LxMLS - Lab Guide

June 11, 2013

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, and to Matplotlib and Numpy, two modules for plotting and scientific computing in Python, respectively. 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 Python Basics

Running Python code

We will start by creating and running a dummy program in Python which simply prints the "Hello World!" message to the standard output (this is usually the first program you code when learning a new programming language).

There are two main ways in which you can run code in Python:

From a file – Create a file named `yourfile.py` and write your program in it, using your favorite text editor:

```
print 'Hello World!'
```

After saving and closing the file, you can run your code by calling:

```
python yourfile.py
```

in the command line. This will run the program and display the message "Hello World!". After, the control will return to the command line.

In the interactive command line – Start the interactive command line in Python using the command `python`. After this, you can run Python code by simply writing it and pressing enter. In our lab sessions, we will use 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 on clusters of machines, but we will not make use of that functionality.

To run IPython, simply type `ipython` on your command line[1]. For interactive numeric use, the `--pylab` flag imports numpy and matplotlib (the two libraries we will extensively use in the lab sessions) for you and sets up interactive graphs:

```
IPython --pylab
```

[1]Note that in some systems, e.g. Linux, you may need to run the command lower-cased.

You can then run Python commands in the IPython command line

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

but you can also run Python code written into a file.

```
 In[]: run ./yourfile.py
Out[]: Hello, World!
```

Keep in mind that you can easily switch between these two modes. You can quickly test commands in the command line directly and e.g. inspect variables. Larger sections of code can be stored and run from files.

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 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 (?), check the website: `http://ipython.scipy.org/moin/`

Exiting

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

0.1.2 Python by Example

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 all the items between indexes i and $j-1$, inclusive.

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

Loops and Indentation

A loop allows a section of code to be repeated a certain number of times, until a stop condition is reached. For instance, when the list you are iterating has reached its end or when a variable has reached a certain value (in this case, you should not forget to update the value of that variable inside the code of the loop). In Python you have `while` and `for` loop statements. The following two example programs output exactly the same using both statements: the even numbers from 2 to 8.

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

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

You can copy and run this from the IPython command line. Alternatively you can write this into your `yourfile.py` file an run it as well. Notice something? It is possible that the code did not act as expected or maybe an error message popped up. This brings us to an important aspect of Python: **indentation**. Indentation is the number of blank spaces at the leftmost of each command. This is how Python differentiates between blocks of commands inside and outside a statement, e.g. `while`, `for` or other. All commands within a statement have the same number of blank spaces at their leftmost. For instance, consider the following code:

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

and its output:

```
1
2
3
```

Exercise 0.2 *Can you then predict the output of the following code?:*

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

Bear in mind that indentation is often the main source of errors when starting to work with Python. Try to get used to it as quickly as possible. It is also recommendable that you use a text editor that can display all characters e.g. blank space, tabs, since these characters can be visually similar but are considered different by Python. One of the most common mistakes by newcomers to Python is to have their files indented with spaces on some lines and with tabs on other lines. Visually it might appear that all lines have proper indentation, but you will get an `IndentationError` message if you try it.

Control Flow

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

```
hour = 16
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!'
```

You can write this command into IPython interactive command line directly or write them into a file and run the file in IPython. Once you do this the function will be available for you to use. Call the function `greet` with different hours of the day (for example, type `greet(16)`) and see that 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.
```

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)
```

The output of the `%prun` command will show the following information for each function that was called during the execution of your code:

- `ncalls`: The number of times this function was called. If this function was used recursively, the output will be two numbers; the first one counts the total function calls with recursions included, the second one excludes recursive calls.
- `tottime`: Total time spent in this function, excluding the time spent in other functions called from within this function.
- `percall`: Same as `tottime`, but divided by the number of calls.

- `cumtime`: Same as `tottime`, but including the time spent in other functions called from within this function.
- `percall`: Same as `cumtime`, but divided by the number of calls.
- `filename:lineno(function)`: Tells you where this function was defined.

Debugging in Python

During the lab sessions we will use the previously described IPython iterative command line which allows you to execute a script, command by command. This should limit the need for debugging tools. However, there will be situations in which we will use and extend modules that involve more elaborated code and statements, like classes and nested functions. Although desirable, it should not be necessary for you to fully understand the whole code to carry out the exercises. It will suffice to understand the algorithm as explained in the theoretical part of the class and the local context of the part of the code where we will be working.

The simplest way to do this is to run the code and stop the execution at a given point (called break-point) to get a quick glimpse of the variable structures and to inspect the execution flow of your program. For that, you can use the ipdb module.

In the following example, we use this module to inspect the `greet` function:

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

Load the new definition of the function into IPython by writting this code in a file and running it. Now, if you try `greet(50)` the code execution should stop at the place where you located the break-point (that is, in the `print 'Good evening!'` statement). You can now run new commands or inspect variables. For this purpose there are a number of commands you can use. The complete list can be found at `http://docs.python.org/library/pdb.html`, but we provide here a short table with the most useful:

(h)elp	Starts the help menu
(p)rint	Prints a variable
(p)retty(p)rint	Prints a variable, with line break (useful for lists)
(n)ext line	Jumps to next line
(s)tep	Jumps inside of the function we stopped at
c(ont(inue))	Continues execution until finding breakpoint or finishing
(r)eturn	Continues execution until current function returns
b(reak) n	Sets a breakpoint in in line n
l(ist) [n], [m]	Prints 11 lines around current line. Optionally starting in line n or between lines n, m
w(here)	Shows which function called the function we are in, and upwards (stack[2])
u(p)	Goes one level up the stack (frame of the function that called the function we are on)
d(down)	Goes one level down the stack
blank	Repeat the last command
expression	Executes the python expression as if it was in current frame

Table 1: Basic pdb/ipdb commands, parentheses indicates abbreviation

So getting back to our example, we can type n(ext) once to execute the line we stopped at

```
ipdb> n
> ./lxmls-toolkit/yourfile.py(8)greet()
      7                 import ipdb;ipdb.set_trace()
----> 8                 print 'Good evening!'
```

[2]Note that since we are inside the IPython command line, the IPython functions will also appear at the top.

Now we can inspect the variable `hour` using the p(retty)p(rint) option

```
ipdb> pp hour
50
```

From here we could keep advancing with the n(ext) option or set a b(reak) point and type c(ontinue) to jump to a new position. We could also execute any python expression which is valid in the current frame (the function we stopped at). This is particularly useful to find out why code crashes, as we can try different alternatives without the need to restart the code again.

0.1.3 Exceptions

Occasionaly, a syntactically correct code statement may produce an error when an attempt is made to execute it. These kind of errors are called *exceptions* in Python. For example, try executing the following:

```
10/0
```

A *ZeroDivisionError* exception was raised, and no output was returned. Exceptions can also be forced to occur by the programmer, with customized error messages (for a complete list of built-in exceptions, see `http://docs.python.org/2/library/exceptions.html`).

```
raise ValueError("Invalid input value.")
```

Exercise 0.4 *Rewrite the code in Exercise 0.3 in order to raise a ValueError exception when the hour is less than 0 or more than 24.*

Handling of exceptions is made with the *try* statement:

```
while True:
    try:
        x = int(raw_input("Please enter a number: ")
        break
    except ValueError:
        print "Oops! That was no valid number. Try again..."
```

It works by first executing the *try* clause. If no exception occurs, the *except* clause is skipped; if an exception does occur, and if its type matches the the exception named in the *except* keyword, the except clause is executed; otherwise, the exception is raised and execution is oborted (if it is not caught by outer *try* statements).

Extending basic Functionalities with Modules

In Python you can load new functionalities into the language by using the `import`, `from` and `as` keywords. For example we can load the numpy module as

```
import numpy as np
```

then we can run the following on the IPython command line

```
np.var?
np.random.normal?
```

The import will make the numpy tools available through the alias `np`. This shorter alias prevents the code from getting too long if we load lots of modules. The first command will display the help for the method `numpy.var` using the previously commented symbol `?`. Note that in order to display the help you need the full name of the function including the module name or alias. Modules have also submodules that can be accessed the same way, as shown in the second example.

0.1.4 Matplotlib – Plotting in Python

Matplotlib[3] 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 integers 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)
plt.show()
```

0.1.5 Numpy – Scientific Computing with Python

Numpy[4] is a library 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.

[3] http://matplotlib.org/

[4] http://www.numpy.org/

- M[i,j], to access the item in the i^{th} row and j^{th} column;
- M[i:j,:], to get the all the rows between the i^{th} and $j-1^{th}$;
- 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)

# These two lines do exactly the same thing
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. Note that integer division in Python 2.x returns the floor division (use floats – e.g. 5.0/2.0 – to obtain rationals). The exact value is 1/3. How close is the approximation?

0.2 Essential Linear Algebra

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

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

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

$$Ax = b$$

with

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

0.2.1 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 0.8 *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 array of type float64. By default, arrays contain 64 bit[5] 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
```

[5]On your computer, particularly if you have an older computer, `int` might denote 32 bits integers

You can also 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.

0.2.2 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 0.9 *You can multiply two matrices 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[k,j]
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**, is given by

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

```
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 $(xy^T)_{ij} = x_i y_j$,

$$xy^T \in \mathbb{R}^{m\times n} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \begin{bmatrix} y_1 & y_2 & \dots & y_n \end{bmatrix} = \begin{bmatrix} x_1y_1 & x_1y_2 & \dots & x_1y_n \\ x_2y_1 & x_2y_2 & \dots & x_2y_n \\ \vdots & \vdots & \ddots & \vdots \\ x_my_1 & x_my_2 & \dots & x_my_n \end{bmatrix}.$$

```
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 $A \in \mathbb{R}^{n\times n}$, $AI = A = IA$.

```
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, $(A^T)^T = A; \qquad (AB)^T = B^T A^T; \qquad (A+B)^T = A^T + B^T$

  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, $\mathrm{tr}(A) = \sum_{i=1}^{n} A_{ii}$

0.2.3 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 : $\|x\|_1 = \sum_{i=1}^{n} |x_i| \qquad \ell_\infty$ norm : $\|x\|_\infty = \max_i |x_i|$.

0.2.4 Linear Independence, Rank, and Orthogonal Matrices

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

$$x_j = \sum_{i \neq j} \alpha_i x_i$$

for some $j \in \{1, \ldots, n\}$ and some scalar values $\alpha_1, \ldots, \alpha_{i-1}, \alpha_{i+1}, \ldots, \alpha_n \in \mathbb{R}$.

- The **rank** of a matrix is the number of linearly independent columns, which is always equal to the number of linearly independent rows.
- For $A \in R^{m \times n}$, rank$(A) \leq$ min(m, n). If rank(A) =min(m, n), then A is said to be **full rank**.
- For $A \in R^{m \times n}$, rank(A)=rank(A^T).
- For $A \in R^{m \times n}$, $B \in R^{n \times p}$, rank$(AB) \leq$ min(rank(A),rank(B)).
- For $A, B \in R^{m \times n}$, rank$(A + B) \leq$ rank(A) + rank(B).
- Two vectors $x, y \in \mathbb{R}^n$ are **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

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

0.3 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, …, 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, …, 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, …, 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:

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

An important case is when events are **independent**, this is also a usual approximation which lends several practical advantages for the computation of the 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 $P(B) > 0$, the **conditional probability** of A given that 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 $A_1, \dots, A_k$ 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 Bayes' theorem.

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

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

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

Remark 0.2 *In Bayesian Statistical Inference, the Bayes' theorem is used to compute the estimates of distribution parameters from data. Here, 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.*

0.3.1 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 $F_X(x) = P(X \le x)$.*

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*

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

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, $\int\limits_{-\infty}^{\infty} f_X(x)dx = 1$ and for every $a \leq b$*

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

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

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

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:

0.3.2 Bernoulli

The **Bernoulli distribution** is a discrete probability distribution that takes value 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

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

0.3.3 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

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

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

0.3.4 Categorical

The **Categorical distribution** (often conflated with the Multinomial distribution, in fields like 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 $p_n = \sum_{i=1}^{n-1} p_i$.

0.3.5 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 $p_1, ..., p_n$ represent the occurrence probability of the respective outcome.

0.3.6 Gaussian Distribution

A very important theorem in probability theory is the **Central Limit Theorem**. 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:

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

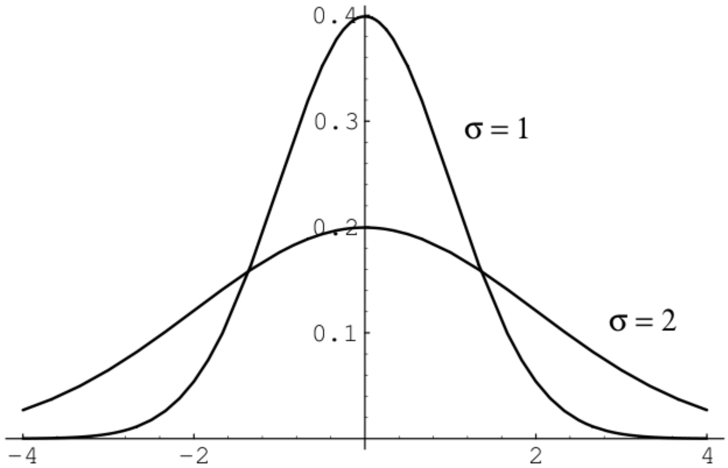


Figure 1: Normal density for two sets of parameter values.

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

0.3.7 Maximum Likelihood Estimation

Until now we assumed that, for every distribution, the parameters θ are known and are used when we calculate $p(x|\theta)$. There are some cases where the values of the parameters are easy to infer, such as the probability p of 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 or the structure of the coin to infer a parameter for p, a person could simply throw the coin n times, count the number of heads h and set $p = \frac{h}{n}$. By doing so, the person is using the data x to estimate θ.

With this in mind, we will now generalize this process by defining the probability $p(\theta|x)$ as the 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. The likelihood 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 independently and identically distributed (i.i.d.) 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 (see Section 0.3.2) 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 see that the likelihood is the 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 be the value for which $\mathcal{L}(p, x)$ is the 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 to

$$p_{mle} = 0.6$$

Exercise 0.10 *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. This dataset has 928 pairs of numbers.*

- *Use the* `load()` *function in the* `galton.py` *file to load the dataset. The file is located under the* `lxmls/readers` *folder. Type the following in your Python interpreter:*

```
import galton as galton
GaltonData = galton.load()
```

- *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.randn` *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?*

0.3.8 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 $\mathcal{F}$.*

0.4 Numerical optimization

Most problems in machine learning require minimization/maximization of functions (likelihoods, risk, energy, entropy, etc.,). Let x^* be the value of x which minimizes the value of some function $f(x)$. Mathematically, this is written as

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

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 it is too cumbersome (or impossible) to solve these equations analytically, and they must be tackled numerically. 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 (**???**).

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

0.4.1 Convex Functions

One important property of a function $f(x)$ is whether it is a **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 particularly useful since you can guarantee that the minimizing sequence converges to the true global minimum of the function, while in non-convex functions you can only guarantee that it will reach a local minimum.

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 benefit of a convex function. 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 (*i.e.*, lower) place than if you drop the ball from the right side. This is the main problem with non-convex functions: there are no guarantees about the quality of the local minimum you find.

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 $f : \mathbb{R}^n \to R$ 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 slope of the

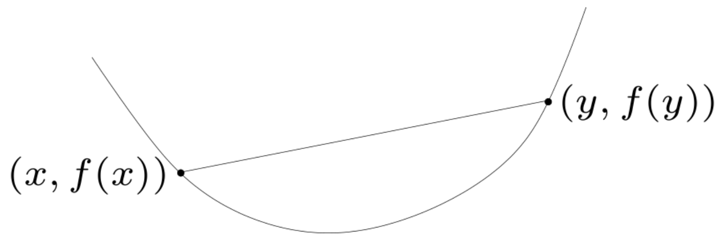


Figure 2: Illustration of a convex function. The line segment between any two points on the graph lies entirely above the curve.

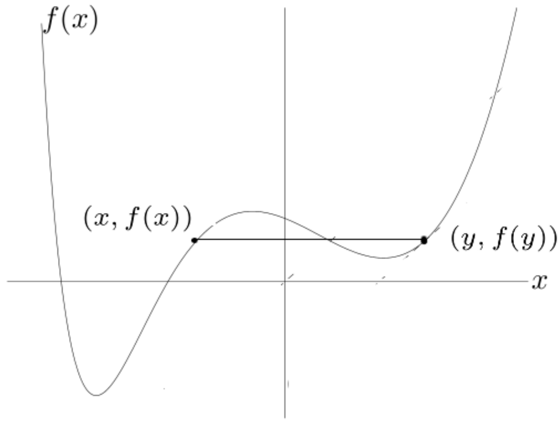


Figure 3: Illustration of a non-convex function. Note the line segment intersecting the curve.

function in that interval.

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

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

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 2: Some derivative examples

An important rule of derivation is the chain rule. 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 $h(x) = \exp(x^2)$, this can be decomposed as $h(x) = f(g(x)) = f(u) = \exp(u)$, where $u = g(x) = x^2$ and has derivative $\frac{\partial h}{\partial x} = \frac{\partial f}{\partial u} \cdot \frac{\partial u}{\partial x} = \exp(u) \cdot 2x = \exp(x^2) \cdot 2x$*

Exercise 0.11 *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))
```

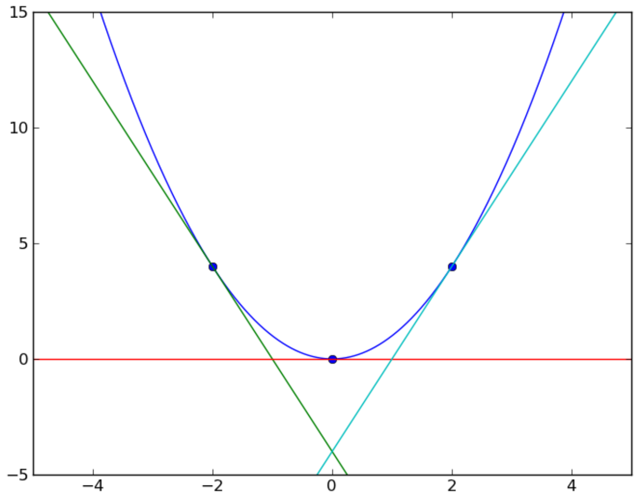


Figure 4: Illustration of the gradient of the function $f(x^2)$ at three different points $x = [-2, 0.2]$. Note that at point $x = 0$ the gradient is zero which corresponds to the minimum of the function.

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 x 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 used 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: $x_1 = x_0 - \eta \nabla_{x_0} f(x)$. Then, you compute the gradient at this new point, $\nabla_{x_1} f(x)$, and take a step of length η on the direction of the negative gradient to find a new point: $x_2 = x_1 - \eta \nabla_{x_1} f(x)$. You proceed 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 a 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** (**?**).

- **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) (**?**).

- **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, $\beta = -\nabla f(x)$. 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 $-\beta\nabla f(x_i)$.
5: $x_{i+1} \leftarrow x_i - \eta\beta\nabla f(x_i)$
6: $i \leftarrow i + 1$
7: **until** stopping criterion is satisfied.

Exercise 0.12 *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):
    return (x+2)**2 - 16*np.exp(-((x-2)**2))
```

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

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

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
```

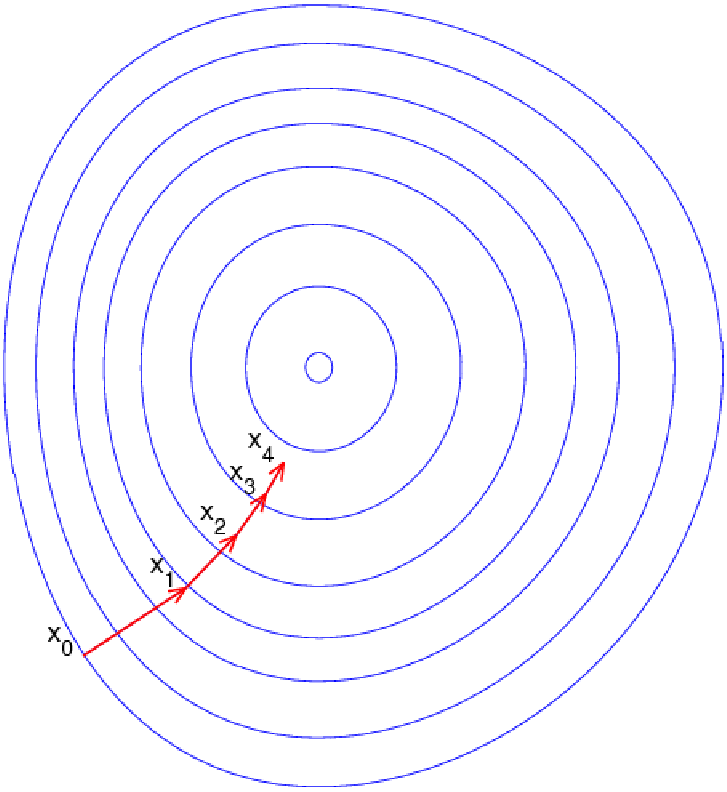


Figure 5: Illustration of gradient descent. The blue circles correspond to contours of the function (each blue circle is a set of points which have the same function value), while the red lines correspond to steps taken in the negative gradient direction.

```
    max_iter = 100
    x_new = start_x
    res = []
    for i in xrange(max_iter):
        x_old = x_new
        #Use beta egual 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 too 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)
plt.plot(res[:,0],res[:,1],'+')
plt.show()
```

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 $x_0 = 8$.

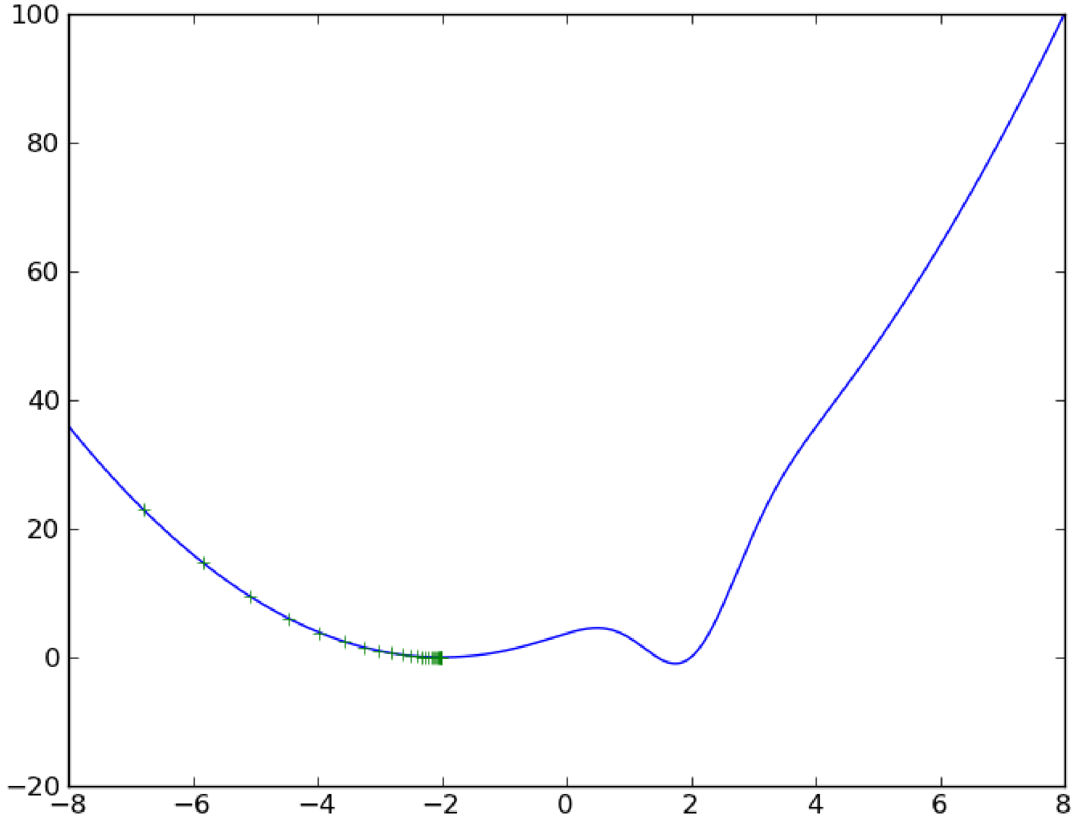


Figure 6: Example of running gradient descent starting on point $x_0 = -8$ for function $f(x) = (x+2)^2 - 16\exp\left(-(x-2)^2\right)$. The function is represented in blue, while the points of the minimizing sequence are displayed as green plus signs.

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

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.

0.5 Python Exercises

0.5.1 Numpy and Matplotlib

Exercise 0.13 1. *Consider the function* $f(x) = (x+2)^2 - 16\exp\left(-(x-2)^2\right)$. *Draw a plot around the* $x \in [-8, 8]$ *region.*

2. *What is* $\frac{\partial f}{\partial x}$*?*

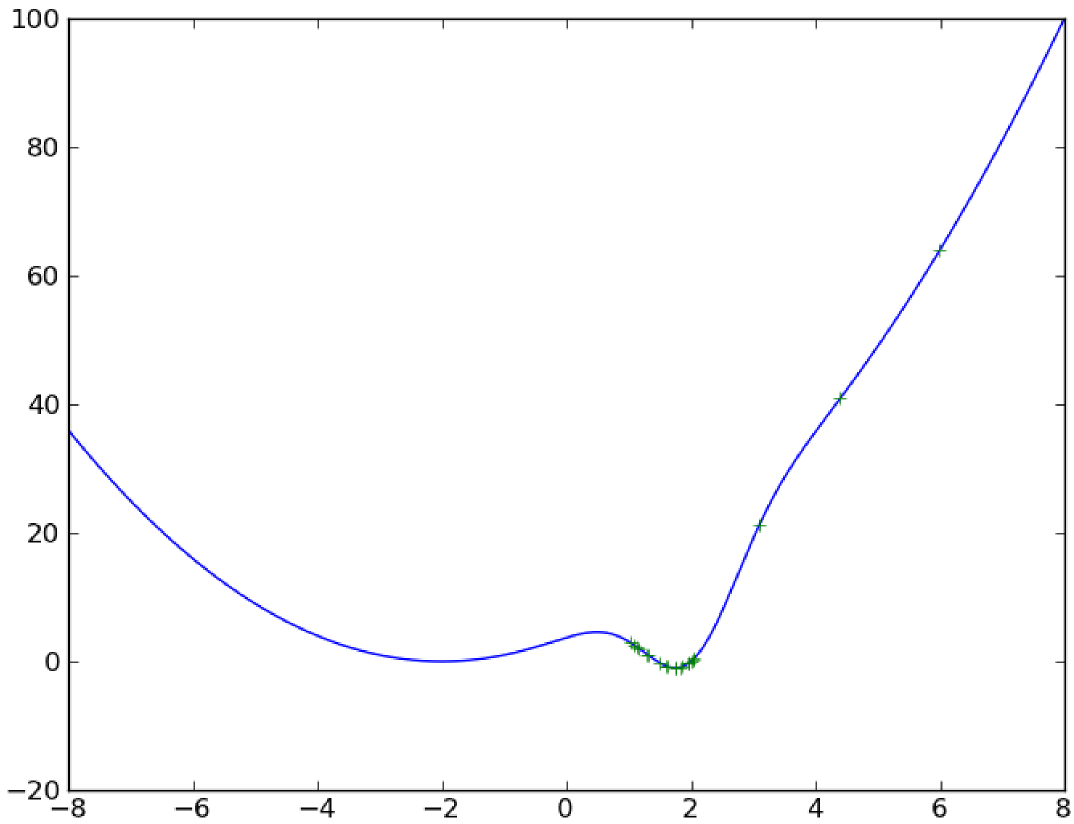


Figure 7: Example of running gradient descent starting on point $x_0 = 8$ for function $f(x) = (x+2)^2 - 16\exp\left(-(x-2)^2\right)$. The function is represented in blue, while the points of the minimizing sequence are displayed as green plus signs.

