LxMLS - Lab Guide

July 20, 2012

Day 0

Basic Tutorials

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

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

0.1 Python

0.1.1 Running Python

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

```
python yourfile.py
```

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

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

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

```
ipython --pylab
```

In some systems you may need to run the command lower-cased:

```
ipython -pylab
```

Help and Documentation

There are several ways to get help on IPython:

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

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

Profiling

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

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

%prun myfunction(22)
```

Exiting

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

0.1.2 Python by Example

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

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

Data Structures

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

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

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

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

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

Loops

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

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

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

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

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

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

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

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

Control Flow

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

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

Functions

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

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

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

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

```
greet(50)
Invalid hour: it should be between 0 and 24.
greet(-5)
Invalid hour: it should be between 0 and 24.
```

Indentation

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

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

```
1
2
3
```

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

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

0.1.3 Debugging in Python

Python is really easy to program after you get used to a few personality traits e.g. the obligatory spacing. During the lab sessions we will also be using the iPython iterative command line which allows you to execute a script command by command. This should limit the need for debugging tools. There will be situations however in which we will be using and extending modules which involve multiple classes and functions. To get a quick glimpse of the variable structures and what the code does, Python offers the pdb debugging module. In the lab we will be using an improved version of this module called ipdb, which incorporates various advantages like colored text. This module has multiple working modes but we be centering ourselves in the simplest yet most effective mode, just write

```
import ipdb;ipdb.set_trace()
```

at the position of the code that you want to inspect and run your code. The code will run normally as expected until it reaches this statement. Then the execution will stop and the pdb command line will open, for example

```
> ./lxmls-toolkit/lxmls/classifiers/gaussian_naive_bayes.py(27)train()
-> nr_x,nr_f = x.shape
(Pdb)
```

This would be the same as locating a breakpoint in any debugging environment. The difference is that we now navigate through the code from a command line rather than an editor. For this purpose there are a number of commands you can use. The complete list can be found here 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	Print a variable
(p)retty(p)rint	Print a variable, with line break (useful for lists)
(n)ext line	Jump to next line
(s)tep	Jump inside of the function we stopped at
c(ont(inue))	Continue execution until finding breakpoint or finishing
(r)eturn	Continue execution until current function returns
b(reak) n	Set a breakpoint in in line n
l(ist) [n], [m]	Print 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[1])
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 last command
expression	Executes python expression as if it were 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

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

```
(Pdb) n
> ./lxmls-toolkit/lxmls/classifiers/gaussian_naive_bayes.py(29)train()
-> classes = np.unique(y)
```

Now we can inspect the two variables we just created using the p(retty)p(rint) option

```
(Pdb) pp nr_x,nr_f
(50, 2)
```

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 crushes, as we can try different alternatives without the need to restart the code again.

0.1.4 Plotting in Python - Matplotlib

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

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

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

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

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

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

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

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

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

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

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

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

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

0.1.5 Numpy

Numpy is a library needed for scientific computing with Python.

Multidimensional Arrays

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

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

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

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

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

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

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

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

Mathematical Operations

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

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

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

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

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

```
import numpy as np

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

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

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

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

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

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

0.2 Essential Linear Algebra

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

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

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

$$Ax = b$$

with

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

0.2.1 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 floating point numbers. You can specify the particular array type by using the keyword dtype.*

