LxMLS - Lab Guide

July 8, 2019

Day 0

Basic Tutorials

0.1 Today's assignment

In this class we will introduce several fundamental concepts needed further ahead. We start with an introduction to Python, the programming language we will use in the lab sessions, and to Matplotlib and Numpy, two modules for plotting and scientific computing in Python, respectively. Afterwards, we present several notions on probability theory and linear algebra. Finally, we focus on numerical optimization. The goal of this class is to give you the basic knowledge for you to understand the following lectures. We will not enter in too much detail in any of the topics.

0.2 Manually Installing the Tools in your own Computer

0.2.1 Desktops vs. Laptops

If you have decided to use one of our provided desktops, all installation procedures have been carried out. You merely need to go to the `lxmls-toolkit-student` folder inside your home directory and start working! You may go directly to section 0.3. If you wish to use your own laptop, you will need to install Python, the required Python libraries and download the LXMLS code base. It is important that you do this as soon as possible (before the school starts) to avoid unnecessary delays. Please follow the install instructions.

0.2.2 Downloading the labs version from GitHub student branch

The code of LxMLS is available online at GitHub. There are two branches of the code: the `master` branch contains fully functional code. **important**: The `student` branch contains the same code with some parts deleted, which you must complete in the following exercises. Download the `student` code by going to

`https://github.com/LxMLS/lxmls-toolkit`

and select the `student` branch in the dropdown menu. This will reload the page to the corresponding branch. Now you just need to click the `clone or download` button to obtain the lab tools in a zip format:

`lxmls-toolkit-student.zip`

After this you can unzip the file where you want to work and enter the unzipped folder. This will be the place where you will work.

0.2.3 Installing Python from Scratch with Anaconda

If you are new to Python the best option right now is the Anaconda platform. You can find installers for Windows, Linux and OSX platforms here

`https://www.anaconda.com/download/`
`https://anaconda.org/pytorch/pytorch`

Finally install the LXMLS toolkit symbolically. This will allow you to modify the code and see the changes take place immediately.

```
python setup.py develop
```

The guide supports both Python2 and Python3. We strongly recommend that you use Python3 as Python2 is being deprecated.

0.2.4 Installing with Pip

If you are familiar with Python you will probably be used to the pip package installer. In this case it might be more easy for you to install the packages yourself using pip and a virtual environment. This will avoid conflicting with existing python installations. To install and create a virtual environment do

```
cd lxmls-toolkit-student
sudo pip install virtualenv
virtualenv venv
source venv/bin/activate
pip install --editable .
```

Note: If you are experiencing issues on Windows, you would probably have to install pytorch first.

```
pip install https://download.pytorch.org/whl/cpu/torch-1.1.0-cp36-cp36m-win_amd64.whl
pip install https://download.pytorch.org/whl/cpu/torchvision-0.3.0-cp36-cp36m-win_amd64.wh
```

0.2.5 (Advanced Users) Virtualizing Python version as well

If you also need to install a new python version, e.g. your systems Python is still Python2 and you can not change this, you can virtualize different Python versions as well. Have a look at pyenv. This will hijack your python binary and allow you switch between Python 2, 3 or install concrete versions for a particular folder. To install pyenv

```
git clone https://github.com/pyenv/pyenv.git ~/.pyenv
```

and add the following to your .bashrc or .bash_profile

```
export PYENV_ROOT="$HOME/.pyenv"
export PATH="$PYENV_ROOT/bin:$PATH"
eval "$(pyenv init -)
```

be careful if you already redefine Python's path in these files. If you do not feel comfortable with this its better to ask for help. Once installed you can do the following in the LxMLS folder

```
source ~/.bashrc
pyenv install 3.6.0
pyenv local 3.6.0
```

to install and set the version of python for that folder to Python 3.6.0

0.2.6 (Advanced Users) Forking and cloning the code on GitHub

It might be the case that you feel very comfortable with scientific Python, know some git/GitHub and want to extend/improve our code base. In that case you can directly clone the project with

```
git clone https://github.com/LxMLS/lxmls-toolkit.git
cd lxmls-toolkit/
git checkout student
pip install --editable .
```

If you want to contribute to the code-base, you can make pull requests to the *develop* branch or raise issues.

0.2.7 Deciding on the IDE and interactive shell to use

An Integrated Development Environment (IDE) includes a text editor and various tools to debug and interpret complex code.

Important: As the labs progress you will need an IDE, or at least a good editor and knowledge of pdb/ipdb. This will not be obvious the first days since we will be seeing simpler examples.

Easy IDEs to work with Python are PyCharm and Visual Studio Code, but feel free to use the software you feel more comfortable with. PyCharm and other well known IDEs like Spyder are provided with the Anaconda installation.

Aside of an IDE, you will need an interactive command line to run commands. This is very useful to explore variables and functions and quickly debug the exercises. For the most complex exercises you will still need an IDE to modify particular segments of the provided code. As interactive command line we recommend the Jupyter notebook. This also comes installed with Anaconda and is part of the pip-installed packages. The Jupyter notebook is described in the next section. In case you run into problems or you feel uncomfortable with the Jupyter notebook you can use the simpler iPython command line.

0.2.8 Jupyter Notebook

Jupyter is a good choice for writing Python code. It is an interactive computational environment for data science and scientific computing, where you can combine code execution, rich text, mathematics, plots and rich media. The Jupyter Notebook is a web application that allows you to create and share documents, which contains live code, equations, visualizations and explanatory text. It is very popular in the areas of data cleaning and transformation, numerical simulation, statistical modeling, machine learning and so on. It supports more than 40 programming languages, including all those popular ones used in Data Science such as Python, R, and Scala. It can also produce many different types of output such as images, videos, LaTex and JavaScript. More over with its interactive widgets, you can manipulate and visualize data in real time.
The main features and advantages using the Jupyter Notebook are the following:

- In-browser editing for code, with automatic syntax highlighting, indentation, and tab completion/introspection.
- The ability to execute code from the browser, with the results of computations attached to the code which generated them.
- Displaying the result of computation using rich media representations, such as HTML, LaTeX, PNG, SVG, etc. For example, publication-quality figures rendered by the matplotlib library, can be included inline.
- In-browser editing for rich text using the Markdown markup language, which can provide commentary for the code, is not limited to plain text.
- The ability to easily include mathematical notation within markdown cells using LaTeX, and rendered natively by MathJax.