3. *Use gradient descent to find a local minimum starting from $x_0 = -4$ and $x_0 = +4$, with $\eta = .01$. Plot all of the intermediate estimates that you obtain in the same plot.*

```
import numpy as np
import matplotlib.pyplot as plt
X = np.linspace(-8, 8, 1000)
Y = (X+2)**2 - 16*np.exp(-((X-2)**2))

# derivative of the function f
def get_Y_dev(x):
    return (2*x+4)-16*(-2*x + 4)*np.exp(-((x-2)**2))

def grad_desc(start_x, eps, prec):
    '''
    runs the gradient descent algorithm and returns the list of estimates

    example of use grad_desc(X, 0.01, 0.00001)
    '''
    x_new = start_x
    x_old = start_x + prec * 2
    res = [x_new]
    while abs(x_old-x_new) > prec:
        x_old = x_new
        x_new = x_old - eps * get_Y_dev(x_new)
        res.append(x_new)
```

```python
    return np.array(res)
```

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.

Exercise 0.14 • *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?*

Exercise 0.15 *Consider the linear regression problem (ordinary least squares), with a single response variable*

$$y = x^T w + \varepsilon$$

The linear regression problem *is, given a set $\{y^{(i)}\}_i$ of samples of y and the corresponding $\boldsymbol{x}^{(i)}$ vectors, estimate $\boldsymbol{w}$ to minimise the sum of the ε variables. Traditionally this is solved analytically to obtain a closed form solution (although this is* ***not the way in which it should be computed****, linear algebra packages have an optimised solver, with numpy, use* `numpy.linalg.lstsq`*).*

Alternatively, we can define the error function for each possible $\boldsymbol{w}$:

$$e(\boldsymbol{w}) = \sum_i \left({\boldsymbol{x}^{(i)}}^T \boldsymbol{w} - y^{(i)} \right)^2 .$$

1. *Derive the gradient of the error $\frac{\partial e}{\partial w_j}$.*
2. *Implement a solver based on this for two dimensional problems (i.e., $\boldsymbol{w} \in R^2$).*
3. *Use this method on the Galton dataset from the previous exercise to estimate the relationship between father and son's height. Try two formulas*
$$s = f w_1 + \varepsilon, \tag{9}$$
where s is the son's height, and f is the father heights; and
$$s = f w_1 + 1 w_0 + \varepsilon \tag{10}$$
where the input variable is now two dimensional: $(f, 1)$. This allows the intercept to be non-zero.
4. *Plot the regression line you obtain with the points from the previous exercise.*
5. *Use the* `np.linalg.lstsq` *function and compare to your solution.*

0.5.2 Debugging

Exercise 0.16 *Use the debugger to debug the* `buggy.py` *script which attempts to repeatedly perform the following computation:*

1. *Start $x_0 = 0$*
2. *Iterate*
   (a) *$x'_{t+1} = x_t + r$, where r is a random variable.*
   (b) *if $x'_{t+1} >= 1.$, then stop.*

(c) *if* $x'_{t+1} <= 0.$, *then* $x_{t+1} = 0$

(d) *else* $x_{t+1} = x'_{t+1}$.

3. *Return the number of iterations.*

Having repeated this computation a number of times, the programme prints the average. Unfortunately, the program has a few bugs, which you need to fix.

Day 1

Classification

This day will serve as an introduction to machine learning. We recall some fundamental concepts about decision theory and classification. We also present some widely used models and algorithms and try to provide the main motivation behind them. There are several textbooks that provide a thorough description of some of the concepts introduced here: for example, **?**),**?**), **?**), **?**), **?**), **?**), to name just a few. The concepts that we introduce in this chapter will be revisited in later chapters, where the same algorithms and models will be adapted to structured inputs and outputs. For now, we concern only with multi-class classification (with just a few classes).

Today's assignment

The assignment of today's class is to implement a classifier called Naive Bayes, and use it to perform sentiment analysis on a corpus of book reviews from Amazon.

1.1 Pre-assignment

1.1.1 Notation

In what follows, we denote by $\mathcal{X}$ our *input set* (also called *observation set*), and by $\mathcal{Y}$ our *output set*. We will make no assumptions about the set $\mathcal{X}$, which can be continuous or discrete. In this lecture, we consider *classification* problems, where $\mathcal{Y} = \{c_1, \ldots, c_K\}$ is a finite set, consisting of K *classes* (also called *labels*). For example, $\mathcal{X}$ can be a set of documents in natural language, and $\mathcal{Y}$ a set of topics, the goal being to assign a topic to each document.

We use upper-case letters for denoting random variables, and lower-case letters for value assignments to those variables: for example,

- X is a random variable taking values on $\mathcal{X}$,
- Y is a random variable taking values on $\mathcal{Y}$,
- $x \in \mathcal{X}$ and $y \in \mathcal{Y}$ are particular values for X and Y.

We consider *events* such as $X = x$, $Y = y$, etc. Throughout, we use modified notation and let $P(y)$ denote the *probability* associated with the event $Y = y$ (instead of writing $P_Y(Y = y)$). *Joint* and *conditional* probabilities are denoted respectively as $P(x, y) \triangleq P_{X,Y}(X = x \wedge Y = y)$ and $P(x|y) \triangleq P_{X|Y}(X = x \mid Y = y)$. From the laws of probabilities:

$$P(x, y) = P(y|x)P(x) = P(x|y)P(y), \tag{1.1}$$

for all $x \in \mathcal{X}$ and $y \in \mathcal{Y}$.

Quantities that are predicted or estimated from the data will be appended a hat-symbol: for example, estimations of the probabilities above are denoted as $\hat{P}(y)$, $\hat{P}(x, y)$ and $\hat{P}(y|x)$; and a prediction of an output will be denoted $\hat{y}$.

We assume that a *training dataset* $\mathcal{D}$ is provided which consists of M input-output pairs (called *examples* or *instances*):

$$\mathcal{D} = \{(x^1, y^1), \ldots, (x^M, y^M)\} \subseteq \mathcal{X} \times \mathcal{Y}. \tag{1.2}$$

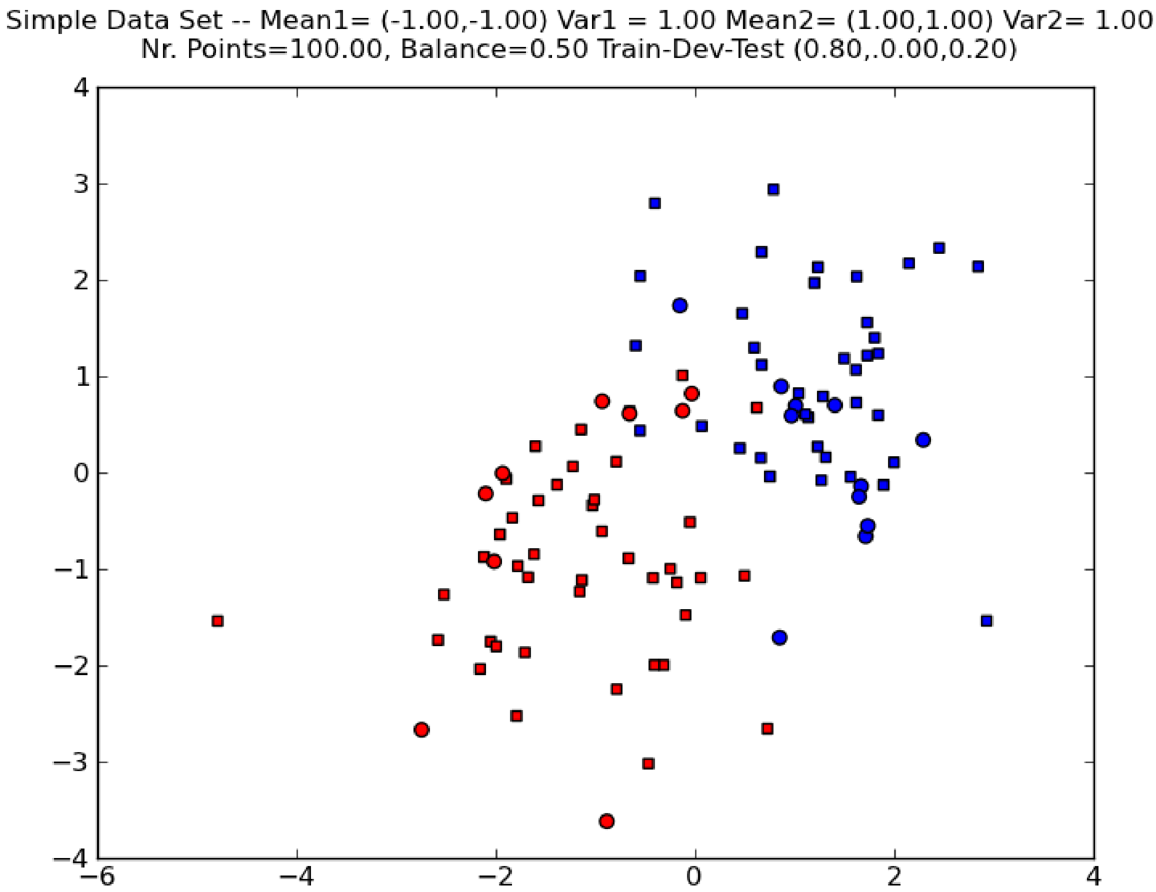


Figure 1.1: Example of a dataset. The input set consists in points in the real plane, $\mathcal{X} = \mathbb{R}^2$, and the output set consists of two classes (Red and Blue). Training points are represented as squares, while test points are represented as circles.

The goal of (supervised) machine learning is to use the training dataset $\mathcal{D}$ to learn a function h (called a *classifier*) that maps from $\mathcal{X}$ to $\mathcal{Y}$: this way, given a new instance $x \in \mathcal{X}$ (test example), the machine makes a prediction $\hat{y}$ by evaluating h on x, i.e., $\hat{y} = h(x)$.

1.1.2 Generative Classifiers: Naïve Bayes

If we knew the *true* distribution $P(X, Y)$, the best possible classifier (Bayes optimal) would be one which predicts according to

$$\begin{aligned}
\hat{y} &= \arg\max_{y \in \mathcal{Y}} P(y|x) = \arg\max_{y \in \mathcal{Y}} \frac{P(x,y)}{P(x)} \\
&=^{\dagger} \arg\max_{y \in \mathcal{Y}} P(x,y) \\
&= \arg\max_{y \in \mathcal{Y}} P(y)P(x|y), \qquad (1.3)
\end{aligned}$$

where in † we used the fact that $P(x)$ is constant with respect to y. Generative classifiers try to estimate the probability distributions $P(Y)$ and $P(X|Y)$ (which are respectively called the *class prior* and the *class conditionals*).

Figure 1.2 shows an example of the Bayes optimal decision boundary for a toy example with $K = 2$ classes, $M = 100$ points, class priors $P(y_1) = P(y_2) = 0.5$, and class conditionals $P(x|y_i)$ given by 2-D Gaussian distributions with the same variance but different means.

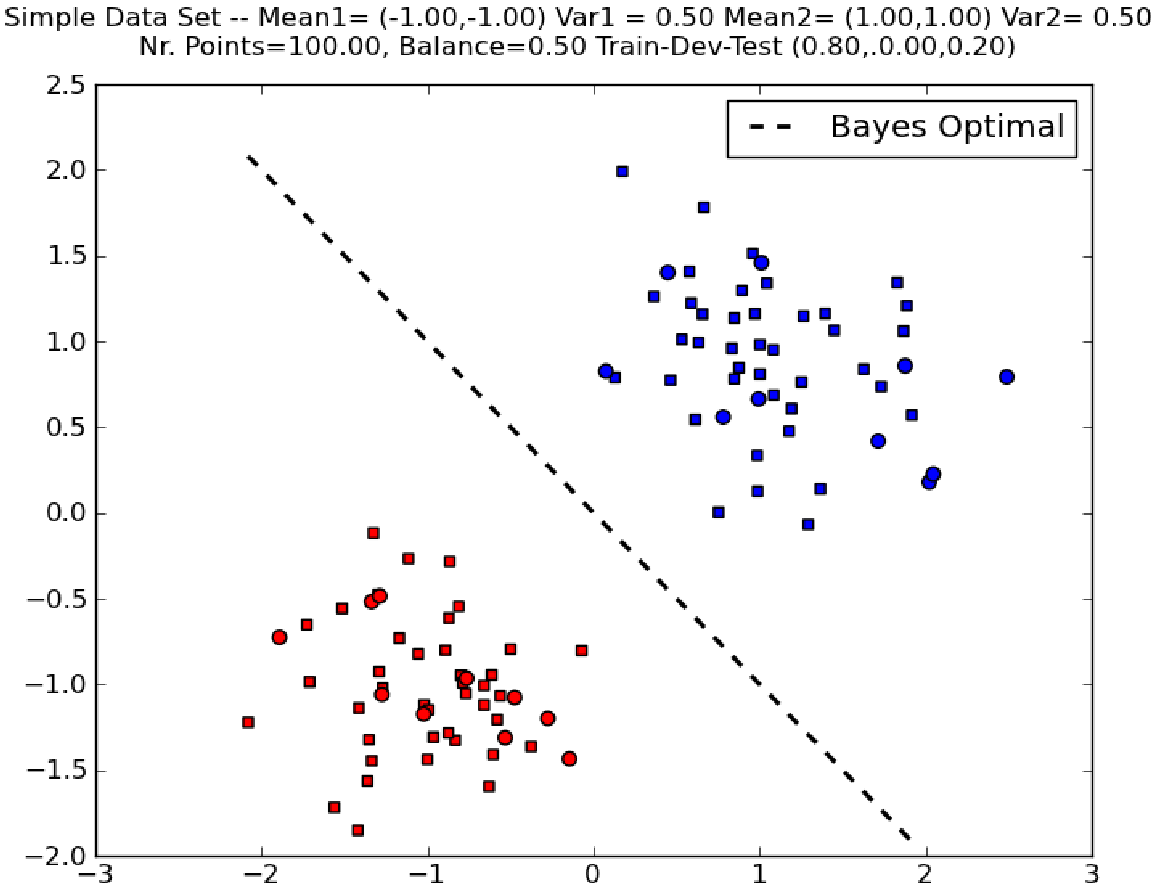


Figure 1.2: Example of a dataset together with the corresponding Bayes optimal decision boundary. The input set consists in points in the real plane, $\mathcal{X} = \mathcal{R}$, and the output set consists of two classes (Red and Blue). Training points are represented as squares, while test points are represented as circles.

Generative models assume that the data are generated according to the following generative story (independently for each $m = 1, \ldots, M$):

1. A class $y_m \sim P(Y)$ is drawn from the class prior distribution;
2. An input $x_m \sim P(X|Y = y_m)$ is drawn from the corresponding class conditional.

Training a generative model amounts to *estimating* these probabilities using the dataset $\mathcal{D}$, yielding estimates $\hat{P}(y)$ and $\hat{P}(x|y)$. This estimation is usually called *training*, or *learning*.

After we are done training, we are given a new input $x \in \mathcal{X}$, and we want to make a prediction according to

$$\hat{y} \quad = \quad \arg\max_{y \in \mathcal{Y}} \hat{P}(y)\hat{P}(x|y), \tag{1.4}$$

using the probabilities estimated in the training stage. This is usually called *inference* or *decoding*.

We are left with two important problems:

1. How should the distributions $\hat{P}(Y)$ and $\hat{P}(X|Y)$ be "defined"? (i.e., what kind of independence assumptions should they state, or how should they factor?)
2. How should parameters be estimated from the training data $\mathcal{D}$?

The first problem strongly depends on the application at hand. Quite often, there is a natural decomposition of the input variable X into J components,

$$X = (X_1, \ldots, X_J). \tag{1.5}$$

The naïve Bayes method makes the following assumption: $X_1, \ldots, X_J$ *are conditionally independent given the class*. Mathematically, this means that

$$P(X|Y) \quad = \quad \prod_{j=1}^{J} P(X_j|Y). \tag{1.6}$$

Note that this independence assumption greatly reduces the number of parameters to be estimated (degrees of freedom) from $O(\exp(J))$ to $O(J)$, hence estimation of $\hat{P}(X|Y)$ becomes much simpler, as we shall see. It also makes the overall computation much more efficient for large J and it decreases the risk of overfitting the data. On the other hand, if the assumption is over-simplistic it may increase the risk of under-fitting.

For the second problem, one of the simplest ways to solve it is using *maximum likelihood estimation*, which aims to maximize the probability of the training sample, assuming that each point was generated independently. This probability (call it $P(\mathcal{D})$) factorizes as

$$\begin{aligned} P(\mathcal{D}) &= \prod_{m=1}^{M} P(x^m, y^m) \\ &= \prod_{m=1}^{M} P(y^m) \prod_{j=1}^{J} P(x_j^m | y^m). \end{aligned} \tag{1.7}$$

1.1.3 Example: Multinomial Naïve Bayes for Document Classification

We now consider a more realistic scenario where the naïve Bayes classifier may be applied. Suppose that the task is *document classification*: $\mathcal{X}$ is the set of all possible documents, and $\mathcal{Y} = \{y_1, \ldots, y_K\}$ is a set of classes for those documents. Let $\mathcal{V} = \{w_1, \ldots, w_J\}$ be the vocabulary, i.e., the set of words that occur in some document.

A very popular document representation is through a "bag-of-words": each document is seen as a collection of words along with their frequencies; word ordering is ignored. We are going to see that this is equivalent to a naïve Bayes assumption with the *multinomial model*.We associate to each class a multinomial distribution, which ignores word ordering, but takes into consideration the frequency with which each word appears in a document. For simplicity, we assume that all documents have the same length L.[1] Each document x is assumed to have been generated as follows. First, a class y is generated according to $P(y)$. Then, x is generated by sequentially picking words from $\mathcal{V}$ with replacement. Each word w_j is picked with probability $P(w_j|y)$. For example, the probability of generating a document $x = w_{j_1} \ldots w_{j_L}$ (*i.e.*, a sequence of L words $w_{j_1}, \ldots, w_{j_L}$) is

$$P(x|y) = \prod_{l=1}^{L} P(w_{j_l}|y) = \prod_{j=1}^{J} P(w_j|y)^{n_j(x)}, \tag{1.8}$$

where $n_j(x)$ is the number of occurrences of word w_j in document x.

Hence, the assumption is that word occurrences (*tokens*) are independent given the class. The parameters that need to be estimated are $\hat{P}(y_1), \ldots, \hat{P}(y_K)$, and $\hat{P}(w_j|y_k)$ for $j = 1, \ldots, J$ and $k = 1, \ldots, K$. Given a training sample $\mathcal{D} = \{(x^1, y^1), \ldots, (x^M, y^M)\}$, denote by $\mathcal{I}_k$ the indices of those instances belonging to the kth class. The maximum likelihood estimates of the quantities above are:

$$\hat{P}(y_k) = \frac{|\mathcal{I}_k|}{M}, \qquad \hat{P}(w_j|y_k) = \frac{\sum_{m \in \mathcal{I}_k} n_j(x^m)}{\sum_{i=1}^{J} \sum_{m \in \mathcal{I}_k} n_i(x^m)}. \tag{1.9}$$

In words: the class priors' estimates are their relative frequencies (as before), and the class-conditional word probabilities are the relative frequencies of those words across documents with that class.

1.2 Assignment

Exercise 1.1 *In this exercise we will use the the Amazon sentiment analysis data (**?**), where the goal is to classify text documents as expressing a* positive *or* negative *sentiment (i.e., a classification problem with two labels). We are going to focus on book reviews. To load the data, type:*

[1] We can get rid of this assumption by defining a distribution on the document length. Everything stays the same if that distribution is uniform up to a maximum document length.

```python
import sentiment_reader as srs
import naive_bayes as nb

scr = srs.SentimentCorpus("books")
```

This will load the data in a bag-of-words representation where rare words (occurring less than 5 *times in the training data) are removed.*

1. *Open the file* `multinomial_naive_bayes.py`*. Inside the* `MultinomialNaiveBayes` *class you will find the* `train` *method. We have already placed some code in that file to help you get started.*

2. *Run naïve Bayes with the multinomial model on the Amazon dataset (sentiment classification) and report results both for training and testing:*

```python
import multinomial_naive_bayes as mnbb

mnb = mnbb.MultinomialNaiveBayes()
params_nb_sc = mnb.train(scr.train_X,scr.train_y)
y_pred_train = mnb.test(scr.train_X,params_nb_sc)
acc_train = mnb.evaluate(scr.train_y, y_pred_train)
y_pred_test = mnb.test(scr.test_X,params_nb_sc)
acc_test = mnb.evaluate(scr.test_y, y_pred_test)
print "Multinomial Naive Bayes Amazon Sentiment Accuracy train: %f test: %f"%(
    acc_train,acc_test)
```

3. *Observe that words that were not observed at training time cause problems at test time. Why? To solve this problem, apply a simple* add-one *smoothing technique: replace the expression in Eq. 1.9 for the estimation of the conditional probabilities by*

$$\hat{P}(w_j|c_k) = \frac{1 + \sum_{m \in \mathcal{I}_k} n_j(x^m)}{J + \sum_{i=1}^{J} \sum_{m \in \mathcal{I}_k} n_i(x^m)}.$$

   where J is the number of distinct words.

   This is a widely used smoothing strategy which has a Bayesian interpretation: it corresponds to choosing a uniform prior for the word distribution on both classes, and to replace the maximum likelihood criterion by a maximum a posteriori *approach. This is a form of* regularization*, preventing the model from* overfitting *on the training data. See* e.g. **??**) *for more information. Report the new accuracies.*

1.3 Post-assignment

1.3.1 Features and Discriminative Classifiers

In the previous sections we discussed generative classifiers. Those classifiers require us to model the class prior and class conditional distributions. Recall, however, that a classifier is *any* function which maps objects $x \in \mathcal{X}$ onto classes $y \in \mathcal{Y}$. While it's often useful to model how the data was generated, it's not required. Classifiers which do not model these distributions are called *discriminative* classifiers.

For the purpose of understanding discriminative classifiers, it is useful to think about each $x \in \mathcal{X}$ as an abstract object which is subject to a set of descriptions or measurements, which are called *features*. A feature is simply a real number that describes the value of some property of x. For example, in the previous section, the features of a document were the number of times each word w_j appeared in it.

Let $g_1(x), \ldots, g_J(x)$ be J features of x. We call the vector

$$\boldsymbol{g}(x) = (g_1(x), \ldots, g_J(x)) \tag{1.10}$$

a *feature vector representation* of x. The map $\boldsymbol{g} : \mathcal{X} \to \mathbb{R}^J$ is called a *feature mapping*.

In NLP applications, features are often binary-valued and result from evaluating propositions such as:

$$g_1(x) \triangleq \begin{cases} 1, & \text{if sentence } x \text{ contains the word } \textit{Ronaldo} \\ 0, & \text{otherwise.} \end{cases} \tag{1.11}$$

$$g_2(x) \triangleq \begin{cases} 1, & \text{if all words in sentence } x \text{ are capitalized} \\ 0, & \text{otherwise.} \end{cases} \tag{1.12}$$

$$g_3(x) \triangleq \begin{cases} 1, & \text{if } x \text{ contains any of the words } \textit{amazing}, \textit{excellent} \text{ or } \textit{:-)} \\ 0, & \text{otherwise.} \end{cases} \tag{1.13}$$

In this example, the feature vector representation of the sentence "Ronaldo shoots and scores an amazing goal!" would be $\boldsymbol{g}(x) = (1, 0, 1)$.

In multi-class learning problems, rather than associating features only with the input objects, it is useful to consider *joint feature mappings* $\boldsymbol{f} : \mathcal{X} \times \mathcal{Y} \to \mathbb{R}^D$. In that case, the *joint feature vector* $\boldsymbol{f}(x, y)$ can be seen as a collection of joint input-output measurements. For example:

$$f_1(x,y) \triangleq \begin{cases} 1, & \text{if } x \text{ contains } \textit{Ronaldo}, \text{ and topic } y \text{ is } \texttt{sport} \\ 0, & \text{otherwise.} \end{cases} \tag{1.14}$$

$$f_2(x,y) \triangleq \begin{cases} 1, & \text{if } x \text{ contains } \textit{Ronaldo}, \text{ and topic } y \text{ is } \texttt{politics} \\ 0, & \text{otherwise.} \end{cases} \tag{1.15}$$

A very simple form of defining a joint feature mapping which is often employed is via:

$$\begin{aligned} \boldsymbol{f}(x,y) &\triangleq \boldsymbol{g}(x) \otimes \boldsymbol{e}_y \\ &= (0, \ldots, 0, \underbrace{\boldsymbol{g}(x)}_{y\text{th slot}}, 0, \ldots, 0) \end{aligned} \tag{1.16}$$

where $\boldsymbol{g}(x) \in \mathbb{R}^J$ is a input feature vector, $\otimes$ is the Kronecker product ($[\boldsymbol{a} \otimes \boldsymbol{b}]_{ij} = a_i b_j$) and $\boldsymbol{e}_y \in \mathbb{R}^K$, with $[\boldsymbol{e}_y]_c = 1$ iff $y = c$, and 0 otherwise. Hence $\boldsymbol{f}(x,y) \in \mathbb{R}^D$ with $D = JK$. NOTA-MA: De acordo com a defenicao dada de produto kronecker, o vector $f(x,y)$ dado em (1.16) devia ser 2D e $\boldsymbol{f}(x,y) \in \mathbb{R}^{J \times K}$.

Linear classifiers are very popular in natural language processing applications. They make their decision based on the rule:

$$\hat{y} = \arg\max_{y \in \mathcal{Y}} \boldsymbol{w} \cdot \boldsymbol{f}(x,y). \tag{1.17}$$

where

- $\boldsymbol{w} \in \mathbb{R}^D$ is a *weight vector*;
- $\boldsymbol{f}(x,y) \in \mathbb{R}^D$ is a *feature vector*;
- $\boldsymbol{w} \cdot \boldsymbol{f}(x,y) = \sum_{d=1}^{D} w_d f_d(x,y)$ is the inner product between $\boldsymbol{w}$ and $\boldsymbol{f}(x,y)$.

Hence, each feature $f_d(x,y)$ has a weight w_d and, for each class $y \in \mathcal{Y}$, a score is computed by linearly combining all the weighted features. All these scores are compared, and a prediction is made by choosing the class with the largest score.

Remark 1.1 *With the design above (Eq. 1.16), and decomposing the weight vector as $\boldsymbol{w} = (\boldsymbol{w}_{c_1}, \ldots, \boldsymbol{w}_{c_K})$, we have that*

$$\boldsymbol{w} \cdot \boldsymbol{f}(x,y) = \boldsymbol{w}_y \cdot \boldsymbol{g}(x). \tag{1.18}$$

In words: each class $y \in \mathcal{Y}$ gets its own weight vector $\boldsymbol{w}_y$, and one defines a input feature vector $\boldsymbol{g}(x)$ that only looks at the input $x \in \mathcal{X}$. This representation is very useful when features only depend on input x since it allows a more compact representation. Note that the number of features is normally very large.