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

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

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 matrix by looping over both indexes and multiplying the individual entries.*

```
a = np.array([[2,3],[3,4]])
b = np.array([[1,1],[1,1]])
a_dim1, a_dim2 = a.shape
b_dim1, b_dim2 = b.shape
c = np.zeros([a_dim1,b_dim2])
for i in xrange(a_dim1):
    for j in xrange(b_dim2):
        for k in xrange(a_dim2):
            c[i,j] += a[i,k]*b[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 $(xy^T)_{ij} = x_i y_j$,

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

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

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

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

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

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

print I
print np.dot(I,x)

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

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

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

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

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

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

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

0.2.4 Linear Independence and Rank

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

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

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 $A \in R^{m \times n}$, $\text{rank}(A) \leq \min(m, n)$. If $\text{rank}(A) = \min(m, n)$, then A is said to be **full rank**.
- For $A \in R^{m \times n}$, $\text{rank}(A) = \text{rank}(A^T)$.
- For $A \in R^{m \times n}$, $B \in R^{n \times p}$, $\text{rank}(AB) \leq \min(\text{rank}(A), \text{rank}(B))$.
- For $A, B \in R^{m \times n}$, $\text{rank}(A + B) \leq \text{rank}(A) + \text{rank}(B)$.
- Two vectors $x, y \in \mathbb{R}^n$ 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

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

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 $A_1, \ldots, A_n$ is a sequence of sets then $\bigcup_{i=1}^{n} A_i = \{\omega \in \mathcal{X} \mid \omega \in A_i \text{ for at least one i}\}$. We say that $A_1, \ldots, A_n$ are **disjoint** or **mutually exclusive** if $A_i \cap A_j = \varnothing$ whenever $i \neq j$.

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

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

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

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

One can derive many properties of P from these axioms:

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

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

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

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

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

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

for every finite subset J of I.

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

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

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

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

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

Using Equations **??** and **??**, one can derive the famous Bayes' theorem.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

A discussion of a few important distributions and related properties:

0.3.2 Bernoulli

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

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

0.3.3 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

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

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

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

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

0.3.5 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 for the respective outcome.

0.3.6 Gaussian Distribution

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

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

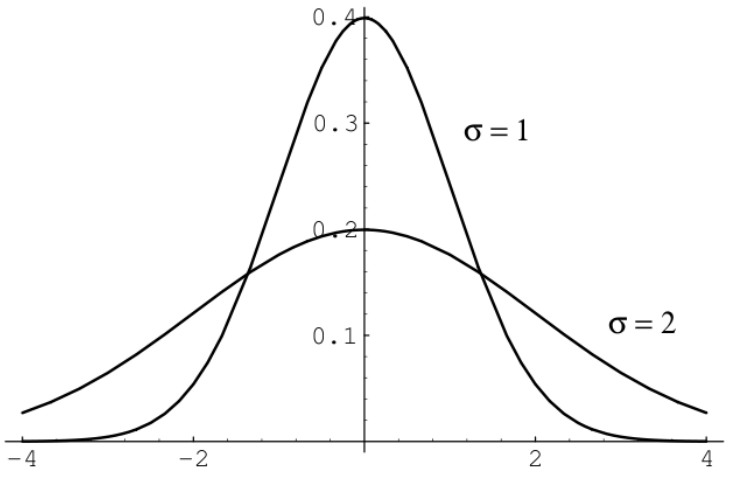


Figure 1: Normal density for two sets of parameter values.

Fig. **??**, 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 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 and count the number of heads h and set $p = \frac{h}{n}$. In doing this, the person is using the data x to estimate θ.

With this in mind, we will know generalize this process. We define the probability $p(\theta|x)$, which is the probability of the parameter θ, given the data x.

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

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

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

To understand this concept better, we go back to the tampered coin example again. Suppose that we throw the coin 5 times and get the sequence [1,1,1,1,1] (1=head, 0=tail). Using the Bernoulli distribution **??** 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 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 compute a sequence of points $x^{(0)}, x^{(1)}, \ldots \in \text{domain}(f)$ such that hopefully $x^t = x^*$. Such a sequence is called the **minimizing sequence** for the problem.

0.4.1 Convex Functions

One important 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 **??** and **??** 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 **??**, 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 **??** 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

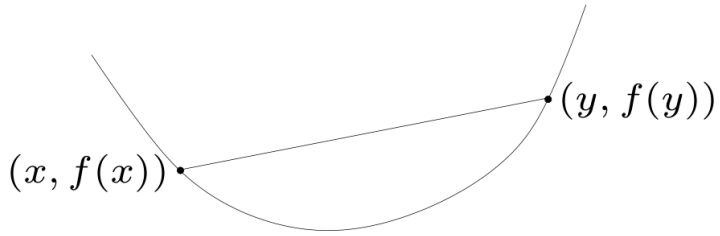


Figure 2: Illustration of a convex function. The line segment between any two points on the graph lies entirely above the curve.

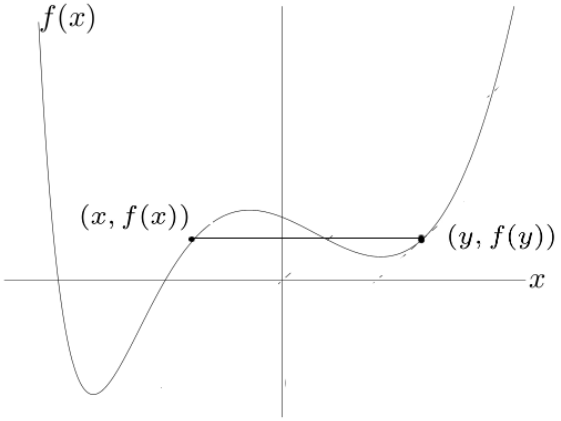


Figure 3: Illustration of a non-convex function. Note the line segment intersecting the curve.

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 function in that interval.

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

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

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 **??** shows derivatives of some functions that we will be using during the school.
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* $h(x) = exp(x^2)$.
$h(x) = f(g(x)) = f(u) = exp(u)$, *where* $u = g(x) = x^2$.
$\frac{\partial h}{\partial x} = \frac{\partial f}{\partial u} \cdot \frac{\partial u}{\partial x} = exp(u) \cdot 2x = exp(x^2) \cdot 2x$

Example 0.4 *Consider the function* $f(x) = x^2$ *and its derivative* $\frac{\partial f}{\partial x}$. *Look at the derivative of that function at points [-2,0,2], draw the tangent to the graph in that point.*
$\frac{\partial f}{\partial x}(-2) = -4$, $\frac{\partial f}{\partial x}(0) = 0$, *and* $\frac{\partial f}{\partial x}(2) = 4$
For example, the tangent equation for $x = -2$ *is* $y = -4x - b$, *where* $b = f(-2)$
The following code plots the function and the derivatives on those points using matplotlib (See Figure **??***).*

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

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

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

The **gradient** of a function is a generalization of the derivative concept we just saw before, for several dimensions. Lets assume we have a function $f(x)$ where $x \in \mathbb{R}^2$, so 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

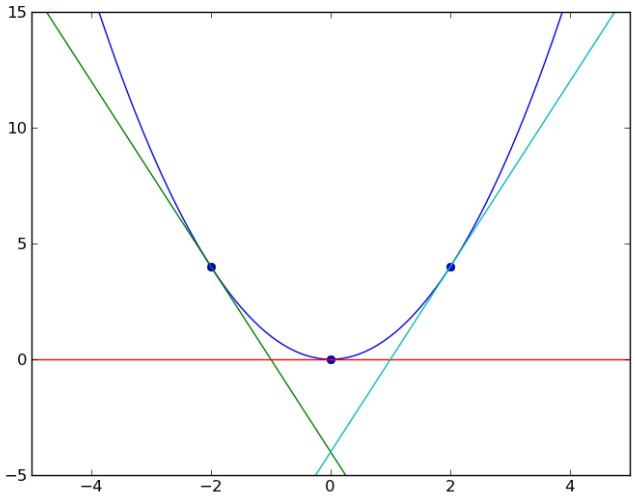


Figure 4: Illustration of the gradient of the function $f(x^2)$ at three different points $x = [-2, 0.2]$. Note that at point $x = 0$ the gradient is zero which corresponds to the minimum of the function.

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

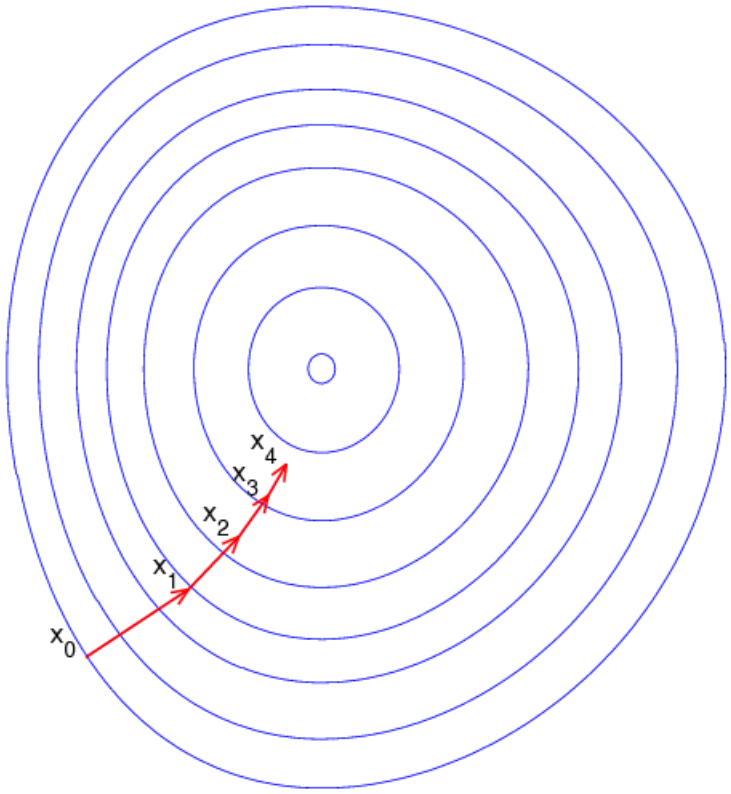


Figure 5: Illustration of gradient descent. The blue circles correspond to contours of the function (each blue circle is a set of points which have the same function value), while the red lines correspond to steps taken in the negative gradient direction.

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

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

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

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

Figure **??** *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.

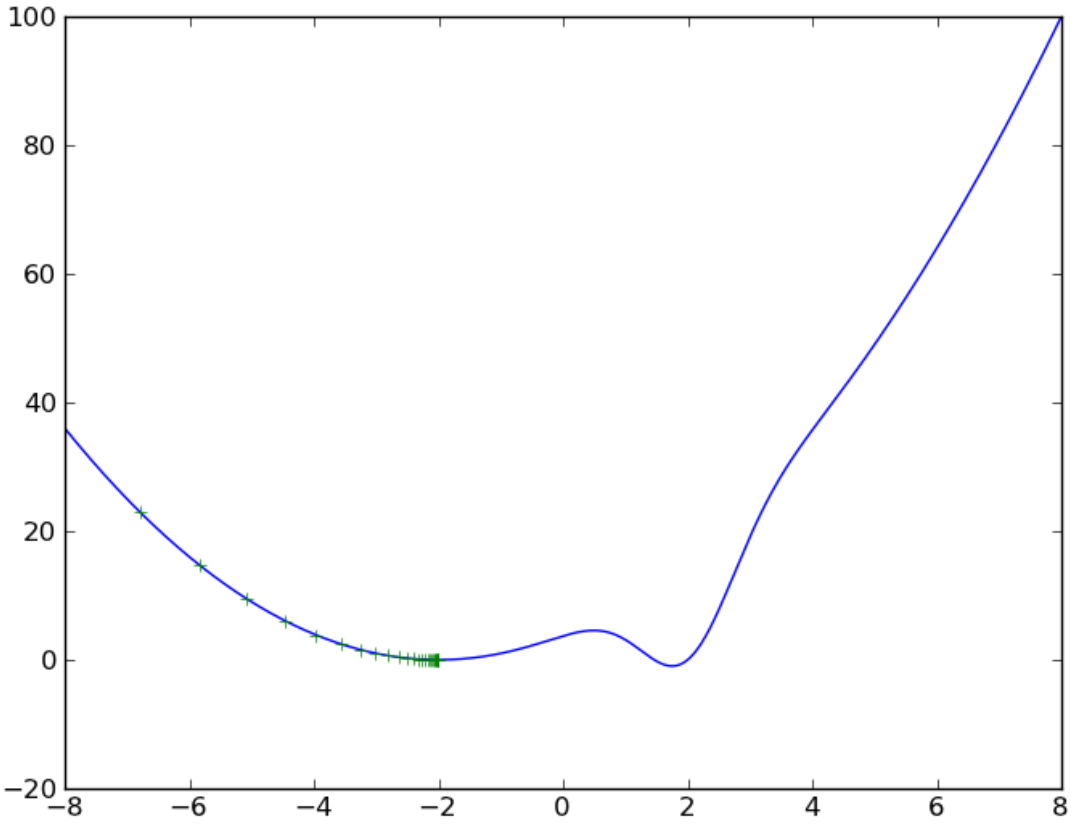


Figure 6: Example of running gradient descent starting on point $x_0 = -8$ for function $f(x) = (x+2)^2 - 16\exp\left(-(x-2)^2\right)$. The function is represented in blue, while the points of the minimizing sequence are displayed as green plus signs.

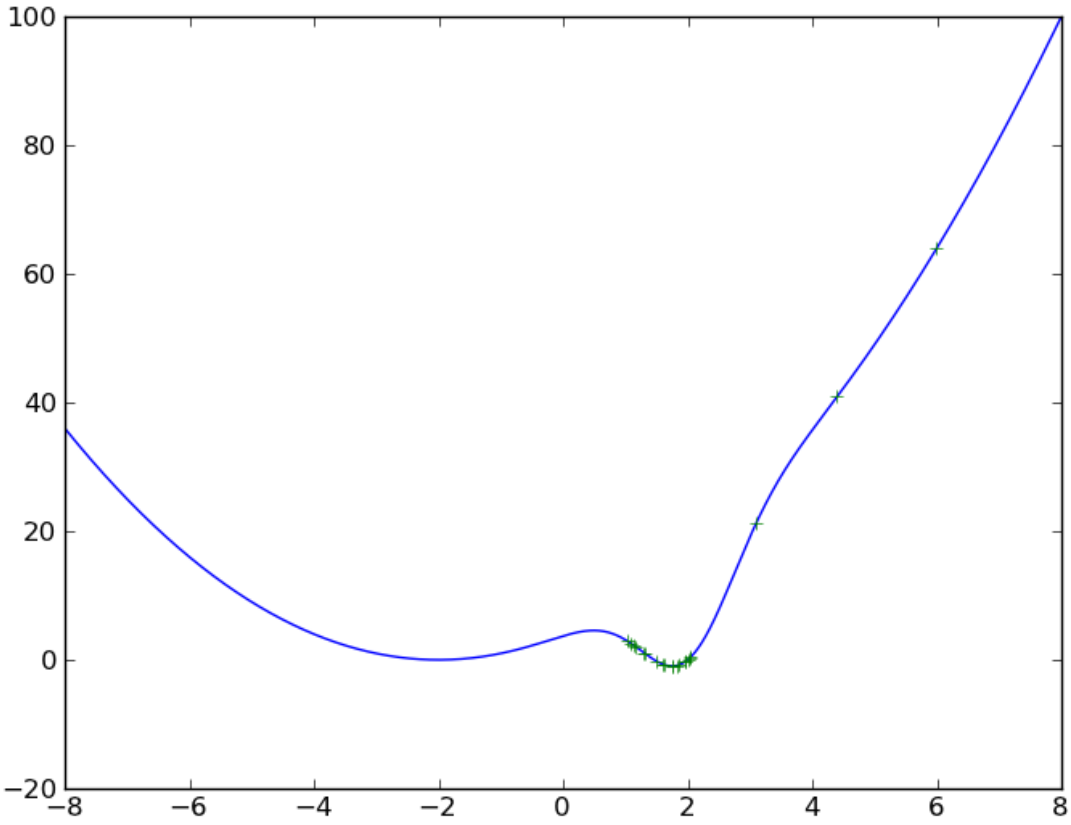


Figure 7: Example of running gradient descent starting on point $x_0 = 8$ for function $f(x) = (x+2)^2 - 16\exp\left(-(x-2)^2\right)$. The function is represented in blue, while the points of the minimizing sequence are displayed as green plus signs.

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).

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:

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

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 input-output pairs (called *examples* or *instances*):

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

The goal of (supervised) machine learning is to use $\mathcal{D}$ 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., $\hat{y} = h(x)$.

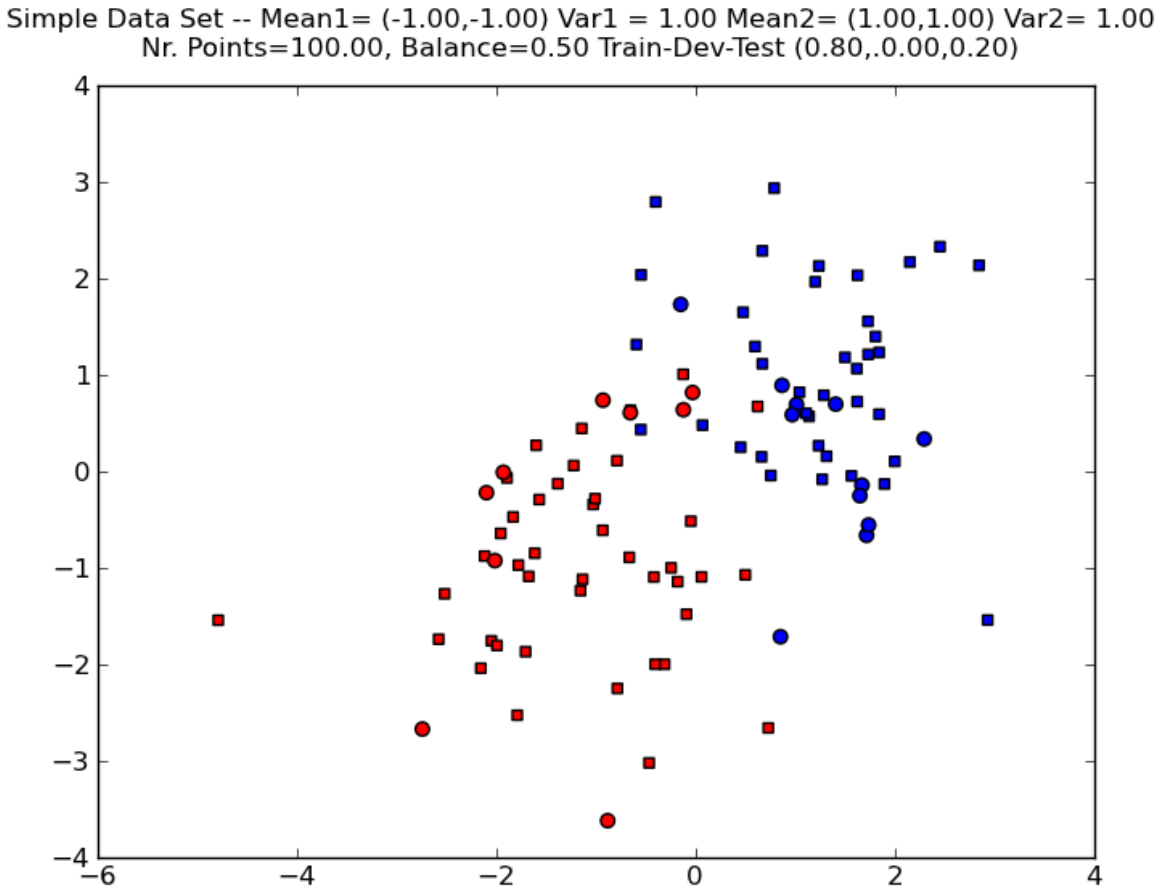


Figure 1.1: Example of a dataset. The input set consists in points in the real plane, $\mathcal{X} = \mathbb{R}^2$, and the output set consists of two classes (Red and Blue). Training points are represented as squares, while test points are represented as circles.

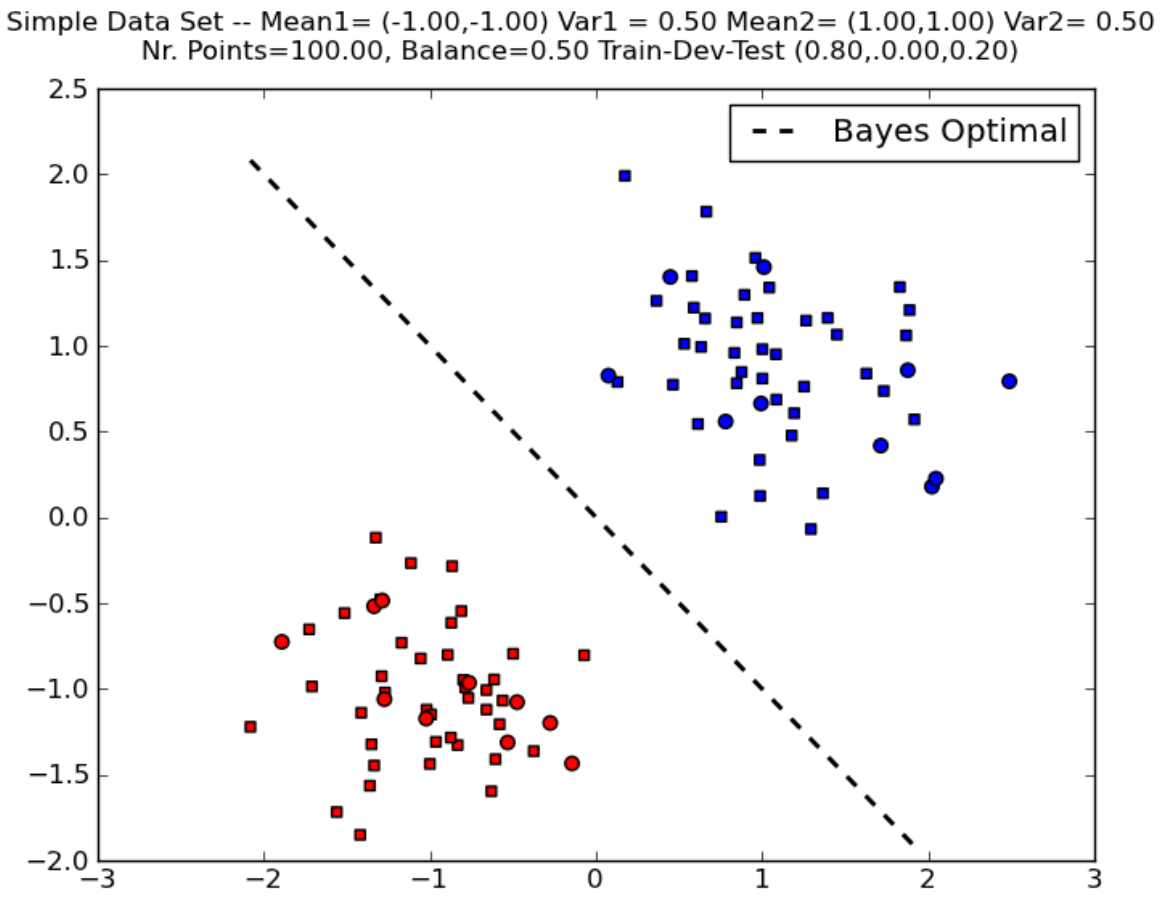


Figure 1.2: Example of a dataset together with the corresponding Bayes optimal decision boundary. The input set consists in points in the real plane, $\mathcal{X} = \mathcal{R}$, and the output set consists of two classes (Red and Blue). Training points are represented as squares, while test points are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

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

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

where in † we used the fact that $P(x)$ is constant with respect to y. The probability distributions $P(Y)$ and $P(X|Y)$ are respectively called the *class prior* and the *class conditionals*. Figure **??** shows an example of the Bayes optimal decision boundary for a toy example. Generative models assume 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 is usually called training time.

At test time, 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 at training time. 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 (in particular 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.

The *maximum likelihood criterion* aims to maximize the probability of the training sample, assuming it was generated iid. This probability (call it $P(\mathcal{D})$) factorizes as

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

1.2.1 Example: 2-D Gaussians

We first illustrate the naïve Bayes assumption with a toy example. Suppose that $\mathcal{X} = \mathbb{R}^2$ and $\mathcal{Y} = \{1, 2\}$. Assume that each class-conditional is a two-dimensional Gaussian distribution with fixed covariance, i.e., $P(X_1, X_2|Y = y) = \mathcal{N}(\boldsymbol{\mu}_y, \boldsymbol{\Sigma}_y)$.

According to the naïve Bayes assumption, $\hat{P}(X_1, X_2|Y) = \hat{P}(X_1|Y)\hat{P}(X_2|Y)$ (remark: this is equivalent to assuming that the $\boldsymbol{\Sigma}_y$ are diagonal!). For simplicity, we also assume that the two classes have unit variance. Then, we have $\hat{P}(X_1|Y = y) = \mathcal{N}(\mu_{y1}, 1.0)$ and $\hat{P}(X_2|Y = y) = \mathcal{N}(\mu_{y2}, 1.0)$. Figure **??** shows an example a dataset of two gaussians with unit variance, where $\mu_{y1} = [-1, -1]$ and $\mu_{y1} = [1, 1]$. Figure **??** shows the same example but where both gaussians have $\Sigma = 0.5$, together with the Bayes optimal decision boundary. The parameters that need to be estimated are the class-conditional means $\mu_{11}, \mu_{12}, \mu_{21}, \mu_{22}$ and the class priors $\hat{P}(Y = 1)$ and $\hat{P}(Y = 2)$. Given a training sample $\mathcal{D} = \{(x^1, y^1), \ldots, (x^M, y^M)\}$, denote by $\mathcal{I}_1 \subseteq \{1, \ldots, M\}$ the indices of those instances belonging to class 1, and by $\mathcal{I}_2 \subseteq \{1, \ldots, M\}$ the indices of the ones that belong to class 2. The maximum likelihood estimates of the quantities above are:

$$\begin{aligned} \hat{P}(Y = 1) = \frac{|\mathcal{I}_1|}{M}, \quad \hat{P}(Y = 2) = \frac{|\mathcal{I}_2|}{M} \\ \mu_{11} = \frac{1}{|\mathcal{I}_1|} \sum_{m \in \mathcal{I}_1} x_1^m, \quad \mu_{12} = \frac{1}{|\mathcal{I}_1|} \sum_{m \in \mathcal{I}_1} x_2^m \\ \mu_{21} = \frac{1}{|\mathcal{I}_2|} \sum_{m \in \mathcal{I}_2} x_1^m, \quad \mu_{22} = \frac{1}{|\mathcal{I}_2|} \sum_{m \in \mathcal{I}_2} x_2^m. \end{aligned} \tag{1.8}$$

In words: the class priors' estimates are their relative frequencies, and the class-conditional means' estimates are the sample means.

Exercise 1.1 *Start by importing all the libraries necessary for this lab through the following preamble:*

