing sequence converges to the true global minimum of the function, while in non-convex functions you can only guarantee to reach a local minimum.

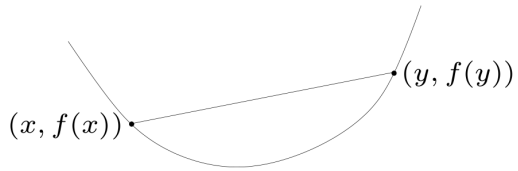

Figure 2: Illustration of a convex function. The line segment between any two points on the graph lies entirely above the curve.

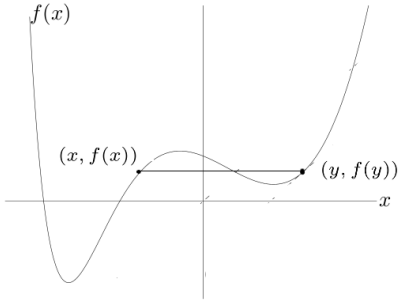

Figure 3: Illustration of a non-convex function. Note the line segment intersecting the curve.

Intuitively, imagine dropping a ball on either side of Figure 2, the ball will role to the bottom of the bowl independently from where it is dropped. This is the main benefits of a convex functions. On the other hand if you drop a ball from the left side of Figure 3 it will reach a different position than if you drop a ball from its right side. Moreover, dropping it from the left side will lead you to a much better place than if you drop the ball from the right side. This is the main problem with non-convex function, there are no guarantees about the quality of the local minimum you find.

Function $f(x)$	Derivative $\frac{\partial f}{\partial x}$
x^2	$2x$
x^n	nx^{n-1}
$\log(x)$	$\frac{1}{x}$
$\exp(x)$	$\exp(x)$
$\frac{1}{x}$	$-\frac{1}{x^2}$

Table 1: Some derivative examples

More formally, some concepts to understand about convex functions are:
A **line segment** between points x_1 and x_2: contains all points such that

$$x = \theta x_1 + (1 - \theta)x_2$$

where $0 \leq \theta \leq 1$.

A **convex set** contains the line segment between any two points in the set

$$x_1, x_2 \in C, \quad 0 \leq \theta \leq 1 \quad \Rightarrow \quad \theta x_1 + (1 - \theta)x_2 \in C$$

A function $f : \mathbb{R}^n \rightarrow R$ is a **convex function** if the domain of f is a convex set and

$$f(\theta x + (1 - \theta)y) \leq \theta f(x) + (1 - \theta)f(y)$$

for all $x, y \in$ domain of f, $0 \leq \theta \leq 1$

0.4.2 Derivative and Gradient

The **derivative** of a function is a measure of how the function varies with its input variables. Given an interval $[a, b]$ one can compute how the function varies within that interval by calculating the average of the function in that interval.

$$\frac{f(a) - f(b)}{a - b} \tag{6}$$

The derivative can be seen as the limit as the interval goes to zero, and it gives us the slope of the function at that point.

$$\frac{\partial f}{\partial x} = \lim_{h=0} \frac{f(x + h) - f(x)}{h} \tag{7}$$

Table 1 shows derivatives of some functions that we will be using during the school.

An important rule of derivation is the chain. Consider $h = f \circ g$, and $u = g(x)$, then:

$$\frac{\partial h}{\partial x} = \frac{\partial f}{\partial u} \cdot \frac{\partial g}{\partial x} \tag{8}$$

Example 0.3 *Consider the function* $h(x) = exp(x^2)$.
$h(x) = f(g(x)) = f(u) = exp(u)$, *where* $u = g(x) = x^2$.
$\frac{\partial h}{\partial x} = \frac{\partial f}{\partial u} \cdot \frac{\partial u}{\partial x} = exp(u) \cdot 2x = exp(x^2) \cdot 2x$

Example 0.4 *Consider the function* $f(x) = x^2$ *and its derivative* $\frac{\partial f}{\partial x}$. *Look at the derivative of that function at points [-2,0,2], draw the tangent to the graph in that point.*