The basic commands you should know are

Esc Enter	Enter command mode Enter edit mode
up/down Ctrl + Enter Shift + Enter	Change between cells Runs code on selected cell Runs code on selected cell, jumps to next cell
restart button	Deletes all variables (useful for troubleshooting)

Table 1: Basic Jupyter commands

A more detailed user guide can be found here:

`http://jupyter-notebook-beginner-guide.readthedocs.io/en/latest/index.html`

0.3 Solving the Exercises

In the student branch we provide the `solve.py` script. This can be used to solve the exercises of each day, e.g.,

```
python ./solve.py linear_classifiers
```

where the solvable days are: linear_classifiers, parsing, non-linear_classifiers, non-linear_sequence_models, sequence_models. You can also undo the solving of an exercise by using

```
python ./solve.py --undo linear_classifiers
```

Note that this script just downloads the master or student versions of certain files from the GitHub repository. It needs an Internet connection. Since some exercises require you to have the exercises of the previous days completed, the monitors may ask you to use this function. **Important:** Remember to save your own version of the code, otherwise it will be overwritten!

0.4 Python

0.4.1 Python Basics

Pre-requisites

At this point you should have installed the needed packages. You need also to feel comfortable with an IDE to edit code and an interactive command line. See previous sections for the details. Your work folder will be

```
lxmls-toolkit-student
```

from there, start your interactive command line of choosing, e.g.,

```
jupyter-notebook
```

and proceed with the following sections.

Running Python code

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

From a file – Create a file named `yourfile.py` and write your program in it, using the IDE of your choice, e.g., PyCharm:

```
print('Hello World!')
```

After saving and closing the file, you can run your code by using the run functionality in your IDE. If you wish to run from a command line instead do

```
python yourfile.py
```

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

In the interactive command line – Start your preferred interactive command line, e.g., Jupyter-notebook. There, you can run Python code by simply writing it and pressing enter (ctr+enter in Jupyter).

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

However, you can also run Python code written into a file.

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

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

Help and Documentation

There are several ways to get help on Jupyter:

- Adding a question mark to the end of a function or variable and pressing Enter brings up associated documentation. Unfortunately, not all packages are well documented. Numpy and matplotlib are pleasant exceptions;
- `help('if')` gets the online documentation for the `if` keyword;
- `help()`, enters the help system.
- When at the help system, type `q` to exit.

0.4.2 Python by Example

Basic Math Operations

Python supports all basic arithmetic operations, including exponentiation. For example, the following code:

```
print(3 + 5)
print(3 - 5)
print(3 * 5)
print(3 / 5)
print(3 ** 5)
```

will produce the following output in Python 2:

```
8
-2
15
0
243
```

and the following output in Python 3:

```
8
-2
15
0.6
243
```

Important: Notice that in Python 2 division is always considered as integer division, hence the result being 0 on the example above. To force a floating point division in Python 2 you can force one of the operands to be a floating point number:

```
print(3 / 5.0)
0.6
```

For Python 3, the division is considered float point, so the operation (3 / 5) or (3 / 5.0) is always 0.6.

Also, notice that the symbol `**` is used as exponentation operator, unlike other major languages which use the symbol `^`. In fact, the `^` symbol has a different meaning in Python (bitwise XOR) so, in the beginning, be sure to double-check your code if it uses exponentiation and it is giving unexpected results.

Data Structures

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

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

A string should be surrounded by either apostrophes (') or quotes (“). You can access a list with the following:

- `len(L)`, which returns the number of items in L;
- `L[i]`, which returns the item at index i (the first item has index 0);
- `L[i:j]`, which returns a new list, containing all the items between indexes i and $j-1$, inclusively.

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

Loops and Indentation

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

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

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

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

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

and its output:

```
1
2
3
```

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

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

Bear in mind that indentation is often the main source of errors when starting to work with Python. Try to get used to it as quickly as possible. It is also recommendable to use a text editor that can display all characters e.g. blank space, tabs, since these characters can be visually similar but are considered different by Python. One of the most common mistakes by newcomers to Python is to have their files indented with spaces on some lines and with tabs on other lines. Visually it might appear that all lines have proper indentation, but you will get an `IndentationError` message if you try it. The recommended[1] way is to use 4 spaces for each indentation level.

Control Flow

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

```
hour = 16
if hour < 12:
    print('Good morning!')
elif hour >= 12 and hour < 20:
    print('Good afternoon!')
else:
    print('Good evening!')
```

Functions

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

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

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

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

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

[1] The PEP8 document (`www.python.org/dev/peps/pep-0008`) is the official coding style guide for the Python language.

Profiling

If you are interested in checking the performance of your program, you can use the command `%prun` in Jupyter. For example:

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

%prun myfunction(22)
```

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

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

Debugging in Python

During the lab sessions, there will be situations in which we will use and extend modules that involve elaborated code and statements, like classes and nested functions. Although desirable, it should not be necessary for you to fully understand the whole code to carry out the exercises. It will suffice to understand the algorithm as explained in the theoretical part of the class and the local context of the part of the code where we will be working. For this to be possible is very important that you learn to use an IDE.

An alternative to IDEs, that can also be useful for quick debugging in Jupyter, is the pdb module. This will stop the execution at a given point (called break-point) to get a quick glimpse of the variable structures and to inspect the execution flow of your program. The ipdb is an improved version of pdb that has to be installed separately. It provides additional functionalities like larger context windows, variable auto complete and colors. Unfortunately ipdb has some compatibility problems with Jupyter. We therefore recommend to use ipdb only in spartan configurations such as vim+ipdb as IDE.
In the following example, we use this module to inspect the `greet` function:

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

Load the new definition of the function by writing this code in a file or a Jupyter cell and running it. Now, if you try `greet(50)` the code execution should stop at the place where you located the break-point (that is, in the `print('Good evening!')` statement). You can now run new commands or inspect variables. For this purpose there are a number of commands you can use[2], but we provide here a short table with the most useful ones:
Getting back to our example, we can type n(ext) once to execute the line we stopped at

[2]The complete list can be found at `http://docs.python.org/library/pdb.html`

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

Table 2: Basic pdb/ipdb commands, parentheses indicates abbreviation

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

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