```
import sys
import matplotlib.pyplot as plt
sys.path.append("readers/")
sys.path.append("classifiers/")
sys.path.append("distributions/")
sys.path.append("util/")

import simple_data_set as sds
import linear_classifier as lcc
import gaussian_naive_bayes as gnbc
import naive_bayes as nb
```

Now, generate a training and a test dataset like in the previous example, each with $M = 100$ *points,* 50 *of each class. Assume the following class-conditionals:* $P(X|Y = 1) \sim N((-1,-1), \sigma^2\boldsymbol{I})$ *and* $P(X|Y = 2) \sim N((1,1), \boldsymbol{I})$*, for* $\sigma = 1.0$*. To do this, run the following command from the* `code` *directory:*

```
sd = sds.SimpleDataSet(nr_examples=100, g1 = [[-1,-1],1], g2 = [[1,1],1], balance=0.
    5, split=[0.5, 0, 0.5])
```

You can visualize your data and see the Bayes optimal surface boundary by typing:

```
fig,axis = sd.plot_data()
```

Note: you might need to type `plt.show()` *in order to show the figure.*

Now, run naïve Bayes on this dataset. To do that, use the class `GaussianNaiveBayes`*, which is defined in the file* `GaussianNaiveBayes.py` *under the classification directory. Report your estimates, as well as training set and testing set accuracies:*

```
gnb = gnbc.GaussianNaiveBayes()
params_nb_sd = gnb.train(sd.train_X, sd.train_y)

print "Estimated Means"
print gnb.means
print "Estimated Priors"
print gnb.prior
y_pred_train = gnb.test(sd.train_X,params_nb_sd)
acc_train = gnb.evaluate(sd.train_y, y_pred_train)
y_pred_test = gnb.test(sd.test_X,params_nb_sd)
acc_test = gnb.evaluate(sd.test_y, y_pred_test)
print "Gaussian Naive Bayes Simple Dataset Accuracy train: %f test: %f"%(acc_train,
    acc_test)
```

To visualize the surface boundary estimated by naïve Bayes, type:

```
fig,axis = sd.add_line(fig,axis,params_nb_sd,"Naive Bayes","red")
```

Do not worry for now about why the surface boundaries look the way they look. This is going to be the subject of §??.

Repeat the exercise above for different values of σ^2*, different balances, and different sample sizes. What do you observe?*

1.2.2 Example: Multinomial Model 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 $\mathcal{Y} = \{c_1, \ldots, c_K\}$ is a set of *topics* 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 multiset 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*.[1] 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.[2] 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—*tokens*—$w_{j_1}, \ldots, w_{j_L}$) is

$$P(x|y) = \prod_{l=1}^{L} P(w_{j_l}|y) = \prod_{j=1}^{J} P(w_j|y)^{n_j(x)}, \tag{1.9}$$

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 $\hat{P}(c_1), \ldots, \hat{P}(c_K)$, and $\hat{P}(w_j|c_k)$ 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:

$$\hat{P}(c_k) = \frac{|\mathcal{I}_k|}{M}, \qquad \hat{P}(w_j|c_k) = \frac{\sum_{m \in \mathcal{I}_k} n_j(x^m)}{\sum_{i=1}^{J} \sum_{m \in \mathcal{I}_k} n_i(x^m)}. \tag{1.10}$$

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.

Exercise 1.2 *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:*

```
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. *Create a file* `MultinomialNaiveBayes.py` *and implement the naïve Bayes with the multinomial model in a new class* `MultinomialNaiveBayes`*. (Hint: look at the implementation of* `GaussianNaiveBayes` *for inspiration).*

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

```
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)
```

[1] Another popular model for documents is the Bernoulli model, which only looks at the presence/absence of a word in a document, rather than word frequency. See **??**) for further information.

[2] 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.

```
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.* **??** *for the estimation of the conditional probabilities by*

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

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 Features and Linear Classifiers

In the previous section, we assumed a particular representation for the input objects $x \in \mathcal{X}$: points in a 2D Euclidean space (in the Gaussian example) and bag-of-words representations of text documents (in the sentiment data example).

The methods discussed in this lecture are also applicable to a wide range of problems, regardless of the intricacies of our input objects. 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 instance in the toy examples described above you can think of $\mathcal{X}$ as a set of points and the features to be its 2D coordinates. Let $g_1(x), \ldots, g_J(x)$ be J features of x. We call the vector

$$\boldsymbol{g}(x) = (g_1(x), \ldots, g_J(x)) \tag{1.11}$$

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:

$$g_1(x) \triangleq \begin{cases} 1, & \text{if sentence } x \text{ contains the word } \textit{Ronaldo} \\ 0, & \text{otherwise.} \end{cases} \tag{1.12}$$

$$g_2(x) \triangleq \begin{cases} 1, & \text{if all words in sentence } x \text{ are capitalized} \\ 0, & \text{otherwise.} \end{cases} \tag{1.13}$$

$$g_3(x) \triangleq \begin{cases} 1, & \text{if } x \text{ contains any of the words } \textit{amazing}, \textit{excellent} \text{ or } \textit{:-)} \\ 0, & \text{otherwise.} \end{cases} \tag{1.14}$$

In this example, the feature vector representation of the sentence x

> Ronaldo kicked the ball and scored 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:

$$f_1(x,y) \triangleq \begin{cases} 1, & \text{if } x \text{ contains } \textit{Ronaldo}, \text{ and topic } y \text{ is } \texttt{sport} \\ 0, & \text{otherwise.} \end{cases} \tag{1.15}$$

$$f_2(x,y) \triangleq \begin{cases} 1, & \text{if } x \text{ contains } \textit{Ronaldo}, \text{ and topic } y \text{ is } \texttt{politics} \\ 0, & \text{otherwise.} \end{cases} \tag{1.16}$$

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

$$\begin{aligned} \boldsymbol{f}(x,y) &\triangleq \boldsymbol{g}(x) \otimes \boldsymbol{e}_y \\ &= (0,\ldots,0,\underbrace{\boldsymbol{g}(x)}_{y\text{th slot}},0,\ldots,0) \end{aligned} \tag{1.17}$$

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 $D = JK$.

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

$$\hat{y} = \arg\max_{y \in \mathcal{Y}} \boldsymbol{w} \cdot \boldsymbol{f}(x,y). \tag{1.18}$$

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.* **??***), and decomposing the weight vector as* $\boldsymbol{w} = (\boldsymbol{w}_{c_1},\ldots,\boldsymbol{w}_{c_K})$*, we have that*

$$\boldsymbol{w} \cdot \boldsymbol{f}(x,y) = \boldsymbol{w}_y \cdot \boldsymbol{g}(x). \tag{1.19}$$

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 (in fact, so is the 2-D Gaussian Bayes classifier—try to show this). 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 model for document classification this is equivalent to:*

$$\begin{aligned} \hat{y} &= \arg\max_{y \in \mathcal{Y}} \log \hat{P}(y) + \log \hat{P}(x|y) \\ &= \arg\max_{y \in \mathcal{Y}} \log \hat{P}(y) + \sum_{j=1}^{J} n_j(x) \log \hat{P}(w_j|y) \\ &= \arg\max_{y \in \mathcal{Y}} \boldsymbol{w}_y \cdot \boldsymbol{g}(x), \end{aligned} \tag{1.20}$$

where

$$\begin{aligned} \boldsymbol{w}_y &= \left(b_y, \log \hat{P}(w_1|y),\ldots,\log \hat{P}(w_J|y)\right) \\ b_y &= \log \hat{P}(y) \\ \boldsymbol{g}(x) &= (1, n_1(x),\ldots,n_J(x)). \end{aligned} \tag{1.21}$$

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

$$\hat{y} = \arg\max_{y \in \mathcal{Y}} \boldsymbol{w}_y \cdot \boldsymbol{g}(x). \tag{1.22}$$

Exercise 1.3 *Show that the Gaussian naïve Bayes classifier with shared and given variance is also a linear classifier, and derive the formulas for* $\boldsymbol{w}_y$*,* b_y*. You should obtain the formulas that are implemented in the* `train` *method of* `GaussianNaiveBayes`*.*

Look again at the decision boundary that you have found in Exercise **??** *and compare it with the Bayes optimal classifier.*

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 $\hat{\boldsymbol{w}} \leftarrow \frac{1}{t}\sum_{i=1}^{t} \boldsymbol{w}^i$

1.4 Online Algorithms: Perceptron and MIRA

1.4.1 Perceptron

Perhaps the oldest algorithm to train a linear classifier is the *perceptron* (**?**), which we depict as Alg. **??**.[3]

The perceptron algorithm works as follows: at each round, it takes an input datum, 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. **??** 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, $\boldsymbol{w}$ 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 under this assumption, the perceptron will eventually reach a separating hyperplane $\boldsymbol{w}$.

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

Exercise 1.4 *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:*

```
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.*

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

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 $\hat{\boldsymbol{w}} \leftarrow \frac{1}{t}\sum_{i=1}^{t} \boldsymbol{w}^i$

1.4.2 Margin Infused Relaxed Algorithm (MIRA)

The MIRA algorithm (**??**) has achieved very good performance in NLP problems. At each round t, MIRA updates the weight vector by solving the following optimization problem:

$$\boldsymbol{w}^{t+1} \leftarrow \arg\min_{\boldsymbol{w}, \xi} \quad \xi + \tfrac{\lambda}{2}\|\boldsymbol{w} - \boldsymbol{w}^t\|^2 \tag{1.23}$$

$$\text{s.t.} \quad \boldsymbol{w} \cdot \boldsymbol{f}(x^m, y^m) \geq \boldsymbol{w} \cdot \boldsymbol{f}(x^m, \hat{y}) + 1 - \xi \tag{1.24}$$

$$\xi \geq 0, \tag{1.25}$$

where $\hat{y} = \arg\max_{y' \in \mathcal{Y}} \boldsymbol{w}^t \cdot \boldsymbol{f}(x^m, y')$ is the prediction using the model with weight vector $\boldsymbol{w}^t$. By inspecting Eq. **??** we see that MIRA attempts to achieve a tradeoff between *conservativeness* (penalizing large changes from the previous weight vector via the term $\frac{\lambda}{2}\|\boldsymbol{w} - \boldsymbol{w}^t\|^2$) and *correctness* (by requiring, through the constraints, that the new model $\boldsymbol{w}^{t+1}$ "separates" the true output from the prediction with a margin (although slack $\xi \geq 0$ is allowed).[4] Note that, if the prediction is correct ($\hat{y} = y^m$) the solution of the problem Eq. **??** leaves the weight vector unchanged ($\boldsymbol{w}^{t+1} = \boldsymbol{w}^t$). This quadratic programming problem has a closed form solution:[5]

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

with

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

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):

$$\rho(\hat{y}, y) = \begin{cases} 1 & \text{if } \hat{y} \neq y \\ 0 & \text{otherwise.} \end{cases} \tag{1.26}$$