Remark 1.2 *The multinomial naïve Bayes classifier described in the previous section is an instance of a linear classifier. Recall that the naïve Bayes classifier predicts according to $\hat{y} = \arg\max_{y \in \mathcal{Y}} \hat{P}(y)\hat{P}(x|y)$. Taking logs, in the multinomial*

Algorithm 2 Averaged perceptron

input: dataset $\mathcal{D}$, number of rounds R
initialize $t = 0, \boldsymbol{w}^t = \mathbf{0}$
for $r = 1$ **to** R **do**
  $\mathcal{D}_s = \text{shuffle}(\mathcal{D})$
  for $i = 1$ **to** M **do**
    $m = \mathcal{D}_s(i)$
    $t = t + 1$
    take training pair (x^m, y^m) and predict using the current model:

$$\hat{y} \leftarrow \arg\max_{y' \in \mathcal{Y}} \boldsymbol{w}^t \cdot \boldsymbol{f}(x^m, y')$$

    update the model: $\boldsymbol{w}^{t+1} \leftarrow \boldsymbol{w}^t + \boldsymbol{f}(x^m, y^m) - \boldsymbol{f}(x^m, \hat{y})$
  end for
end for
output: the averaged model $\hat{\boldsymbol{w}} \leftarrow \frac{1}{t} \sum_{i=1}^{t} \boldsymbol{w}^i$

model for document classification this is equivalent to:

$$\begin{aligned}
\hat{y} &= \arg\max_{y \in \mathcal{Y}} \log \hat{P}(y) + \log \hat{P}(x|y) \\
&= \arg\max_{y \in \mathcal{Y}} \log \hat{P}(y) + \sum_{j=1}^{J} n_j(x) \log \hat{P}(w_j|y) \\
&= \arg\max_{y \in \mathcal{Y}} \boldsymbol{w}_y \cdot \boldsymbol{g}(x), \qquad (1.19)
\end{aligned}$$

where

$$\begin{aligned}
\boldsymbol{w}_y &= \big(b_y, \log \hat{P}(w_1|y), \ldots, \log \hat{P}(w_J|y)\big) \\
b_y &= \log \hat{P}(y) \\
\boldsymbol{g}(x) &= (1, n_1(x), \ldots, n_J(x)). \qquad (1.20)
\end{aligned}$$

Hence, the multinomial model yields a prediction rule of the form

$$\hat{y} = \arg\max_{y \in \mathcal{Y}} \boldsymbol{w}_y \cdot \boldsymbol{g}(x). \qquad (1.21)$$

1.3.2 Online Discriminative Algorithms: Perceptron and MIRA

We now discuss two discriminative classification algorithms. These two algorithms are called *online* (or *stochastic*) algorithms because they only process one data point (in our example, one document) at a time. Algorithms which look at the whole dataset at once are called *offline*, or *batch* algorithms, and will be discussed later.

Perceptron

Perhaps the oldest algorithm to train a linear classifier is the *perceptron* (**?**), which we depict as Alg. 2.[2] NOTA-MA: O preceptron algorithm consiste no metodo de gradiente quando a funo de custo e o erro quadratico, certo? Assim, pode ter varios passos (neste caso fixou-se o valor do passo em 1), e tambm tem uma verso bach.

The perceptron algorithm works as follows: at each round, it takes an element x from the data set, and uses the current model to make a prediction. If the prediction is correct, nothing happens. Otherwise, the model is corrected by adding the feature vector w.r.t. the correct output and subtracting the feature vector w.r.t. the predicted (wrong) output. Then, we proceed to the next round. Alg. 2 is remarkably simple; yet it often reaches a very good performance, often better than the Naïve Bayes model, and usually not much worse than maximum entropy models or SVMs (which will be described in the next section).

A weight vector $\boldsymbol{w}$ defines a *separating hyperplane* if it classifies all the training data correctly, *i.e.*, if $y^m = \arg\max_{y \in \mathcal{Y}} \boldsymbol{w} \cdot \boldsymbol{f}(x^m, y)$ hold for $m = 1, \ldots, M$. A dataset $\mathcal{D}$ is *separable* if such a weight vector exists (in general,

[2] Actually, we are showing a more robust variant of the perceptron, which averages the weight vector as a post-processing step.

w is not unique). A very important property of the perceptron algorithm is the following: if $\mathcal{D}$ is separable, then the number of mistakes made by the perceptron algorithm until it finds a separating hyperplane is *finite*. This means that if the data are separable, the perceptron will eventually find a separating hyperplane w.

There are other variants of the perceptron (e.g., with regularization) which we omit for brevity.

Exercise 1.2 *We provide an implementation of the perceptron algorithm in the class* `Perceptron` *(file* `perceptron.py`*).*

1. *Run the perceptron algorithm on the simple dataset previously generated and report its train and test set accuracy: NOTA-MA: Falta definir o sd ("'simple dataset"') anterirormente. Apenas se definiu o dataset da Amazon.*

```
import perceptron as percc

perc = percc.Perceptron()
params_perc_sd = perc.train(sd.train_X,sd.train_y)
y_pred_train = perc.test(sd.train_X,params_perc_sd)
acc_train = perc.evaluate(sd.train_y, y_pred_train)
y_pred_test = perc.test(sd.test_X,params_perc_sd)
acc_test = perc.evaluate(sd.test_y, y_pred_test)
print "Perceptron Simple Dataset Accuracy train: %f test: %f"%(acc_train,acc_test)
```

2. *Plot the decision boundary found:*

```
fig,axis = sd.add_line(fig,axis,params_perc_sd,"Perceptron","blue")
```

   Change the code to save the intermediate weight vectors, and plot them every five iterations. What do you observe?

3. *Run the perceptron algorithm on the Amazon dataset.*

Margin Infused Relaxed Algorithm (MIRA)

The MIRA algorithm (**??**) has achieved very good performance in NLP problems. Recall that the Perceptron takes an input pattern and, if its prediction is wrong, adds the quantity $[f(x^m, y^m) - f(x^m, \hat{y})]$ to the weight vector. MIRA changes this by adding $\eta^t[f(x^m, y^m) - f(x^m, \hat{y})]$ to the weight vector. The difference is the step size η^t, which depends on the iteration t.

There is a theoretical basis for this algorithm, which we now briefly explain. At each round t, MIRA updates the weight vector by solving the following optimization problem:

$$w^{t+1} \leftarrow \arg\min_{w,\xi} \quad \xi + \tfrac{\lambda}{2}\|w - w^t\|^2 \tag{1.22}$$
$$\text{s.t.} \quad w \cdot f(x^m, y^m) \geq w \cdot f(x^m, \hat{y}) + 1 - \xi \tag{1.23}$$
$$\xi \geq 0, \tag{1.24}$$

where $\hat{y} = \arg\max_{y' \in \mathcal{Y}} w^t \cdot f(x^m, y')$ is the prediction using the model with weight vector w^t. By inspecting Eq. 1.22 we see that MIRA attempts to achieve a tradeoff between *conservativeness* (penalizing large changes from the previous weight vector via the term $\frac{\lambda}{2}\|w - w^t\|^2$) and *correctness* (by requiring, through the constraints, that the new model w^{t+1} "separates" the true output from the prediction with a margin (although slack $\xi \geq 0$ is allowed).[3] Note that, if the prediction is correct ($\hat{y} = y^m$) the solution of the problem Eq. 1.22 leaves the weight vector unchanged ($w^{t+1} = w^t$). This quadratic programming problem has a closed form solution:[4]

$$w^{t+1} \leftarrow w^t + \eta^t(f(x^m, y^m) - f(x^m, \hat{y})),$$

with

$$\eta^t = \min\left\{\lambda^{-1}, \frac{w^t \cdot f(x^m, \hat{y}) - w^t \cdot f(x^m, y^m) + \rho(\hat{y}, y^m)}{\|f(x^m, y^m) - f(x^m, \hat{y})\|^2}\right\},$$

[3]The intuition for this large margin separation is the same for support vector machines, which will be discussed in §1.3.3.

[4]Note that the perceptron updates are identical, except that we always have $\eta_t = 1$.

Algorithm 3 MIRA

input: dataset $\mathcal{D}$, parameter λ, number of rounds R
initialize $t = 0, \boldsymbol{w}^t = \mathbf{0}$
for $r = 1$ **to** R **do**
  $\mathcal{D}_s = \text{shuffle}(\mathcal{D})$
  for $i = 1$ **to** M **do**
    $m = \mathcal{D}_s(i)$
    $t = t + 1$
    take training pair (x^m, y^m) and predict using the current model:

$$\hat{y} \leftarrow \arg\max_{y' \in \mathcal{Y}} \boldsymbol{w}^t \cdot \boldsymbol{f}(x^m, y')$$

    compute loss: $\ell^t = \boldsymbol{w}^t \cdot \boldsymbol{f}(x^m, \hat{y}) - \boldsymbol{w}^t \cdot \boldsymbol{f}(x^m, y^m) + \rho(\hat{y}, y^m)$
    compute stepsize: $\eta^t = \min \left\{ \lambda^{-1}, \frac{\ell^t}{\|\boldsymbol{f}(x^m,y^m) - \boldsymbol{f}(x^m,\hat{y})\|^2} \right\}$
    update the model: $\boldsymbol{w}^{t+1} \leftarrow \boldsymbol{w}^t + \eta^t (\boldsymbol{f}(x^m, y^m) - \boldsymbol{f}(x^m, \hat{y}))$
  end for
end for
output: the averaged model $\hat{\boldsymbol{w}} \leftarrow \frac{1}{t} \sum_{i=1}^{t} \boldsymbol{w}^i$

where $\rho : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}_+$ is a non-negative cost function, such that $\rho(\hat{y}, y)$ is the cost incurred by predicting $\hat{y}$ when the true output is y; we assume $\rho(y, y) = 0$ for all $y \in \mathcal{Y}$. For simplicity, we focus here on the 0/1-cost (but keep in mind that other cost functions are possible):

$$\rho(\hat{y}, y) = \begin{cases} 1 & \text{if } \hat{y} \neq y \\ 0 & \text{otherwise.} \end{cases} \tag{1.25}$$

MIRA is depicted in Alg. 3. For other variants of MIRA, see **?**).

Exercise 1.3 *Implement the MIRA algorithm (Hint: use the perceptron algorithm as a starting point and modify it as necessary). Do this by creating a file* `Mira.py` *and implement class* `Mira`*. Then, repeat the perceptron exercise now using MIRA, for several values of λ:*

```
import mira as mirac

mira = mirac.Mira()
mira.regularizer = 1.0 # This is lambda
params_mira_sd = mira.train(sd.train_X,sd.train_y)
y_pred_train = mira.test(sd.train_X,params_mira_sd)
acc_train = mira.evaluate(sd.train_y, y_pred_train)
y_pred_test = mira.test(sd.test_X,params_mira_sd)
acc_test = mira.evaluate(sd.test_y, y_pred_test)
print "Mira Simple Dataset Accuracy train: %f test: %f"%(acc_train,acc_test)
fig,axis = sd.add_line(fig,axis,params_mira_sd,"Mira","green")

params_mira_sc = mira.train(scr.train_X,scr.train_y)
y_pred_train = mira.test(scr.train_X,params_mira_sc)
acc_train = mira.evaluate(scr.train_y, y_pred_train)
y_pred_test = mira.test(scr.test_X,params_mira_sc)
acc_test = mira.evaluate(scr.test_y, y_pred_test)
print "Mira Amazon Sentiment Accuracy train: %f test: %f"%(acc_train,acc_test)
```

Compare the results achieved and separating hiperplanes found.

1.3.3 Batch Discriminative Classifiers: Maximum Entropy and Support Vector Machines

The algorithms described in the last section (perceptron and MIRA) are called *online* or *stochastic* algorithms, because they look at one data point at a time. We now describe two discriminative classifiers which look at all points at once; these are called *offline* or *batch* algorithms.

Maximum Entropy Classifiers

The notion of *entropy* in the context of Information Theory (**?**) is one of the most significant advances in mathematics in the twentieth century. The principle of *maximum entropy* (which appears under different names, such as "maximum mutual information" or "minimum Kullback-Leibler divergence") plays a fundamental role in many methods in statistics and machine learning (**?**). [5] The basic rationale is that choosing the model with the highest entropy (subject to constraints that depend on the observed data) corresponds to making the fewest possible assumptions regarding what was unobserved, making uncertainty about the model as large as possible.

For example, if we throw a die and want to estimate the probability of its outcomes, the distribution with the highest entropy would be the uniform distribution (each outcome having of probability a 1/6). Now suppose that we are only told that outcomes $\{1,2,3\}$ occurred 10 times in total, and $\{4,5,6\}$ occurred 30 times in total, then the principle of maximum entropy would lead us to estimate $P(1) = P(2) = P(3) = 1/12$ and $P(1) = P(2) = P(3) = 1/4$ (i.e., outcomes would be uniform within each of the two groups). For an introduction of maximum entropy models, along with pointers to the literature, see `http://www.cs.cmu.edu/~aberger/maxent.html`.

This example could be presented in a more formal way. Suppose that we want to use binary features to represent the outcome of the die throw. We use two features: $f_{123}(x,y) = 1$ if and only if $y \in \{1,2,3\}$, and $f_{456}(x,y) = 1$ if and only if $y \in \{4,5,6\}$. Our observations state that in 40 throws, we observed f_{123} 10 times (25%) and f_{456} 30 times (75%). The maximum entropy principle states that we want to find the parameters $\boldsymbol{w}$ of our model, and consequently the probability distribution $P_{\boldsymbol{w}}(Y|X)$, which makes f_{123} have an expected value of 0.25 and f_{456} have an expected value of 0.75. These constraints, $E[f_{123}] = 0.25$ and $E[f_{456}] = 0.75$, are known as *first moment matching constraints*.[6]

An important fundamental result, which we will not prove here, is that the maximum entropy distribution $P_{\boldsymbol{w}}(Y|X)$ under first moment matching constraints is a *log-linear model*. [7] It has the following parametric form:

$$P_{\boldsymbol{w}}(y|x) = \frac{\exp(\boldsymbol{w} \cdot \boldsymbol{f}(x,y))}{Z(\boldsymbol{w},x)} \tag{1.26}$$

The denominator in Eq. 1.26 is called the *partition function*:

$$Z(\boldsymbol{w},x) = \sum_{y' \in \mathcal{Y}} \exp(\boldsymbol{w} \cdot \boldsymbol{f}(x,y')). \tag{1.27}$$

An important property of the partition function is that the gradient of its logarithm equals the feature expectations:

$$\begin{aligned} \nabla_{\boldsymbol{w}} \log Z(\boldsymbol{w},x) &= E_{\boldsymbol{w}}[\boldsymbol{f}(x,Y)] \\ &= \sum_{y' \in \mathcal{Y}} P_{\boldsymbol{w}}(y'|x)\boldsymbol{f}(x,y'). \end{aligned} \tag{1.28}$$

The average conditional log-likelihood is:

$$\begin{aligned} \mathcal{L}(\boldsymbol{w};\mathcal{D}) &= \frac{1}{M} \log P_{\boldsymbol{w}}(y^1,\ldots,y^M|x^1,\ldots,x^M) \\ &= \frac{1}{M} \log \prod_{m=1}^{M} P_{\boldsymbol{w}}(y^m|x^m) \\ &= \frac{1}{M} \sum_{m=1}^{M} \log P_{\boldsymbol{w}}(y^m|x^m) \\ &= \frac{1}{M} \sum_{m=1}^{M} \left(\boldsymbol{w} \cdot \boldsymbol{f}(x^m,y^m) - \log Z(\boldsymbol{w},x^m)\right). \end{aligned} \tag{1.29}$$

We try to find the parameters $\boldsymbol{w}$ that maximize the log-likelihood $\mathcal{L}(\boldsymbol{w};\mathcal{D})$; to avoid overfitting, we add a

[5] For an excellent textbook on Information Theory, we recommend **?**).

[6] In general, these constraints mean that feature expectations under that distribution $\frac{1}{M}\sum_m E_{Y\sim P_{\boldsymbol{w}}}[\boldsymbol{f}(x_m,Y)]$ must match the observed relative frequencies $\frac{1}{M}\sum_m \boldsymbol{f}(x_m,y_m)$.

[7] Also called a a Boltzmann distribution, or an exponential family of distributions.

regularization term that penalizes values of $\boldsymbol{w}$ that have a high magnitude. The optimization problem becomes:

$$\begin{aligned}\hat{\boldsymbol{w}} &= \arg\max_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{w};\mathcal{D}) - \frac{\lambda}{2}\|\boldsymbol{w}\|^2 \\ &= \arg\min_{\boldsymbol{w}} -\mathcal{L}(\boldsymbol{w};\mathcal{D}) + \frac{\lambda}{2}\|\boldsymbol{w}\|^2. \qquad (1.30)\end{aligned}$$

Here we use the squared L_2-norm as the regularizer,[8] but other norms are possible. The scalar $\lambda \geq 0$ controls the amount of regularization. Unlike the naïve Bayes examples, this optimization problem does not have a closed form solution in general; hence we need to resort to numerical optimization (see section 0.4). Let $F_\lambda(\boldsymbol{w};\mathcal{D}) = -\mathcal{L}(\boldsymbol{w};\mathcal{D}) + \frac{\lambda}{2}\|\boldsymbol{w}\|^2$ be the objective function in Eq. 1.30. This function is convex, which implies that a local optimum of Eq. 1.30 is also a global optimum. $F_\lambda(\boldsymbol{w};\mathcal{D})$ is also differentiable: its gradient is

$$\begin{aligned}\nabla_{\boldsymbol{w}} F_\lambda(\boldsymbol{w};\mathcal{D}) &= \frac{1}{M}\sum_{m=1}^{M}(-\boldsymbol{f}(x^m,y^m) + \nabla_{\boldsymbol{w}} \log Z(\boldsymbol{w},x^m)) + \lambda \boldsymbol{w} \\ &= \frac{1}{M}\sum_{m=1}^{M}(-\boldsymbol{f}(x^m,y^m) + E_{\boldsymbol{w}}[\boldsymbol{f}(x^m,Y)]) + \lambda \boldsymbol{w}. \qquad (1.31)\end{aligned}$$

A batch gradient method to optimize Eq. 1.30 is shown in Alg. 4. Essentially, Alg. 4 iterates through the following updates until convergence:

$$\begin{aligned}\boldsymbol{w}^{t+1} &\leftarrow \boldsymbol{w}^t - \eta_t \nabla_{\boldsymbol{w}} F_\lambda(\boldsymbol{w}^t;\mathcal{D}) \\ &= (1-\lambda\eta_t)\boldsymbol{w}^t + \eta_t \frac{1}{M}\sum_{m=1}^{M}(\boldsymbol{f}(x^m,y^m) - E_{\boldsymbol{w}}[\boldsymbol{f}(x^m,Y)]). \qquad (1.32)\end{aligned}$$

Convergence is ensured for suitable stepsizes η_t. Monotonic decrease of the objective value can also be ensured if η_t is chosen with a suitable line search method, such as Armijo's rule (**?**). In practice, more sophisticated methods exist for optimizing Eq. 1.30, such as conjugate gradient or L-BFGS. The latter is an example of a quasi-Newton method, which only requires gradient information, but uses past gradients to try to construct second order (Hessian) approximations.

In large-scale problems (very large M) batch methods are slow. *Online* or *stochastic* optimization are attractive alternative methods. Stochastic gradient methods make "noisy" gradient updates by considering only a single instance at the time. The resulting algorithm is shown as Alg. 5. At each round t, an instance $m(t)$ is chosen, either randomly (stochastic variant) or by cycling through the dataset (online variant). The stepsize sequence must decrease with t: typically, $\eta_t = \eta_0 t^{-\alpha}$ for some $\eta_0 > 0$ and $\alpha \in [1,2]$, tuned in a development partition or with cross-validation.

Exercise 1.4 *We provide an implementation of the L-BFGS algorithm for training maximum entropy models in the class* `MaxEnt_batch`*, as well as an implementation of the SGD algorithm in the class* `MaxEnt_online`*. NOTA-MA: A sigla SGD (stochastic gradient descent?) no foi definida; no e percebe o que .*

1. *Train a maximum entropy model using L-BFGS on the Simple data set (try different values of λ). Compare the results with the previous methods. Plot the decision boundary.*

```
import max_ent_batch as mebc

me_lbfgs = mebc.MaxEnt_batch()
me_lbfgs.regularizer = 1.0
params_meb_sd = me_lbfgs.train(sd.train_X,sd.train_y)
y_pred_train = me_lbfgs.test(sd.train_X,params_meb_sd)
acc_train = me_lbfgs.evaluate(sd.train_y, y_pred_train)
y_pred_test = me_lbfgs.test(sd.test_X,params_meb_sd)
acc_test = me_lbfgs.evaluate(sd.test_y, y_pred_test)
print "Max-Ent batch Simple Dataset Accuracy train: %f test: %f"%(acc_train,acc_test
    )

fig,axis = sd.add_line(fig,axis,params_meb_sd,"Max-Ent-Batch","orange")
```

[8] In a Bayesian perspective, this corresponds to choosing independent Gaussian priors $p(w_d) \sim \mathcal{N}(0;1/\lambda^2)$ for each dimension of the weight vector.

Algorithm 4 Batch Gradient Descent for Maximum Entropy

1: **input:** $\mathcal{D}$, λ, number of rounds T,
learning rate sequence $(\eta_t)_{t=1,...,T}$
2: initialize $\boldsymbol{w}^1 = \boldsymbol{0}$
3: **for** $t = 1$ **to** T **do**
4: **for** $m = 1$ **to** M **do**
5: take training pair (x^m, y^m) and compute conditional probabilities using the current model, for each $y' \in \mathcal{Y}$:

$$P_{\boldsymbol{w}^t}(y'|x^m) = \frac{\exp(\boldsymbol{w}^t \cdot \boldsymbol{f}(x^m, y'))}{Z(\boldsymbol{w}, x)}$$

6: compute the feature vector expectation:

$$E_{\boldsymbol{w}}[\boldsymbol{f}(x^m, Y)] = \sum_{y' \in \mathcal{Y}} P_{\boldsymbol{w}^t}(y'|x^m)\boldsymbol{f}(x^m, y')$$

7: **end for**
8: choose the stepsize η_t using, *e.g.*, Armijo's rule
9: update the model:

$$\boldsymbol{w}^{t+1} \leftarrow (1 - \lambda\eta_t)\boldsymbol{w}^t + \eta_t M^{-1} \sum_{m=1}^{M} (\boldsymbol{f}(x^m, y^m) - E_{\boldsymbol{w}}[\boldsymbol{f}(x^m, Y)])$$

10: **end for**
11: **output:** $\hat{\boldsymbol{w}} \leftarrow \boldsymbol{w}^{T+1}$

2. *Train a maximum entropy model using L-BFGS, on the Amazon dataset (try different values of λ) and report training and test set accuracy. What do you observe?*

```
params_meb_sc = me_lbfgs.train(scr.train_X,scr.train_y)
y_pred_train = me_lbfgs.test(scr.train_X,params_meb_sc)
acc_train = me_lbfgs.evaluate(scr.train_y, y_pred_train)
y_pred_test = me_lbfgs.test(scr.test_X,params_meb_sc)
acc_test = me_lbfgs.evaluate(scr.test_y, y_pred_test)
print "Max-Ent Batch Amazon Sentiment Accuracy train: %f test: %f"%(acc_train,
    acc_test)
```

3. *Now, fix $\lambda = 1.0$ and train with SGD (you might try to adjust the initial step). Compare the objective values obtained during training with those obtained with L-BFGS. What do you observe?*

```
import max_ent_online as meoc

me_sgd = meoc.MaxEnt_online()
me_sgd.regularizer = 1.0
params_meo_sc = me_sgd.train(scr.train_X,scr.train_y)
y_pred_train = me_sgd.test(scr.train_X,params_meo_sc)
acc_train = me_sgd.evaluate(scr.train_y, y_pred_train)
y_pred_test = me_sgd.test(scr.test_X,params_meo_sc)
acc_test = me_sgd.evaluate(scr.test_y, y_pred_test)
print "Max-Ent Online Amazon Sentiment Accuracy train: %f test: %f"%(acc_train,
    acc_test)
```

Support Vector Machines

Support vector machines are also a discriminative approach, but they are not a probabilistic model at all. The basic idea is that, if the goal is to accurately predict outputs (according to some cost function), we should focus on that goal in the first place, rather than trying to estimate a probability distribution ($P(Y|X)$ or $P(X,Y)$),

Algorithm 5 SGD for Maximum Entropy

input: $\mathcal{D}$, λ, number of rounds T,
learning rate sequence $(\eta_t)_{t=1,\ldots,T}$
initialize $\boldsymbol{w}^1 = \mathbf{0}$
for $t = 1$ **to** T **do**
  choose $m = m(t)$ randomly
  take training pair (x^m, y^m) and compute conditional probabilities using the current model, for each $y' \in \mathcal{Y}$:

$$P_{\boldsymbol{w}^t}(y'|x^m) = \frac{\exp(\boldsymbol{w}^t \cdot \boldsymbol{f}(x^m, y'))}{Z(\boldsymbol{w}, x)}$$

  compute the feature vector expectation:

$$E_{\boldsymbol{w}}[\boldsymbol{f}(x^m, Y)] = \sum_{y' \in \mathcal{Y}} P_{\boldsymbol{w}^t}(y'|x^m)\boldsymbol{f}(x^m, y')$$

  update the model:

$$\boldsymbol{w}^{t+1} \leftarrow (1 - \lambda\eta_t)\boldsymbol{w}^t + \eta_t\left(\boldsymbol{f}(x^m, y^m) - E_{\boldsymbol{w}}[\boldsymbol{f}(x^m, Y)]\right)$$

end for
output: $\hat{\boldsymbol{w}} \leftarrow \boldsymbol{w}^{T+1}$

which is a more difficult problem. As **?**) puts it, "do not solve an estimation problem of interest by solving a more general (harder) problem as an intermediate step."

We next describe the *primal* problem associated with multi-class support vector machines (**?**), which is of primary interest in natural language processing. There is a significant amount of literature about Kernel Methods (**??**) mostly focused on the *dual* formulation. We will not discuss non-linear kernels or this dual formulation here.[9]

Consider $\rho(y', y)$ as a non-negative cost function, representing the cost of assigning a label y' when the correct label was y. For simplicity, we focus here on the $0/1$-cost defined by Equation 1.25 (but keep in mind that other cost functions are possible). The *hinge loss*[10] is the function

$$\ell(\boldsymbol{w}; x, y) = \max_{y' \in \mathcal{Y}} \left[\boldsymbol{w} \cdot \boldsymbol{f}(x, y') - \boldsymbol{w} \cdot \boldsymbol{f}(x, y) + \rho(y', y)\right]. \tag{1.33}$$

Note that the objective of Eq. 1.33 becomes zero when $y' = y$. Hence, we always have $\ell(\boldsymbol{w}; x, y) \geq 0$. Moreover, if ρ is the $0/1$ cost, we have $\ell(\boldsymbol{w}; x, y) = 0$ if and only if the weight vector is such that the model makes a correct prediction with a *margin* greater than 1: *i.e.*, $\boldsymbol{w} \cdot \boldsymbol{f}(x, y) \geq \boldsymbol{w} \cdot \boldsymbol{f}(x, y') + 1$ for all $y' \neq y$. Otherwise, a positive loss is incurred. The idea behind this formulation is that not only do we want to make a correct prediction, but we want to make a *confident* prediction; this is why we have a loss unless the prediction is correct with some margin.