```
pdb> pp hour
50
```

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

0.4.3 Exceptions

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

```
10/0
```

A *ZeroDivisionError* exception was raised, and no output was returned. Exceptions can also be forced to occur by the programmer, with customized error messages [3].

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

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

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

[3] For a complete list of built-in exceptions, see `http://docs.python.org/3/library/exceptions.html`

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

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

Extending basic Functionalities with Modules

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

```
import numpy as np
```

Then we can run the following on the Jupyter command line:

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

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

Organizing your Code with your own modules

Creating you own modules is extremely simple. you can for example create the file in your work directory

`my_tools.py`

and store there the following code

```
def my_print(input):
    print(input)
```

From Jupyter you can now import and use this tool as

```
import my_tools
my_tools.my_print("This works!")
```

Important: When you modify a module, you need to reload the notebook page for the changes to take effect. Autoreload is set by defect in the schools notebooks. Autoreload is set by defect in the school's notebooks. for the latter. Other ways of importing one or all the tools from a module are

```
from my_tools import my_print  # my_print directly accesible in code
from my_tools import *         # will make all functions in my_tools accessible
```

However, this makes reloading the module more complicated. You can also store tools ind different folders. For example, if you store the previous example in the folder

`day0_tools`

and store inside an empty file called

`__init__.py`

then the following import will work

```
import day0_tools.my_tools
```

0.4.4 Matplotlib – Plotting in Python

Matplotlib[4] 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 Jupyter, which will introduce you to some of the basic numeric and plotting operations.*

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

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

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

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

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

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

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

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

[4] http://matplotlib.org/

0.4.5 Numpy – Scientific Computing with Python

Numpy[5] is a library for scientific computing with Python.

Multidimensional Arrays

The main object of numpy is the multidimensional array. A multidimensional array is a table with all elements of the same type and can have several dimensions. Numpy provides various functions to access and manipulate multidimensional arrays. In one dimensional arrays, you can index, slice, and iterate as you can with lists. In a two dimensional array M, you can perform these operations along several dimensions.

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

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

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

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

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

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

Mathematical Operations

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

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

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

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

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

```
import numpy as np

A = np.arange(100)

# These two lines do exactly the same thing
print(np.mean(A))
print(A.mean())
```

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

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

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

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

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

Use numpy to implement this for $f(x) = x^2$*. You should not need to use any loops. Note that integer division in Python 2.x returns the floor division (use floats – e.g.* $5.0/2.0$ *– to obtain rationals). The exact value is* $1/3$*. How close is the approximation?*

0.5 Essential Linear Algebra

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

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

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

$$\mathbf{Ax} = \mathbf{b}$$

with

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

0.5.1 Notation

We use the following notation:

- By $\mathbf{A} \in \mathbb{R}^{m \times n}$, we denote a **matrix** with m rows and n columns, where the entries of A are real numbers.
- By $\mathbf{x} \in \mathbb{R}^n$, 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 $\mathbf{x}^T$, the transpose of $\mathbf{x}$.
- The ith element of a vector $\mathbf{x}$ is denoted x_i:

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

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[6] 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
```

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.

0.5.2 Some Matrix Operations and Properties

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

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

Exercise 0.9 *You can multiply two matrices by looping over both indexes and multiplying the individual entries.*

```
a = np.array([[2,3],[3,4]])
b = np.array([[1,1],[1,1]])
a_dim1, a_dim2 = a.shape
b_dim1, b_dim2 = b.shape
c = np.zeros([a_dim1,b_dim2])
for i in range(a_dim1):
    for j in range(b_dim2):
        for k in range(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: $(\mathbf{AB})\mathbf{C} = \mathbf{A}(\mathbf{BC})$.
- Matrix multiplication is distributive: $\mathbf{A}(\mathbf{B}+\mathbf{C}) = \mathbf{AB} + \mathbf{AC}$.

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

- Matrix multiplication is (generally) not commutative : $\mathbf{AB} \neq \mathbf{BA}$.
- Given two vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ the product $\mathbf{x}^T\mathbf{y}$, called **inner product** or **dot product**, is given by

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

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

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

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

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

- The **identity matrix**, denoted $\mathbf{I} \in \mathbb{R}^{n \times n}$, 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 $\mathbf{A} \in \mathbb{R}^{n \times n}$, $\mathbf{AI} = \mathbf{A} = \mathbf{IA}$.

```
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 $\mathbf{A} \in \mathbb{R}^{m \times n}$, the transpose $\mathbf{A}^T \in \mathbb{R}^{n \times m}$ is the $n \times m$ matrix whose entries are given by $(A^T)_{ij} = A_{ji}$.

  Also, $(\mathbf{A}^T)^T = \mathbf{A}$; $(\mathbf{AB})^T = \mathbf{B}^T\mathbf{A}^T$; $(\mathbf{A} + \mathbf{B})^T = \mathbf{A}^T + \mathbf{B}^T$

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

0.5.3 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

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

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

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

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

0.5.4 Linear Independence, Rank, and Orthogonal Matrices

A set of vectors $\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\} \subset \mathbb{R}^m$ 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

$$\mathbf{x}_j = \sum_{i \neq j} \alpha_i \mathbf{x}_i$$

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

$$\mathbf{U}^T\mathbf{U} = \mathbf{I} = \mathbf{U}\mathbf{U}^T.$$

0.6 Probability Theory

Probability is the most used 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* $A = \{HH, HT\}$.