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

Exercise 1.5 *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*:*

[4] The intuition for this large margin separation is the same for support vector machines, which will be discussed in §**??**.
[5] Note that the perceptron updates are identical, except that we always have $\eta_t = 1$.

```
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.

1.5 Discriminative Classifiers: Maximum Entropy and Support Vector Machines

Unlike the naïve Bayes classifier, the algorithms described in the last section (perceptron and MIRA) directly focus on finding a separating hyperplane to discriminate among the classes, rather than attempting to model the probability $P(X,Y)$ that generates the data. This kind of methods are called *discriminative* (by opposition to the *generative* ones). This section presents two important discriminative classifiers, with widespread use in NLP applications: maximum entropy and support vector machines.

1.5.1 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 (**?**). For an excellent textbook on Information Theory, we recommend **?**). 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, trying to making uncertainty about the model as large as possible. For example, if we have a dice 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 partition the set of possible outcomes in two groups and are only told about how many times outcomes on each of the groups have occurred. If we know 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`. A fundamental result is that the maximum entropy distribution $P_{\boldsymbol{w}}(Y|X)$ under *first moment matching constraints* (which 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 $\frac{1}{M}\sum_m \boldsymbol{f}(x_m,y_m)$) is a *log-linear model*. The dual of that optimization problem is that of maximizing likelihood in a log-linear model (in the binary case, called *logistic regression* model).

The maximum entropy distribution[6] has the following parametric form:

$$P_{\boldsymbol{w}}(y|x) = \frac{\exp(\boldsymbol{w}\cdot\boldsymbol{f}(x,y))}{Z(\boldsymbol{w},x)} \tag{1.27}$$

[6] Also called a log-linear model, a Boltzmann distribution, or an exponential family of distributions.

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

$$Z(\boldsymbol{w}, x) = \sum_{y' \in \mathcal{Y}} \exp(\boldsymbol{w} \cdot \boldsymbol{f}(x, y')). \tag{1.28}$$

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

$$\begin{aligned} \nabla_w \log Z(\boldsymbol{w}, x) &= E_{\boldsymbol{w}}[\boldsymbol{f}(x, Y)] \\ &= \sum_{y' \in \mathcal{Y}} P_{\boldsymbol{w}}(y'|x) \boldsymbol{f}(x, y'). \end{aligned} \tag{1.29}$$

Maximum entropy models are trained *discriminatively*: this means that, instead of maximizing the *joint* likelihood $P_{\boldsymbol{w}}(x^1, \ldots, x^M, y^1, \ldots, y^M)$ (like generative approaches, such as naïve Bayes, do), one maximizes directly the *conditional* likelihood $P_{\boldsymbol{w}}(y^1, \ldots, y^M | x^1, \ldots, x^M)$. The rationale is that one does not need to worry about modeling the input variables if all we want is an accurate estimate of $P(Y|X)$, which is what matters for prediction. The average conditional log-likelihood is:

$$\begin{aligned} \mathcal{L}(\boldsymbol{w}; \mathcal{D}) &= \frac{1}{M} \log P_{\boldsymbol{w}}(y^1, \ldots, y^M | x^1, \ldots, x^M) \\ &= \frac{1}{M} \log \prod_{m=1}^{M} P_{\boldsymbol{w}}(y^m | x^m) \\ &= \frac{1}{M} \sum_{m=1}^{M} \log P_{\boldsymbol{w}}(y^m | x^m) \\ &= \frac{1}{M} \sum_{m=1}^{M} \left(\boldsymbol{w} \cdot \boldsymbol{f}(x^m, y^m) - \log Z(\boldsymbol{w}, x^m) \right). \end{aligned} \tag{1.30}$$

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

$$\begin{aligned} \hat{\boldsymbol{w}} &= \arg\max_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{w}; \mathcal{D}) - \frac{\lambda}{2} \|\boldsymbol{w}\|^2 \\ &= \arg\min_{\boldsymbol{w}} -\mathcal{L}(\boldsymbol{w}; \mathcal{D}) + \frac{\lambda}{2} \|\boldsymbol{w}\|^2. \end{aligned} \tag{1.31}$$

Here we use the squared L_2-norm as the regularizer,[7] 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 **??**). 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. **??**. This function is convex, which implies that a local optimum of Eq. **??** is also a global optimum. $F_\lambda(\boldsymbol{w}; \mathcal{D})$ is also differentiable: its gradient is

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

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

$$\begin{aligned} \boldsymbol{w}^{t+1} &\leftarrow \boldsymbol{w}^t - \eta_t \nabla_{\boldsymbol{w}} F_\lambda(\boldsymbol{w}^t; \mathcal{D}) \\ &= (1 - \lambda \eta_t) \boldsymbol{w}^t + \eta_t \frac{1}{M} \sum_{m=1}^{M} \left(\boldsymbol{f}(x^m, y^m) - E_{\boldsymbol{w}}[\boldsymbol{f}(x^m, Y)] \right). \end{aligned} \tag{1.33}$$

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

[7] 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

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**
  for $m = 1$ **to** M **do**
    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:

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

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

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

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

methods exist for optimizing Eq. **??**, 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. **??**. 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 1.6 *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`*.*

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")
```

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?*

Algorithm 5 SGD for Maximum Entropy

1: **input:** $\mathcal{D}$, λ, number of rounds T,
learning rate sequence $(\eta_t)_{t=1,\ldots,T}$
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 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: update the model:

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

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

```
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)
```

1.5.2 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 $P(X, Y)$), 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

Algorithm 6 Stochastic Subgradient Descent for SVMs

1: **input:** $\mathcal{D}$, λ, number of rounds T,
learning rate sequence $(\eta_t)_{t=1,\ldots,T}$
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 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:** $\hat{\boldsymbol{w}} \leftarrow \boldsymbol{w}^{T+1}$

formulation here.[8]

Consider $\rho(y', y)$ as a non-negative cost function. For simplicity, we focus here on the $0/1$-cost defined by Equation **??** (but keep in mind that other cost functions are possible). The *hinge loss*[9] is the function

$$\ell(\boldsymbol{w}; x, y) = \max_{y' \in \mathcal{Y}} \boldsymbol{w} \cdot \boldsymbol{f}(x, y') - \boldsymbol{w} \cdot \boldsymbol{f}(x, y) + \rho(y', y). \tag{1.34}$$

Note that the objective of Eq. **??** 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.

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

$$\hat{\boldsymbol{w}} \quad = \quad \arg\min_{\boldsymbol{w}} \sum_{m=1}^{M} \ell(\boldsymbol{w}; x^m, y^m) + \frac{\lambda}{2}\|\boldsymbol{w}\|^2, \tag{1.35}$$

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

$$\arg\min_{\boldsymbol{w}, \boldsymbol{\xi}} \qquad \sum_{m=1}^{M} \xi_m + \tfrac{\lambda}{2}\|\boldsymbol{w}\|^2 \tag{1.36}$$

$$\text{s.t.} \quad \boldsymbol{w} \cdot \boldsymbol{f}(x^m, y^m) \geq \boldsymbol{w} \cdot \boldsymbol{f}(x^m, \tilde{y}^m) + 1 - \xi_m, \quad \forall m, \tilde{y}^m \in \mathcal{Y} \setminus \{y^m\}. \tag{1.37}$$

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 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 **??** does not have a closed form solution. Moreover, unlike maximum entropy models, the objective function in **??** 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. **??** is illustrated as Alg. **??**. The similarity with maximum entropy models (Alg. **??**) 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/`

[8] 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.

[9] 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.

	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.

`~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.* **??**–**??***) and the online algorithms seen above (perceptron and MIRA).*

Exercise 1.7 *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.

1.6 Comparison

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

Exercise 1.8

- *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?*

1.7 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 datapoints 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.

The problem setting is the following: let $\mathcal{X} = \{\bar{\mathbf{x}}^1, \ldots, \bar{\mathbf{x}}^D\}$ be a training set of independent and identically-distributed random variables. In this work $\bar{\mathbf{x}}^d$ (for notation simplicity we will drop the superscript d when considering an isolated example) corresponds to a sentence in natural language and decomposes as a sequence of observations of length N: $\bar{\mathbf{x}} = \mathbf{x}_1 \ldots \mathbf{x}_N$. Each $\mathbf{x}_n$ is a discrete random variable (a *word*), taking a value v from a finite vocabulary $\mathcal{V}$. Each $\bar{\mathbf{x}}$ has an unknown hidden structure $\bar{\mathbf{y}}$ that we want to predict. The structures are sequences $\bar{\mathbf{y}} = \mathbf{y}_1 \ldots \mathbf{y}_N$ of the same length N as the observations. Each hidden state $\mathbf{y}_n$ is a discrete random variable and can take a value y from a discrete vocabulary $\mathbf{Y}$.

Notation	
$\mathcal{X}$	training set
D	number of training examples
$\bar{\mathbf{x}} = \mathbf{x}_1 \ldots \mathbf{x}_N$	observation sequence
N	size of the observation sequence
$\mathbf{x}_i$	observation at position i in the sequence
$\mathcal{V}$	observation values set
$\lvert\mathcal{V}\rvert$	number of distinct observation types
v_i	particular observation, $i \in \lvert\mathcal{V}\rvert$
$\bar{\mathbf{y}} = \mathbf{y}_1 \ldots \mathbf{y}_N$	hidden state sequence
$\mathbf{y}_i$	hidden state at position i in the sequence
$\mathbf{Y}$	hidden states value set
$\lvert\mathbf{Y}\rvert$	number of distinct hidden value types
y_i	particular hidden value, $i \in \lvert\mathbf{Y}\rvert$

Table 2.1: General notation used in this class

We focus on the well known Hidden Markov Model (HMM) on section **??**, where we describe how to estimate its parameters from labeled data **??**. We will then move to how to find the most likely hidden sequence (decoding) given an observation sequence and a parameter set **??**. 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. Finally, section **??** will describe the task of part-of-speech tagging, and how HMM are suitable for this task.

2.1 Hidden Markov Models

The Hidden Markov Model (HMM) is one of the most common sequence probabilistic models, and has been applied to a wide variety of tasks. More generally, an HMM is a particular instance of a chain directed probabilistic graphical model, or a Bayesian network. 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, in our case $\mathbf{x}$, and the *hidden* variables, in our case $\mathbf{y}$. In the HMM terminology, the observed variables are called *observations*, and the hidden variables are called *states*.

As you may find out in today's lab session, implementing the inference routines of the HMM can be challenging, and debugging can become hard in large datasets. We thus start with a small and very simple (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 (w);*
- *shopping (s);*
- *cleaning his apartment (c);*
- *playing tennis (t).*

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 *(r) or* sunny *(s). 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 is independent of everything else* given the weather on the previous day. *The entire system is that of a 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.

It is useful to artificially introduce a special "STOP" state at the end of the sequence, which marks its end. This is useful for two reasons: it simplifies the notation, and it also allows our model to cope with sequences of any finite size (otherwise, how would our model cope with natural sentences, which can range from one or two words to dozens?).

Figure **??** shows the HMM model for the first sequence of our simple example, already including the "STOP" symbol. The notation is summarized in Table **??**. Note that i, j can go up to $N + 1$ due to the "STOP" symbol.

HMM Example	
$\bar{\mathbf{x}}$	observed sentence "w w s c"
$N = 4$	observation length
i, j	positions in the sentence: $i, j \in \{1 \dots N\}$
$\mathcal{V} = \{w, s, c, t\}$	observation vocabulary
p, q	indexes into the vocabulary $p, q \in \lvert\mathcal{V}\rvert$
$\mathbf{x}_i = v_q, \mathbf{x}_j = v_p$	observation at position $\mathbf{x}_i$ ($\mathbf{x}_j$) has value v_q (v_p)
$\mathbf{Y} = \{r, s\}$	hidden values vocabulary
l, m	indexes into the hidden values vocabulary
$\mathbf{y}_i = y_l, \mathbf{y}_j = y_m$	state at position $\mathbf{y}_i$ ($\mathbf{y}_j$) has hidden value y_l (y_m)

Table 2.2: HMM notation for the running example.

A first order HMM model has the following independence assumptions over the joint distribution $p_\theta(\bar{\mathbf{x}}, \bar{\mathbf{y}})$:

- **Independence of previous states.** The probability of being in a given state y_l at position $\mathbf{y}_i$ only depends on the state y_m of the previous position $\mathbf{y}_{i-1}$. Formally, $p_\theta(\mathbf{y}_i = y_l \mid \mathbf{y}_{i-1} = y_m, \mathbf{y}_{i-2} \dots \mathbf{y}_1) = p_\theta(\mathbf{y}_i = y_l \mid \mathbf{y}_{i-1} = y_m)$, defining a first order Markov chain.[1]
- **Homogeneous transition.** The probability of making a transition from state y_l to state y_m is independent of the particular position in the sentence: for all $i, j \in \{1, \dots, N\}$, $p_\theta(\mathbf{y}_i = y_l \mid \mathbf{y}_{i-1} = y_m) = p_\theta(\mathbf{y}_j = y_l \mid \mathbf{y}_{j-1} = y_m)$, so $p_\theta(\mathbf{y}_i = y_l \mid \mathbf{y}_{i-1} = y_m) = p_\theta(y_l \mid y_m)$.

[1] 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 harder.

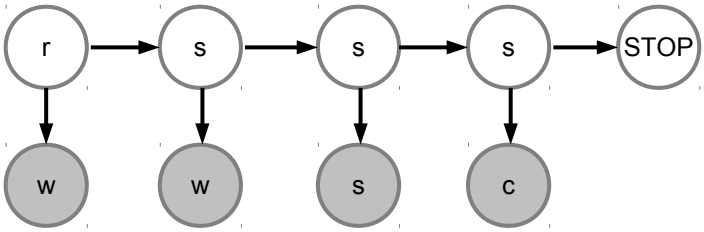


Figure 2.1: HMM structure, for the simple running example.

- **Observation independence.** The probability of observing v_q at position i is fully determined by the state at that position. Formally, $p_\theta(\mathbf{x}_i = v_q \mid \mathbf{y}_i = y_l, \mathbf{y}_{i-1}, \mathbf{y}_{i-2} \dots \mathbf{y}_1) = p_\theta(\mathbf{x}_i = v_q \mid \mathbf{y}_i = y_l)$, and this probability is independent of the particular position, that is $p_\theta(\mathbf{x}_i = v_q \mid \mathbf{y}_i = y_l) = p_\theta(v_q \mid y_l)$.

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

We also need to define a *start probability*, the probability of starting at state y_l. The three probability distributions that define the HMM model are summarized in Table **??**. For each one of them we will use a short notation to simplify the exposition.

HMM distributions		
Name	probability distribution	short notation
initial probability	$p_\theta(\mathbf{y}_1 = y_l)$	π_l
transition probability	$p_\theta(\mathbf{y}_i = y_l \mid \mathbf{y}_{i-1} = y_m)$	$a_{m,l}$
observation probability	$p_\theta(\mathbf{x}_i = v_q \mid \mathbf{y}_i = y_l)$	$b_l(\mathbf{x}_i)$

Table 2.3: HMM probability distributions.