Support vector machines (SVM) tackle the following optimization problem:

$$\hat{\boldsymbol{w}} \quad = \quad \arg\min_{\boldsymbol{w}} \sum_{m=1}^{M} \ell(\boldsymbol{w}; x^m, y^m) + \frac{\lambda}{2}\|\boldsymbol{w}\|^2, \tag{1.34}$$

where we also use the squared L_2-norm as the regularizer. For the $0/1$-cost, the problem in Eq. 1.34 is equivalent to:

$$\arg\min_{\boldsymbol{w}, \xi} \quad \textstyle\sum_{m=1}^{M} \xi_m + \frac{\lambda}{2}\|\boldsymbol{w}\|^2 \tag{1.35}$$

$$\text{s.t.} \quad \boldsymbol{w} \cdot \boldsymbol{f}(x^m, y^m) \geq \boldsymbol{w} \cdot \boldsymbol{f}(x^m, \tilde{y}^m) + 1 - \xi_m, \quad \forall m, \tilde{y}^m \in \mathcal{Y} \setminus \{y^m\}. \tag{1.36}$$

Geometrically, we are trying to choose the linear classifier that yields the largest possible separation margin, while we allow some violations, penalizing the amount of slack via extra variables $\xi_1, \ldots, \xi_m$. There is now a

[9] The main reason why we prefer to discuss the primal formulation with linear kernels is that the resulting algorithms run in linear time (or less), while known kernel-based methods are quadratic with respect to M. In large-scale problems (large M) the former are thus more appealing.

[10] The hinge loss for the $0/1$ cost is sometimes defined as $\ell(\boldsymbol{w}; x, y) = \max\{0, \max_{y' \neq y} \boldsymbol{w} \cdot \boldsymbol{f}(x, y') - \boldsymbol{w} \cdot \boldsymbol{f}(x, y) + 1\}$. Given our definition of $\rho(\hat{y}, y)$, note that the two definitons are equivalent.

Algorithm 6 Stochastic Subgradient Descent for SVMs

1: **input:** $\mathcal{D}$, λ, number of rounds T,
learning rate sequence $(\eta_t)_{t=1,\dots,T}$
2: initialize $\boldsymbol{w}^1 = \boldsymbol{0}$
3: **for** $t = 1$ **to** T **do**
4: choose $m = m(t)$ randomly
5: take training pair (x^m, y^m) and compute the "cost-augmented prediction" under the current model:

$$\tilde{y} = \arg\max_{y' \in \mathcal{Y}} \boldsymbol{w}^t \cdot \boldsymbol{f}(x^m, y') - \boldsymbol{w}^t \cdot \boldsymbol{f}(x^m, y^m) + \rho(y', y)$$

6: update the model:

$$\boldsymbol{w}^{t+1} \leftarrow (1 - \lambda\eta_t)\boldsymbol{w}^t + \eta_t \left(\boldsymbol{f}(x^m, y^m) - \boldsymbol{f}(x^m, \tilde{y})\right)$$

7: **end for**
8: **output:** $\hat{\boldsymbol{w}} \leftarrow \boldsymbol{w}^{T+1}$

trade-off: increasing the slack variables ξ_m makes it easier to satisfy the constraints, but it will also increase the value of the cost function.

Problem 1.34 does not have a closed form solution. Moreover, unlike maximum entropy models, the objective function in 1.34 is non-differentiable, hence smooth optimization is not possible. However, it is still convex, which ensures that any local optimum is the global optimum. Despite not being differentiable, we can still define a *subgradient* of the objective function (which generalizes the concept of gradient), which enables us to apply subgradient-based methods. A stochastic subgradient algorithm for solving Eq. 1.34 is illustrated as Alg. 6. The similarity with maximum entropy models (Alg. 5) is striking: the only difference is that, instead of computing the feature vector expectation using the current model, we compute the feature vector associated with the cost-augmented prediction using the current model.

A variant of this algorithm was proposed by **?**) under the name *Pegasos*, with excellent properties in large-scale settings. Other algorithms and software packages for training SVMs that have become popular are SVMLight (`http://svmlight.joachims.org`) and LIBSVM (`http://www.csie.ntu.edu.tw/~cjlin/libsvm/`), which allow non-linear kernels. These will generally be more suitable for smaller datasets, where high accuracy optimization can be obtained without much computational effort.

Remark 1.3 *Note the similarity between the stochastic (sub-)gradient algorithms (Algs. 5–6) and the online algorithms seen above (perceptron and MIRA).*

Exercise 1.5 *Implement the SVM primal algorithm (Hint: look at the models implemented earlier, you should only need to change a few lines of code). Do this by creating a file* `SVM.py` *and implement class* `SVM`*. Then, repeat the MaxEnt exercise now using SVMs, for several values of λ:*

```
import svm as svmc

svm = svmc.SVM()
svm.regularizer = 1.0 # This is lambda
params_svm_sd = svm.train(sd.train_X,sd.train_y)
y_pred_train = svm.test(sd.train_X,params_svm_sd)
acc_train = svm.evaluate(sd.train_y, y_pred_train)
y_pred_test = svm.test(sd.test_X,params_svm_sd)
acc_test = svm.evaluate(sd.test_y, y_pred_test)
print "SVM Online Simple Dataset Accuracy train: %f test: %f"%(acc_train,acc_test)

fig,axis = sd.add_line(fig,axis,params_svm_sd,"SVM","orange")

params_svm_sc = svm.train(scr.train_X,scr.train_y)
y_pred_train = svm.test(scr.train_X,params_svm_sc)
acc_train = svm.evaluate(scr.train_y, y_pred_train)
y_pred_test = svm.test(scr.test_X,params_svm_sc)
acc_test = svm.evaluate(scr.test_y, y_pred_test)
print "SVM Online Amazon Sentiment Accuracy train: %f test: %f"%(acc_train,acc_test)
```

Compare the results achieved and separating hiperplanes found.

	Naive Bayes	Perceptron	MIRA	MaxEnt	SVMs
Generative/Discriminative	G	D	D	D	D
Performance if true model not in the hipothesis class	Bad	Fair (may not converge)	Good	Good	Good
Performance if features overlap	Fair	Good	Good	Good	Good
Training	Closed Form	Easy	Easy	Fair	Fair
Hyperparameters to tune	1 (smoothing)	0	1	1	1

Table 1.1: Comparison among different models.

1.3.4 Comparison

Table 1.3.4 provides a high-level comparison among the different models discussed in this chapter.

Exercise 1.6 • *Using the simple data set run the different models varying some characteristics of the data, number of points, variance (hence separability), class balance. Use function* run_all_classifiers *in file* labs/run_all_classifiers.py *which receives a dataset and plots all decisions boundaries and accuracies. What can you say about the methods when the amount of data increases? What about when the classes become too unbalanced?*

1.3.5 Final remarks

Some implementations of the discussed algorithms are available on the Web:

- SVMLight: `http://svmlight.joachims.org`
- LIBSVM: `http://www.csie.ntu.edu.tw/~cjlin/libsvm/`
- Maximum Entropy: `http://homepages.inf.ed.ac.uk/lzhang10/maxent_toolkit.html`
- MALLET: `http://mallet.cs.umass.edu/`.

Day 2

Sequence Models

In this class, we relax the assumption that the data points are independently and identically distributed (i.i.d.) by moving to a scenario of *structured prediction*, where the inputs are assumed to have temporal or spacial dependencies. We start by considering sequential models, which correspond to a *chain structure*: for instance, the words in a sentence. In this lecture, we will use part-of-speech tagging as our example task.

We focus on the well known Hidden Markov Model (HMM) on section 2.2, where we describe how to estimate its parameters from labeled data 2.3. We will then move to how to find the most likely hidden sequence (decoding) given an observation sequence and a parameter set 2.4. This section will explain the required inference algorithms (Viterbi and Forward-Backward) for sequence models. These inference algorithms will be fundamental for the rest of this lecture, as well as for the next lecture on *discriminative* training of sequence models. Section 2.5 will describe the task of part-of-speech tagging, and how HMM are suitable for this task. Finally, Section **??** will address unsupervised learning of HMMs through the EM algorithm.

2.1 Notation

The problem setting is the following. We assume a finite set of *observation labels*, $\Sigma := \{w_1, \ldots, w_J\}$, and a finite set of *state labels*, $\Lambda := \{c_1, \ldots, c_K\}$. In part-of-speech tagging, Σ is a vocabulary of word types, and Λ is the set of part-of-speech tags (noun, verb, adjective, *etc.*) Let ε denote the empty string. We denote by $\Sigma^* := \{\varepsilon\} \cup \Sigma \cup \Sigma^2 \cup \ldots$ and $\Lambda^* := \{\varepsilon\} \cup \Lambda \cup \Lambda^2 \cup \ldots$ the Kleene closure of each of the two sets above. Elements of Σ^* and Λ^* are *strings of observations* and *strings of states*, respectively. Throughout this class, we assume our input set is $\mathcal{X} = \Sigma^*$, and our output set is $\mathcal{Y} = \Lambda^*$. In other words, our inputs are observation sequences, $x = x_1 x_2 \ldots x_N$, for some $N \in \mathbb{N}$, where each $x_i \in \Sigma$; given such a x, we seek the corresponding state sequence, $y = y_1 y_2 \ldots y_N$, where each $y_i \in \Lambda$. We will assume two scenarios in this lab:

1. *Supervised learning.* We will train models from a sample set of paired observation and state sequences, $\mathcal{D}_L := \{(x^1, y^1), \ldots, (x^M, y^M)\} \subseteq \mathcal{X} \times \mathcal{Y}$.
2. *Unsupervised learning.* We will also train models from the set of observations only, $\mathcal{D}_U := \{x^1, \ldots, x^M\} \subseteq \mathcal{X}$.

Our notation is summarized in Table 2.1.

2.2 Hidden Markov Models

Hidden Markov Models (HMMs) are one of the most common sequence probabilistic models, and have been applied to a wide variety of tasks. HMMs are particular instances of directed probabilistic graphical models (or Bayesian networks) which have a chain topology. In a Bayesian network, every random variable is represented as a node in a graph, and the edges in the graph are directed and represent probabilistic dependencies between the random variables. For an HMM, the random variables are divided into two sets, the *observed variables*, $X = X_1 \ldots X_N$, and the *hidden variables* $Y = Y_1 \ldots Y_N$. In the HMM terminology, the observed variables are called *observations*, and the hidden variables are called *states*. The states are generated according to a Markov process, in which the ith state Y_i depends only at the previous state Y_{i-1}. We call the probability distributions $P(Y_i|Y_{i-1})$ the *transition probabilities*. Two special states are the `start` symbol, which starts the sequence, and the `stop` symbol, which ends it. We call the distributions $P(Y_1|Y_0 = \texttt{start})$ the *initial probabilities*, and $P(Y_{N+1} =$

Notation	
$\mathcal{D}_L$	training set (including labeled data)
$\mathcal{D}_U$	training set (unlabeled data only)
M	number of training examples
$x = x_1 \ldots x_N$	observation sequence
$y = y_1 \ldots y_N$	state sequence
N	length of the sequence
x_i	observation at position i in the sequence
y_i	state at position i in the sequence
`start`, `stop`	start and stop symbols
Σ	observation set
J	number of distinct observation types
w_j	particular observation, $j \in \{1, \ldots, J\}$
Λ	state set
K	number of distinct states
c_k	particular state, $k \in \{1, \ldots, K\}$

Table 2.1: General notation used in this class

$\texttt{stop}|Y_N)$ the *final probabilities*.[1] In addition, states emit observation symbols according to *emission probabilities* $P(X_i|Y_i)$. In an HMM, it is assumed that all observations are independent given the states that generated them.

As you may find out in today's lab session, implementing the inference routines of the HMM can be challenging. We start with a small and very simple (also very unrealistic!) example. The idea is that you may compute the desired quantities by hand and check if your implementation yields the correct result.

Example 2.1 *Consider a person, which is only interested in four activities:*

- *walking in the park* (`walk`)*;*
- *shopping* (`shop`)*;*
- *cleaning his apartment* (`clean`)*;*
- *playing tennis* (`tennis`)*.*

The choice of what to do on a given day is determined exclusively by the weather at that day. The weather can be either `rainy` *or* `sunny`*. Now, suppose that we observe what the person did on a sequence of days; can we use that information to predict the weather each of those days? To tackle this problem, we assume that the weather behaves as a discrete Markov chain: the weather on a given day depends only on the previous day's weather. The entire system can be described as an hidden Markov model (HMM).*

Assume we are asked to predict what was the weather on two different sequences of days given the following observations: "`walk walk shop clean`*" and "*`clean walk tennis walk`*." This will be our test set.*

To train our model, we will be given access to three different sequences of days, containing both the activities and the weather on those days, namely: "`walk/rainy walk/sunny shop/sunny clean/sunny`*", "*`walk/rainy walk/rainy shop/rainy clean/sunny`*" and "*`walk/sunny shop/sunny shop/sunny clean/sunny.`*" This will be our training set.*

Figure 2.2 shows the HMM model for the first sequence of our simple example, already including the `start` *and* `stop` *symbols. The notation is summarized in Table 2.2.*

Exercise 2.1 *Load the simple sequence dataset. From the ipython command line (note: start ipython from the* `lxmls` *directory), create a simple sequence object and look at the training and test set.*

```
>>> import readers.simple_sequence as ssr
>>> simple = ssr.SimpleSequence()
>>> print simple.train
[walk/rainy walk/sunny shop/sunny clean/sunny , walk/rainy walk/rainy shop/rainy clean/
    sunny , walk/sunny shop/sunny shop/sunny clean/sunny ]
>>> print simple.test
```

[1]Note that the initial and final probabilities are asymmetric.

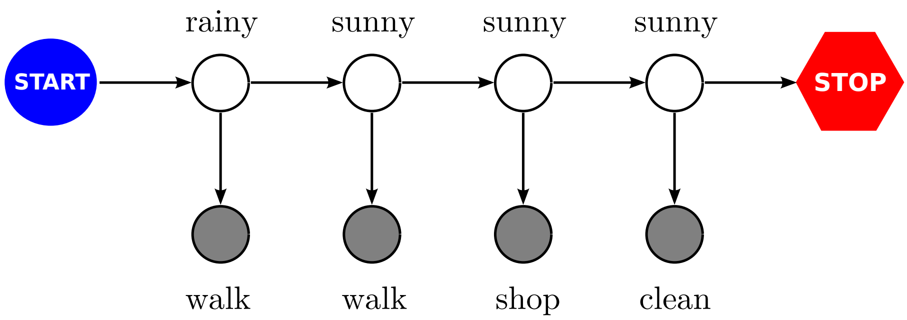


Figure 2.1: HMM structure, for the simple running example.

HMM Example	
x	observed sequence "`walk walk shop clean`"
$N = 4$	observation length
i	position in the sentence: $i \in \{1 \dots N\}$
$\Sigma = \{\texttt{walk}, \texttt{shop}, \texttt{clean}, \texttt{tennis}\}$	observation set
j	index into the observation set $j \in \{1, \dots, J\}$
$X_i = w_j$	observation at position i has value w_j
$\Lambda = \{\texttt{rainy}, \texttt{sunny}\}$	state set
k	index into state set $k \in \{1, \dots, K\}$
$Y_i = c_k, Y_{i-1} = c_l$	state at position i $(i-1)$ has value c_k (c_l)

Table 2.2: HMM notation for the running example.

```
[walk/rainy walk/sunny shop/sunny clean/sunny , clean/sunny walk/sunny tennis/sunny walk/
    sunny ]
```

A first order HMM model has the following independence assumptions over the joint distribution $P(X = x, Y = y)$:

- **Independence of previous states.** The probability of being in a given state at position i only depends on the state of the previous position $i-1$. Formally, $P(Y_i = y_i | Y_{i-1} = y_{i-1}, Y_{i-2} = y_{i-2}, \dots, Y_1 = y_1) = P(Y_i = y_i | Y_{i-1} = y_{i-1})$, defining a first order Markov chain.[2]
- **Homogeneous transition.** The probability of making a transition from state c_l to state c_k is independent of the particular position in the sentence: for all $i, t \in \{1, \dots, N\}$, $P(Y_i = c_k | Y_{i-1} = c_l) = P(Y_{i+t} = c_k | Y_{i+t-1} = c_l)$, so we can simply write $P(Y_i = c_k | Y_{i-1} = c_l) = P_{\text{trans}}(c_k | c_l)$.
- **Observation independence.** The probability of observing $X_i = x_i$ at position i is fully determined by the state Y_i at that position. Formally, $P(X_i = x_i | Y_1 = y_1, \dots, Y_i = y_i, \dots, Y_N = y_N) = P(X_i = x_i | Y_i = y_i)$, and this probability is independent of the particular position, so we can write $P(X_i = w_j | Y_i = c_k) = P(X_{i+t} = w_j | Y_{i+t} = c_k) = P_{\text{emiss}}(w_j | c_k)$ for every i and t.

These conditional independence assumptions are crucial to allow efficient inference, as will be described.

We also need to define *initial probabilities*, the probability of starting at each state, and *final probabilities*, the probability of ending the sequence given that we are at a particular state. The distributions that define the HMM model are summarized in Table 2.3. For each one of them we will use a short notation to simplify the exposition.

The joint distribution can be expressed as:

$$P(X_1 = x_1, \dots, X_N = x_N, Y_1 = y_1, \dots, Y_N = y_N) =$$
$$P_{\text{init}}(y_1 | \texttt{start}) \times \left(\prod_{i=1}^{N-1} P_{\text{trans}}(y_{i+1} | y_i) \right) \times P_{\text{final}}(\texttt{stop} | y_N) \times \prod_{i=1}^{N} P_{\text{emiss}}(x_i | y_i), \tag{2.1}$$

[2]The order of the Markov chain depends on the number of previous positions taken into account. The remainder of the exposition can be easily extended to higher order HMMs, giving the model more generality, but making inference more expensive.

HMM distributions			
Name	probability distribution	short notation	array size
initial probability	$P(Y_1 = c_k \| Y_0 = \texttt{start})$	$P_{\text{init}}(c_k \| \texttt{start})$	K
transition probability	$P(Y_i = c_k \| Y_{i-1} = c_l)$	$P_{\text{trans}}(c_k \| c_l)$	$K \times K$
final probability	$P(Y_{N+1} = \texttt{stop} \| Y_N = c_k)$	$P_{\text{final}}(\texttt{stop} \| c_k)$	K
emission probability	$P(X_i = w_j \| Y_i = c_k)$	$P_{\text{emiss}}(w_j \| c_k)$	$J \times K$

Table 2.3: HMM probability distributions.

which for the example from Figure 2.1 is:

$$\begin{aligned} &P(X_1 = x_1, \ldots, X_4 = x_4, Y_1 = y_1, \ldots, Y_4 = y_4) = \\ &\quad P_{\text{init}}(\texttt{rainy}|\texttt{start}) \times P_{\text{trans}}(\texttt{sunny}|\texttt{rainy}) \times P_{\text{trans}}(\texttt{sunny}|\texttt{sunny}) \times P_{\text{trans}}(\texttt{sunny}|\texttt{sunny}) \times \\ &\quad P_{\text{final}}(\texttt{stop}|\texttt{sunny}) \times P_{\text{emiss}}(\texttt{walk}|\texttt{rainy}) \times P_{\text{emiss}}(\texttt{walk}|\texttt{sunny}) \times P_{\text{emiss}}(\texttt{shop}|\texttt{sunny}) \\ &\quad \times P_{\text{emiss}}(\texttt{clean}|\texttt{sunny}). \end{aligned} \tag{2.2}$$

In the next section we turn our attention to estimating the different probability distributions of the model.

2.3 Finding the Maximum Likelihood Parameters

One important problem in HMMs is to estimate the model parameters, *i.e.*, the distributions depicted in Table 2.3. We will refer to the set of all these parameters as θ. In a supervised setting, the HMM model is trained to maximize the joint log-likelihood of the data. Given a dataset $\mathcal{D}_L$, the objective being optimized is:

$$\arg\max_{\theta} \sum_{m=1}^{M} \log P_{\theta}(X = x^m, Y = y^m), \tag{2.3}$$

where $P_{\theta}(X = x^m, Y = y^m)$ is given by Eq. 2.1.

In some applications (*e.g.* speech recognition) the observation variables are continuous, hence the emission distributions are real-valued (*e.g.* mixtures of Gaussians). In our case, both the state set and the observation set are discrete (and finite), therefore we use multinomial distributions for the emission and transition probabilities. Multinomial distributions are attractive for several reasons: first of all, they are easy to implement; secondly, the maximum likelihood estimation of the parameters has a simple closed form. The parameters are just normalized counts of events that occur in the corpus (the same as the Naïve Bayes from previous class).

Given our labeled corpus $\mathcal{D}_L$, the estimation process consists of counting how many times each event occurs in the corpus and normalize the counts to form proper probability distributions. Let us define the following quantities, called sufficient statistics, that represent the counts of each event in the corpus:

$$\textbf{Initial Counts}: \quad C_{\text{init}}(c_k) = \sum_{m=1}^{M} \mathbf{1}(y_1^m = c_k); \tag{2.4}$$

$$\textbf{Transition Counts}: \quad C_{\text{trans}}(c_k, c_l) = \sum_{m=1}^{M} \sum_{i=2}^{N} \mathbf{1}(y_i^m = c_k \wedge y_{i-1}^m = c_l); \tag{2.5}$$

$$\textbf{Final Counts}: \quad C_{\text{final}}(c_k) = \sum_{m=1}^{M} \mathbf{1}(y_N^m = c_k); \tag{2.6}$$

$$\textbf{Emission Counts}: \quad C_{\text{emiss}}(w_j, c_k) = \sum_{m=1}^{M} \sum_{i=1}^{N} \mathbf{1}(x_i^m = w_j \wedge y_i^m = c_k); \tag{2.7}$$

Note that **1** is an indicator function that has the value 1 when the particular event happens, and zero otherwise. In words, the previous equations amount to going through the training corpus and counting how often each even occurs. For example, Eq. 2.5 counts how often state c_k follows state c_j. Therefore, $C_{\text{trans}}(\texttt{sunny}, \texttt{rainy})$ contains the number of times that a sunny day followed a rainy day.

After computing the counts, one can perform some sanity checks to make sure the implementation is correct. Summing over all entries of each count table we should observe the following:

- **Initial Counts** – Should sum to the number of sentences: $\sum_{k=1}^{K} C_{\text{init}}(c_k) = M$
- **Transition/Final Counts** – Should sum to the number of tokens: $\sum_{k,l=1}^{K} C_{\text{trans}}(c_k, c_l) + \sum_{k=1}^{K} C_{\text{final}}(c_k) = MN$
- **Observation Counts** – Should sum to the number of tokens: $\sum_{j=1}^{J} \sum_{k=1}^{K} C_{\text{emiss}}(w_j, c_k) = MN$.

Using the sufficient statistics (counts) the parameter estimates are:

$$P_{\text{init}}(c_k|\texttt{start}) = \frac{C_{\text{init}}(c_k)}{\sum_{l=1}^{K} C_{\text{init}}(c_l)} \tag{2.8}$$

$$P_{\text{final}}(\texttt{stop}|c_l) = \frac{C_{\text{final}}(c_l)}{\sum_{k=1}^{K} C_{\text{trans}}(c_k, c_l) + C_{\text{final}}(c_l)} \tag{2.9}$$

$$P_{\text{trans}}(c_k|c_l) = \frac{C_{\text{trans}}(c_k, c_l)}{\sum_{p=1}^{K} C_{\text{trans}}(c_p, c_l) + C_{\text{final}}(c_l)} \tag{2.10}$$

$$P_{\text{emiss}}(w_j|c_k) = \frac{C_{\text{emiss}}(w_j, c_k)}{\sum_{q=1}^{J} C_{\text{emiss}}(w_q, c_k)} \tag{2.11}$$

Exercise 2.2 *The provided function* train_supervised *from the* hmm.py *file implements the above parameter estimates. Run this function given the simple dataset above and look at the estimated probabilities. Are they correct? You can also check the variables ending in* _counts *instead of* _probs *to see the raw counts (for example, typing* hmm.initial_probs *will show you the raw counts of initial states). How are the counts related to the probabilities?*

```
>>> import sequences.hmm as hmmc
>>> hmm = hmmc.HMM(simple.x_dict, simple.y_dict)
>>> hmm.train_supervised(simple.train)
>>> print "Initial Probabilities:", hmm.initial_probs
[ 0.66666667  0.33333333]
>>> print "Transition Probabilities:", hmm.transition_probs
[[ 0.5     0.   ]
 [ 0.5     0.625]]
>>> print "Final Probabilities:", hmm.final_probs
[ 0.     0.375]
>>> print "Emission Probabilities", hmm.emission_probs
[[ 0.75    0.25 ]
 [ 0.25    0.375]
 [ 0.      0.375]
 [ 0.      0.   ]]
```

2.4 Decoding a Sequence

Given the learned parameters and a new observation sequence $x = x_1 \ldots x_N$, we want to find "the best" sequence of hidden states $y^* = y_1^* \ldots y_N^*$; this is called the *decoding* problem. There are several ways to define what we mean by the best y^*—for instance, we may will to minimize the probability of error on each hidden variable Y_i; or, we may want the best assignment to the sequence $Y_1 \ldots Y_N$ as a whole.

The first way, normally called **posterior decoding** or **minimum risk decoding**, consists in picking the highest state posterior for each position i in the sequence:

$$y_i^* = \arg\max_{y_i \in \Lambda} P(Y_i = y_i | X_1 = x_1, \ldots, X_N = x_N). \tag{2.12}$$

This method does not guarantee that the sequence $y^* = y_1^* \ldots y_N^*$ will be a valid sequence of the model. For instance there might be a transition probability between two of the best state posteriors with probability zero.

The second approach, called **Viterbi decoding**, consists in picking the best global hidden state sequence:

$$\begin{aligned} y^* &= \arg\max_{y = y_1 \ldots y_N} P(Y_1 = y_1, \ldots, Y_N = y_N | X_1 = x_1, \ldots, X_N = x_N) \\ &= \arg\max_{y = y_1 \ldots y_N} P(Y_1 = y_1, \ldots, Y_N = y_N, X_1 = x_1, \ldots, X_N = x_N). \end{aligned} \tag{2.13}$$

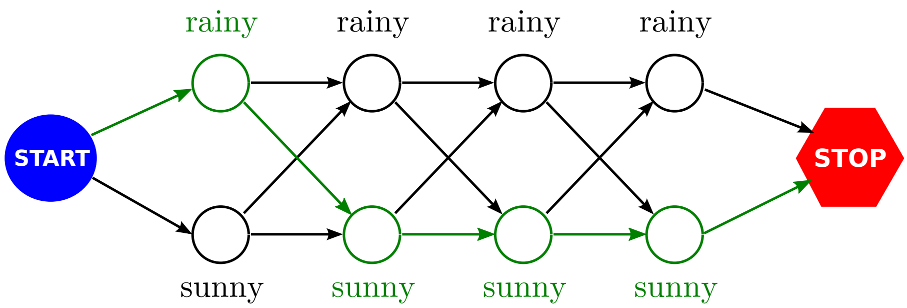


Figure 2.2: Trellis representation of the HMM in Figure 2.1, for the observation sequence "`walk walk shop clean`", where each hidden variable can take the values `rainy` or `sunny`.