$\frac{\partial f}{\partial x}(-2) = -4$, $\frac{\partial f}{\partial x}(0) = 0$, *and* $\frac{\partial f}{\partial x}(2) = 4$

For example, the tangent equation for $x = -2$ *is* $y = -4x - b$, *where* $b = f(-2)$

The following code plots the function and the derivatives on those points using matplotlib (See Figure 4).

```
a = np.arange(-5,5,0.01)
f_x = np.power(a,2)
plt.plot(a,f_x)

plt.xlim(-5,5)
plt.ylim(-5,15)

k= np.array([-2,0,2])
plt.plot(k,k**2,"bo")
for i in k:
    plt.plot(a, (2*i)*a - (i**2))
```

The **gradient** of a function is a generalization of the derivative concept we just saw before, for several dimensions. Lets assume we have a function $f(x)$ where $x \in \mathbb{R}^2$ so can be seen as a pair $x = x_1, x_2$ then the gradient measures the slope of the function in both directions. $\nabla_x f(x) = [\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}]$.

0.4.3 Gradient Based Methods

Gradient based methods are probably the most common methods used for finding the minimizing sequence for a given function. The methods use in this class will make use of the function value $f(x)$ as well as the gradient of the function $\nabla_x f(x)$. The simplest method is the **Gradient descent** method, an unconstrained first-order optimization algorithm.

The intuition of this method is as follows: You start at a given point x_0 and compute the gradient at that point $\nabla_{x_0} f(x)$. You then take a step of length η on the direction of the negative gradient to find a new point: $x_1 = x_0 - \eta \nabla_{x_0} f(x)$.

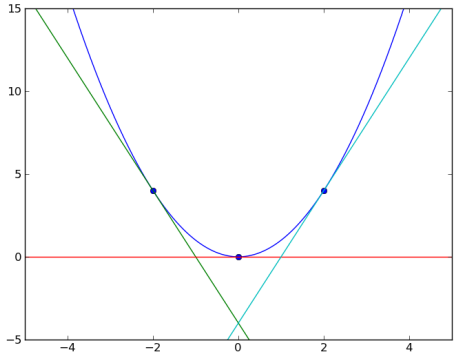

Figure 4: Illustration of the gradient of the function $f(x^2)$ at three different points $x = [-2, 0.2]$. Note that at point $x = 0$ the gradient is zero which corresponds to the minimum of the function.

You proceed until the algorithm until you have reached a minimum (local or global). Recall from the previous subsection that you can identify the minimum by testing if the norm of the gradient is zero $|\nabla f(x)| = 0$.

There are several practical concerns even with this basic algorithm to ensure both that the algorithm converges (reaches the minimum) and that it does so in fast way (by fast we mean the number of function and gradient evaluations).

- **Step Size** η A first question is how to find the step length η. One condition is that *eta* should guarantee sufficient decrease in the function value. We will not cover these methods here but the most common ones are **Backtracking line search** or the **Wolf Line Search** (Nocedal and Wright, 1999).

- **Descent Direction** A second problem is that using the negative gradient as direction can lead to a very slow convergence. Different methods that change the descent direction by multiplying the gradient by a matrix β have been proposed that guarantee a faster convergence. Two notable methods are the Conjugate Gradient (CG) and the Limited Memory Quasi Newton methods(LBFGS) (Nocedal and Wright, 1999).

- **Stopping Criteria** Finally, it will normally not be possible to reach full convergence either because it will be too slow, or because of numerical issues (computers cannot perform exact arithmetic). So normally we need to define a stopping criteria for the algorithm. Three common criteria (that are normally used together) are: a maximum number of iterations; the gradient norm be smaller than a given threshold $|\nabla f(x)| \leq \eta_1$, or

the normalized difference in the function value be smaller than a given threshold $\frac{|f(x_t)-f(x_{t-1})|}{\max(|f(x_t)|,|f(x_{t-1})|)} \leq \eta_2$

Algorithm 1 shows the general gradient based algorithm. Note that for the simple gradient descent algorithm β is the identity matrix and the descent direction is just the negative gradient of the function $|-\nabla f(x)|$. Figure 5 shows an illustration of the gradient descent algorithm.