The joint distribution can be expressed as:

$$p_\theta(\bar{\mathbf{x}}, \bar{\mathbf{y}}) = \pi_{\mathbf{y}_1} b_{\mathbf{y}_1}(\mathbf{x}_1)[\prod_{i=2}^{N} a_{\mathbf{y}_{i-1},\mathbf{y}_i} b_{\mathbf{y}_i}(\mathbf{x}_i)] a_{\mathbf{y}_i,STOP}, \tag{2.1}$$

which for the example from Figure **??** is:

$$p_\theta(\bar{\mathbf{x}}, \bar{\mathbf{y}}) = \pi_r b_r(''w'') a_{r,s} b_s(''w'') a_{s,s} b_s(''s'') a_{s,s} b_s(''c'') a_{s,STOP}. \tag{2.2}$$

Here is a detailed explanation of the terms:

$$p_\theta(\bar{\mathbf{x}}, \bar{\mathbf{y}}) = \underbrace{\pi_r}_{\text{prob. of starting at state ``r''}} \overbrace{b_r(''w'')}^{\text{prob. of observing ``w'' if state is ``r''}} \underbrace{a_{r,s}}_{\text{prob. of going from state ``r'' to state ``s''}} \overbrace{b_s(''w'')}^{\text{prob. of observing ``w'' if state is ``s''}} \underbrace{a_{s,s} b_s(''s'') a_{s,s} b_s(''c'')}_{\text{etc...}} \overbrace{a_{s,STOP}}^{\text{prob. of going from state ``s'' to state ``STOP''}} \tag{2.3}$$

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.*

```
 In[]: run readers/simple_sequence.py
 In[]: simple = SimpleSequence()
 In[]: simple.train
Out[]: [w/r w/s s/s c/s , w/r w/r s/r c/s , w/s s/s s/s c/s ]
 In[]: simple.test
Out[]: [w/r w/s s/s c/s , c/s w/s t/s w/s ]
```

In the next section we turn our attention to estimating the different probability distributions of the model: π_l, $a_{m,l}$ and $b_l(\mathbf{x}_i)$.

short notation	probability distribution	—parameters—	constraint
π_j	$p_\theta(\mathbf{y}_1 = y_j)$	$\lvert\mathbf{Y}\rvert$	$\sum_{y \in \mathbf{Y}} \pi_j = 1;$
$a_{l,m}$	$p_\theta(\mathbf{y}_i = y_l \mid \mathbf{y}_{i-1} = y_m)$	$\lvert\mathbf{Y}\rvert(\lvert\mathbf{Y}\rvert + 1)$	$\sum_{y_l \in \mathbf{Y}} a_{m,l} = 1;$
$b_q(l)$	$p_\theta(\mathbf{x}_i = v_q \mid \mathbf{y}_i = y_l)$	$\lvert\mathbf{Y}\rvert\lvert\mathcal{V}\rvert$	$\sum_{v_q \in \mathcal{V}} b_q(l) = 1.$

Table 2.4: Multinomial parametrization of the HMM distributions.

2.2 Finding the Maximum Likelihood Parameters

So far we have not committed to any form for the probability distributions π_l, $a_{m,l}$ and $b_l(\mathbf{x}_i)$. In both applications addressed in this class, both the observations and the hidden variables are discrete. The most common approach is to model each of these probability distributions as multinomial distributions, summarized in Table **??**. Note that the number of parameters of $a_{l,m}$ is $|\mathbf{Y}|(|\mathbf{Y}| + 1)$ because of the special "STOP" symbol.

We will refer to the set of all parameters as θ. The HMM model is trained to maximize the Log Likelihood of the data. Given a dataset $\mathcal{D}$ the objective being optimized is:

$$\arg\max_{\theta} \sum_{\bar{\mathbf{x}}, \bar{\mathbf{y}} \in \mathcal{D}} \log p_\theta(\bar{\mathbf{x}}, \bar{\mathbf{y}}) \tag{2.4}$$

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 a labeled corpus, 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:

$$\textbf{Initial Counts}: \quad ic(y_l) = \sum_{\bar{\mathbf{x}}, \bar{\mathbf{y}} \in \mathcal{D}} \mathbf{1}(\mathbf{y}_1 = y_j); \tag{2.5}$$

$$\textbf{Transition Counts}: \quad tc(y_l, y_m) = \sum_{\bar{\mathbf{x}}, \bar{\mathbf{y}} \in \mathcal{D}} \sum_{i=1}^{N} \mathbf{1}(\mathbf{y}_i = y_l \mid \mathbf{y}_{i-1} = y_m); \tag{2.6}$$

$$\textbf{State Counts}: \quad sc(v_q, y_l) = \sum_{\bar{\mathbf{x}}, \bar{\mathbf{y}} \in \mathcal{D}} \sum_{i=1}^{N} \mathbf{1}(\mathbf{x}_i = v_q \mid \mathbf{y}_i = y_l) \tag{2.7}$$

Note that $\mathbf{1}$ 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. (**??**) counts how often state y_l follows state y_m. Therefore, $tc(\text{"}s\text{"}, \text{"}r\text{"})$ contains the number of times that a sunny day ("s") followed a rainy day ("r").

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.
- **Transition Counts** - Should sum to the number of tokens.
- **Observation Counts** - Should sum to the number of tokens.

Exercise 2.2 *Convince yourself that the sanity checks described above are true. Collect the counts from a supervised corpus using method* collect_counts_from_corpus *and use the provided function* sanity_check_counts *to perform these checks on the counts table.*

```
In []: run sequences/hmm.py
In []: hmm = HMM(simple)
In []: hmm.collect_counts_from_corpus(simple.train)
In []: hmm.sanity_check_counts(simple.train)
Init Counts match
Final Counts match
Transition Counts match
Observations Counts match
```

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

$$\hat{\pi}_l = \frac{ic(y_l)}{\sum_{y_m \in \mathbf{Y}} ic(y_m)} \tag{2.8}$$

$$\hat{a}_{l,m} = \frac{tc(y_l, y_m)}{\sum_{y_l \in \mathbf{Y}} tc(y_l, y_m)} \tag{2.9}$$

$$\hat{b}_l(v_q = o) = \frac{sc(v_q, y_l)}{\sum_{v_p \in \mathcal{V}} sc(v_p, y_l)} \tag{2.10}$$

Exercise 2.3 *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.init_probs *will show you the raw counts of initial states). How are the counts related to the probabilities?*

```
In[]:  run sequences/hmm.py
In[]: hmm = HMM(simple)
In[]: hmm.train_supervised(simple.train)
In []: hmm.init_probs
Out[]:
array([[ 0.66666667],
       [ 0.33333333]])
In []: hmm.transition_probs
Out[]:
array([[ 0.5  ,  0.   ],
       [ 0.5  ,  0.625],
       [ 0.   ,  0.375]])
In []: hmm.observation_probs
Out[]:
array([[ 0.75 ,  0.25 ],
       [ 0.25 ,  0.375],
       [ 0.   ,  0.375],
       [ 0.   ,  0.   ]])
```

2.3 Finding the most likely sequence - Decoding

Given the learned parameters and an observation $\bar{\mathbf{x}}$, we want to find the best hidden state sequence $\bar{\mathbf{y}}^*$; this is called *decoding*. There are several ways to define what we mean by the best $\bar{\mathbf{y}}^*$; for instance, it could be the best assignment to each hidden variable $\mathbf{y}_i$, or the best assignment to the sequence $\bar{\mathbf{y}}$ as a whole.

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

$$\bar{\mathbf{y}}^* = \arg\max_{\mathbf{y}_1 \cdots \mathbf{y}_N} b_i(\mathbf{y}_i) \tag{2.11}$$

where $\gamma_i(\mathbf{y}_i)$ is the posterior probability $P(\mathbf{y}_i|\bar{\mathbf{x}})$. This method does not guarantee that the sequence $\bar{\mathbf{y}}^*$ is a valid sequence of the model. For instance there might be a transition probability between two of the best node posteriors with probability zero.

The second approach, called **Viterbi decoding**, consists in picking the best global hidden state sequence $\bar{\mathbf{y}}$.

$$\bar{\mathbf{y}}^* = \arg\max_{\bar{\mathbf{y}}} p_\theta(\bar{\mathbf{x}}, \bar{\mathbf{y}}). \tag{2.12}$$

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 **??**).

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

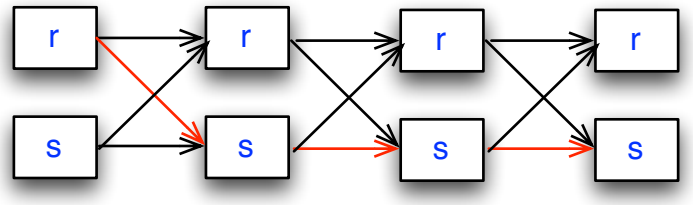


Figure 2.2: Trellis representation of the HMM in Figure **??**, for the observation sequence "w w s c", where each hidden variable can take the values r, s.

Each column represents a position in the sentence, and each row represents a possible state. For the decoding algorithms described in the following sections, it is useful to define a re-parametrization of the model in equation (**??**), in terms of node potentials $\phi_n(l)$ (associating a number to each box in Figure **??**) and edge potentials $\phi_{n-1,n}(l,m)$ (associating a number to each edge in Figure **??**). For our example, this re-parametrization is given by

$$p_\theta(\bar{\mathbf{x}}, \bar{\mathbf{y}}) = \phi_1(r)\phi_1(r,s)\phi_2(s)\phi_2(s,s)\phi_3(s)\phi_3(s,s)\phi_4(s). \tag{2.13}$$

In other words, to do this re-parameterization we need to find expressions for the potential variables, (the ϕ's) such that (**??**) and (**??**) are equal. The solution is given by

$$\phi_{i-1,i}(l,m) = a_{l,m} \tag{2.14}$$

and

$$\phi_i(l) = \begin{cases} b_{\mathbf{x}_i}(l)\pi_l & \text{i = 1} \\ b_{\mathbf{x}_i}(l) & \text{i = 2,...,N-1} \\ b_{\mathbf{x}_i}(l)a_{l,STOP} & \text{i = N} \end{cases} \tag{2.15}$$

Exercise 2.4 *Convince yourself that equation* **??** *is equivalent to equation* **??**, *as long as you use the correspondences in equations* (**??**) *and* (**??**).

Use the given function build_potentials *on the first training sequence and confirm that the values are correct. You should get the same values as presented below.*

```
In[]: run sequences/hmm.py
In[]: hmm = HMM(simple)
In[]: hmm.train_supervised(simple.train)
In[]: node_potentials,edge_potentials = hmm.build_potentials(simple.train.seq_list[0])
In []: node_potentials
Out[]:
array([[ 0.5       ,  0.75      ,  0.25      ,  0.        ],
       [ 0.08333333,  0.25      ,  0.375     ,  0.140625  ]])
In []: edge_potentials
Out[]:
array([[[ 0.5  ,  0.5  ,  0.5  ],
        [ 0.5  ,  0.5  ,  0.5  ]],

       [[ 0.   ,  0.   ,  0.   ],
        [ 0.625,  0.625,  0.625]]])
```

2.3.1 Posterior Decoding

Posterior decoding consists in picking the highest state posterior for each position in the sequence independently:

$$\bar{\mathbf{y}}^* = \arg\max_{\mathbf{y}_1 \cdots \mathbf{y}_N} \gamma_i(\mathbf{y}_i). \tag{2.16}$$

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

Figure 2.3: Forward trellis for the first sentence of the training data at position 1 (left) and at position 2 (right)

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.

$$\textbf{Sequence Posterior}: \quad p_\theta(\bar{\mathbf{y}} \mid \bar{\mathbf{x}}) = \frac{p_\theta(\bar{\mathbf{x}}, \bar{\mathbf{y}})}{p_\theta(\bar{\mathbf{x}})}; \tag{2.17}$$

$$\textbf{State Posterior}: \quad \gamma_i(y_l) = p_\theta(\mathbf{y}_i = y_l \mid \bar{\mathbf{x}}); \tag{2.18}$$

$$\textbf{Transition Posterior}: \quad \xi_i(y_l, y_m) = p_\theta(\mathbf{y}_i = y_l, \mathbf{y}_{i+1} = y_m \mid \bar{\mathbf{x}}). \tag{2.19}$$

To compute the posteriors a first step is to be able to compute the likelihood of the sequence $p_\theta(\bar{\mathbf{x}})$, which corresponds to summing the probability of all possible hidden state sequences.

$$\textbf{Likelihood}: \quad p_\theta(\bar{\mathbf{x}}) = \sum_{\bar{\mathbf{y}}} p_\theta(\bar{\mathbf{x}}, \bar{\mathbf{y}}). \tag{2.20}$$

The number of possible hidden state sequences is exponential in the length of the sentence ($|\mathbf{Y}|^N$), 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 **??**. 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 $\bar{\mathbf{y}}$) 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, 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.

$$\textbf{Forward Probability}: \quad \alpha_i(y_l) = p_\theta(\mathbf{y}_i = y_l, \mathbf{x}_1 \dots \mathbf{x}_i) \tag{2.21}$$

The forward probability represents the probability that in position i we are in state $\mathbf{y}_i = y_l$ and that we have observed $\bar{\mathbf{x}}$ up to that position. The forward probability is defined by the following recurrence (we will use the potentials we defined earlier, in (**??**) and (**??**)):

$$\alpha_1(y_l) = \phi_1(l) \tag{2.22}$$

$$\alpha_i(y_l) = \left[\sum_{y_m \in \mathbf{Y}} \phi_{i-1,i}(m,l) \alpha_{i-1}(y_m) \right] \phi_i(l) \tag{2.23}$$

At position 1, the probability of being in state "r" and observing word "w" is just the node marginal for that position: $\alpha_1(r) = \phi_1(r)$ (see Figure **??** left). At position 2 the probability of being in state "s" and observing the sequence of words "w w" corresponds to the sum of all possible ways of reaching position 2 in state "s", namely, "rs" and "ss" (see Figure **??** right). The probability of the former is $\phi_1(r)\phi_1(r,s)\phi_2(s)$ and that of the latter is $\phi_1(s)\phi_1(s,s)\phi_2(s)$, so we get:

$$\begin{aligned}
\alpha_2(s) &= \phi_1(r)\phi_1(r,s)\phi_2(s) + \phi_1(s)\phi_1(s,s)\phi_2(s) \\
\alpha_2(s) &= [\phi_1(r)\phi_1(r,s) + \phi_1(s)\phi_1(s,s)]\,\phi_2(s) \\
\alpha_2(s) &= \sum_{\mathbf{y} \in \mathbf{Y}} [\phi_1(\mathbf{y})\phi_1(\mathbf{y},s)\phi_2(s)] \\
\alpha_2(s) &= \sum_{\mathbf{y} \in \mathbf{Y}} [\alpha_1(\mathbf{y})\phi_1(\mathbf{y},s)\phi_2(s)]
\end{aligned}$$