Both approaches rely on dynamic programming, making use of the independence assumptions of the HMM model. They use an alternative representation of the HMM called a Trellis (Figure 2.2).

This representation unfolds all possible states for each position and makes explicit the independence assumption: each position only depends on the previous position. Figure 2.2 shows the trellis for the particular example in Figure 2.1.

Each column represents a position in the sentence, and each row represents a possible state. We can associate to each arrow in Figure 2.2 a *transition probability* (or, in the arrows that depart from the start symbol, a *initial probability*, and in the arrows that point to the stop symbol, a *final probability*), and to each circle in Figure 2.2 an *emission probability*, which is the probability that the observed symbol is emitted by that particular state.

For convenience, we will be working with log-probabilities, rather than probabilities. This will be motivated further in Section **??**, where we describe how operations can be performed efficiently in the log-domain. If we associate to each circle and arrow in Figure 2.2 a score which corresponds to the log-probabilities above, and if we define the score of a path in the trellis connecting the `start` and `stop` symbols as the sum of the scores of the circles and arrows it traverses, then the goal of finding the most likely sequence of states (Viterbi decoding) corresponds to finding the path with the largest score.

In the next exercise, you will compute the trellis scores.

Exercise 2.3 *The trellis scores are given by the following expressions:*

- *For each state c_k:*

$$\text{score}_{\text{init}}(c_k) \quad = \quad \log P_{\text{init}}(Y_1 = c_k|\texttt{start}). \tag{2.14}$$

- *For each position $i \in 1, \ldots, N-1$ and each pair of states c_k and c_l:*

$$\text{score}_{\text{trans}}(i, c_k, c_l) \quad = \quad \log P_{\text{trans}}(Y_{i+1} = c_k|Y_i = c_l). \tag{2.15}$$

- *For each state c_l:*

$$\text{score}_{\text{final}}(i, c_l) \quad = \quad \log P_{\text{final}}(\texttt{stop}|Y_N = c_l). \tag{2.16}$$

- *For each position $i \in 1, \ldots, N$ and state c_k:*

$$\text{score}_{\text{emiss}}(i, c_k) \quad = \quad \log P_{\text{emiss}}(X_i = x_i|Y_i = c_k). \tag{2.17}$$

Convince yourself that the score of a path in the trellis (summing over the scores above) is equivalent gives the log-probability $\log P(X = x, Y = y)$*, as defined in Eq. 2.2. Use the given function* compute_scores *on the first training sequence and confirm that the values are correct. You should get the same values as presented below.*

```
>>> initial_scores, transition_scores, final_scores, emission_scores = hmm.compute_scores
    (simple.train.seq_list[0])
>>> print initial_scores
[-0.40546511 -1.09861229]
>>> print transition_scores
[[[-0.69314718         -inf]
  [-0.69314718 -0.47000363]]
```

```
 [[-0.69314718        -inf]
  [-0.69314718 -0.47000363]]

 [[-0.69314718        -inf]
  [-0.69314718 -0.47000363]]]
>>> print final_scores
[        -inf -0.98082925]
>>> print emission_scores
[[-0.28768207 -1.38629436]
 [-0.28768207 -1.38629436]
 [-1.38629436 -0.98082925]
 [        -inf -0.98082925]]
```

Note that scores which are $-\infty$ correspond to zero-probability events.

2.4.1 Computing in log-domain

We will see that the decoding algorithms will need to multiply twice as many probability terms as the length N of the sequence. This may cause underflowing problems when N is large, since the nested multiplication of numbers smaller than 1 may easily become smaller than the machine precision. To avoid that problem, **?**) presents a scaled version of the decoding algorithms that avoids this problem. An alternative, which is widely used, is computing in the log-domain. That is, instead of manipulating probabilities, manipulate log-probabilities (the scores presented above). Every time we need to multiply probabilities, we can sum their log-representations, since:

$$\log(\exp(a) \times \exp(b)) = a + b. \tag{2.18}$$

Sometimes, we need to *add* probabilities. In the log domain, this requires us to compute

$$\log(\exp(a) + \exp(b)) = a + \log(1 + \exp(b - a)), \tag{2.19}$$

where we assume that a is smaller than b.

Exercise 2.4 *Look at the module* `sequences/log_domain.py`*. This module implements a function* `logsum_pair(logx, logy)` *to add two numbers represented in the log-domain; it returns their sum also represented in the log-domain. The function* `logsum(logv)` *sums all components of an array represented in the log-domain. This will be used later in our decoding algorithms. To observe why this is important, type the following:*

```
>>> import numpy as np
>>> a = np.random.rand(10)
>>> np.log(sum(np.exp(a)))
2.8397172643228661
>>> np.log(sum(np.exp(10*a)))
10.121099917705818
>>> np.log(sum(np.exp(100*a)))
93.159220940569128
>>> np.log(sum(np.exp(1000*a)))
inf
>>> from sequences.log_domain import *
>>> logsum(a)
2.8397172643228665
>>> logsum(10*a)
10.121099917705818
>>> logsum(100*a)
93.159220940569114
>>> logsum(1000*a)
925.88496219586864
```

2.4.2 Posterior Decoding

Posterior decoding consists in picking the highest state posterior for each position in the sequence independently; for each $i = 1, \ldots, N$:

$$y_i^* = \arg\max_{y_i \in \Lambda} P(Y_i = y_i | X = x). \tag{2.20}$$

For a given sequence, the **sequence posterior distribution** is the probability of a particular hidden state sequence given that we have observed a particular sentence. Moreover, we will be interested in two other posteriors distributions: the **state posterior distribution**, corresponding to the probability of being in a given state in a certain position given the observed sentence; and the **transition posterior distribution**, which is the probability of making a particular transition, from position i to $i+1$, given the observed sentence.

$$\textbf{Sequence Posterior}: \quad P(Y = y | X = x) = \frac{P(X = x, Y = y)}{P(X = x)}; \tag{2.21}$$

$$\textbf{State Posterior}: \quad P(Y_i = y_i | X = x); \tag{2.22}$$

$$\textbf{Transition Posterior}: \quad P(Y_{i+1} = y_{i+1}, Y_i = y_i | X = x). \tag{2.23}$$

To compute the posteriors, a first step is to be able to compute the likelihood of the sequence $P(X = x)$, which corresponds to summing the probability of all possible hidden state sequences.

$$\textbf{Likelihood}: \quad P(X = x) = \sum_{y \in \Lambda^N} P(X = x, Y = y). \tag{2.24}$$

The number of possible hidden state sequences is exponential in the length of the sentence ($|\Lambda|^N$), which makes summing over all of them hard. In this particular small example there are $2^4 = 16$ such paths and we can actually enumerate them explicitly and calculate their probability using Equation 2.1. But this is as far as it goes: for part-of-speech induction with a small tagset of 12 tags and a medium size sentence of length 10, there are $12^{10} = 61917364224$ such paths. Yet, we must be able to compute this sum (sum over $y \in \Lambda^N$) to compute the above likelihood formula; this is called the inference problem. For sequence models, there is a well known dynamic programming algorithm, the **Forward-Backward** (FB) algorithm, that allows the computation to be performed in linear time,[3] by making use of the independence assumptions.

The FB algorithm relies on the independence of previous states assumption, which is illustrated in the trellis view by only having arrows between consecutive states. The FB algorithm defines two auxiliary probabilities, the forward probability and the backward probability.

$$\textbf{Forward Probability}: \quad \mathrm{forward}(i, c_k) = P(Y_i = c_k, X_1 = x_1, \ldots, X_i = x_i) \tag{2.25}$$

The forward probability represents the probability that in position i we are in state $Y_i = c_k$ and that we have observed $x_1, \ldots, x_i$ up to that position. The forward probability is defined by the following recurrence rule:

$$\mathrm{forward}(1, c_k) = P_{\mathrm{init}}(c_k | \mathtt{start}) \times P_{\mathrm{emiss}}(x_1 | c_k) \tag{2.26}$$

$$\mathrm{forward}(i, c_k) = \left(\sum_{c_l \in \Lambda} P_{\mathrm{trans}}(c_k | c_l) \times \mathrm{forward}(i-1, c_l) \right) \times P_{\mathrm{emiss}}(x_i | c_k) \tag{2.27}$$

$$\mathrm{forward}(N+1, \mathtt{stop}) = \sum_{c_l \in \Lambda} P_{\mathrm{final}}(\mathtt{stop} | c_l) \times \mathrm{forward}(N, c_l). \tag{2.28}$$

Using the forward trellis one can compute the likelihood simply as:

$$P(X = x) = \mathrm{forward}(N+1, \mathtt{stop}). \tag{2.29}$$

Although the forward probability is enough to calculate the likelihood of a given sequence, we will also need the backward probability to calculate the state posteriors. The backward probability is similar to the forward probability, but operates in the inverse direction. It represents the probability of observing $x_{i+1}, \ldots, x_N$ from position $i+1$ up to N, given that at position i we are at state $Y_i = c_l$:

$$\textbf{Backward Probability}: \quad \mathrm{backward}(i, c_l) = P(X_{i+1} = x_{i+1}, \ldots, X_N = x_N | Y_i = c_l). \tag{2.30}$$

[3] The runtime is linear with respect to the sequence length. More precisely, the runtime is $O(N|\Lambda|^2)$. A naive enumeration would cost $O(|\Lambda|^N)$.

Algorithm 7 Forward-Backward algorithm

input: sentence $x_1, \ldots, x_N$, scores $P_{\text{init}}, P_{\text{trans}}, P_{\text{final}}, P_{\text{emiss}}$
Forward pass: Compute the forward probabilities
Initialization
for $c_k \in \Lambda$ **do**
  $\text{forward}(1, c_k) = P_{\text{init}}(c_k|\texttt{start}) \times P_{\text{emiss}}(x_1|c_k)$
end for
for $i = 2$ **to** N **do**
  for $c_k \in \Lambda$ **do**
    $\text{forward}(i, c_k) = \left(\sum_{c_l \in \Lambda} P_{\text{trans}}(c_k|c_l) \times \text{forward}(i-1, c_l)\right) \times P_{\text{emiss}}(x_i|c_k)$
  end for
end for
Backward pass: Compute the backward probabilities
Initialization
for $c_l \in \Lambda$ **do**
  $\text{backward}(N, c_l) = P_{\text{final}}(\texttt{stop}|c_l)$
end for
for $i = N-1$ **to** 1 **do**
  $\text{backward}(i, c_l) = \sum_{c_k \in \Lambda} P_{\text{trans}}(c_k|c_l) \times \text{backward}(i+1, c_k) \times P_{\text{emiss}}(x_{i+1}|c_k)$
end for
output: The forward and backward probabilities.

The backward recurrence is given by:

$$\begin{aligned}
\text{backward}(N, c_l) &= P_{\text{final}}(\texttt{stop}|c_l) && (2.31)\\
\text{backward}(i, c_l) &= \sum_{c_k \in \Lambda} P_{\text{trans}}(c_k|c_l) \times \text{backward}(i+1, c_k) \times P_{\text{emiss}}(x_{i+1}|c_k) && (2.32)\\
\text{backward}(0, \texttt{start}) &= \sum_{c_k \in \Lambda} P_{\text{init}}(c_k|\texttt{start}) \times \text{backward}(1, c_k) \times P_{\text{emiss}}(x_1|c_k). && (2.33)
\end{aligned}$$

Using the backward trellis one can compute the likelihood simply as:.

$$P(X = x) = \text{backward}(0, \texttt{start}). \tag{2.34}$$

Moreover, with the forward and backward probabilities, we can compute the likelihood of a given sentence using any position i in the sentence as follows:

$$\begin{aligned}
P(X = x) &= \sum_{c_k \in \Lambda} P(X_1 = x_1, \ldots, X_N = x_N, Y_i = c_k)\\
&= \sum_{c_k \in \Lambda} \underbrace{P(X_1 = x_1, \ldots, X_i = x_i, Y_i = c_k)}_{\text{forward}(i, c_k)} \times \underbrace{P(X_{i+1} = x_{i+1}, \ldots, X_N = x_N | Y_i = c_k)}_{\text{backward}(i, c_k)}\\
&= \sum_{c_k \in \Lambda} \text{forward}(i, c_k) \times \text{backward}(i, c_k). && (2.35)
\end{aligned}$$

This equation will work for any choice of i. Although redundant, this fact is useful when implementing an HMM as a sanity check that the computations are being performed correctly, since one can compute this expression for several i; they should all yield the same value.

Algorithm 7 shows the pseudo code for the forward backward algorithm.

Exercise 2.5 *Run the provided forward-backward algorithm on the first train sequence. Observe that both the forward and the backward passes give the same log-likelihood.*

```
>>> log_likelihood, forward = hmm.decoder.run_forward(initial_scores, transition_scores,
    final_scores, emission_scores)
>>> print 'Log-Likelihood =', log_likelihood
Log-Likelihood = -5.06823232601
>>> log_likelihood, backward = hmm.decoder.run_backward(initial_scores, transition_scores
```

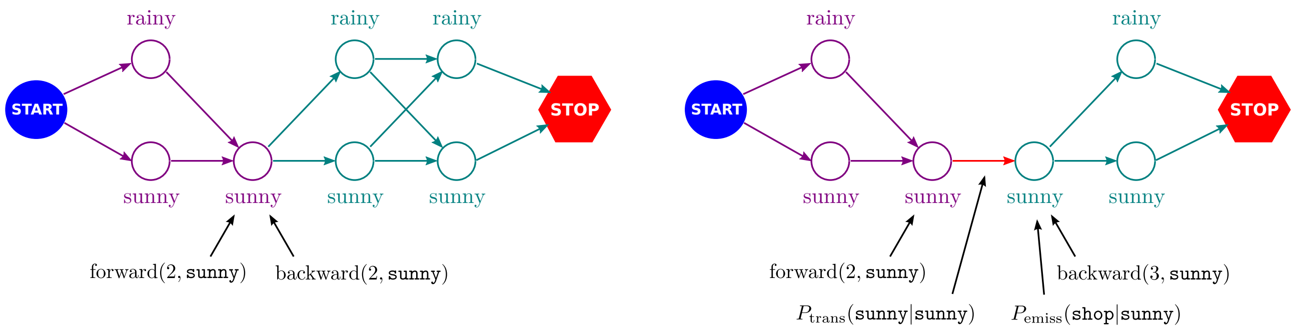


Figure 2.3: A graphical representation of the components in the state and transition posteriors.

```
    , final_scores, emission_scores)
>>> print 'Log-Likelihood =', log_likelihood
Log-Likelihood = -5.06823232601
```

Given the forward and backward probabilities, one can compute both the state and transition posteriors (you can hint why by looking at the term inside the sum in Eq. 2.35).

$$\textbf{State Posterior}: \quad P(Y_i = y_i|X = x) = \frac{\text{forward}(i, y_i) \times \text{backward}(i, y_i)}{P(X = x)}; \tag{2.36}$$

$$\textbf{Transition Posterior}: \quad P(Y_i = y_i, Y_{i+1} = y_{i+1}|X = x) = \frac{\text{forward}(i, y_i) \times P_{\text{trans}}(y_{i+1}|y_i) \times P_{\text{emiss}}(x_{i+1}|y_{i+1}) \times \text{backward}(i+1, y_{i+1})}{P(X = x)}. \tag{2.37}$$

A graphical representation of these posteriors is illustrated in Figure 2.3. On the left it is shown that $\text{forward}(i, y_i)\text{backward}(i, y_i)$ returns the sum of all paths that contain the state y_i, weighted by $P(X = x)$; on the right we can see that $\text{forward}(i, y_i)P_{\text{trans}}(y_{i+1}|y_i)P_{\text{emiss}}(x_{i+1}|y_{i+1})\text{backward}(i + 1, y_{i+1})$ returns the same for all paths containing the edge from y_i to y_{i+1}. As a practical example, given that the person perform the sequence of actions "`walk walk shop clean`", we want to know the probability of having been raining in the second day. The state posterior probability for this event can be seen as the probability that the sequence of actions above was generated by a sequence of weathers and where it was raining in the second day. In this case, the possible sequences would be all the sequences which have `rainy` in the second position.

Using the state posteriors, we are ready to perform posterior decoding. The strategy is to compute the state posteriors for each position $i \in \{1, \ldots, N\}$ and each state $c_k \in \Lambda$, and then pick the arg-max at each position:

$$\widehat{y}_i := \arg\max_{y_i \in \Lambda} P(Y_i = y_i|X = x). \tag{2.38}$$

Exercise 2.6 *Compute the node posteriors for the first training sentence (use the provided* compute_posteriors *function), and look at the output. Note that the state posteriors are a proper probability distribution (the columns of the result sum to 1).*

```
>>> initial_scores, transition_scores, final_scores, emission_scores = hmm.compute_scores
    (simple.train.seq_list[0])
>>> state_posteriors, _, _ = hmm.compute_posteriors(initial_scores,
                                                      transition_scores,
                                                      final_scores,
                                                      emission_scores)

>>> print state_posteriors
[[ 0.95738152  0.04261848]
 [ 0.75281282  0.24718718]
 [ 0.26184794  0.73815206]
 [ 0.          1.        ]]
```

Exercise 2.7 *Run the posterior decode on the first test sequence, and evaluate it.*

```
>>> y_pred = hmm.posterior_decode(simple.test.seq_list[0])
>>> print "Prediction test 0:", y_pred
walk/rainy walk/rainy shop/sunny clean/sunny
>>> print "Truth test 0:", simple.test.seq_list[0]
walk/rainy walk/sunny shop/sunny clean/sunny
```

Do the same for the second test sentence:

```
>>> y_pred = hmm.posterior_decode(simple.test.seq_list[1])
>>> print "Prediction test 1:", y_pred
clean/rainy walk/rainy tennis/rainy walk/rainy
>>> print "Truth test 1:", simple.test.seq_list[1]
clean/sunny walk/sunny tennis/sunny walk/sunny
```

What is wrong? Note the observations for the second test sentence: the observation `tennis` *was never seen at training time, so the probability for it will be zero (no matter what state). This will make all possible state sequences have zero probability. As seen in the previous lecture, this is a problem with generative models, which can be corrected using smoothing (among other options).*

Change the train_supervised *method to add smoothing:*

```
def train_supervised(self,sequence_list, smoothing):
```

Try, for example, adding 1 to all the counts, and repeating this exercise with that smoothing. What do you observe?

```
>>> hmm.train_supervised(simple.train, smoothing=0.1)
>>> y_pred = hmm.posterior_decode(simple.test.seq_list[0])
>>> print "Prediction test 0 with smoothing:", y_pred
walk/rainy walk/rainy shop/sunny clean/sunny
>>> print "Truth test 0:", simple.test.seq_list[0]
walk/rainy walk/sunny shop/sunny clean/sunny
>>> y_pred = hmm.posterior_decode(simple.test.seq_list[1])
>>> print "Prediction test 1 with smoothing:", y_pred
clean/sunny walk/sunny tennis/sunny walk/sunny
>>> print "Truth test 1:", simple.test.seq_list[1]
clean/sunny walk/sunny tennis/sunny walk/sunny
```

2.4.3 Viterbi Decoding

Viterbi decoding consists in picking the best global hidden state sequence $\widehat{y}$ as follows:

$$\widehat{y} = \arg\max_{y \in \Lambda^N} P(Y = y | X = x) = \arg\max_{y \in \Lambda^N} P(X = x, Y = y). \tag{2.39}$$

The Viterbi algorithm is very similar to the forward procedure of the FB algorithm, making use of the same trellis structure to efficiently represent the exponential number of sequences without prohibitive computation costs. In fact, the only difference from the forward-backward algorithm is in the recursion 2.27 where instead of *summing* over all possible hidden states, we take their *maximum*.

$$\textbf{Viterbi} \quad \text{viterbi}(i, y_i) = \max_{y_1 \cdots y_{i-1}} P(Y_1 = y_1, \ldots Y_i = y_i, X_1 = x_1, \ldots, X_i = x_i) \tag{2.40}$$

The Viterbi trellis represents the path with maximum probability in position i when we are in state $Y_i = y_i$ and that we have observed $x_1, \ldots, x_i$ up to that position. The Viterbi algorithm is defined by the following

Algorithm 8 Viterbi algorithm

input: sentence $x_1, \ldots, x_N$, scores $P_{\text{init}}, P_{\text{trans}}, P_{\text{final}}, P_{\text{emiss}}$
Forward pass: Compute the best paths for every end state
Initialization
for $c_k \in \Lambda$ **do**
  $\text{viterbi}(1, c_k) = P_{\text{init}}(c_k|\texttt{start}) \times P_{\text{emiss}}(x_1|c_k)$
end for
for $i = 2$ **to** N **do**
  for $c_k \in \Lambda$ **do**
    $\text{viterbi}(i, c_k) = \left(\max_{c_l \in \Lambda} P_{\text{trans}}(c_k|c_l) \times \text{viterbi}(i-1, c_l)\right) \times P_{\text{emiss}}(x_i|c_k)$
    $\text{backtrack}(i, c_k) = \left(\arg\max_{c_l \in \Lambda} P_{\text{trans}}(c_k|c_l) \times \text{viterbi}(i-1, c_l)\right) \times P_{\text{emiss}}(x_i|c_k)$
  end for
end for
$\max_{y \in \Lambda^N} P(X = x, Y = y) := \max_{c_l \in \Lambda} P_{\text{final}}(\texttt{stop}|c_l) \times \text{viterbi}(N, c_l)$

Backward pass: backtrack to obtain the most likely path
$\widehat{y}_N = \arg\max_{c_l \in \Lambda} P_{\text{final}}(\texttt{stop}|c_l) \times \text{viterbi}(N, c_l)$
for $i = N - 1$ **to** 1 **do**
  $\widehat{y}_i = \text{backtrack}(i + 1, \widehat{y}_{i+1})$
end for
output: the viterbi path $\widehat{y}$.

recurrence:

$$\text{viterbi}(1, c_k) = P_{\text{init}}(c_k|\texttt{start}) \times P_{\text{emiss}}(x_1|c_k) \quad (2.41)$$

$$\text{viterbi}(i, c_k) = \left(\max_{c_l \in \Lambda} P_{\text{trans}}(c_k|c_l) \times \text{viterbi}(i-1, c_l)\right) \times P_{\text{emiss}}(x_i|c_k) \quad (2.42)$$

$$\text{backtrack}(i, c_k) = \left(\arg\max_{c_l \in \Lambda} P_{\text{trans}}(c_k|c_l) \times \text{viterbi}(i-1, c_l)\right) \times P_{\text{emiss}}(x_i|c_k) \quad (2.43)$$

$$\text{viterbi}(N+1, \texttt{stop}) = \max_{c_l \in \Lambda} P_{\text{final}}(\texttt{stop}|c_l) \times \text{viterbi}(N, c_l) \quad (2.44)$$

$$\text{backtrack}(N+1, \texttt{stop}) = \arg\max_{c_l \in \Lambda} P_{\text{final}}(\texttt{stop}|c_l) \times \text{viterbi}(N, c_l). \quad (2.45)$$

Algorithm 8 shows the pseudo code for the Viterbi algorithm. Note the similarity with the forward algorithm. The only differences are:

- Maximizing instead of summing;
- Keeping the argmax's to backtrack.

Exercise 2.8 *Implement a method for performing Viterbi decoding in file* `sequence_classification_decoder.py`.

```
        def run_viterbi(self, initial_scores, transition_scores, final_scores,
            emission_scores):
```

Hint: look at the implementation of `run_forward`.

This method will be called by

```
def viterbi_decode(self, sequence)
```

in the module `sequence_classifier.py`.

Test your method on both test sentences and compare the results with the ones given.

```
>>> hmm.train_supervised(simple.train, smoothing=0.1)
>>> y_pred, score = hmm.viterbi_decode(simple.test.seq_list[0])
>>> print "Viterbi decoding Prediction test 0 with smoothing:", y_pred, score
walk/rainy walk/rainy shop/sunny clean/sunny  -6.02050124698
>>> print "Truth test 0:", simple.test.seq_list[0]
walk/rainy walk/sunny shop/sunny clean/sunny
>>> y_pred, score = hmm.viterbi_decode(simple.test.seq_list[1])
>>> print "Viterbi decoding Prediction test 1 with
smoothing:", y_pred, score
clean/sunny walk/sunny tennis/sunny walk/sunny  -11.713974074
>>> print "Truth test 1:", simple.test.seq_list[1]
clean/sunny walk/sunny tennis/sunny walk/sunny
```

Note: since we didn't run the train_supervised *method again, we are still using the result of the training using smoothing. Therefore, you should compare these results to the ones of posterior decoding with smoothing.*

2.5 Part-of-Speech Tagging (POS)

Part of speech tagging is one the most important NLP tasks. The task is to assign each word a grammatical category, *i.e.* Noun, Verb, Adjective, ... In English, using the Penn Treebank (PTB) (**?**), the current state of the art for part of speech tagging is around 97% for a variety of methods [4].

In the rest of this class we will use a subset of the PTB corpus, but instead of using the original 45 tags we will use a reduced tag set of 12 tags, to make the algorithms faster for the class. In this task, x is a sentence (*i.e.*, a sequence of word tokens) and y is the sequence of possible PoS tags.

The first step is to load the corpus. We will start by loading 1000 sentences for training and 200 sentences both for development and testing. Then we train the HMM model by maximum likelihood estimation.

```
>>> import readers.pos_corpus as pcc
>>> corpus = pcc.PostagCorpus()
>>> train_seq = corpus.read_sequence_list_conll("../data/train-02-21.conll",max_sent_len=
    15,max_nr_sent=1000)
>>> test_seq = corpus.read_sequence_list_conll("../data/test-23.conll",max_sent_len=15,
    max_nr_sent=1000)
>>> dev_seq = corpus.read_sequence_list_conll("../data/dev-22.conll",max_sent_len=15,
    max_nr_sent=1000)
>>> hmm = hmmc.HMM(corpus.word_dict, corpus.tag_dict)
>>> hmm.train_supervised(train_seq)
>>> hmm.print_transition_matrix()
```

Look at the transition probabilities of the trained model (see Figure 2.4), and see if they match your intuition about the English language (e.g. adjectives tend to come before nouns).