Algorithm 1 Gradient Descent

1: **given** a starting point $x_0, i = 0$
2: **repeat**
3: Compute step size η
4: Compute descent direction β
5: $x_{i+1} \leftarrow x_i + \eta\beta\nabla f(x_i)$
6: $i \leftarrow i + 1$
7: **until** stopping criterion is satisfied.

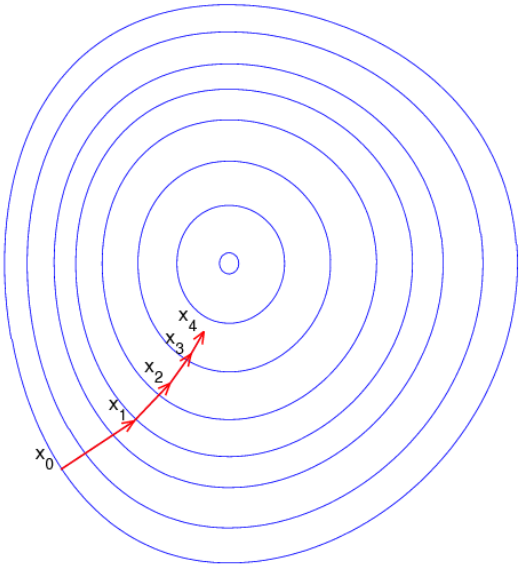

Figure 5: Illustration of gradient descent. The blue circles correspond to the function values at different points, while the red lines correspond to steps taken in the negative gradient direction.

Exercise 0.11 *Consider the function* $f(x) = (x+2)^2 - 16\exp\left(-(x-2)^2\right)$. *Make a function that computes the function value given* x.

```
def get_y(x):
    value =  pow((x+2),2) - 16*math.exp(-x*(x-2))
    return value
```

Draw a plot around $x \in [-8, 8]$.

```
x = np.arange(-8,8,0.001)
y = map(lambda l: get_y(l),x)
plot(x,y)
```

Calculate the derivative of the function $f(x)$*, implement the function* get_grad(x).

```
def get_grad(x):
    return (2*x+4)-16*(-2*x + 4)*np.exp(-((x-2)**2))
```

Use the method gradient_descent *to find the minimum of this function. Convince yourself that the code is doing the proper thing. Look at the constants we defined. Note, that we are using a simple approach to pick the step size (always have the value step_size) which is not necessarily correct.*

```
def gradient_descent(start_x,func,grad):
    # Precision of the solution
    prec = 0.0001
    #Use a fixed small step size
    step_size = 0.1
    #max iterations
    max_iter = 100
    x_new = start_x
    res = []
    for i in xrange(max_iter):
        x_old = x_new
        #Use beta equal to -1 for gradient descent
        x_new = x_old - step_size * get_grad(x_new)
        f_x_new = get_y(x_new)
        f_x_old = get_y(x_old)
        res.append([x_new,f_x_new])
        if(abs(f_x_new -f_x_old) < prec):
            print "change in function values to small, leaving"
            return np.array(res)
    print "exceeded maximum number of iterations, leaving"
    return np.array(res)
```

Run the gradient descent algorithm starting from $x_0 = -8$ and plot the minimizing sequence.

```
x_0 = -8
res = gradient_descent(x_0,get_y,get_grad)
plot(res[:,0],res[:,1],'+')
```