Algorithm 7 Forward Backward algorithm

input: sentence $\bar{\mathbf{x}}$, parameters θ
Forward pass: Compute the forward probabilities
Initialization
for $y_l \in \mathbf{Y}$ **do**
  $\alpha_1(y_l) = \phi_1(y_l)$
end for
for $i = 2$ **to** N **do**
  for $y_l \in \mathbf{Y}$ **do**
    $\alpha_i(y_l) = \left[\sum_{m \in \mathbf{Y}} \phi_{i-1,i}(m,l)\alpha_{i-1}(y_m)\right] \phi_i(l)$
  end for
end for
Backward pass: Compute the backward probabilities
Initialization
for $y_l \in \mathbf{Y}$ **do**
  $\beta_N(y_l) = 1$
end for
for $i = N - 1$ **to** 1 **do**
  $\beta_i(y_l) = \sum_{y_m \in \mathbf{Y}} \phi_{i,i+1}(l,m)\phi_{i+1}(m)\beta_{i+1}(y_m)$.
end for
output: The forward and backward probabilities α and β

Using the forward trellis one can compute the likelihood by summing over all possible hidden states for the last position.

$$p_\theta(\bar{\mathbf{x}}) = \sum_{\bar{\mathbf{y}}} p_\theta(\bar{\mathbf{x}}, \bar{\mathbf{y}}) = \sum_{y \in \mathbf{Y}} \alpha_N(y). \tag{2.24}$$

Although the forward probability is enough to calculate the likelihood of a given sequence, we will also need the backward probability to calculate the node and edge posteriors. The backward probability, represents the probability of observing $\bar{\mathbf{x}}$ from position $i+1$ up to N, given that at position i we are at state $\mathbf{y}_i = y_l$:

$$\textbf{Backward Probability}: \quad \beta_i(y_l) = p_\theta(\mathbf{x}_{i+1}...\mathbf{x}_N | \mathbf{y}_i = y_l). \tag{2.25}$$

The backward recurrence is given by:

$$\beta_N(y_l) = 1 \tag{2.26}$$
$$\beta_i(y_l) = \sum_{y_m \in \mathbf{Y}} \phi_{i,i+1}(l,m)\phi_{i+1}(m)\beta_{i+1}(y_m). \tag{2.27}$$

The backward probability is similar to the forward probability, but operates in the inverse direction. At position N there are no more observations, and the backward probability is set to 1. At position i, the probability of having observed the future and being in state y_l, is given by the sum for all possible states of the probability of having transitioned from position i with state y_l to position $i+1$ with state y_m, and observing $\mathbf{x}_{i+1}$ at time $i+1$ and the future from there onward, which is $\beta_{i+1}(\mathbf{y}_{i+1} = y_m)$.

With the FB probabilities one can compute the likelihood of a given sentence using any position in the sentence.

$$p_\theta(\bar{\mathbf{x}}) = \sum_{\bar{\mathbf{y}}} p_\theta(\bar{\mathbf{x}}, \bar{\mathbf{y}}) = \sum_{y \in \mathbf{Y}} \alpha_i(y)\beta_i(y). \tag{2.28}$$

This equation will work for any choice of i. Note that for time N, $\beta_N(y) = 1$ and we get back to equation **??**. 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. The FB algorithm may fail for long sequences 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 FB algorithm that avoids this problem.

Algorithm **??** shows the pseudo code for the forward backward algorithm.

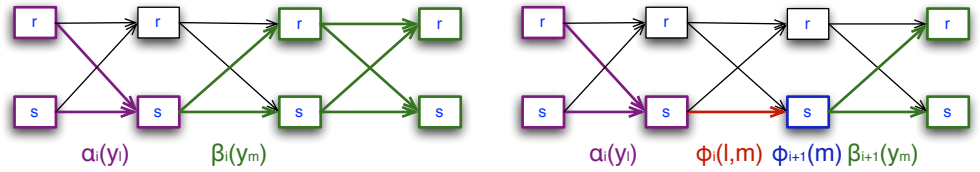


Figure 2.4: A graphical representation of the components in the state and transition posteriors.

Exercise 2.5 *Run the provided forward-backward algorithm on the first train sequence. Use the provided function that makes use of Equation* **??** *to make sure your implementation is correct:*

```
In[]:  run sequences/hmm.py
In[]: hmm = HMM(simple)
In[]: hmm.train_supervised(simple.train)
In[]: forward,backward =  hmm.forward_backward(simple.train.seq_list[0])
In []: sanity_check_forward_backward(forward,backward)
[[ 0.00629354]
 [ 0.00629354]
 [ 0.00629354]
 [ 0.00629354]]
Out[8]: True
```

Moreover, given the forward and backward probabilities one can compute both the state and transition posteriors.

$$\textbf{State Posterior:} \quad \gamma_i(y_l) = p_\theta(\mathbf{y}_i = y_l \mid \bar{\mathbf{x}}) = \frac{\alpha_i(y_l)\beta_i(y_l)}{p_\theta(\bar{\mathbf{x}})}; \tag{2.29}$$

$$\begin{aligned}\textbf{Transition Posterior:} \quad \xi_i(y_l, y_m) &= p_\theta(\mathbf{y}_i = y_l, \mathbf{y}_{i+1} = y_m \mid \bar{\mathbf{x}}) \\ &= \frac{\alpha_i(y_l)\phi_{i,i+1}(l,m)\phi_{i+1}(m)\beta_{i+1}(y_m)}{p_\theta(\bar{\mathbf{x}})}.\end{aligned} \tag{2.30}$$

A graphical representation of these posteriors is illustrated in figure **??**. On the left it is shown that $\alpha_i(y_l)\beta_i(y_l)$ returns the sum of all paths that contain the state y_i, weighted by $p_\theta(\bar{\mathbf{x}}, \bar{\mathbf{y}})$; on the right we can see that $\alpha_i(y_l)\phi_{i,i+1}(l,m)\phi_{i+1}(m)\beta_{i+1}(y_m)$ returns the same for all paths containing the edge from y_l to y_m. Thus, these posteriors can be seen as the ratio of the number of paths that contain the given state or transition (weighted by $p_\theta(\bar{\mathbf{x}}, \bar{\mathbf{y}})$) and the number of possible paths in the graph $p_\theta(\bar{\mathbf{x}})$. As an practical example, given that the person perform the sequence of actions "w w s c", 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 "walk walk shop clean" was generated by a sequence of weathers and where it was raining in the second day. In this case, the possible sequences would be "r r r r", "s r r r", "r r s r", "r r r s", "s r s r", "s r r s", "r r s s", and "s r s s" (*i.e.*, all the sequences which have "r" in the second position).

Using the node posteriors, we are ready to perform posterior decoding. Algorithm **??** shows the posterior decoding algorithm.

Exercise 2.6 *Given the node and edge posterior formulas* **??**,**??** *and the forward and backward formulas* **??**,**??**, *convince yourself that formulas* **??**,**??** *are correct.*

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

```
In[]:  run sequences/hmm.py
In[]: hmm = HMM(simple)
In[]: hmm.train_supervised(simple.train)
In [15]: node_posteriors = hmm.get_node_posteriors(simple.train.seq_list[0])

In [16]: node_posteriors
Out[16]:
array([[ 0.95738152,  0.75281282,  0.26184794,  0.        ],
```

Algorithm 8 Posterior Decoding algorithm

input: The forward and backward probabilities α and β.
Compute Likelihood: Compute the likelihood of the sentence
$L = 0$
for $y_l \in \mathbf{Y}$ **do**
  $p_\theta(\bar{\mathbf{x}}) = p_\theta(\bar{\mathbf{x}}) + \alpha_N(y_l)$
end for
$\hat{\mathbf{y}} = []$
for $i = 1$ **to** N **do**
  $max = 0$
  for $y_l \in \mathbf{Y}$ **do**
    $\gamma_i(y_l) = \frac{\alpha_i(y_l)\beta_i(y_l)}{p_\theta(\bar{\mathbf{x}})}$
    if $\gamma_i(y_l) > max$ **then**
      $max = \gamma_i(y_l)$
      $\hat{y}_i = y_l$
    end if
  end for
end for
output: the posterior path $\hat{\mathbf{y}}$

```
        [ 0.04261848,  0.24718718,  0.73815206,  1.        ]])
```

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

```
In[]: run sequences/hmm.py
In[]: hmm = HMM(simple)
In[]: hmm.train_supervised(simple.train)
In[]: y_pred = hmm.posterior_decode(simple.test.seq_list[0])
In[]: y_pred
Out[]: w/r w/r s/s c/s
In[]: simple.test.seq_list[0]
Out[]: w/r w/s s/s c/s
```

Do the same for the second test sentence:

```
In[]: y_pred = hmm.posterior_decode(simple.test.seq_list[1])
In[]: y_pred
Out[]: c/r w/r t/r w/r
In[]: simple.test.seq_list[1]
Out[]: c/s w/s t/s w/s
```

What is wrong? Note the observations for the second test sentence: the observation "t" 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?

Note that if you use smoothing when training, the sanity checks mentioned at the start of this chapter are no longer true. For example, the sum of all the transition counts is no longer equal to the number of tokens – it is larger.

Algorithm 9 Viterbi algorithm

input: sentence $\bar{\mathbf{x}}$, parameters θ
Forward pass: compute the maximum paths for every end state
Initialization
for $y_l \in \mathbf{Y}$ **do**
  $\delta_1(y_l) = \phi_1(y_l)$
end for
for $i = 2$ **to** N **do**
  for $y_l \in \mathbf{Y}$ **do**
    $\delta_i(y_l) = \left[\max_{m \in \mathbf{Y}} \phi_{i-1,i}(m,l)\delta_{i-1}(y_m)\right] \phi_i(l)$
    $\psi_i(y_l) = m$
  end for
end for
Backward pass: Build the most likely path
$\hat{\mathbf{y}} = []$
$\hat{y}_N = \arg\max_{y_m \in |\mathbf{Y}|} \delta_N(y_l)$
for $i = N$ **to** 2 **do**
  $\hat{y}_i = \psi_{i+1}(y_{i+1})$
end for
output: the viterbi path $\hat{\mathbf{y}}$

2.3.2 Viterbi Decoding

Viterbi decoding consists in picking the best global hidden state sequence $\bar{\mathbf{y}}$.

$$\bar{\mathbf{y}}^* = \arg\max_{\bar{\mathbf{y}}} p_\theta(\bar{\mathbf{x}}, \bar{\mathbf{y}}). \tag{2.31}$$

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 **??** where instead of summing over all possible hidden states, we take their maximum.

$$\textbf{Viterbi} \quad \delta_i(y_l) = \arg\max_{y_1 \dots y_i} p_\theta(\mathbf{y}_i = y_l, \mathbf{x}_1 \dots \mathbf{x}_i) \tag{2.32}$$

The viterbi trellis represents the path with maximum probability in position i when we are in state $\mathbf{y}_i = y_l$ and that we have observed $\bar{\mathbf{x}}$ up to that position. The viterbi algorithm is defined by the following recurrence (we again use the potentials defined in equations (**??**) and (**??**)):

$$\delta_1(y_l) = \phi_1(l) \tag{2.33}$$

$$\delta_i(y_l) = \left[\max_{y_1 \dots y_i} \phi_{i-1,i}(m,l)\delta_{i-1}(y_m)\right] \phi_i(l) \tag{2.34}$$

$$\psi_i(y_l) = \left[\arg\max_{y_1 \dots y_i} \phi_{i-1,i}(m,l)\delta_{i-1}(y_m)\right] \tag{2.35}$$

Algorithm **??** shows the pseudo code for the Viterbi algorithm.

Exercise 2.8 *Implement a method for performing viterbi decoding. You can use our implementation of posterior decoding (the function* posterior_decode*) for reference.*

```
        def viterbi_decode(self,seq):
```

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

```
In []: y_pred = hmm.viterbi_decode(simple.test.seq_list[0])
In []: y_pred
Out[]: w/r w/r s/r c/s
```

```
In []: y_pred = hmm.viterbi_decode(simple.test.seq_list[1])
In []: y_pred
Out[]: c/s w/s t/s w/s
```

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.*

2.4 Part-of-Speech Tagging (POS)

Part of speech tagging is probably one the most common 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 [2].

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, $\bar{\mathbf{x}}$ is a sentence and $\bar{\mathbf{y}}$ is the sequence of possible PoS tags.

The first step is to load the pos corpus. We will start by loading 1000 sentences for training and 200 sentences both for development and testing, and training the HMM model.

```
In []: run readers/pos_corpus.py
In []: corpus = PostagCorpus()
In []: train_seq = corpus.read_sequence_list_conll("../data/train-02-21.conll",
    max_sent_len=15,max_nr_sent=1000)
unknown tag po
unknown tag pr
unknown tag wd
unknown tag pd
unknown tag wr
unknown tag sy
In []: test_seq = corpus.read_sequence_list_conll("../data/test-23.conll",max_sent_len=15
    ,max_nr_sent=1000)
unknown tag pr
unknown tag po
unknown tag wr
unknown tag wd
unknown tag pd
unknown tag sy
In []: dev_seq = corpus.read_sequence_list_conll("../data/dev-22.conll",max_sent_len=15,
    max_nr_sent=1000)
unknown tag wd
unknown tag wr
unknown tag po
unknown tag pr
unknown tag pd
In []: corpus.add_sequence_list(train_seq)
In []: run sequences/hmm.py
In []: hmm = HMM(corpus)
In []: hmm.train_supervised(train_seq)
In []: hmm.print_transition_matrix()
```

Ignore the warnings – they are a consequence of our choice of using only a reduced set of 12 tags instead of the full set of POS tags.