A 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 for the computation of the joint probability.

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

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

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

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

for every finite subset J of I.

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

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

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

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

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

Using Equations 2 and 3, one can derive the 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, the Bayes' theorem is used to compute the estimates of distribution parameters from data. Here, prior is the initial* belief *about the parameters, likelihood is the distribution function of the*

parameter (usually trained from data) and posterior is the updated belief *about the parameters.*

0.6.1 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:

0.6.2 Bernoulli

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

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

0.6.3 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.

0.6.4 Categorical

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

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

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

0.6.5 Multinomial

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

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

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

0.6.6 Gaussian Distribution

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

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

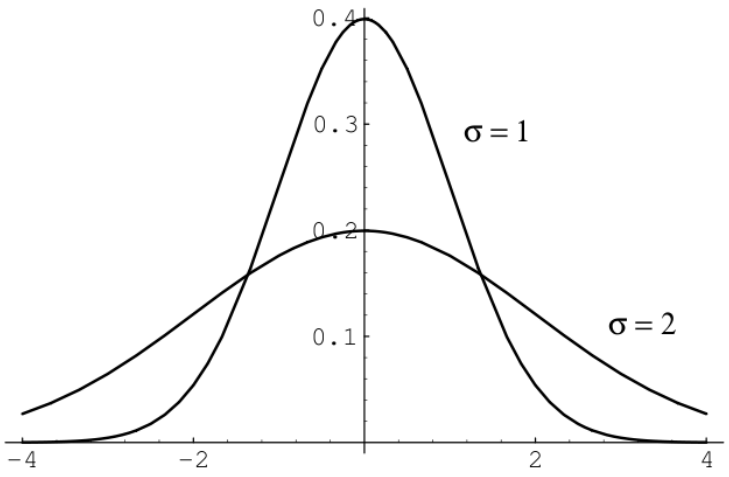

Figure 1: Normal density for two sets of parameter values.

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

0.6.7 Maximum Likelihood Estimation

Until now we assumed that, for every distribution, the parameters θ are known and are used when we calculate $p(x|\theta)$. There are some cases where the values of the parameters are easy to infer, such as the probability p of

getting a head using a fair coin, used on a Bernoulli or Binomial distribution. However, in many problems, these values are complex to define and it is more viable to estimate the parameters using the data x. For instance, in the example above with the coin toss, if the coin is somehow tampered to have a biased behavior, rather than examining the dynamics or the structure of the coin to infer a parameter for p, a person could simply throw the coin n times, count the number of heads h and set $p = \frac{h}{n}$. By doing so, the person is using the data x to estimate θ.

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

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

where $x_1, ..., x_n$ are independently and identically distributed (i.i.d.) samples.

To understand this concept better, we go back to the tampered coin example again. Suppose that we throw the coin 5 times and get the sequence [1,1,1,1,1] (1=head, 0=tail). Using the Bernoulli distribution (see Section 0.6.2) f to model this problem, we get the following likelihood values:

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

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

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

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

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

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

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

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

And the MLE estimate would be given by:

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

which resolves to

$$p_{mle} = 0.6$$

Exercise 0.10 *Over the next couple of exercises we will make use of the Galton dataset, a dataset of heights of fathers and sons from the 1877 paper that first discussed the "regression to the mean" phenomenon. This dataset has 928 pairs of numbers.*