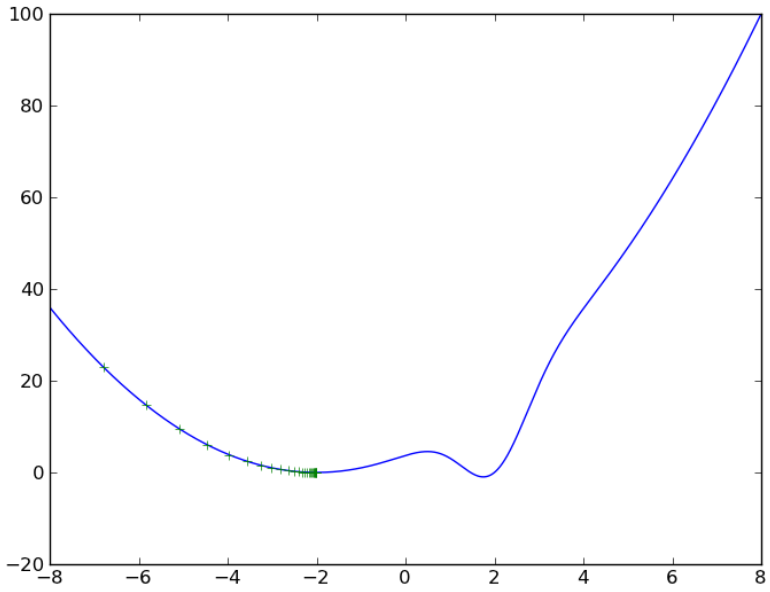

Figure 6: Example of running gradient descent starting on point $x_0 = -8$ for function $f(x) = (x+2)^2 - 16\exp\left(-(x-2)^2\right)$. The function is represented in blue, while the points of the minimizing sequence are displayed as green squares.

Figure 6 shows the resulting minimizing sequence. Note that the algorithm converged to a minimum, but since the function is not convex it converged only to a local minimum.

Now try the same exercise starting from the initial point $x_0 = 8$.

```
x_0 = 8
res = gradient_descent(x_0,get_y,get_grad)
plot(res[:,0],res[:,1],'+')
```

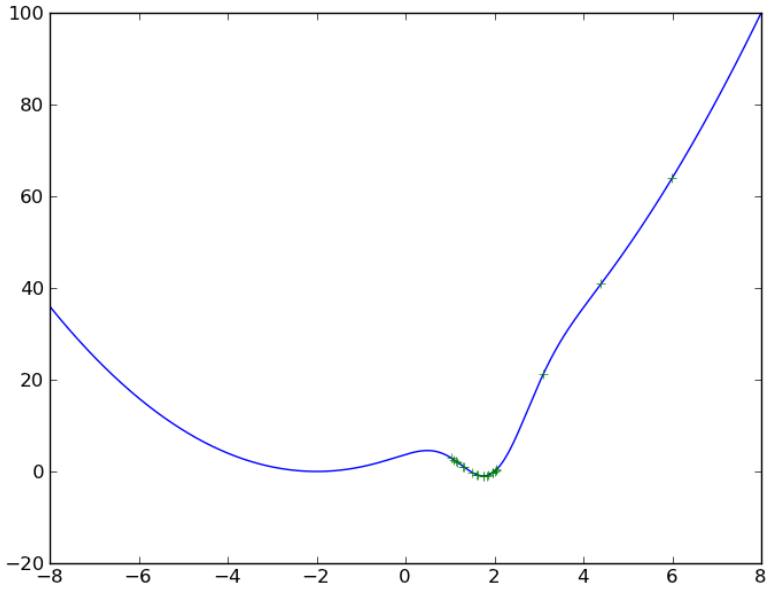

Figure 7: Example of running gradient descent starting on point $x_0 = 8$ for function $f(x) = (x+2)^2 - 16\exp\left(-(x-2)^2\right)$. The function is represented in blue, while the points of the minimizing sequence are displayed as green squares.

Figure 7 shows the resulting minimizing sequence. Note that now the algorithm converged to the global minimum. However, note that to get to the global minimum the sequence of points jumped from one side of the minimum to the other. This is a consequence of using a wrong step size (in this case too large).

Repeat the previous exercise changing both the values of the step-size and the precision. What do you observe.

During this school we will rely on the numerical optimization methods provided by Scipy (scientific computing library in python), which are very efficient implementations.