Look at the transition probabilities of the trained model (see Figure **??**), 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:*

```
In []: viterbi_pred_train = hmm.viterbi_decode_corpus(train_seq.seq_list)
In []: posterior_pred_train = hmm.posterior_decode_corpus(train_seq.seq_list)
```

[2] See ACL state of the art wiki

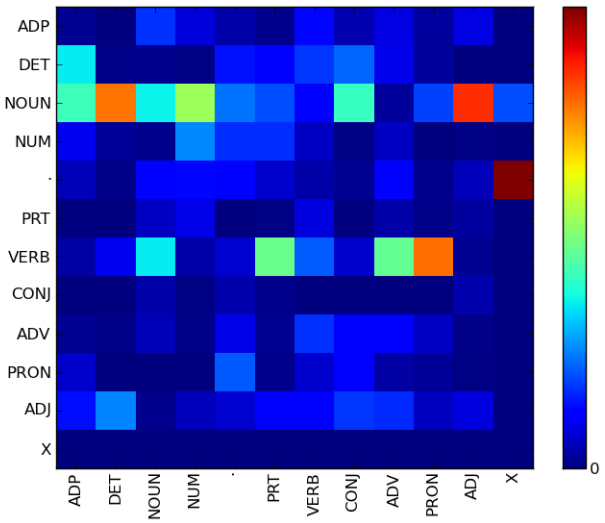


Figure 2.5: Transition probabilities of the trained model. Each column is previous state and row is current state. Note the high probability of having Noun after Adjective, or of having Verb after Noun, as expected.

```
In []: eval_viterbi_train =   hmm.evaluate_corpus(train_seq.seq_list,viterbi_pred_train)
In []: eval_posterior_train = hmm.evaluate_corpus(train_seq.seq_list,posterior_pred_train
    )
In []: eval_viterbi_train
Out[]: 0.9811604369175269
In []: eval_posterior_train
Out[]: 0.9811604369175269

In []: viterbi_pred_test = hmm.viterbi_decode_corpus(test_seq.seq_list)
In []: posterior_pred_test = hmm.posterior_decode_corpus(test_seq.seq_list)
In []: eval_viterbi_test =   hmm.evaluate_corpus(test_seq.seq_list,viterbi_pred_test)
In []: eval_posterior_test = hmm.evaluate_corpus(test_seq.seq_list,posterior_pred_test)
In []: eval_viterbi_test
Out[]: 0.5224796989502872
In []: eval_posterior_test
Out[]: 0.3798772034066152
```

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*).*

```
In []: hmm.pick_best_smoothing(train_seq,dev_seq,[0,0.1,0.01,1])
Smoothing 0.000000 --  Train Set Accuracy: Posterior Decode 0.981, Viterbi Decode: 0.981
sequences/hmm.py:214: RuntimeWarning: invalid value encountered in double_scalars
  posteriors[current_state,pos] = forward[current_state,pos]*backward[current_state,pos]/
      likelihood
Smoothing 0.000000 -- Test Set Accuracy: Posterior Decode 0.401, Viterbi Decode: 0.539
Smoothing 0.100000 --  Train Set Accuracy: Posterior Decode 0.969, Viterbi Decode: 0.965
Smoothing 0.100000 -- Test Set Accuracy: Posterior Decode 0.865, Viterbi Decode: 0.854
Smoothing 0.010000 --  Train Set Accuracy: Posterior Decode 0.981, Viterbi Decode: 0.980
Smoothing 0.010000 -- Test Set Accuracy: Posterior Decode 0.843, Viterbi Decode: 0.835
Smoothing 1.000000 --  Train Set Accuracy: Posterior Decode 0.884, Viterbi Decode: 0.878
Smoothing 1.000000 -- Test Set Accuracy: Posterior Decode 0.821, Viterbi Decode: 0.812
Out[]: 0.1
```

As before, the warnings are due to tags in the dev set which were not present in the train set – just ignore them. Using the best smoothing value (0.1) evaluate the accuracy on the test set.

```
In []: hmm.train_supervised(train_seq,smoothing=0.1)
```

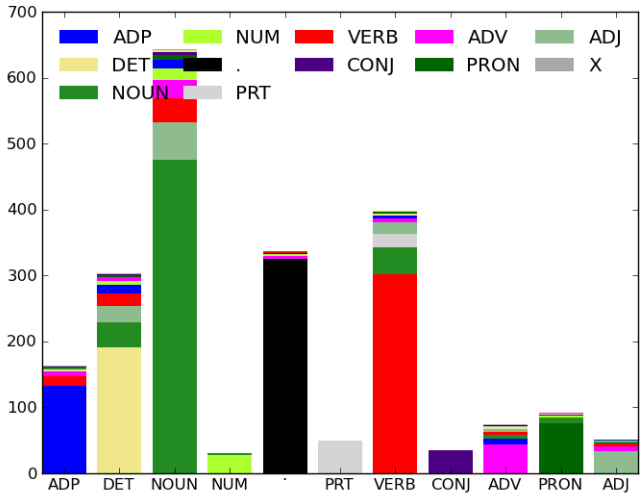


Figure 2.6: Confusion Matrix for the previous example. Predict tags are columns and the true tags corresponds to the constituents of each column.

```
In []: pred = hmm.viterbi_decode_corpus(test_seq.seq_list)
In []: eval_test = hmm.evaluate_corpus(test_seq.seq_list,pred)
In []: eval_test
Out[]: 0.8377896613190731
```

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 **??**.

```
In []: run sequences/confusion_matrix.py
In []: cm = build_confusion_matrix(test_seq.seq_list,pred,len(corpus.int_to_tag),hmm.
    nr_states)
In []: plot_confusion_bar_graph(cm,corpus.int_to_tag,xrange(hmm.nr_states),tag_colors)
In[]: plt.show()
```

Exercise 2.10 *So far we have only worked with a limited dataset of 1000 words. Try increasing the number of sentences to 10000. What do you observe?*

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, $p_\theta(\bar{\mathbf{x}}, \bar{\mathbf{y}})$ whereas discriminative ones only model the conditional probability of the sequence, given the observed data, $p_\theta(\bar{\mathbf{y}}|\bar{\mathbf{x}})$.

Table **??** 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 **??**, 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

$$\arg\max_{\bar{\mathbf{y}}} p_\theta(\bar{\mathbf{y}}|\bar{\mathbf{x}}) = \arg\max_{\bar{\mathbf{y}}} \theta \cdot f(\bar{\mathbf{x}}, \bar{\mathbf{y}}) \tag{3.1}$$

As in the previous section, $\bar{\mathbf{y}}$ is a sequence so the maximization is over an exponential number of objects, making it intractable by brute force methods. Again we will make a first order Markov independence assumption, and so the features will decompose as the model. Therefore, expression **??** can be written as:

$$\arg\max_{\bar{\mathbf{y}}} \sum_N \sum_{\bar{\mathbf{y}}} \theta \cdot f(n, \mathbf{y}_n, \bar{\mathbf{x}}_n) + \sum_N \sum_{\mathbf{y}_n \in \mathbf{Y}} \theta \cdot f(n, \mathbf{y}_n, \mathbf{y}_{n-1}, \bar{\mathbf{x}}_n) \tag{3.2}$$

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 the presence or absence of a given condition.

Table **??** depicts the features that are implicit in the HMM, which was the subject of the previous chapter. These features are indicators of initial, observation and transition events. The fact that we were using a gen-

Classification	**Sequences**
Generative	
Naïve Bayes **??**	Hidden Markov Models **??**
Discriminative	
Perceptron **??**	Structured Perceptron **??**
Maximum Entropy **??**	Conditional Random Fields **??**

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

Condition	Name
$\mathbf{y}_i = l$ & $t = 0$	Initial State
$\mathbf{y}_i = l$ & $\mathbf{y}_{i-1} = m$	Transition Features
$\bar{\mathbf{x}}_i = a$ & $\mathbf{y}_i = l$	Observation Features

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

Condition	Name
$\mathbf{y}_i = l$ & $t = 0$	Initial State
$\mathbf{y}_i = l$ & $\mathbf{y}_{i-1} = m$	Transition Features
$\bar{\mathbf{x}}_i = a$ & $\mathbf{y}_i = l$	Observation Features
$\bar{\mathbf{x}}_i = a$ & a is uppercased & $\mathbf{y}_i = l$	Uppercase Features
$\bar{\mathbf{x}}_i = a$ & a contains digit & $\mathbf{y}_i = l$	Digit Features
$\bar{\mathbf{x}}_i = a$ & a contains hyphen & $\mathbf{y}_i = l$	Hyphen Features
$\bar{\mathbf{x}}_i = a$ & $a_{[0..i]} \forall i \in [1,2,3]$ $\mathbf{y}_i = l$	Prefix Features
$\bar{\mathbf{x}}_i = a$ & $a_{[N-i..N]} \forall i \in [1,2,3]$ & $\mathbf{y}_i = l$	Suffix Features

Table 3.3: Extended feature set. Some features in this set could not be included in the HMM model.

erative model has forced us (in some sense) to make strong independence assumptions. However, since we now move to a discriminative approach, where we model $P(\bar{\mathbf{y}}|\bar{\mathbf{x}})$ rather than $P(\bar{\mathbf{x}}, \bar{\mathbf{y}})$, 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* $\bar{\mathbf{x}}$. 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 **??** 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!

We consider two kinds of features: *node features*, which form a vector $\boldsymbol{f}_N(\bar{\mathbf{x}}, y_i)$, and *edge features*, which form a vector $\boldsymbol{f}_E(\bar{\mathbf{x}}, y_i, y_{i-1})$.[1] These feature vectors will receive parameter vectors $\boldsymbol{\theta}_N$ and $\boldsymbol{\theta}_E$. Similarly as in the previous chapter, we consider:

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

$$\psi_V(\bar{\mathbf{x}}, \mathbf{y}_i) = \exp(\boldsymbol{\theta}_V \cdot \boldsymbol{f}_V(\bar{\mathbf{x}}, \mathbf{y}_i)). \tag{3.3}$$

- *Edge Potentials.* These are scores for the transitions. They are given by

$$\psi_E(\bar{\mathbf{x}}, \mathbf{y}_i, \mathbf{y}_{i-1}) = \exp(\boldsymbol{\theta}_E \cdot \boldsymbol{f}_E(\bar{\mathbf{x}}, \mathbf{y}_i, \mathbf{y}_{i-1})). \tag{3.4}$$

Let $\boldsymbol{\theta} = (\boldsymbol{\theta}_N, \boldsymbol{\theta}_E)$. The conditional probability $P(\bar{\mathbf{y}}|\bar{\mathbf{x}})$ is then defined as follows:

$$P(\bar{\mathbf{y}}|\bar{\mathbf{x}}) = \frac{1}{Z(\boldsymbol{\theta}, \bar{\mathbf{x}})} \exp\left(\sum_i \boldsymbol{\theta}_V \cdot \boldsymbol{f}_V(\bar{\mathbf{x}}_i, \mathbf{y}_i) + \sum_i \boldsymbol{\theta}_E \cdot \boldsymbol{f}_E(\bar{\mathbf{x}}_i, \mathbf{y}_i, \mathbf{y}_{i-1})\right) \tag{3.5}$$

$$= \frac{1}{Z(\boldsymbol{\theta}, x)} \prod_i \psi_V(\bar{\mathbf{x}}_i, \mathbf{y}_i) \psi_E(\bar{\mathbf{x}}_i, \mathbf{y}_i, \mathbf{y}_{i-1}), \tag{3.6}$$

[1] To make things simpler, we will assume later on that edge features do not depend on the input $\bar{\mathbf{x}}$—but they could, without changing at all the decoding algorithm.

Algorithm 10 Averaged Structured perceptron

1: **input:** dataset $\mathcal{D}$, 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 $(\bar{\mathbf{x}}^m, \bar{\mathbf{y}}^m)$ and predict using the current model:

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

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

where

$$Z(\boldsymbol{\theta}, x) = \sum_{\mathbf{y} \in \mathbf{Y}} \prod_i \psi_V(\bar{\mathbf{x}}_i, \mathbf{y}_i)\psi_E(\bar{\mathbf{x}}_i, \mathbf{y}_i, \mathbf{y}_{i-1}) \tag{3.7}$$

is the partition function.

There are three important problems that need to be solved:

1. Given $\bar{\mathbf{x}}$, computing the most likely output sequence $\bar{\mathbf{y}}$ (the one which maximizes $P(\bar{\mathbf{y}}|\bar{\mathbf{x}})$.
2. Compute the posterior marginals $P(\mathbf{y}_i|\bar{\mathbf{x}})$ at each position i.
3. Compute the partition function.

Interestingly, all these problems can be solved by using the same algorithms (just changing the potentials) that were already implemented for HMMs: the Viterbi algorithm (for 1) and the forward-backward algorithm (for 2–3).

3.2 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 §**??** on the perceptron algorithm and compare it against the structured perceptron (Algorithm **??**).

There are only two differences:

- Instead of finding $\arg\max_{y' \in \mathcal{Y}}$ for a given variable, it finds the $\arg\max_{\bar{\mathbf{y}}'}$, the best sequence. We can do this by using the Viterbi algorithm with the node and edge potentials (actually, the log of those potentials) defined in Eqs. **??**–**??**, along with the assumption that the features decompose as the model, as explained in the previous section.
- Instead of updating the features for the entire y' (in this case $\bar{\mathbf{y}}$) we update the features only at the positions where the labels are different (note that step 6 of algorithm **??** just keeps the current value of the weights if the predicted and true labels are the same).

Exercise 3.1 *In this exercise you will test the structured perceptron algorithm using different feature sets for Part-of-Speech Tagging. Start with the code below, which uses the ID feature set from table* **??**.

```
import sequences.structured_perceptron as spc
import sequences.crf_batch as crfc
import readers.pos_corpus as pcc
import sequences.id_feature as idfc
import sequences.extended_feature as exfc
```

```
print "Perceptron Exercise"
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)
corpus.add_sequence_list(train_seq)
id_f = idfc.IDFeatures(corpus)
id_f.build_features()
sp = spc.StructuredPercetron(corpus,id_f)
sp.nr_rounds = 20
sp.train_supervised(train_seq.seq_list)
```

You will get some messages about "unknown tags", which are a consequence of using a reduced set of 12 POS tags instead of the full set. You will also receive feedback when each epoch is finished.

After training is done, evaluate the learned model on the training, development and test sets.