- *Use the* `load()` *function in the* `galton.py` *file to load the dataset. The file is located under the* `lxmls/readers` *folder*[7]*. Type the following in your Python interpreter:*

```
import galton as galton
galton_data = 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?*

0.6.8 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 $\mathcal{F}$.*

0.7 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 (Nocedal and Wright, 1999; Bertsekas et al., 1995; Boyd and Vandenberghe, 2004).

The most common way to solve the problems when no closed form solution is available is to resort to an iterative algorithm. In this Section, we will see some of these iterative optimization techniques. These iterative algorithms construct a sequence of points $x^{(0)}, x^{(1)}, \ldots \in \text{domain}(f)$ such that hopefully $x^t = x^*$ after a number of iterations. Such a sequence is called the **minimizing sequence** for the problem.

0.7.1 Convex Functions

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

Intuitively, imagine dropping a ball on either side of Figure 2, the ball will roll 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

[7] You might need to inform python about the location of the lxmls labs toolkit. To do so you need to `import sys` and use the `sys.path.append` function to add the path to the toolkit readers.

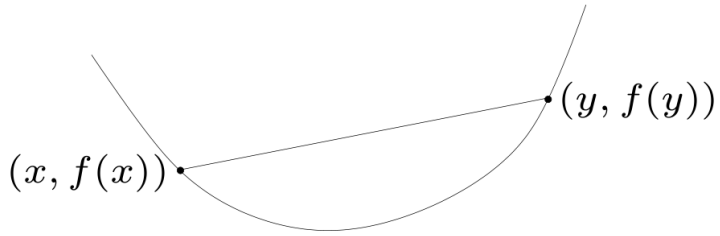

Figure 2: Illustration of a convex function. The line segment between any two points on the graph lies entirely above the curve.

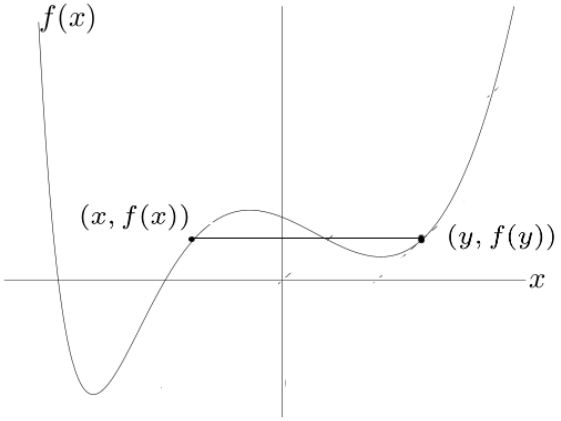

Figure 3: Illustration of a non-convex function. Note the line segment intersecting the curve.

you drop a ball from the left side of Figure 3 it will reach a different position than if you drop a ball from its right side. Moreover, dropping it from the left side will lead you to a much better (*i.e.*, lower) place than if you drop the ball from the right side. This is the main problem with non-convex functions: there are no guarantees about the quality of the local minimum you find.

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

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

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

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

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

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

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

for all $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n, 0 \leq \theta \leq 1$

0.7.2 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:

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

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

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

Table 3: Some derivative examples

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

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

Example 0.3 *Consider the function $h(x) = \exp(x^2)$, this can be decomposed as $h(x) = f(g(x)) = f(u) = \exp(u)$, where $u = g(x) = x^2$ and has derivative $\frac{\partial h}{\partial x} = \frac{\partial f}{\partial u} \cdot \frac{\partial u}{\partial x} = \exp(u) \cdot 2x = \exp(x^2) \cdot 2x$*

Exercise 0.11 *Consider the function $f(x) = x^2$ and its derivative $\frac{\partial f}{\partial x}$. Look at the derivative of that function at points [-2,0,2], draw the tangent to the graph in that point $\frac{\partial f}{\partial x}(-2) = -4$, $\frac{\partial f}{\partial x}(0) = 0$, and $\frac{\partial f}{\partial x}(2) = 4$. For example, the tangent equation for $x = -2$ is $y = -4x - b$, where $b = f(-2)$. The following code plots the function and the derivatives on those points using matplotlib (See Figure 4).*

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

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

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

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

0.7.3 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 $f(\mathbf{x}) : \mathbb{R}^n \to R$ as well as the gradient of the function $\nabla_{\mathbf{x}} f(\mathbf{x})$. 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 $\mathbf{x}_0$ and compute the gradient at that point $\nabla_{\mathbf{x}_0} f(\mathbf{x})$. You then take a step of length η on the direction of the negative gradient to find a new point: $\mathbf{x}_1 = \mathbf{x}_0 - \eta \nabla_{\mathbf{x}_0} f(\mathbf{x})$. Then, you compute the gradient at this new point, $\nabla_{\mathbf{x}_1} f(\mathbf{x})$, and take a step of length η on the direction of the negative gradient to find a new point: $\mathbf{x}_2 = \mathbf{x}_1 - \eta \nabla_{\mathbf{x}_1} f(\mathbf{x})$. You proceed until you have reached a minimum (local or global). Recall from the previous subsection that you can identify the minimum by testing if the norm of the gradient is zero: $||\nabla f(\mathbf{x})|| = 0$.

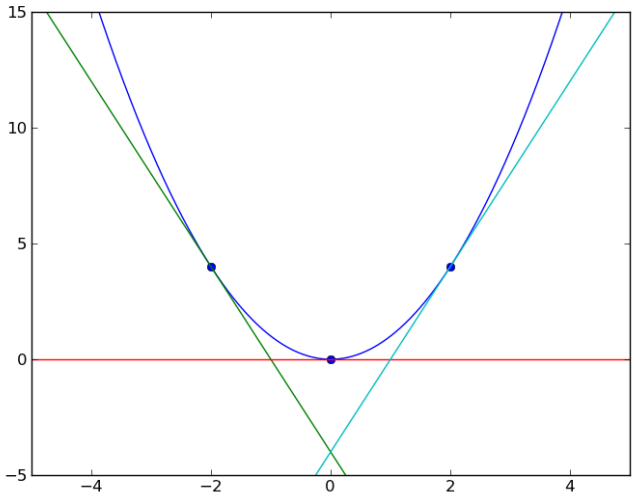

Figure 4: Illustration of the gradient of the function $f(x^2)$ at three different points $x = [-2, 0, 2]$. Note that at point $x = 0$ the gradient is zero which corresponds to the minimum of the function.

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** (Nocedal and Wright, 1999).
- **Descent Direction** A second problem is that using the negative gradient as direction can lead to a very slow convergence. Different methods that change the descent direction by multiplying the gradient by a matrix $\mathbf{B}$ have been proposed that guarantee a faster convergence. Two notable methods are the Conjugate Gradient (CG) and the Limited Memory Quasi Newton methods (LBFGS) (Nocedal and Wright, 1999).
- **Stopping Criteria** Finally, it will normally not be possible to reach full convergence either because it will be too slow, or because of numerical issues (computers cannot perform exact arithmetic). So normally we need to define a stopping criteria for the algorithm. Three common criteria (that are normally used together) are: a maximum number of iterations; the gradient norm be smaller than a given threshold $||\nabla f(\mathbf{x})|| \leq \eta_1$, or the normalized difference in the function value be smaller than a given threshold $\frac{|f(\mathbf{x}_t) - f(\mathbf{x}_{t-1})|}{\max(|f(\mathbf{x}_t)|, |f(\mathbf{x}_{t-1})|)} \leq \eta_2$

Algorithm 1 shows the general gradient based algorithm. Note that for the simple gradient descent algorithm $\mathbf{B}$ is the identity matrix and the descent direction is just the negative gradient of the function.

Algorithm 1 Gradient Descent

1: **given** a starting point $\mathbf{x}_0, i = 0$
2: **repeat**
3: Compute step size η
4: Compute descent direction $-\mathbf{B}\nabla f(x_i)$.
5: $\mathbf{x}_{i+1} \leftarrow \mathbf{x}_i - \eta \mathbf{B} \nabla f(\mathbf{x}_i)$
6: $i \leftarrow i + 1$
7: **until** stopping criterion is satisfied.

Exercise 0.12 *Consider the function* $f(x) = (x+2)^2 - 16\exp\left(-(x-2)^2\right)$. *Make a function that computes the function value given* x.

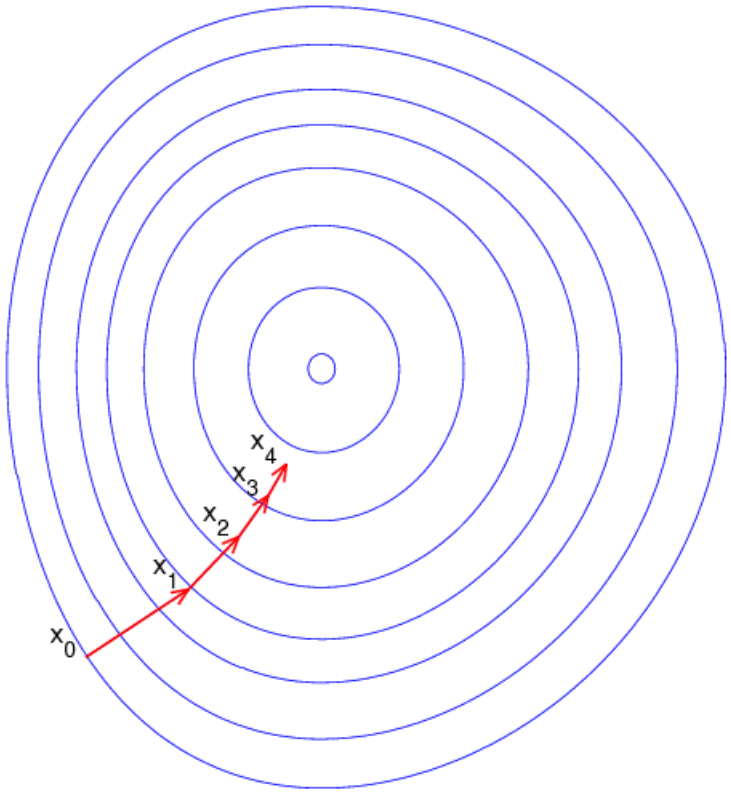

Figure 5: Illustration of gradient descent. The blue circles correspond to contours of the function (each blue circle is a set of points which have the same function value), while the red lines correspond to steps taken in the negative gradient direction.

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

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