Exercise 2.9 *Test the model using both posterior decoding and Viterbi decoding on both the train and test set, using the methods in class HMM:*

```
>>> viterbi_pred_train = hmm.viterbi_decode_corpus(train_seq)
>>> posterior_pred_train = hmm.posterior_decode_corpus(train_seq)
>>> eval_viterbi_train =   hmm.evaluate_corpus(train_seq, viterbi_pred_train)
>>> eval_posterior_train =  hmm.evaluate_corpus(train_seq, posterior_pred_train)
>>> print "Train Set Accuracy: Posterior Decode \%.3f, Viterbi Decode: \%.3f"\%(
    eval_posterior_train,eval_viterbi_train)
Train Set Accuracy: Posterior Decode 0.985, Viterbi Decode: 0.985
>>> viterbi_pred_test = hmm.viterbi_decode_corpus(test_seq)
>>> posterior_pred_test = hmm.posterior_decode_corpus(test_seq)
>>> eval_viterbi_test =   hmm.evaluate_corpus(test_seq,viterbi_pred_test)
>>> eval_posterior_test = hmm.evaluate_corpus(test_seq,posterior_pred_test)
>>> print "Test Set Accuracy: Posterior Decode \%.3f, Viterbi Decode: \%.3f"\%(
    eval_posterior_test,eval_viterbi_test)
```

[4] See ACL state of the art wiki

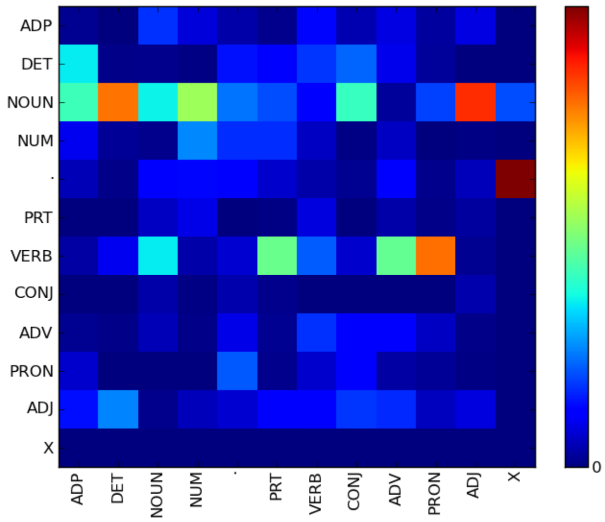


Figure 2.4: Transition probabilities of the trained model. Each column is previous state and row is current state. Note the high probability of having Noun after Adjective, or of having Verb after Noun, as expected.

```
Test Set Accuracy: Posterior Decode 0.350, Viterbi Decode: 0.509
```

What do you observe? Remake the previous exercise but now train the HMM using smoothing. Try different values (0,0.1,0.01,1) and report the results on the train and development set. (Use function pick_best_smoothing*).*

```
>>> best_smothing = hmm.pick_best_smoothing(train_seq, dev_seq, [10,1,0.1,0])
Smoothing 10.000000 --  Train Set Accuracy: Posterior Decode 0.731, Viterbi Decode: 0.691
Smoothing 10.000000 -- Test Set Accuracy: Posterior Decode 0.712, Viterbi Decode: 0.675
Smoothing 1.000000 --  Train Set Accuracy: Posterior Decode 0.887, Viterbi Decode: 0.865
Smoothing 1.000000 -- Test Set Accuracy: Posterior Decode 0.818, Viterbi Decode: 0.792
Smoothing 0.100000 --  Train Set Accuracy: Posterior Decode 0.968, Viterbi Decode: 0.965
Smoothing 0.100000 -- Test Set Accuracy: Posterior Decode 0.851, Viterbi Decode: 0.842
Smoothing 0.000000 --  Train Set Accuracy: Posterior Decode 0.985, Viterbi Decode: 0.985
Smoothing 0.000000 -- Test Set Accuracy: Posterior Decode 0.370, Viterbi Decode: 0.526
>>> hmm.train_supervised(train_seq, smoothing=best_smothing)
>>> viterbi_pred_test = hmm.viterbi_decode_corpus(test_seq)
>>> posterior_pred_test = hmm.posterior_decode_corpus(test_seq)
>>> eval_viterbi_test =   hmm.evaluate_corpus(test_seq, viterbi_pred_test)
>>> eval_posterior_test = hmm.evaluate_corpus(test_seq, posterior_pred_test)
>>> print "Best Smoothing \%f --  Test Set Accuracy: Posterior Decode \%.3f, Viterbi
    Decode: \%.3f"%(best_smothing,eval_posterior_test,eval_viterbi_test)
Best Smoothing 0.100000 --  Test Set Accuracy: Posterior Decode 0.837, Viterbi Decode: 0.
    827
```

Perform some error analysis to understand were the errors are coming from. You can start by visualizing the confusion matrix (true tags vs predicted tags). You should get something like 2.5.

```
>>> import sequences.confusion_matrix as cm
>>> confusion_matrix = cm.build_confusion_matrix(test_seq.seq_list, viterbi_pred_test,
    len(corpus.tag_dict), hmm.get_num_states())
>>> cm.plot_confusion_bar_graph(confusion_matrix, corpus.tag_dict, xrange(hmm.
    get_num_states()), 'Confusion matrix')
>>> plt.show()
```

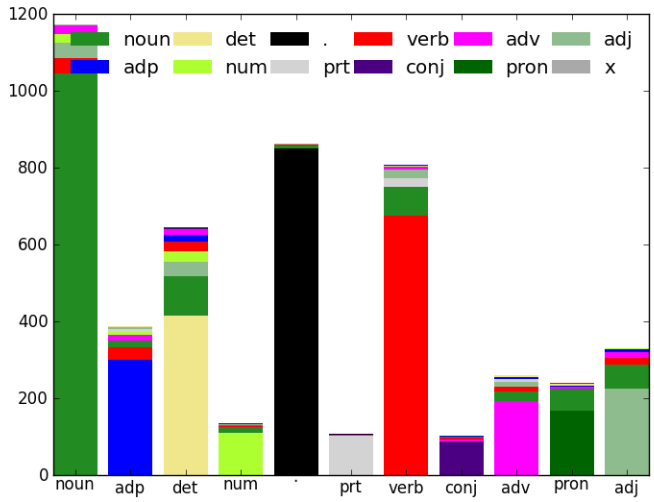


Figure 2.5: Confusion Matrix for the previous example. Predict tags are columns and the true tags corresponds to the constituents of each column.

2.6 Unsupervised Learning of HMMs

[AFM: explain here the EM algorithm]

```
Initial accuracy: 0.303638
Iter: 1 Log Likelihood: -101824.763927
Iter: 1 Accuracy: 0.305441
Iter: 2 Log Likelihood: -78057.108346
Iter: 2 Accuracy: 0.321976
Iter: 3 Log Likelihood: -77813.725501
Iter: 3 Accuracy: 0.357451
Iter: 4 Log Likelihood: -77192.947674
Iter: 4 Accuracy: 0.385109
Iter: 5 Log Likelihood: -76191.800849
Iter: 5 Accuracy: 0.392123
Iter: 6 Log Likelihood: -75242.572729
Iter: 6 Accuracy: 0.391121
Iter: 7 Log Likelihood: -74392.892496
Iter: 7 Accuracy: 0.404249
Iter: 8 Log Likelihood: -73357.542833
Iter: 8 Accuracy: 0.399940
Iter: 9 Log Likelihood: -72135.182778
Iter: 9 Accuracy: 0.399238
Iter: 10 Log Likelihood: -70924.246230
Iter: 10 Accuracy: 0.395430
Iter: 11 Log Likelihood: -69906.561800
Iter: 11 Accuracy: 0.394328
Iter: 12 Log Likelihood: -69140.228623
Iter: 12 Accuracy: 0.390821
Iter: 13 Log Likelihood: -68541.416423
Iter: 13 Accuracy: 0.391522
Iter: 14 Log Likelihood: -68053.456865
Iter: 14 Accuracy: 0.389117
Iter: 15 Log Likelihood: -67667.318961
Iter: 15 Accuracy: 0.386411
Iter: 16 Log Likelihood: -67337.685686
Iter: 16 Accuracy: 0.385409
Iter: 17 Log Likelihood: -67054.571821
Iter: 17 Accuracy: 0.385409
Iter: 18 Log Likelihood: -66769.973881
Iter: 18 Accuracy: 0.385409
```

```
Iter: 19 Log Likelihood: -66442.608458
Iter: 19 Accuracy: 0.385409
```

Day 3

Learning Structured Predictors

In this class, we will continue to focus on sequence classification, but instead of following a *generative* approach (like in the previous chapter) we move towards *discriminative* approaches. Recall that the difference between these approaches is that generative approaches attempt to model the probability distribution of the data, $P(X,Y)$ whereas discriminative ones only model the conditional probability of the sequence, given the observed data, $P(Y|X)$.

Table 3.1 shows how the models for classification have counterparts in *sequential* classification. In fact, in the last chapter we discussed the Hidden Markov model, which can be seen as a generalization of the Naïve Bayes model for sequences. In this chapter, we will see a generalization of the Perceptron algorithm for sequence problems (yielding the Structured Perceptron, **?**) and a generalization of Maximum Entropy model for sequences (yielding Conditional Random Fields, **?**). Note that both these generalizations are not specific for sequences and can be applied to a wide range of models (we will see in tomorrow's lecture how these methods can be applied to parsing). Although we will not cover all the methods described in Chapter 1, bear in mind that all of those have a structured counterpart. It should be intuitive after this lecture how those methods could be extended to structured problems, given the perceptron example. Before we explain the particular methods, the next section will talk a bit about feature representation for sequences.

Throughout this chapter, we will be searching for the solution of

$$\arg\max_{y \in \Lambda^N} P(Y = y|X = x) = \arg\max_{y \in \Lambda^N} \boldsymbol{w} \cdot \boldsymbol{f}(x, y). \tag{3.1}$$

where $\boldsymbol{w}$ is a weight vector, and $\boldsymbol{f}(x,y)$ is a feature vector. We will see that these sort of linear models are more flexible than HMMs, in the sense that they may incorporate more general features while being able to reuse the same decoders (based on the Viterbi and forward-backward algorithms). In fact, *the exercises in this lab will not touch the decoders that have been developed in the previous lab.* Only the training algorithms and the function that compute the scores will change.

As in the previous section, $y = y_1 \ldots y_N$ is a sequence so the maximization is over an exponential number of objects, making it intractable by brute force methods. Again we will make an assumption analogous to the "first order Markov independence property," so that the features may decompose as a sum over nodes and edges in a trellis. This is done by assuming expression 3.1 can be written as:

$$\arg\max_{y \in \Lambda^N} \sum_{i=1}^{N} \underbrace{\boldsymbol{w} \cdot \boldsymbol{f}_{\text{emiss}}(i, x, y_i)}_{\text{score}_{\text{emiss}}} + \underbrace{\boldsymbol{w} \cdot \boldsymbol{f}_{\text{init}}(x, y_1)}_{\text{score}_{\text{init}}} + \sum_{i=1}^{N-1} \underbrace{\boldsymbol{w} \cdot \boldsymbol{f}_{\text{trans}}(i, x, y_i, y_{i+1})}_{\text{score}_{\text{trans}}} + \underbrace{\boldsymbol{w} \cdot \boldsymbol{f}_{\text{final}}(x, y_N)}_{\text{score}_{\text{final}}} \tag{3.2}$$

In other words, the scores $\text{score}_{\text{emiss}}$, $\text{score}_{\text{init}}$, $\text{score}_{\text{trans}}$ and $\text{score}_{\text{final}}$ are now computed as inner products

Classification	**Sequences**
Generative	
Naïve Bayes 1.1.2	Hidden Markov Models 2.2
Discriminative	
Perceptron 1.3.2	Structured Perceptron 3.3
Maximum Entropy 1.3.3	Conditional Random Fields 3.2

Table 3.1: Summary of the methods that we will be covering this lecture.

Condition	Name
$y_i = c_k$ & $i = 0$	Initial Features
$y_i = c_k$ & $y_{i-1} = c_l$	Transition Features
$y_i = c_k$ & $i = N$	Final Features
$x_i = w_j$ & $y_i = c_k$	Emission Features

Table 3.2: `IDFeatures` feature set. This set replicates the features used by the HMM model.

Condition	Name
$y_i = c_k$ & $i = 0$	Initial Features
$y_i = c_k$ & $y_{i-1} = c_l$	Transition Features
$y_i = c_k$ & $i = N$	Final Features
$x_i = w_j$ & $y_i = c_k$	Basic Emission Features
$x_i = w_j$ & w_j is uppercased & $y_i = c_k$	Uppercase Features
$x_i = w_j$ & w_j contains digit & $y_i = c_k$	Digit Features
$x_i = w_j$ & w_j contains hyphen & $y_i = c_k$	Hyphen Features
$x_i = w_j$ & $w_j[0..i] \forall i \in [1,2,3]$ & $y_i = c_k$	Prefix Features
$x_i = w_j$ & $w_j[\lvert w_j\rvert - i..\lvert w_j\rvert] \forall i \in [1,2,3]$ & $y_i = c_k$	Suffix Features

Table 3.3: Extended feature set. Some features in this set could not be included in the HMM model. The features included in the bottom row are all considered emission features for the purpose of our implementation, since they all depend on i, x and y_i.

between weight vectors and feature vectors rather than log-probabilities. The feature vectors depend locally on the output variable (*i.e.*, they only look at a single y_i or a pair y_i, y_{i+1}); however they may depend globally on the observed input $x = x_1, \ldots, x_N$.

3.1 Feature Extraction

In this section we will define two simple feature sets. The first one will only use identity features, and will mimic the features used by the HMM model from the previous section. This will allow us to directly compare the performance of a generative vs a discriminative approach. Note that although not required, all the features we will use in this section are binary features, indicating whether a given condition is satisfied or not.

Table 3.2 depicts the features that are implicit in the HMM, which was the subject of the previous chapter. These features are indicators of initial, transition, final, and emission events. The fact that we were using a generative model has forced us (in some sense) to make strong independence assumptions. However, since we now move to a discriminative approach, where we model $P(Y|X)$ rather than $P(X,Y)$, we are not tied anymore to some of these assumptions. In particular:

- We may use "overlapping" features, *e.g.*, features that fire simultaneously for many instances. For example, we can use a feature for a word, such as a feature which fires for the word "brilliantly", and another for prefixes and suffixes of that word, such as one which fires if the last two letters of the word are "ly". This would lead to an awkward model if we wanted to insist on a generative approach.

- We may use features that depend arbitrarily on the *entire input sequence* x. On the other hand, we still need to resort to "local" features with respect to the *outputs* (*e.g.* looking only at consecutive state pairs), otherwise decoding algorithms will become more expensive.

Table 3.3 shows examples of features that are traditionally used in POS tagging with discriminative models. Of course, we could have much more complex features, looking arbitrarily to the input sequence. We are not going to have them in this exercise only for performance reasons (to have less features and smaller caches). State-of-the-art sequence classifiers can easily reach over one million features!

Our features subdivide into two groups: f_{emiss}, f_{init}, and f_{final} are all instances of *node features*, depending only on a single position in the state sequence (a node in the trellis); f_{trans} are *edge features*, depending on two consecutive positions in the state sequence (an edge in the trellis). Similarly as in the previous chapter, we have the following scores (also called *log-potentials* in the literature on CRFs and graphical models):

- *Initial scores.* These are scores for the initial state. They are given by

$$\text{score}_{\text{init}}(x, y_1) = \boldsymbol{w} \cdot \boldsymbol{f}_{\text{init}}(x, y_1). \tag{3.3}$$

- *Transition scores.* These are scores for two consecutive states at a particular position. They are given by

$$\text{score}_{\text{trans}}(i, x, y_i, y_{i+1}) = \boldsymbol{w} \cdot \boldsymbol{f}_{\text{trans}}(i, x, y_i, y_{i+1}). \tag{3.4}$$

- *Final scores.* These are scores for the final state. They are given by

$$\text{score}_{\text{final}}(x, y_N) = \boldsymbol{w} \cdot \boldsymbol{f}_{\text{final}}(x, y_N). \tag{3.5}$$

- *Emission scores.* These are scores for a state at a particular position. They are given by

$$\text{score}_{\text{emiss}}(i, x, y_i) = \boldsymbol{w} \cdot \boldsymbol{f}_{\text{emiss}}(i, x, y_i). \tag{3.6}$$

Given a weight vector $\boldsymbol{w}$, the conditional probability $P_{\boldsymbol{w}}(Y = y|X = x)$ is then defined as follows:

$$P_{\boldsymbol{w}}(Y = y|X = x) = \frac{1}{Z(\boldsymbol{w}, x)} \exp\left(\boldsymbol{w} \cdot \boldsymbol{f}_{\text{init}}(x, y_1) + \sum_{i=1}^{N-1} \boldsymbol{w} \cdot \boldsymbol{f}_{\text{trans}}(i, x, y_i, y_{i+1}) + \boldsymbol{w} \cdot \boldsymbol{f}_{\text{final}}(x, y_N) + \sum_{i=1}^{N} \boldsymbol{w} \cdot \boldsymbol{f}_{\text{emiss}}(i, x, y_i)\right) \tag{3.7}$$

where the normalizing factor $Z(\boldsymbol{w}, x)$ is called the *partition function*:

$$Z(\boldsymbol{w}, x) = \sum_{y \in \Lambda^N} \exp\left(\boldsymbol{w} \cdot \boldsymbol{f}_{\text{init}}(x, y_1) + \sum_{i=1}^{N-1} \boldsymbol{w} \cdot \boldsymbol{f}_{\text{trans}}(i, x, y_i, y_{i+1}) + \boldsymbol{w} \cdot \boldsymbol{f}_{\text{final}}(x, y_N) + \sum_{i=1}^{N} \boldsymbol{w} \cdot \boldsymbol{f}_{\text{emiss}}(i, x, y_i)\right) \tag{3.8}$$

For decoding, there are three important problems that need to be solved:

1. Given $X = x$, computing the most likely output sequence $\widehat{y}$ (the one which maximizes $P_{\boldsymbol{w}}(Y = y|X = x)$.
2. Compute the posterior marginals $P_{\boldsymbol{w}}(Y_i = y_i|X = x)$ at each position i.
3. Evaluate the partition function $Z(\boldsymbol{w}, x)$.

Interestingly, all these problems can be solved by using the very same algorithms that were already implemented for HMMs: the Viterbi algorithm (for 1) and the forward-backward algorithm (for 2–3). All that changes is the way the scores are computed.

For training, the important problem is that of obtaining the weight vector $\boldsymbol{w}$ that lead to an accurate classifier. We will discuss two possible strategies:

1. Maximizing conditional log-likelihood from a set of labeled data $\{(x^m, y^m)\}_{m=1}^M$, yielding **conditional random fields**. This corresponds to the following optimization problem:

$$\widehat{\boldsymbol{w}} = \arg\max_{\boldsymbol{w}} \sum_{m=1}^{M} \log P_{\boldsymbol{w}}(Y = y^m|X = x^m). \tag{3.9}$$

   To avoid overfitting, it is common to regularize with the Euclidean norm function, which is equivalent to considering a zero-mean Gaussian prior on the weight vector. The problem becomes:

$$\widehat{\boldsymbol{w}} = \arg\max_{\boldsymbol{w}} \sum_{m=1}^{M} \log P_{\boldsymbol{w}}(Y = y^m|X = x^m) - \frac{\lambda}{2}\|\boldsymbol{w}\|^2. \tag{3.10}$$

   This is precisely the structured variant of the logistic regression method discussed in Chapter 1. Unlike HMMs, this problem does not have a closed form solution and has to be solved with numerical optimization.

2. Alternatively, running the **structured perceptron** algorithm to obtain a weight vector $\boldsymbol{w}$ that accurately classifies the training data. We will see that this simple strategy achieves results which are competitive with conditional log-likelihood maximization.

Algorithm 9 SGD for Conditional Random Fields

input: $\mathcal{D}$, λ, number of rounds T, learning rate sequence $(\eta_t)_{t=1,\ldots,T}$
initialize $\boldsymbol{w}^1 = \mathbf{0}$
for $t = 1$ **to** T **do**
  choose $m = m(t)$ randomly
  take training pair (x^m, y^m) and compute the probability $P_{\boldsymbol{w}}(Y = y^m | X = x^m)$ using the current model $\boldsymbol{w}$ and Eq. 3.7.
  for every i, y_i', and y_{i+1}', compute marginal probabilities $P(Y_i = y_i' | X = x)$ and $P(Y_i = y_i', Y_{i+1} = y_{i+1}' | X = x^m)$ using the current model $\boldsymbol{w}$, for each node and edge, through the forward-backward algorithm.
  compute the feature vector expectation:

$$\begin{aligned} E_{\boldsymbol{w}^t}[\boldsymbol{f}(x^m, Y)] = & \sum_{y_1} P(y_1|x^m) \boldsymbol{f}_{\text{init}}(x^m, y_1) + \\ & \sum_{i=1}^{N-1} \sum_{y_i, y_{i+1}} P(y_i, y_{i+1}|x^m) \boldsymbol{f}_{\text{trans}}(i, x^m, y_i, y_{i+1}) + \\ & \sum_{y_N} P(y_N|x^m) \boldsymbol{f}_{\text{final}}(x^m, y_N) + \\ & \sum_{i=1}^{N} \sum_{y_i} P(y_i|x^m) \boldsymbol{f}_{\text{emiss}}(i, x^m, y_i). \end{aligned} \tag{3.11}$$

  update the model:

$$\boldsymbol{w}^{t+1} \leftarrow (1 - \lambda\eta_t)\boldsymbol{w}^t + \eta_t \left(\boldsymbol{f}(x^m, y^m) - E_{\boldsymbol{w}^t}[\boldsymbol{f}(x^m, Y)]\right)$$

end for
output: $\widehat{\boldsymbol{w}} \leftarrow \boldsymbol{w}^{T+1}$

3.2 Conditional Random Fields

Conditional Random Fields (CRF) (**?**) can be seen as an extension of the Maximum Entropy (ME) models to structured problems.[1] They are *globally* normalized models: the probability of a given sentence is given by Equation 3.7. There are only two differences with respect to the standard ME model described a couple of days ago for multi-class classification:

- Instead of computing the posterior marginals $P(Y = y|X = x)$ for all possible configurations of the output variables (which are exponentially many), it assumes the model decompose into "parts" (in this case, nodes and edges), and it computes only the posterior marginals for those parts, $P(Y_i = y_i|X = x)$ and $P(Y_i = y_i, Y_{i+1} = y_{i+1}|X = x)$. Crucially, **we can compute these quantities by using the very same forward-backward algorithm that we have described for HMMs**.

- Instead of updating the features for all possible outputs $y' \in \Lambda^N$, we again exploit the decomposition into parts above and update only "local features" at the nodes and edges.

Algorithm 9 shows the pseudo code to optimize a CRF with the stochastic gradient descent (SGD) algorithm (our toolkit also includes an implmentation of a quasi-Newton method, L-BFGS, which converges faster, but for the purpose of this exercise, we will stick with SGD).

Exercise 3.1 *In this exercise you will train a CRF using different feature sets for Part-of-Speech Tagging. Start with the code below, which uses the ID feature set from table 3.2.*

```
import sequences.crf_online as crfo
import sequences.structured_perceptron as spc
import readers.pos_corpus as pcc
import sequences.id_feature as idfc
import sequences.extended_feature as exfc
```

[1] An earlier, less successful, attempt to perform such an extension was via Maximum Entropy Markov models (MEMM) (**?**). There each factor (a node or edge) is a *locally* normalized maximum entropy model. A shortcoming of MEMMs is its so-called *labeling bias* (**?**), which makes them biased towards states with few successor states (see **?**) for more information).

```
print "CRF Exercise"

corpus = pcc.PostagCorpus()
train_seq = corpus.read_sequence_list_conll("../data/train-02-21.conll",max_sent_len=10,
    max_nr_sent=1000)
test_seq = corpus.read_sequence_list_conll("../data/test-23.conll",max_sent_len=10,
    max_nr_sent=1000)
dev_seq = corpus.read_sequence_list_conll("../data/dev-22.conll",max_sent_len=10,
    max_nr_sent=1000)
feature_mapper = idfc.IDFeatures(train_seq)
feature_mapper.build_features()

crf_online = crfo.CRFOnline(corpus.word_dict, corpus.tag_dict, feature_mapper)
crf_online.num_epochs = 20
crf_online.train_supervised(train_seq)

Epoch: 0 Objective value: -5.779018
Epoch: 1 Objective value: -3.192724
Epoch: 2 Objective value: -2.717537
Epoch: 3 Objective value: -2.436614
Epoch: 4 Objective value: -2.240491
Epoch: 5 Objective value: -2.091833
Epoch: 6 Objective value: -1.973353
Epoch: 7 Objective value: -1.875643
Epoch: 8 Objective value: -1.793034
Epoch: 9 Objective value: -1.721857
Epoch: 10 Objective value: -1.659605
Epoch: 11 Objective value: -1.604499
Epoch: 12 Objective value: -1.555229
Epoch: 13 Objective value: -1.510806
Epoch: 14 Objective value: -1.470468
Epoch: 15 Objective value: -1.433612
Epoch: 16 Objective value: -1.399759
Epoch: 17 Objective value: -1.368518
Epoch: 18 Objective value: -1.339566
Epoch: 19 Objective value: -1.312636
```

You will receive feedback when each epoch is finished, note that running the 20 epochs might take a while. After training is done, evaluate the learned model on the training, development and test sets.

```
pred_train = crf_online.viterbi_decode_corpus(train_seq)
pred_dev = crf_online.viterbi_decode_corpus(dev_seq)
pred_test = crf_online.viterbi_decode_corpus(test_seq)

eval_train = crf_online.evaluate_corpus(train_seq, pred_train)
eval_dev = crf_online.evaluate_corpus(dev_seq, pred_dev)
eval_test = crf_online.evaluate_corpus(test_seq, pred_test)

print "CRF - ID Features Accuracy Train: %.3f Dev: %.3f Test: %.3f"%(eval_train,eval_dev,
    eval_test)
```

You should get values similar to these:

```
Out[]: CRF -
ID Features Accuracy Train: 0.949 Dev: 0.846 Test: 0.858
```

Compare with the results achieved with the HMM model (0.837 on the test set, from the previous lecture). Even using a similar feature set a CRF yields better results than the HMM from the previous lecture. Perform some error analysis and figure out what are the main errors the tagger is making. Compare them with the errors made by the HMM model. (Hint: use the methods developed in the previous lecture to help you with the error analysis).

Exercise 3.2 *Repeat the previous exercise using the extended feature set. Compare the results.*

```
feature_mapper = exfc.ExtendedFeatures(train_seq)
feature_mapper.build_features()

crf_online = crfo.CRFOnline(corpus.word_dict, corpus.tag_dict, feature_mapper)
crf_online.num_epochs = 20
crf_online.train_supervised(train_seq)

Epoch: 0 Objective value: -7.141596
Epoch: 1 Objective value: -1.807511
Epoch: 2 Objective value: -1.218877
Epoch: 3 Objective value: -0.955739
Epoch: 4 Objective value: -0.807821
Epoch: 5 Objective value: -0.712858
Epoch: 6 Objective value: -0.647382
Epoch: 7 Objective value: -0.599442
Epoch: 8 Objective value: -0.562584
Epoch: 9 Objective value: -0.533411
Epoch: 10 Objective value: -0.509885
Epoch: 11 Objective value: -0.490548
Epoch: 12 Objective value: -0.474318
Epoch: 13 Objective value: -0.460438
Epoch: 14 Objective value: -0.448389
Epoch: 15 Objective value: -0.437800
Epoch: 16 Objective value: -0.428402
Epoch: 17 Objective value: -0.419990
Epoch: 18 Objective value: -0.412406
Epoch: 19 Objective value: -0.405524

pred_train = crf_online.viterbi_decode_corpus(train_seq)
pred_dev = crf_online.viterbi_decode_corpus(dev_seq)
pred_test = crf_online.viterbi_decode_corpus(test_seq)

eval_train = crf_online.evaluate_corpus(train_seq, pred_train)
eval_dev = crf_online.evaluate_corpus(dev_seq, pred_dev)
eval_test = crf_online.evaluate_corpus(test_seq, pred_test)

print "CRF - Extended Features Accuracy Train: %.3f Dev: %.3f Test: %.3f"%(eval_train,
    eval_dev,eval_test)
```

You should get values close to the following:

```
CRF - Extended Features Accuracy Train: 0.984 Dev: 0.899 Test: 0.894
```

Compare the errors obtained with the two different feature sets. Do some error analysis: what errors were correct by using more features? Can you think of other features to use to solve the errors found?