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

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

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

You should get values similar to these:

```
Out[]: Structured Percetron - ID Features Accuracy Train: 0.968 Dev: 0.864 Test: 0.852
```

Compare with the results achieved with the HMM model (0.838 on the test set, from the previous lecture). Even using a similar feature set the structured perceptron yields better results than the HMM from the previous lecture. Perform some error analysis and figure out what are the main errors the perceptron 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.*

```
ex_f = exfc.ExtendedFeatures(corpus)
ex_f.build_features()
sp = spc.StructuredPercetron(corpus,ex_f)
sp.nr_rounds = 20
sp.train_supervised(train_seq.seq_list)

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

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

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

You should get values close to the following:

```
Structured Percetron - Extended Features Accuracy Train: 0.970 Dev: 0.913 Test: 0.903
```

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.

3.3 Conditional Random Fields

Conditional Random Fields (CRF) (**?**) can be seen as an extension of the Maximum Entropy (ME) models to structured problems.[2]

CRFs are *globally* normalized models: the probability of a given sentence is given by Equation **??**. Going from a maximum entropy model (in multi-class classification) to a CRF mimics the transition discussed above from perceptron to structured perceptron:

- Instead of finding the posterior marginal $P(y'|x)$ for a given variable, it finds the posterior marginals for all factors (nodes and edges), $P(\bar{\mathbf{y}}_i|\bar{\mathbf{x}})$ and $P(\bar{\mathbf{y}}_i, \bar{\mathbf{y}}_{i-1}|\bar{\mathbf{x}})$. We can compute these quantities by using the forward-backward algorithm with the node and edge potentials defined in Eqs. **??**–**??**, along with the assumption that the features decompose as the model, as explained in the previous section.
- The features are updated factor wise (i.e., for each node and edge).

Algorithm **??** shows the pseudo code to optimize a CRF with a batch gradient method (in the exercise, we will use a quasi-Newton method, L-BFGS). Again, we can also take an online approach to optimization, but here we will stick with the batch one.

Exercise 3.3 *Repeat Exercises* **??**–**??** *using a CRF model instead of the perceptron algorithm. Report the results.*

Here is the code for the simple feature set (note: this code doesn't give feedback while it's running, and it can take several minutes to run, even on a fast computer; just be patient):

```
crf = crfc.CRF_batch(corpus,id_f)
crf.train_supervised(train_seq.seq_list)

pred_train = crf.viterbi_decode_corpus(train_seq.seq_list)
pred_dev = crf.viterbi_decode_corpus(dev_seq.seq_list)
pred_test = crf.viterbi_decode_corpus(test_seq.seq_list)

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

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

[2] 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).

Algorithm 11 Batch Gradient Descent for Conditional Random Fields

input: $\mathcal{D}$, λ, number of rounds T, learning rate sequence $(\eta_t)_{t=1,\ldots,T}$
initialize $\boldsymbol{\theta}^1 = \mathbf{0}$
for $t = 1$ **to** T **do**
  for $m = 1$ **to** M **do**
    take training pair (x^m, y^m) and compute conditional probabilities using the current model, for each $\bar{\mathbf{y}}$:

$$P_{\boldsymbol{\theta}^t}(\bar{\mathbf{y}}|\bar{\mathbf{x}}) = \frac{1}{Z(\boldsymbol{\theta}^t, \bar{\mathbf{x}})} \exp\left(\sum_i \boldsymbol{\theta}_V^t \cdot \boldsymbol{f}_V(\bar{\mathbf{x}}, \mathbf{y}_i) + \sum_i \boldsymbol{\theta}_E^t \cdot \boldsymbol{f}_E(\bar{\mathbf{x}}, \mathbf{y}_i, \mathbf{y}_{i-1})\right)$$

    compute the feature vector expectation:

$$E_{\boldsymbol{\theta}^t}[\boldsymbol{f}(\bar{\mathbf{x}}^m, \bar{\mathbf{y}}^m)] = \sum_{\bar{\mathbf{y}}} P_{\boldsymbol{\theta}^t}(\bar{\mathbf{y}}^m|\bar{\mathbf{x}}^m)\boldsymbol{f}(\bar{\mathbf{x}}^m, \bar{\mathbf{y}}^m)$$

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

$$\boldsymbol{\theta}^{t+1} \leftarrow (1 - \lambda\eta_t)\boldsymbol{\theta}^t + \eta_t M^{-1} \sum_{m=1}^{M} \left(\boldsymbol{f}(\bar{\mathbf{x}}^m, \bar{\mathbf{y}}^m) - E_{\boldsymbol{\theta}^t}[\boldsymbol{f}(\bar{\mathbf{x}}^m, \bar{\mathbf{y}}^m)]\right)$$

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

You should get close to these results (apart from some warnings which you can safely ignore):

```
CRF - ID Features Accuracy Train: 0.936 Dev: 0.858 Test: 0.853
```

Here is the code for the extended feature set:

```
crf = crfc.CRF_batch(corpus,ex_f)
crf.train_supervised(train_seq.seq_list)

pred_train = crf.viterbi_decode_corpus(train_seq.seq_list)
pred_dev = crf.viterbi_decode_corpus(dev_seq.seq_list)
pred_test = crf.viterbi_decode_corpus(test_seq.seq_list)

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

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

And here are the expected results:

```
CRF - Extended Features Accuracy Train: 0.595 Dev: 0.587 Test: 0.591
```

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.

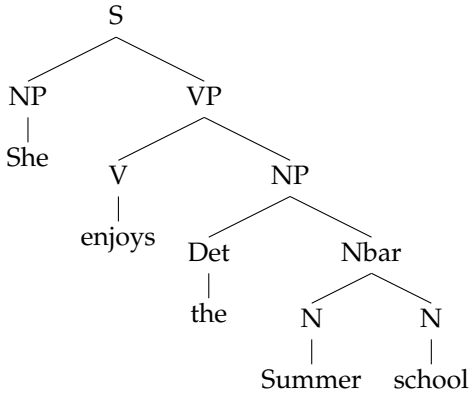
[2] This does not mean that natural languages are context free. There is an immense body of work on grammar formalisms that relax the "context-free" assumption, and those formalisms have been endowed with a probabilistic framework as well. Examples are: lexical functional grammars, head-driven phrase 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:



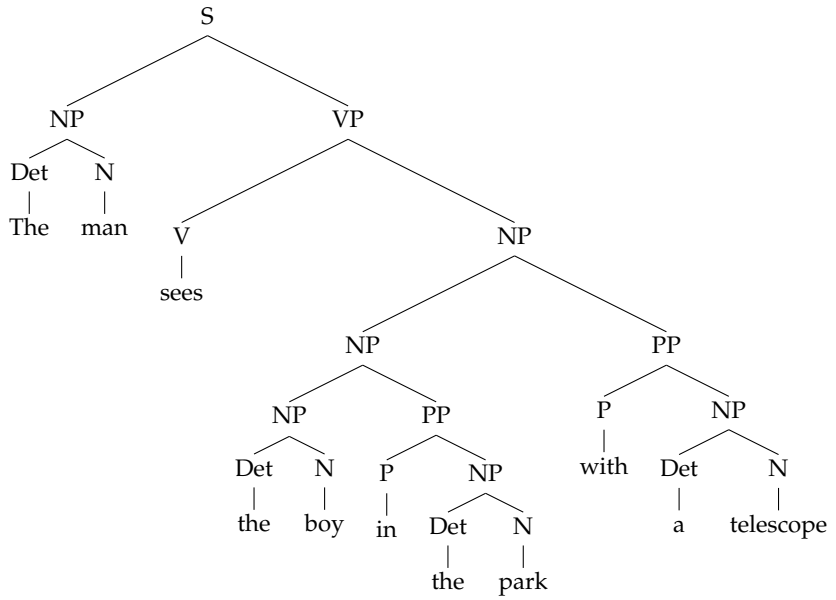
4.1.2 Ambiguity

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

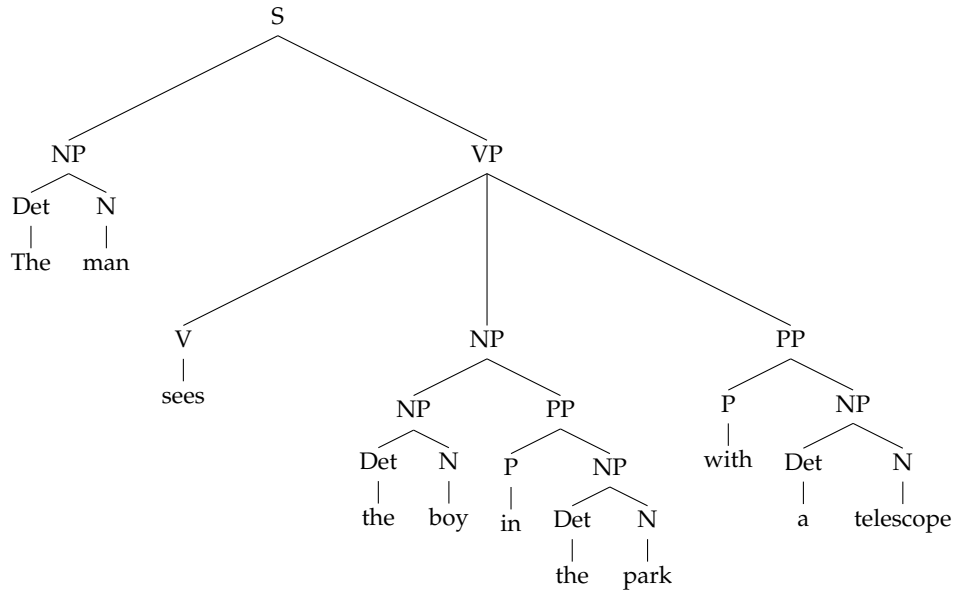
```
The man sees the boy in the park with a telescope.
```

for which all the following parse trees are plausible interpretations:

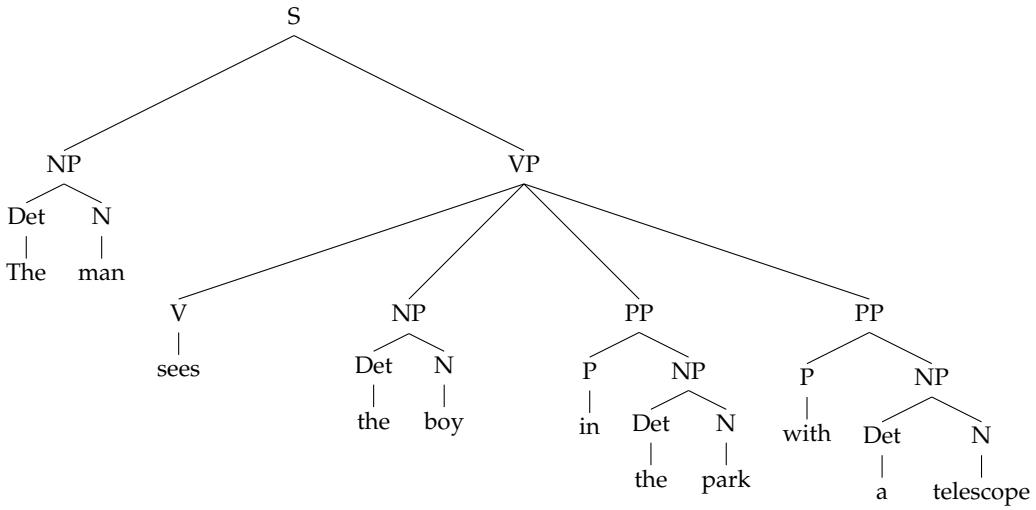
1. The boy is in a park and he has a telescope:



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 $G_{\boldsymbol{\theta}} = \langle \mathcal{N}, \mathcal{T}, \mathcal{R}, \mathrm{S}, \boldsymbol{\theta} \rangle$, where $\langle \mathcal{N}, \mathcal{T}, \mathcal{R}, \mathrm{S} \rangle$ 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 $Z \to XY$ is endowed a conditional probability

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

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

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

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

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

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 12 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

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

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. **??**.

[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.

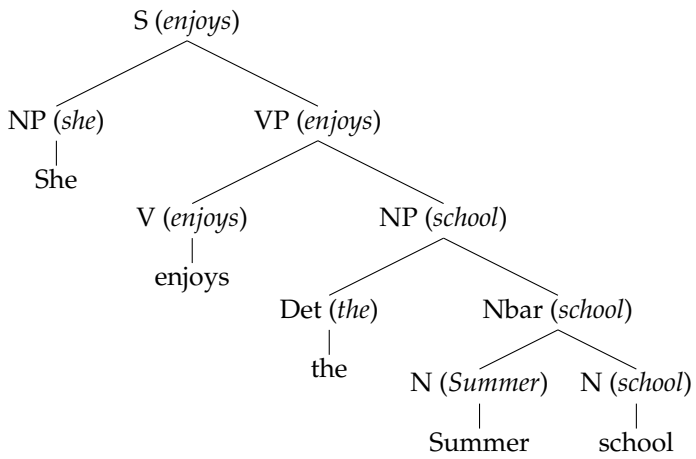


Figure 4.1: A lexicalized parse tree for the sentence *She enjoys the Summer school*.

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. **??** 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. **??**. If we drop the phrase constituents and keep only the head words, the parse tree would become:

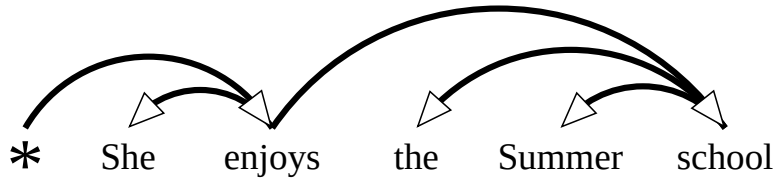
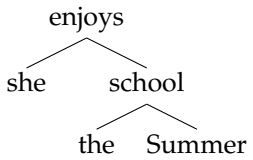


Figure 4.2: A dependency tree for the sentence *She enjoys the Summer school*. Note the additional dummy root symbol (*) which is included for convenience.



This representation is called a *dependency tree*; it can be alternatively represented as shown in Fig. **??**. 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. **??** 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.

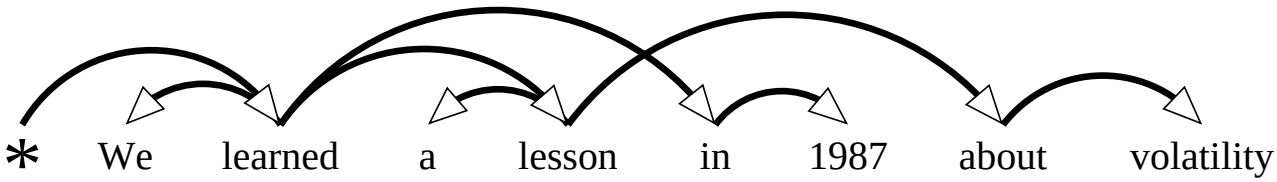


Figure 4.3: A non-projective parse tree.

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 $O(N^2)$.

```
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:*

```
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:*

```
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`*:*

```
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 (**?**).