```
x = np.arange(-8, 8, 0.001)
y = get_y(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_scalar(start_x, func, grad, step_size=0.1, prec=0.0001):
    max_iter=100
    x_new = start_x
    res = []
    for i in range(max_iter):
        x_old = x_new
```

```
        # Use negative step size for gradient descent
        x_new = x_old - step_size * grad(x_new)
        f_x_new = func(x_new)
        f_x_old = func(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 = np.arange(-8, 8, 0.001)
y = get_y(x)
plt.plot(x, y)

x_0 = -8
res = gradient_descent_scalar(x_0, get_y, get_grad)
plt.plot(res[:,0], res[:,1], 'r+')
plt.show()
```

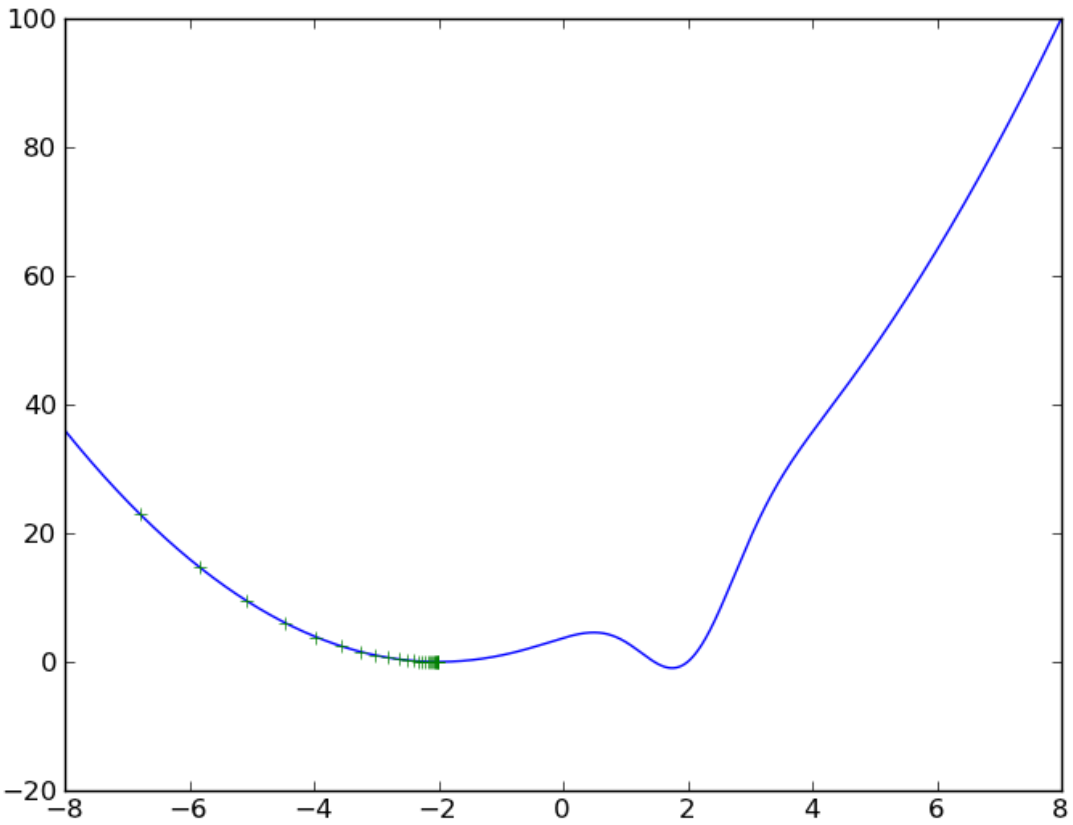

Figure 6: Example of running gradient descent starting on point $x_0 = -8$ for function $f(x) = (x+2)^2 - 16\exp\left(-(x-2)^2\right)$. The function is represented in blue, while the points of the minimizing sequence are displayed as green plus signs.

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

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