The main lesson to learn from this exercise is that, usually, if you are not satisfied by the accuracy of your algorithm, you can perform some error analysis and find out which errors your algorithm is making. You can then add more features which attempt to improve those specific errors (this is known as *feature engineering*). This can lead to two problems:

- More features will make training and decoding more expensive. For example, if you add features that depend on the current word and the previous word, the number of new features is the square of the number of different words, which is quite large. For example, the Penn Treebank has around 40000 different words, so you are adding a lot of new features, even though not all pairs of words will ever occur. Features that depend on three words (previous, current, and next) are even more numerous.
- If features are very specific, such as the (previous word, current word, next word) one just mentioned, they might occur very rarely in the training set, which leads to overfit problems. Some of these problems (not all) can be mitigated with techniques such as smoothing, which you already learned about.

Algorithm 10 Averaged Structured perceptron

1: **input:** $\mathcal{D}$, number of rounds T
2: initialize $\boldsymbol{w}^1 = \mathbf{0}$
3: **for** $t = 1$ **to** T **do**
4: choose $m = m(t)$ randomly
5: take training pair (x^m, y^m) and predict using the current model $\boldsymbol{w}$, through the Viterbi algorithm:

$$\widehat{y} \leftarrow \arg\max_{y' \in \Lambda^N} \boldsymbol{w}^t \cdot \boldsymbol{f}(x^m, y')$$

6: update the model: $\boldsymbol{w}^{t+1} \leftarrow \boldsymbol{w}^t + \boldsymbol{f}(x^m, y^m) - \boldsymbol{f}(x^m, \widehat{y})$
7: **end for**
8: **output:** the averaged model $\widehat{\boldsymbol{w}} \leftarrow \frac{1}{T}\sum_{t=1}^{T} \boldsymbol{w}^t$

3.3 Structured Perceptron

The structured perceptron (**?**), namely its averaged version, is a very simple algorithm that relies on Viterbi decoding and very simple additive updates. In practice this algorithm is very easy to implement and behaves remarkably well in a variety of problems. These two characteristics make the structured perceptron algorithm a natural first choice to try and test a new problem or a new feature set.

Recall what you learned from §1.3.2 on the perceptron algorithm and compare it against the structured perceptron (Algorithm 10).

There are only two differences, which mimic the ones already seen for the comparison between CRFs and multi-class ME models:

- Instead of explicitly enumerating all possible output configurations (exponentially many of them) to compute $\widehat{y} := \arg\max_{y' \in \mathcal{Y}} \boldsymbol{w} \cdot \boldsymbol{f}(x^m, y')$, it finds the best sequence through the Viterbi algorithm.
- Instead of updating the features for the entire $\widehat{y}$, it updates only the node and edge features at the positions where the labels are different—i.e., where mistakes are made.

Exercise 3.3 *Implement the structured perceptron algorithm. To do this, edit file* `structured_perceptron.py` *and implement the function*

```
def perceptron_update(self, sequence):
    pass
```

This function should apply one round of the perceptron algorithm, updating the weights for a given sequence, and returning the number of predicted labels (which equals the sequence length) and the number of mistaken labels.

Hint: adapt the function

```
def gradient_update(self, sequence, eta):
```

defined in file `crf_online.py`*. You will need to replace the computation of posterior marginals by the Viterbi algorithm, and to change the parameter updates according to Algorithm 10.*

Exercise 3.4 *Repeat Exercises 3.1–3.2 using the structured perceptron algorithm instead of a CRF. Report the results.*

Here is the code for the simple feature set:

```
feature_mapper = idfc.IDFeatures(train_seq)
feature_mapper.build_features()

print "Perceptron Exercise"

sp = spc.StructuredPerceptron(corpus.word_dict, corpus.tag_dict, feature_mapper)
sp.num_epochs = 20
sp.train_supervised(train_seq)

Epoch: 0 Accuracy: 0.656806
```

```
Epoch: 1 Accuracy: 0.820898
Epoch: 2 Accuracy: 0.879176
Epoch: 3 Accuracy: 0.907432
Epoch: 4 Accuracy: 0.925239
Epoch: 5 Accuracy: 0.939956
Epoch: 6 Accuracy: 0.946284
Epoch: 7 Accuracy: 0.953790
Epoch: 8 Accuracy: 0.958499
Epoch: 9 Accuracy: 0.955114
Epoch: 10 Accuracy: 0.959235
Epoch: 11 Accuracy: 0.968065
Epoch: 12 Accuracy: 0.968212
Epoch: 13 Accuracy: 0.966740
Epoch: 14 Accuracy: 0.971302
Epoch: 15 Accuracy: 0.968653
Epoch: 16 Accuracy: 0.970419
Epoch: 17 Accuracy: 0.971891
Epoch: 18 Accuracy: 0.971744
Epoch: 19 Accuracy: 0.973510

pred_train = sp.viterbi_decode_corpus(train_seq)
pred_dev = sp.viterbi_decode_corpus(dev_seq)
pred_test = sp.viterbi_decode_corpus(test_seq)



eval_train = sp.evaluate_corpus(train_seq, pred_train)
eval_dev = sp.evaluate_corpus(dev_seq, pred_dev)
eval_test = sp.evaluate_corpus(test_seq, pred_test)

print "Structured Perceptron - ID Features Accuracy Train: %.3f Dev: %.3f Test: %.3f"%(
    eval_train,eval_dev,eval_test)
```

```
Structured Perceptron - ID Features Accuracy Train: 0.984 Dev: 0.835 Test: 0.840
```

Here is the code for the extended feature set:

```
feature_mapper = exfc.ExtendedFeatures(train_seq)
feature_mapper.build_features()
sp = spc.StructuredPerceptron(corpus.word_dict, corpus.tag_dict, feature_mapper)
sp.num_epochs = 20
sp.train_supervised(train_seq)

Epoch: 0 Accuracy: 0.764386
Epoch: 1 Accuracy: 0.872701
Epoch: 2 Accuracy: 0.903458
Epoch: 3 Accuracy: 0.927594
Epoch: 4 Accuracy: 0.938484
Epoch: 5 Accuracy: 0.951141
Epoch: 6 Accuracy: 0.949816
Epoch: 7 Accuracy: 0.959529
Epoch: 8 Accuracy: 0.957616
Epoch: 9 Accuracy: 0.962325
Epoch: 10 Accuracy: 0.961148
Epoch: 11 Accuracy: 0.970567
Epoch: 12 Accuracy: 0.968212
Epoch: 13 Accuracy: 0.973216
Epoch: 14 Accuracy: 0.974393
Epoch: 15 Accuracy: 0.973951
Epoch: 16 Accuracy: 0.976600
Epoch: 17 Accuracy: 0.977483
Epoch: 18 Accuracy: 0.974834
Epoch: 19 Accuracy: 0.977042
```

```
pred_train = sp.viterbi_decode_corpus(train_seq)
pred_dev = sp.viterbi_decode_corpus(dev_seq)
pred_test = sp.viterbi_decode_corpus(test_seq)

eval_train = sp.evaluate_corpus(train_seq, pred_train)
eval_dev = sp.evaluate_corpus(dev_seq, pred_dev)
eval_test = sp.evaluate_corpus(test_seq, pred_test)

print "Structured Perceptron - Extended Features Accuracy Train: %.3f Dev: %.3f Test: %.
    3f"%(eval_train,eval_dev,eval_test)
```

And here are the expected results:

```
Structured Perceptron - Extended Features Accuracy Train: 0.984 Dev: 0.888 Test: 0.890
```

Day 4

Syntax and Parsing

In this lab we will implement some exercises related with *parsing*.

4.1 Phrase-based Parsing

4.1.1 Context Free Grammars

Let $\mathcal{T}$ be an *alphabet* (*i.e.*, a finite set of symbols), and denote by $\mathcal{T}^*$ its Kleene closure, *i.e.*, the infinite set of strings produced with those symbols:

$$\mathcal{T}^* = \varnothing \cup \mathcal{T} \cup \mathcal{T}^2 \cup \ldots$$

A *language* L is a subset of $\mathcal{T}^*$. The "complexity" of a language L can be loosely defined by how hard it is to construct a machine (an *automaton*) capable of distinguishing the words in L from the elements of $\mathcal{T}^*$ which are not in L.[1] If L is finite, a very simple automaton can be built which just memorizes the strings in L. The next simplest case is that of *regular languages*, which are recognizable by *finite state machines*. These are the languages that can be expressed by regular expressions. An example (where $\mathcal{T} = \{a, b\}$) is the language $L = \{ab^n aa^n \mid n \in \mathbb{N}\}$, which corresponds to the regular expression $ab * a+$. *Hidden Markov models* (studied in previous lectures) can be seen as a stochastic version of finite state machines.

A step higher in the hierarchy of languages leads to *context-free languages*, which are more complex than regular languages. These are languages that are generated by *context-free grammars*, and recognizable by *push-down automata* (which are slightly more complex than finite state machines). In this section we describe context-free grammars and how they can be made probabilistic. This will yield models that are more powerful than hidden Markov models, and are specially amenable for modeling the syntax of natural languages.[2]

A *context-free grammar* (CFG) is a tuple $G = \langle \mathcal{N}, \mathcal{T}, \mathcal{R}, \mathtt{S} \rangle$ where:

1. $\mathcal{N}$ is a finite set of *non-terminal* symbols. Elements of $\mathcal{N}$ are denoted by upper case letters $(A, B, C, \ldots)$. Each non-terminal symbol is a syntactic category: it represents a different type of phrase or clause in the sentence.

2. $\mathcal{T}$ is a finite set of *terminal* symbols (disjoint from $\mathcal{N}$). Elements of $\mathcal{T}$ are denoted by lower case letters $(a, b, c, \ldots)$. Each terminal symbol is a surface word: terminal symbols make up the actual content of sentences. The set $\mathcal{T}$ is called the *alphabet* of the language defined by the grammar G.

3. $\mathcal{R}$ is a set of *production rules*, *i.e.*, a finite relation from $\mathcal{N}$ to $(\mathcal{N} \cup \mathcal{T})^*$. G is said to be in Chomsky normal form (CNF) if production rules in $\mathcal{R}$ are either of the form $A \to BC$ or $A \to a$.

4. $\mathtt{S}$ is a *start symbol*, used to represent the whole sentence. It must be an element of $\mathcal{N}$.

Any CFG can be transformed to be in CNF without loosing any expressive power in terms of the language it generates. Hence, we henceforth assume that G is in CNF without loss of generality.

To see how CFGs can model the syntax of natural languages, consider the following sentence,

[1] We recommend the classic book by **?**) for a thorough introduction on the subject of automata theory and formal languages.

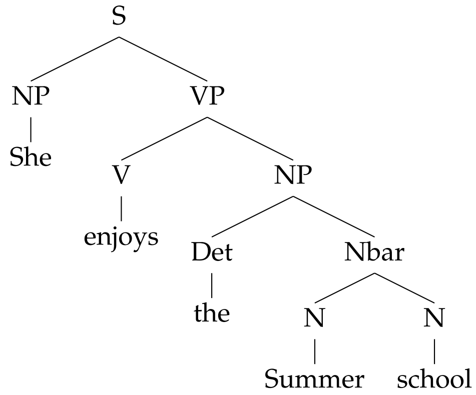
[2] This does not mean that natural languages are context free. There is an immense body of work on grammar formalisms that relax the "context-free" assumption, and those formalisms have been endowed with a probabilistic framework as well. Examples are: lexical functional grammars, head-driven phrase structured grammars, combinatorial categorial grammars, tree adjoining grammars, etc. Some of these formalisms are *mildly context sensitive*, a relaxation of the "context-free" assumption which still allows polynomial parsing algorithms. There is also equivalence in expressive power among several of these formalisms.

```
She enjoys the Summer school.
```

along with a grammar (in CNF) with the following production rules:

```
S --> NP VP
NP --> Det N
NP --> She
VP --> V NP
V --> enjoys
Det --> the
Nbar --> N N
N --> Summer
N --> school
```

With this grammar, we may derive the following parse tree:



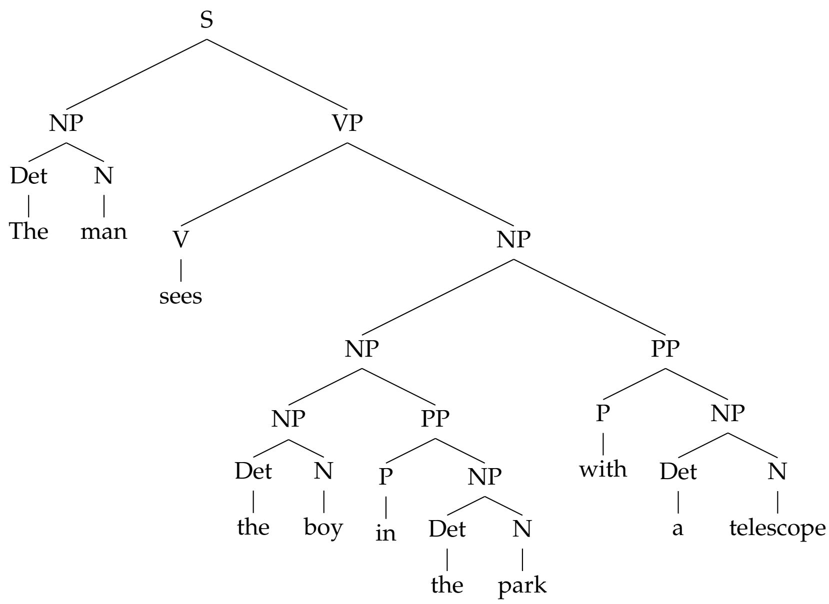
4.1.2 Ambiguity

A fundamental characteristic of natural languages is *ambiguity*. For example, consider the sentence

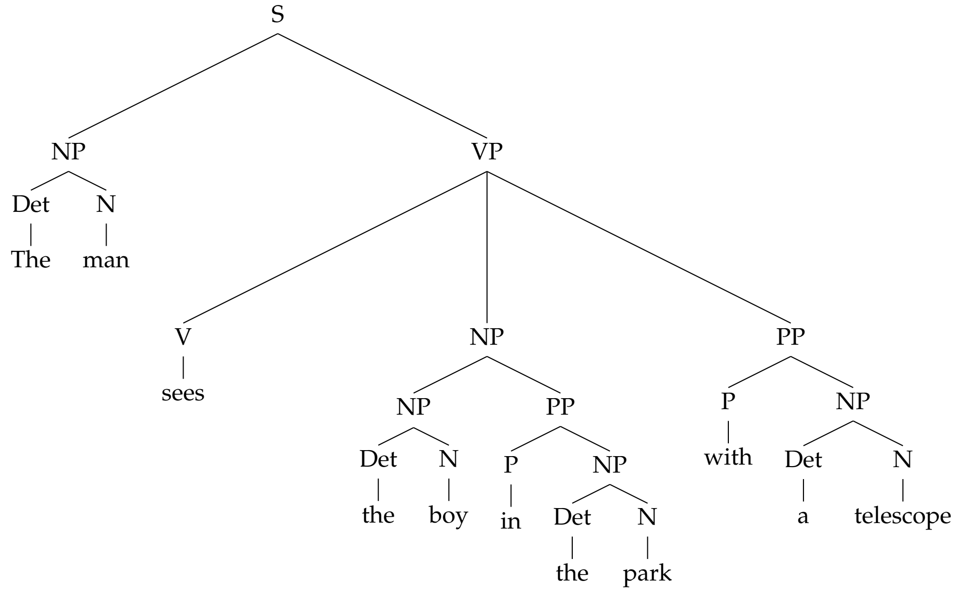
```
The man sees the boy in the park with a telescope.
```

for which all the following parse trees are plausible interpretations:

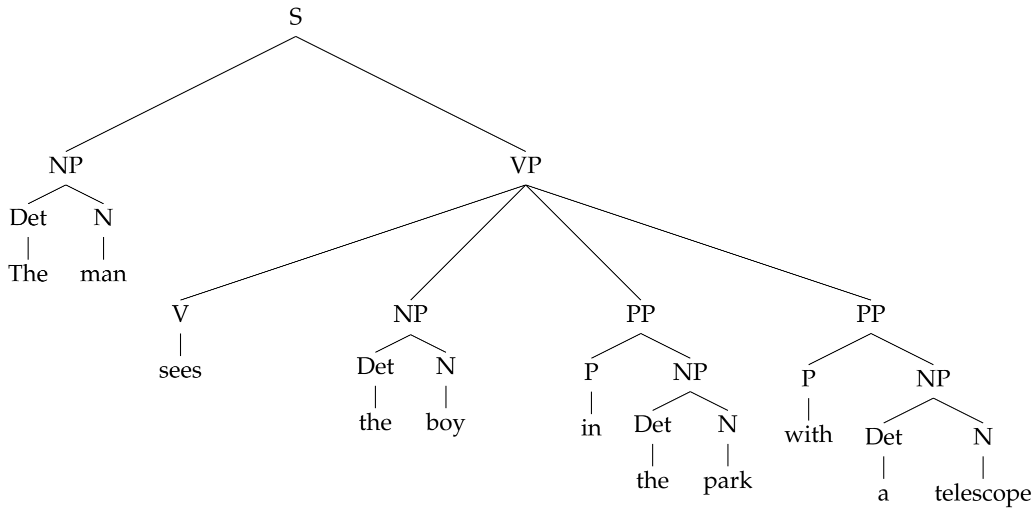
1. The boy is in a park and he has a telescope:



2. The boy is in a park, and the man sees him using a telescope as an instrument:



3. The man is in the park and he has a telescope, through which he sees a boy somewhere:



The ambiguity is caused by the several places to each the prepositional phrase could be attached. This kind of syntactical ambiguity (*PP-attachment*) is one of the most frequent in natural language.

4.1.3 Probabilistic Context-Free Grammars

A *probabilistic context-free grammar* is a tuple $G_{\boldsymbol{\theta}} = \langle \mathcal{N}, \mathcal{T}, \mathcal{R}, \mathtt{S}, \boldsymbol{\theta} \rangle$, where $\langle \mathcal{N}, \mathcal{T}, \mathcal{R}, \mathtt{S} \rangle$ is a CFG and $\boldsymbol{\theta}$ is a vector of parameters, one per each production rule in $\mathcal{R}$. Assuming that the grammar is in CNF, each rule of the kind $Z \rightarrow XY$ is endowed a conditional probability

$$\theta_{Z \rightarrow XY} = P_{\boldsymbol{\theta}}(XY|Z),$$

and each unary rule of the kind $Z \rightarrow w$ is endowed with a conditional probability

$$\theta_{Z \rightarrow w} = P_{\boldsymbol{\theta}}(w|Z).$$

For these conditional probabilities to be well defined, the entries of $\boldsymbol{\theta}$ must be non-negative and need to normalize properly for each $Z \in \mathcal{N}$:

$$\sum_{X,Y \in \mathcal{N}} \theta_{Z \rightarrow XY} + \sum_{w \in \mathcal{T}} \theta_{Z \rightarrow w} = 1.$$

Let s be a string and t a parse tree derivation for s. For each $r \in \mathcal{R}$, let $n_r(t, s)$ be the number of times production rule r appears in the derivation. According to this generative model, the joint probability of t and s factors as

Algorithm 11 CKY algorithm

input: probabilistic CFG $G_{\boldsymbol{\theta}}$ in CNF, sentence $s = w_1 \dots w_N$

{Initialization}
for $i = 1$ **to** N **do**
  for each production rule $r \in \mathcal{R}$ of the form $Z \to w_i$ **do**
    $\delta(i, i, Z) = \theta_{Z \to w_i}$
  end for
end for

{Induction}
for $i = 2$ **to** N **do** {i is length of span}
  for $j = 1$ **to** $N - i + 1$ **do** {j is start of span}
    for each non-terminal $Z \in \mathcal{N}$ **do**
      Set partial probability:

$$\delta(j, j+i-1, Z) = \max_{\substack{X,Y \\ j<k<j+i}} \delta(j, k-1, X) \times \delta(k, j+i-2, Y) \times \theta_{Z \to XY}$$

      Store backpointer:

$$\psi(j, j+i-1, Z) = \arg\max_{\substack{X,Y \\ j<k<j+i}} \delta(j, k-1, X) \times \delta(k, j+i-2, Y) \times \theta_{Z \to XY}$$

    end for
  end for
end for

{Termination}
$P(s, \hat{t}) = \delta(1, N, \mathtt{S})$
Backtrack through ψ to obtain most likely parse tree $\hat{t}$

the product of the conditional probabilities above:

$$P(t, s) = \prod_{r \in \mathcal{R}} \theta_r^{n_r(t,s)}.$$

For example, for the sentence above (*She enjoys the Summer school*) this probability would be

$$\begin{aligned} P(t, s) &= P(\mathtt{NP\ VP}|\mathtt{S}) \times P(\mathtt{She}|\mathtt{NP}) \times P(\mathtt{V\ NP}|\mathtt{VP}) \times P(\mathtt{enjoys}|\mathtt{V}) \\ &\quad \times P(\mathtt{Det\ Nbar}|\mathtt{NP}) \times P(\mathtt{the}|\mathtt{Det}) \times P(\mathtt{N\ N}|\mathtt{Nbar}) \\ &\quad \times P(\mathtt{Summer}|\mathtt{N}) \times P(\mathtt{school}|\mathtt{N}). \end{aligned} \tag{4.1}$$

When a sentence is ambiguous, the most likely parse tree can be obtained by maximizing the conditional probability $P(t|s)$; this quantity is proportional to $P(t, s)$ and therefore the latter quantity can be maximized. The number of possible parse trees, however, grows exponentially with the sentence length, rendering a direct maximization intractable. Fortunately, a generalization of the Viterbi algorithm exists which uses dynamic programming to carry out this computation. This is the subject of the next section.

4.1.4 The CKY Parsing Algorithm

One of the most widely algorithm for parsing natural language sentences is the Cocke-Kasami-Younger (CKY) algorithm. Given a grammar in CNF with $|\mathcal{R}|$ production rules, its runtime complexity for parsing a sentence of length N is $O(N^3|\mathcal{R}|)$. We present here a simple extension of the CKY algorithm that obtains the most likely parse tree of a sentence, along with its probability.[3] This is displayed in Alg. 11.

[3] Similarly, the forward-backward algorithm for computing posterior marginals in sequence models can be extended for context-free parsing. It takes the name *inside-outside algorithm*. See **?**) for details.

Exercise 4.1 *In this simple exercise, you will see the CKY algorithm in action. There is a Javascript applet that illustrates how CKY works (in its non-probabilistic form). Go to* `http://www.diotavelli.net/people/void/demos/cky.html`*, and observe carefully the several steps taken by the algorithm. Write down a small grammar in CNF that yields multiple parses for the ambiguous sentence* The man saw the boy in the park with a telescope*, and run the demo for this particular sentence. What would happen in the probabilistic form of CKY?*

4.1.5 Learning the Grammar

There is an immense body of work on *grammar induction* using probabilistic models (see *e.g.*, **?**) and the references therein, as well as the most recent works of **???**)): this is the problem of learning the parameters of a grammar from plain sentences only. This can be done in an EM fashion (like in sequence models), except that the forward-backward algorithm is replaced by inside-outside. Unfortunately, the performance of unsupervised parsers is far from good, at present days. Much better results have been produced by supervised systems, which, however, require expensive annotation in the form of *treebanks*: this is a corpus of sentences annotated with their corresponding syntactic trees. The following is an example of an annotated sentence in one of the most widely used treebanks, the *Penn Treebank* (`http://www.cis.upenn.edu/~treebank/`):

```
( (S
    (NP-SBJ (NNP BELL) (NNP INDUSTRIES) (NNP Inc.) )
    (VP (VBD increased)
      (NP (PRP$ its) (NN quarterly) )
      (PP-DIR (TO to)
        (NP (CD 10) (NNS cents) ))
      (PP-DIR (IN from)
        (NP
          (NP (CD seven) (NNS cents) )
          (NP-ADV (DT a) (NN share) ))))
    (. .) ))
```

Exercise 4.2 *This exercise will show you that real-world sentences can have complicated syntactic structures. There is a parse tree visualizer in* `http://www.ark.cs.cmu.edu/treeviz/`*. Go to your local* `data/treebanks` *folder and open the file* `PTB_excerpt.txt`*. Copy a few trees from the file, one at a time, and examine their parse trees in the visualizer.*

A treebank makes possible to learn a parser in a supervised fashion. The simplest way is via a generative approach. Instead of counting transition and observation events of an HMM (as we did for sequence models), we now need to count production rules and symbol occurrences to estimate the parameters of a probabilistic context-free grammar. While performance would be much better than that of unsupervised parsers, it would still be rather poor. The reason is that the model we have described so far is oversimplistic: it makes too strong independence assumptions. In this case, the Markovian assumptions are:

1. Each terminal symbol w in some position i is independent of everything else given that it was derived by the rule $Z \to w$ (i.e., given its parent Z);

2. Each pair of non-terminal symbols X and Y spanning positions i to j, with split point k, is independent of everything else given that they were derived by the rule $Z \to XY$ (i.e., given their parent Z).

The next section describes some model refinements that complicate the problem of parameter estimation, but usually allow for a dramatic improvement on the quality of the parser.

4.1.6 Model Refinements

A number of refinements has been made that yield more accurate parsers. We mention just a few:

Parent annotation. This strategy splits each non-terminal symbol in the grammar (*e.g.* Z) by annotating it with all its possible parents (*e.g.* creates nodes $Z^X, Z^Y, \ldots$ every time production rules like $X \to Z\cdot$, $X \to \cdot Z$, $Y \to Z\cdot$, or $Y \to \cdot Z$ exist in the original grammar). This increases the vertical Markovian length of the model, hence weakening the independence assumptions. Parent annotation was initiated by **?**) and carried on in the unlexicalized parsers of **?**) and follow-up works.

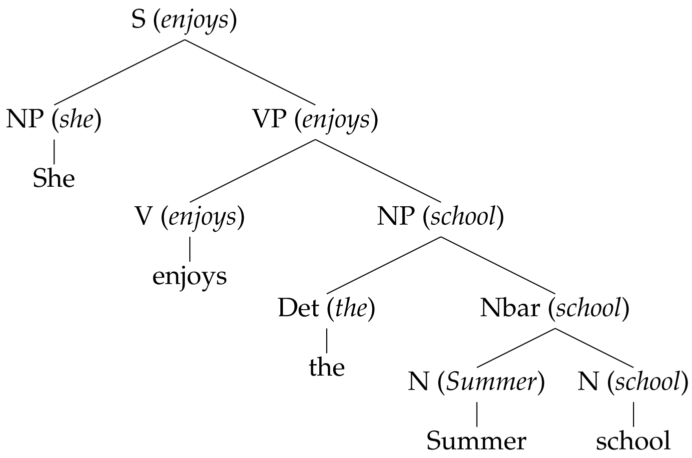


Figure 4.1: A lexicalized parse tree for the sentence *She enjoys the Summer school*.