```
x = np.arange(-8, 8, 0.001)
y = get_y(x)
plt.plot(x, y)

x_0 = 8
res = gradient_descent_scalar(x_0, get_y, get_grad)
plt.plot(res[:,0], res[:,1], 'r+')
plt.show()
```

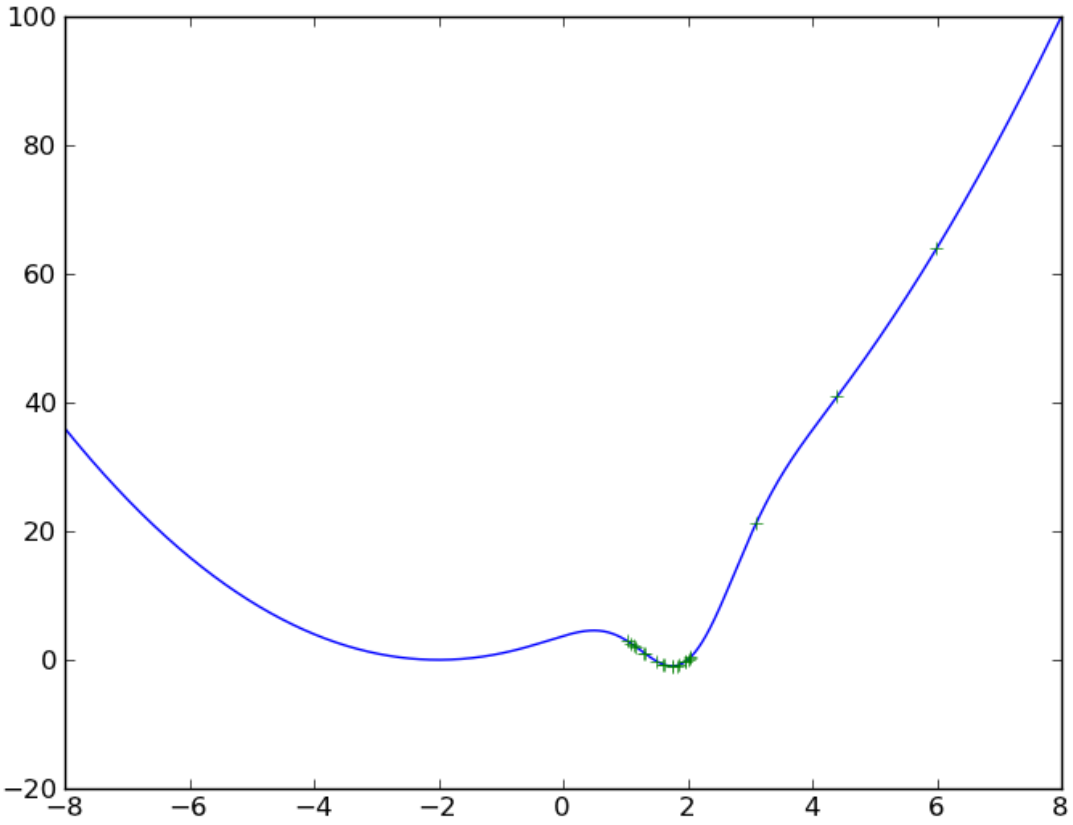

Figure 7: Example of running gradient descent starting on point $x_0 = 8$ for function $f(x) = (x+2)^2 - 16\exp\left(-(x-2)^2\right)$. The function is represented in blue, while the points of the minimizing sequence are displayed as green plus signs.

Figure 7 shows the resulting minimizing sequence. Note that now the algorithm converged to the global minimum. However, note that to get to the global minimum the sequence of points jumped from one side of the minimum to the other. This is a consequence of using a wrong step size (in this case too large). Repeat the previous exercise changing both the values of the step-size and the precision. What do you observe?

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

0.8 Python Exercises

0.8.1 Numpy and Matplotlib

Exercise 0.13 *Consider the linear regression problem (ordinary least squares) on the Galton dataset, with a single response variable*

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

.

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

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

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

1. *Derive the gradient of the error* $\frac{\partial e}{\partial w_j}$.
2. *Implement a solver based on this for two dimensional problems (i.e.,* $\boldsymbol{w} \in R^2$*).*
3. *Use this method on the Galton dataset from the previous exercise to estimate the relationship between father and son's height. Try two formulas*

$$s = f w_1 + \varepsilon, \tag{9}$$

where s *is the son's height, and* f *is the father heights; and*

$$s = f w_1 + 1 w_0 + \varepsilon \tag{10}$$

where the input variable is now two dimensional: $(f, 1)$*. This allows the intercept to be non-zero.*

4. *Plot the regression line you obtain with the points from the previous exercise.*
5. *Use the* `np.linalg.lstsq` *function and compare to your solution.*

Please refer to the notebook for solutions.

0.8.2 Debugging

Exercise 0.14 *Use the debugger to debug the* `buggy.py` *script which attempts to repeatedly perform the following computation:*

1. *Start* $x_0 = 0$
2. *Iterate*
   (a) $x'_{t+1} = x_t + r$*, where* r *is a random variable.*
   (b) *if* $x'_{t+1} >= 1.$*, then stop.*
   (c) *if* $x'_{t+1} <= 0.$*, then* $x_{t+1} = 0$
   (d) *else* $x_{t+1} = x'_{t+1}$.
3. *Return the number of iterations.*

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

Day 1

Linear Classifiers

This day will serve as an introduction to machine learning. We will recall some fundamental concepts about decision theory and classification, 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, Mitchell (1997), Duda et al. (2001), Schölkopf and Smola (2002), Joachims (2002), Bishop (2006), Manning et al. (2008), to name just a few. The concepts that we introduce in this chapter will be revisited in later chapters and expanded to account for non-linear models and structured inputs and outputs. For now, we will concern ourselves only with multi-class classification (with just a few classes) and linear classifiers.

Today's assignment

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

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.

For simplicity reasons, throughout this lecture we will use modified notation and let $P(y)$ denote the *probability* associated with the event $Y = y$ (instead of $P_Y(Y = y)$). Also, *joint* and *conditional* probabilities are denoted as $P(x, y) \triangleq P_{X,Y}(X = x \wedge Y = y)$ and $P(x|y) \triangleq P_{X|Y}(X = x \mid Y = y)$, respectively. From the laws of probabilities:

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

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

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

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

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

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

1.2 Generative Classifiers: Naïve Bayes

If we knew the *true* distribution $P(X, Y)$, the best possible classifier (called 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.