Lexicalization. A particular weakness of PCFGs is that they ignore word context for interior nodes in the parse tree. Yet, this context is relevant in determining the production rules that should be invoked in the derivation. A way of overcoming this limitation is by *lexicalizing* parse trees, *i.e.*, annotating each phrase node with the lexical item (word) which governs that phrase: this is called the *head* word of the phrase. Fig. 4.1 shows an example of a lexicalized parse tree. To account for lexicalization, each non-terminal symbol in the grammar (*e.g.* Z) is split into many symbols, each annotated with a word that may govern that phrase (*e.g.* $Z^{w_1}, Z^{w_2}, \ldots$). This greatly increases the size of the grammar, but it has a significant impact in performance. A string of work involving lexicalized PCFGs includes **???**).

Discriminative models. Similarly as in sequence models (where it was shown how to move from an HMM to a CRF), we may abandon our generative model and consider a discriminative one. An advantage of doing that is that it becomes much easier to adopt non-local input features (*i.e.*, features that depend arbitrarily on the surface string), for example the kind of features obtained via lexicalization, and much more. The CKY parsing algorithm can still be used for decoding, provided the feature vector decompose according to *non-terminal symbols* and *production rules*. In this case, productions and non-terminals will have a score which does not correspond to a log-probability; the partition function and the posterior marginals can be computed with the inside-outside algorithm. See **?**) for an application of structured SVMs to parsing, and **?**) for an application of CRFs.

Latent variables. Splitting the variables in the grammar by introducing latent variables appears as an alternative to lexicalization and parent annotation. There is a string of work concerning latent variable grammars, either for the generative and discriminative cases: **???**). Some related work also considers coarse-to-fine parsing, which iteratively applies more and more refined models: **?**).

History-based parsers. Finally, there is a totally different line of work which models parsers as a sequence of greedy shift-reduce decisions made by a push-down automaton (**??**). When discriminative models are used, arbitrary conditioning can be done on past decisions made by the automaton, allowing to include features that are difficult to handle by the other parsers. This comes at the price of greediness in the decisions taken, which implies suboptimality in maximizing the desired objective function.

4.2 Dependency Parsing

4.2.1 Motivation

Consider again the sentence

She enjoys the Summer school.

along with the lexicalized parse tree displayed in Fig. 4.1. If we drop the phrase constituents and keep only the head words, the parse tree would become:

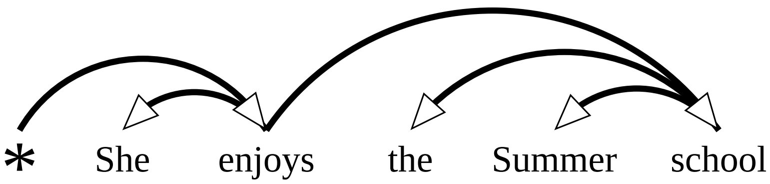
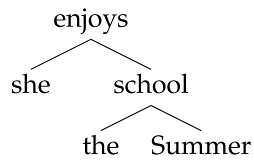


Figure 4.2: A dependency tree for the sentence *She enjoys the Summer school*. Note the additional dummy root symbol (*) which is included for convenience.



This representation is called a *dependency tree*; it can be alternatively represented as shown in Fig. 4.2. Dependency trees retain the lexical relationships involved in lexicalized phrase-based parse trees. However, they drop phrasal constituents, which render non-terminal nodes unnecessary. This has computational advantages (no grammar constant is involved in the complexity of the parsing algorithms) as well as design advantages (no grammar is necessary, and treebank annotations are way simpler, since no internal constituents need to be annotated). It also shifts the focus from internal syntactic structures and generative grammars (**?**) to lexical and transformational grammars (**????**). Arcs connecting words are called *dependency links* or *dependency arcs*. In an arc $\langle h, m \rangle$, the source word h is called the *head* and the target word m is called the *modifier*.

4.2.2 Projective and Non-projective Parsing

Dependency trees constructed using the method just described (*i.e.*, lexicalization of context-free phrase-based trees) always satisfy the following properties:

1. Each word (excluding the dummy root symbol) has exactly one parent.
2. The dummy root symbol has no parents.
3. There are no cycles.
4. The dummy root symbol has exactly one child.
5. All arcs are *projective*. This means that for any arc $\langle h, m \rangle$, all words in its span (*i.e.*, all words lying between h and m) are descendents from h (i.e. there is a directed path of dependency links from h to such word).

Conditions 1–3 ensure that the set of dependency links form a well-formed tree, rooted in the dummy symbol, which spans all the words of the sentence. Condition 4 requires that there is a single link departing from the root. Finally, a tree satisfying condition 5 is said *projective*: it implies that arcs cannot cross (*e.g.*, we cannot have arcs $\langle h, m \rangle$ and $\langle h', m' \rangle$ such that $h < h' < m < m'$).

In many languages (*e.g.*, those which have free-order) we would like to relax the assumption that all trees must be projective. Even in languages which have fixed word order (such as English) there are syntactic phenomena which are awkward to characterize using projective trees arising from the context-free assumption. Usually, such phenomena are characterized with additional linguistic artifacts (e.g., traces, Wh-movement, *etc.*). An example is the sentence (extracted from the Penn Treebank)

We learned a lesson in 1987 about volatility.

There, the prepositional phrase *in 1987* should be attached to the verb phrase headed by *learned* (since this is *when* we learned the lesson), but the other prepositional phrase *about volatility* should be attached to the noun phrase headed by *lesson* (since the *lesson* was about volatility). To explain such phenomenon, context-free grammars need to use additional machinery which allows words to be scrambled (in this case, via a movement transformation and the consequent insertion of a trace). In the dependency-based formalism, we can get rid of all those artifacts altogether by allowing *non-projective* parse trees. These are trees that satisfy conditions 1–3 above, but not necessarily conditions 4 or 5.[4] The dependency tree in Fig. 4.3 is non-projective: note that the arc $\langle lesson, about \rangle$ is not projective.

[4]It is also common to impose conditions 1–4, in which case the tree need not be projective, but it must have a single link departing from the root. The algorithms to be described below can be adapted for this case.

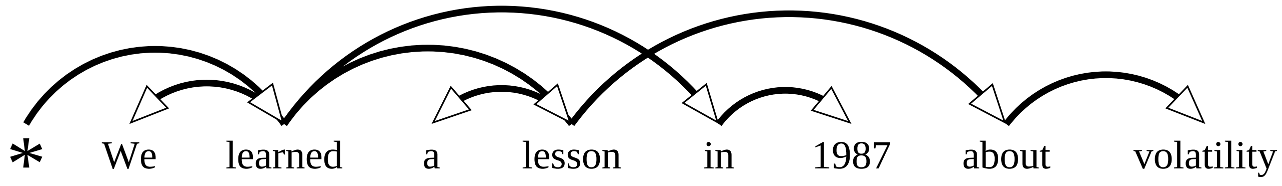


Figure 4.3: A non-projective parse tree.

We end this section by mentioning that dependency trees can have their arcs labeled, to provide more detailed syntactic information. For example, the arc $\langle enjoys, She\rangle$ could be labeled as `SUBJ` to denote that the modifier *She* has a subject function, and the arc $\langle enjoys, school\rangle$ could be labeled as `OBJ` to denote that the modifier *school* has an object function. For simplicity, we resort to *unlabeled* trees, which just convey the backbone structure. The cope with the labels, one can use either a joint model that infers the backbone and labels altogether, or to have a two-stage approach that first gets the backbone structure, and then the arc labels.

4.2.3 Algorithms for Projective Dependency Parsing

We now turn our attention to *algorithms* for obtaining a dependency parse tree. We start by considering a simple kind of models which are called *arc-factored*. These models assign a score $s_{\boldsymbol{\theta}}(h, m)$ to each possible arc $\langle h, m\rangle$ connecting a pair of words; they then score a particular dependency tree t by summing over the individual scores of the arcs that are present in the tree:

$$\text{score}_{\boldsymbol{\theta}}(t) = \sum_{\langle h,m\rangle \in t} s_{\boldsymbol{\theta}}(h, m).$$

As usual, from the point of view of the parsing algorithm, it does not matter whether the scores come from a generative or discriminative approach, and which features were used to compute the scores. The three important inference tasks are:

1. Obtain the tree with the largest score,
$$\hat{t} = \arg\max_t \text{score}_{\boldsymbol{\theta}}(t).$$

2. Compute the partition function (for a log-linear model),
$$Z(\mathbf{s}_{\boldsymbol{\theta}}) = \sum_t \exp(\text{score}_{\boldsymbol{\theta}}(t)),$$
where $\mathbf{s}_{\boldsymbol{\theta}}$ is short-hand notation for the set of all the $s_{\boldsymbol{\theta}}(h, m)$ coefficients.

3. Compute the posterior marginals for all the possible arcs (which for a log-linear model is the gradient of the log-partition function),
$$P_{\boldsymbol{\theta}}(\langle h, m\rangle \in t) = \frac{\partial \log Z(\mathbf{s}_{\boldsymbol{\theta}})}{\partial s_{\boldsymbol{\theta}}(h, m)}.$$

Exercise 4.3 ***(Warning: this exercise is somewhat complex. Feel free to think about it after the lab and ask your questions later!)*** *In* projective *dependency parsing using arc-factored models, the three tasks above can be solved in time $O(N^3)$. Sketch how the most likely dependency tree can be computed by "adapting" the CKY algorithm. (Hint: note that the CKY algorithm builds larger spans by combining smaller spans, and multiplies their weights by the weight of the corresponding production rule. In dependency parsing, each "span" is not represented by a constituent, but rather by the position of its lexical head. Convince yourself that this can only be either the leftmost or the rightmost position, and work out how the two spans can be combined.)*

*The instantiation of the CKY algorithm for projective dependency parsing is called Eisner's algorithm (**?**). Analogously, the partition function and the posterior marginals can be computed by adapting the inside-outside algorithm.*

4.2.4 Algorithms for Non-Projective Dependency Parsing

We turn our attention to *non-projective* dependency parsing. In that case, efficient solutions also exist for the three problems above; interestingly, they are based in combinatorial algorithms which are not related at all with dynamic programming:

- The first problem corresponds to finding the *maximum weighted directed spanning tree* in a directed graph. This problem is well known in combinatorics and can be solved in $O(N^3)$ using Chu-Liu-Edmonds' algorithm (**??**).[5] This has first been noted by **?**).

- The second and third problems can be solved by invoking another important result in combinatorics, the *matrix-tree theorem* (**?**). This fact has been noticed independently by **???**). The cost is that of computing a determinant and inverting a matrix, which can be done in time $O(N^3)$. The procedure is as follows. We first consider the directed weighted graph formed by including all the possible dependency links $\langle h, m\rangle$ (including the ones departing from the dummy root symbol, for which $h = 0$ by convention), along with weights given by $\exp(s_{\boldsymbol{\theta}}(h, m))$, and compute its $(N+1)$-by-$(N+1)$ Laplacian matrix $\boldsymbol{L}$ whose entries are:
$$L_{hm} = \begin{cases} \sum_{h'=0}^{N} \exp(s_{\boldsymbol{\theta}}(h', m)), & \text{if } h = m, \\ -\exp(s_{\boldsymbol{\theta}}(h, m))), & \text{otherwise.} \end{cases} \tag{4.2}$$
Denote by $\hat{L}$ the $(0, 0)$-minor of L, *i.e.*, the matrix obtained from L by removing the first row and column. Consider its determinant $\det \hat{L}$ and its inverse $\hat{L}^{-1}$. Then:

  – the partition function is given by
$$Z(\mathbf{s}_{\boldsymbol{\theta}}) = \det \hat{L}; \tag{4.3}$$

  – the posterior marginals are given by
$$P_{\boldsymbol{\theta}}(\langle h, m\rangle \in t) = \begin{cases} \exp(s_{\boldsymbol{\theta}}(h, m)) \cdot ([\hat{L}^{-1}]_{mm} - [\hat{L}^{-1}]_{mh}) & \text{if } h \neq 0 \\ \exp(s_{\boldsymbol{\theta}}(0, m)) \cdot [\hat{L}^{-1}]_{mm} & \text{otherwise.} \end{cases} \tag{4.4}$$

Exercise 4.4 *In this exercise you are going to experiment with arc-factored non-projective dependency parsers.*

*The CoNLL-X and CoNLL 2008 shared task datasets (**??**) contain dependency treebanks for 14 languages. In this lab, we are going to experiment with the Portuguese and English datasets. We preprocessed those datasets to exclude all sentences with more than 15 words; this yielded the files:*

- `data/deppars/portuguese_train.conll`,
- `data/deppars/portuguese_test.conll`,
- `data/deppars/english_train.conll`,
- `data/deppars/english_test.conll`.

1. *After importing all the necessary libraries, load the Portuguese dataset:*

```
import sys
sys.path.append("parsing/" )

import dependency_parser as depp

dp = depp.DependencyParser()
dp.read_data("portuguese")
```

   Observe the statistics which are shown. How many features are there in total?

2. *We will now have a close look on the* features *that can be used in the parser. Examine the file:*

   `lxmls/parsing/dependency_features.py`.

   The following method takes a sentence and computes a vector of features for each possible arc $\langle h, m\rangle$:

[5] There is a asymptotically faster algorithm by **?**) which solves the same problem in $O(N^2)$.

```python
def create_arc_features(self, instance, h, m, add=False):
  '''Creates features for arc h-->m.'''
```

We grouped the features in several subsets, so that we can conduct some ablation experiments:

- Basic *features that look only at the parts-of-speech of the words that can be connected by an arc;*
- Lexical *features that also look at these words themselves;*
- Distance *features that look at the length and direction of the dependency link* (i.e., *distance between the two words);*
- Contextual *features that look at the context (part-of-speech tags) of the words surrounding h and m.*

In the default configuration, only the basic features are enabled. The total number of features is the quantity observed in the previous question. With this configuration, train the parser by running 10 *epochs of the structured perceptron algorithm:*

```python
dp.train_perceptron(10)
dp.test()
```

What is the accuracy obtained in the test set? (Note: the shown accuracy is the fraction of words whose parent was correctly predicted.)

3. *Repeat the previous exercise by subsequently enabling the lexical, distance and contextual features:*

```python
dp.features.use_lexical = True
dp.read_data("portuguese")
dp.train_perceptron(10)
dp.test()

dp.features.use_distance = True
dp.read_data("portuguese")
dp.train_perceptron(10)
dp.test()

dp.features.use_contextual = True
dp.read_data("portuguese")
dp.train_perceptron(10)
dp.test()
```

For each configuration, write down the number of features and test set accuracies. Observe the improvements obtained when more features were added. Feel free to engineer new features!

4. *Which of the three important inference tasks discussed above (computing the most likely tree, computing the partition function, and computing the marginals) need to be performed in the structured perceptron algorithm? What about a maximum entropy classifier, with stochastic gradient descent? Check your answers by looking at the following two methods in* `code/dependency_parser.py`*:*

```python
def train_perceptron(self, n_epochs):
...

def train_crf_sgd(self, n_epochs, sigma, eta0 = 0.001):
...
```

Repeat the last exercise by training a maximum entropy classifier, with stochastic gradient descent, using $\lambda = 0.01$ *and a initial stepsize of* $\eta_0 = 0.1$*:*

```
dp.train_crf_sgd(10, 0.01, 0.1)
dp.test()
```

Compare the results with those obtained by the perceptron algorithm.

5. *Train a parser for English using your favourite learning algorithm:*

```
dp.read_data("english")
dp.train_perceptron(10)
dp.test()
```

The predicted trees are placed in the file `data/deppars/english_test.conll.pred`*. To get a sense of which errors are being made, you can check the sentences that differ from the gold standard (see the data in* `data/deppars/english_test.conll`*) and visualize those sentences,* e.g.*, in* `http://www.ark.cs.cmu.edu/treeviz/`*.*

4.2.5 Model Refinements

A number of refinements has been made that yield more accurate dependency parsers. We mention just a few:

Sibling and grandparent features. The arc-factored assumption fails to capture correlations between pairs of arcs. The dynamic programming algorithms for the *projective* case can be extended (at some additional cost) to handle features that look at consecutive sibling arcs on the same side of the head (*i.e.*, pairs of arcs of the form $\langle h, m\rangle$ and $\langle h, s\rangle$ with $h < m < s$ or $h > m > s$, such that no arc $\langle h, r\rangle$ exists with r between m and s. This has been done by **?**). Similarly, grandparents can also be accommodated with similar extensions (**?**). These are called "second-order models."

For the non-projective case, however, any extension beyond the arc-factored model becomes NP-hard (**?**). Yet, approximate algorithms have been proposed to handle "second-order models" that seem to work well: a greedy method (**?**), loopy belief propagation (**?**), a linear programming relaxation (**?**), and a dual decomposition method (**?**).

Third-order models. For the projective case, third order models have also been considered (**?**)

Transition-based parsers. Like in the phrase-based case, there is a totally different line of work which models parsers as a sequence of greedy shift-reduce decisions (**??**). These parsers seem to be very fast (expected linear time) and only slightly less accurate than the state-of-the-art. Solutions have been worked out for the non-projective case also (**?**).

Bibliography

Bertsekas, D., Homer, M., Logan, D., and Patek, S. (1995). *Nonlinear programming*. Athena Scientific.

Bishop, C. (2006). *Pattern recognition and machine learning*, volume 4. Springer New York.

Blitzer, J., Dredze, M., and Pereira, F. (2007). Biographies, bollywood, boom-boxes and blenders: Domain adaptation for sentiment classification. In *Annual Meeting-Association For Computational Linguistics*, volume 45, page 440.

Bottou, L. (1991). *Une Approche Theorique de l'Apprentissage Connexionniste: Applications a la Reconnaissance de la Parole*. PhD thesis.

Boyd, S. and Vandenberghe, L. (2004). *Convex optimization*. Cambridge Univ Pr.

Buchholz, S. and Marsi, E. (2006). CoNLL-X shared task on multilingual dependency parsing. In *Proc. of CoNLL*.

Carreras, X. (2007). Experiments with a higher-order projective dependency parser. In *Proc. of CoNLL*.

Charniak, E. (1997). Statistical parsing with a context-free grammar and word statistics. In *Proceedings of the National Conference on Artificial Intelligence*, pages 598–603. Citeseer.

Charniak, E., Johnson, M., Elsner, M., Austerweil, J., Ellis, D., Haxton, I., Hill, C., Shrivaths, R., Moore, J., Pozar, M., et al. (2006). Multilevel coarse-to-fine pcfg parsing. In *Proceedings of the main conference on Human Language Technology Conference of the North American Chapter of the Association of Computational Linguistics*, pages 168–175. Association for Computational Linguistics.

Chomsky, N. (1965). *Aspects of the Theory of Syntax*, volume 119. The MIT press.

Chu, Y. J. and Liu, T. H. (1965). On the shortest arborescence of a directed graph. *Science Sinica*, 14:1396–1400.

Cohen, S., Gimpel, K., and Smith, N. (2008). Logistic normal priors for unsupervised probabilistic grammar induction. In *In NIPS*. Citeseer.

Collins, M. (1999). *Head-driven statistical models for natural language parsing*. PhD thesis, University of Pennsylvania.

Collins, M. (2002). Discriminative training methods for hidden markov models: Theory and experiments with perceptron algorithms. In *Proceedings of the ACL-02 conference on Empirical methods in natural language processing-Volume 10*, pages 1–8. Association for Computational Linguistics.

Cover, T., Thomas, J., Wiley, J., et al. (1991). *Elements of information theory*, volume 6. Wiley Online Library.

Covington, M. (1990). Parsing discontinuous constituents in dependency grammar. *Computational Linguistics*, 16(4):234–236.

Crammer, K., Dekel, O., Keshet, J., Shalev-Shwartz, S., and Singer, Y. (2006). Online Passive-Aggressive Algorithms. *JMLR*, 7:551–585.

Crammer, K. and Singer, Y. (2002). On the algorithmic implementation of multiclass kernel-based vector machines. *The Journal of Machine Learning Research*, 2:265–292.

Duda, R., Hart, P., and Stork, D. (2001). *Pattern classification*, volume 2. Wiley New York.

Edmonds, J. (1967). Optimum branchings. *Journal of Research of the National Bureau of Standards*, 71B:233–240.

Eisner, J. (1996). Three new probabilistic models for dependency parsing: An exploration. In *Proceedings of the 16th conference on Computational linguistics-Volume 1*, pages 340–345. Association for Computational Linguistics.

Eisner, J. and Satta, G. (1999). Efficient parsing for bilexical context-free grammars and head automaton grammars. In *Proc. of ACL*.

Finkel, J., Kleeman, A., and Manning, C. (2008). Efficient, feature-based, conditional random field parsing. *Proceedings of ACL-08: HLT*, pages 959–967.

Henderson, J. (2003). Inducing history representations for broad coverage statistical parsing. In *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology-Volume 1*, pages 24–31. Association for Computational Linguistics.

Hopcroft, J., Motwani, R., and Ullman, J. (1979). *Introduction to automata theory, languages, and computation*, volume 3. Addison-wesley Reading, MA.

Huang, L. and Sagae, K. (2010). Dynamic programming for linear-time incremental parsing. In *Proc. of ACL*, pages 1077–1086.

Hudson, R. (1984). *Word grammar*. Blackwell Oxford.

Jaynes, E. (1982). On the rationale of maximum-entropy methods. *Proceedings of the IEEE*, 70(9):939–952.

Joachims, T. (2002). *Learning to Classify Text Using Support Vector Machines: Methods, Theory and Algorithms*. Kluwer Academic Publishers.

Johnson, M. (1998). Pcfg models of linguistic tree representations. *Computational Linguistics*, 24(4):613–632.

Klein, D. and Manning, C. (2002). A generative constituent-context model for improved grammar induction. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, pages 128–135. Association for Computational Linguistics.

Klein, D. and Manning, C. (2003). Accurate unlexicalized parsing. In *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics-Volume 1*, pages 423–430. Association for Computational Linguistics.

Koo, T. and Collins, M. (2010). Efficient third-order dependency parsers. In *Proc. of ACL*, pages 1–11.

Koo, T., Globerson, A., Carreras, X., and Collins, M. (2007). Structured prediction models via the matrix-tree theorem. In *Proc. EMNLP*.

Koo, T., Rush, A. M., Collins, M., Jaakkola, T., and Sontag, D. (2010). Dual decomposition for parsing with non-projective head automata. In *EMNLP*.

Lafferty, J., McCallum, A., and Pereira, F. (2001). Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Procs. of ICML*, pages 282–289.

Magerman, D. (1995). Statistical decision-tree models for parsing. In *Proceedings of the 33rd annual meeting on Association for Computational Linguistics*, pages 276–283. Association for Computational Linguistics.

Manning, C., Raghavan, P., and Schütze, H. (2008). *Introduction to information retrieval*, volume 1. Cambridge University Press Cambridge, UK.

Manning, C. and Schütze, H. (1999). *Foundations of statistical natural language processing*, volume 59. MIT Press.

Marcus, M., Marcinkiewicz, M., and Santorini, B. (1993). Building a large annotated corpus of English: The Penn Treebank. *Computational linguistics*, 19(2):313–330.

Martins, A. F. T., Smith, N. A., and Xing, E. P. (2009). Concise integer linear programming formulations for dependency parsing. In *Proc. of ACL-IJCNLP*.

McCallum, A., Freitag, D., and Pereira, F. (2000). Maximum entropy markov models for information extraction and segmentation. In *Proceedings of the Seventeenth International Conference on Machine Learning*, pages 591–598. Citeseer.

McDonald, R., Lerman, K., and Pereira, F. (2006). Multilingual dependency analysis with a two-stage discriminative parser. In *Proc. of CoNLL*.

McDonald, R. and Satta, G. (2007). On the complexity of non-projective data-driven dependency parsing. In *Proc. of IWPT*.

McDonald, R. T., Pereira, F., Ribarov, K., and Hajic, J. (2005). Non-projective dependency parsing using spanning tree algorithms. In *Proc. of HLT-EMNLP*.

Melčuk, I. (1988). *Dependency syntax: theory and practice*. State University of New York Press.

Mitchell, T. (1997). *Machine learning*.

Nivre, J. (2009). Non-projective dependency parsing in expected linear time. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP: Volume 1-Volume 1*, pages 351–359. Association for Computational Linguistics.

Nivre, J., Hall, J., Nilsson, J., Eryiğit, G., and Marinov, S. (2006). Labeled pseudo-projective dependency parsing with support vector machines. In *Procs. of CoNLL*.

Nocedal, J. and Wright, S. (1999). *Numerical optimization*. Springer verlag.

Pérez, F. and Granger, B. E. (2007). IPython: a System for Interactive Scientific Computing. *Comput. Sci. Eng.*, 9(3):21–29.

Petrov, S. and Klein, D. (2007). Improved inference for unlexicalized parsing. In *Human Language Technologies 2007: The Conference of the North American Chapter of the Association for Computational Linguistics; Proceedings of the Main Conference*, pages 404–411, Rochester, New York. Association for Computational Linguistics.

Petrov, S. and Klein, D. (2008a). Discriminative log-linear grammars with latent variables. In Platt, J., Koller, D., Singer, Y., and Roweis, S., editors, *Advances in Neural Information Processing Systems 20 (NIPS)*, pages 1153–1160, Cambridge, MA. MIT Press.

Petrov, S. and Klein, D. (2008b). Sparse multi-scale grammars for discriminative latent variable parsing. In *Proceedings of the 2008 Conference on Empirical Methods in Natural Language Processing*, pages 867–876, Honolulu, Hawaii. Association for Computational Linguistics.

Rabiner, L. (1989). A tutorial on hidden markov models and selected applications in speech recognition. *In Proc. IEEE*, 77(2):257–286.

Ratnaparkhi, A. (1999). Learning to parse natural language with maximum entropy models. *Machine Learning*, 34(1):151–175.

Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386.

Schölkopf, B. and Smola, A. J. (2002). *Learning with Kernels*. The MIT Press, Cambridge, MA.

Shalev-Shwartz, S., Singer, Y., and Srebro, N. (2007). Pegasos: Primal estimated sub-gradient solver for svm. In *ICML*.

Shannon, C. (1948). A mathematical theory of communication. *Bell Syst. Tech. Journ.*, 27(379):623.

Shawe-Taylor, J. and Cristianini, N. (2004). *Kernel Methods for Pattern Analysis*. CUP.

Smith, D. A. and Eisner, J. (2008). Dependency parsing by belief propagation. In *Proc. of EMNLP*.

Smith, D. A. and Smith, N. A. (2007). Probabilistic models of nonprojective dependency trees. In *Proc. EMNLP-CoNLL*.

Smith, N. and Eisner, J. (2005). Guiding unsupervised grammar induction using contrastive estimation. In *Proc. of IJCAI Workshop on Grammatical Inference Applications*. Citeseer.

Surdeanu, M., Johansson, R., Meyers, A., Màrquez, L., and Nivre, J. (2008). The CoNLL-2008 shared task on joint parsing of syntactic and semantic dependencies. *Proc. of CoNLL*.

Tarjan, R. (1977). Finding optimum branchings. *Networks*, 7(1):25–36.

Taskar, B., Klein, D., Collins, M., Koller, D., and Manning, C. (2004). Max-margin parsing. In *Proc. EMNLP*, pages 1–8.

Tesnière, L. (1959). *Eléments de syntaxe structurale*. Libraire C. Klincksieck.

Tutte, W. (1984). *Graph Theory*. Addison-Wesley, Reading, MA.

Vapnik, N. V. (1995). *The Nature of Statistical Learning Theory*. Springer-Verlag, New York.