Generative classifiers try to estimate the probability distributions $P(Y)$ and $P(X|Y)$, which are respectively called the *class prior* and the *class conditionals*. They assume that the data are generated according to the following generative story (independently for each $m = 1, \ldots, M$):

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

Figure 1.1 shows an example of the Bayes optimal decision boundary for a toy example with $K = 2$ classes, $M = 100$ points, class priors $P(y_1) = P(y_2) = 0.5$, and class conditionals $P(x|y_i)$ given by 2-D Gaussian distributions with the same variance but different means.

1.2.1 Training and Inference

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

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

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

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

We are left with two important problems:

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

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

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

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

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

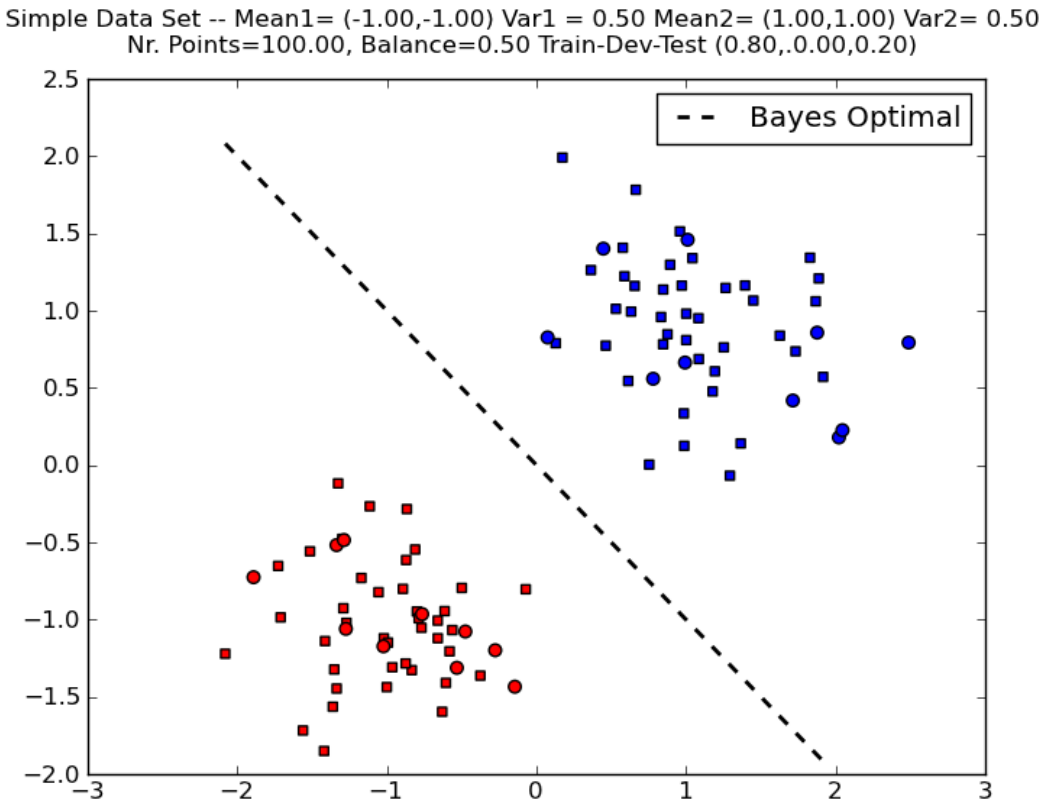

Figure 1.1: Example of a dataset together with the corresponding Bayes optimal decision boundary. The input set consists in points in the real plane, $\mathcal{X} = \mathcal{R}$, and the output set consists of two classes (Red and Blue). Training points are represented as squares, while test points are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

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

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

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

1.2.2 Example: Multinomial Naïve Bayes for Document Classification

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

A very popular document representation is through a "bag-of-words": each document is seen as a collection of words along with their frequencies; word ordering is ignored. We are going to see that this is equivalent to a naïve Bayes assumption with the *multinomial model*. We associate to each class a multinomial distribution, which ignores word ordering, but takes into consideration the frequency with which each word appears in a document. For simplicity, we assume that all documents have the same length L.[1] Each document x is as-

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

sumed 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 $x = w_{j_1} \dots w_{j_L}$ (*i.e.*, a sequence of L words $w_{j_1}, \dots, w_{j_L}$) is

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

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

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

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

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

1.3 Assignment

With the previous theoretical background, you will be able to solve today's assignment.

Exercise 1.1 *In this exercise we will use the Amazon sentiment analysis data (Blitzer et al., 2007), where the goal is to classify text documents as expressing a* positive *or* negative *sentiment (i.e., a classification problem with two classes). We are going to focus on book reviews. To load the data, type:*

```
import lxmls.readers.sentiment_reader as srs

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. *Implement the Naïve Bayes algorithm. Open the file* `multinomial_naive_bayes.py`*, which is inside the* `classifiers` *folder. In the* `MultinomialNaiveBayes` *class you will find the* `train` *method. We have already placed some code in that file to help you get started.*

2. *After implementing, run Naïve Bayes with the multinomial model on the Amazon dataset (sentiment classification) and report results both for training and testing:*

```
import lxmls.classifiers.multinomial_naive_bayes as mnbb

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

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

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

where J is the number of distinct words.

This is a widely used smoothing strategy which has a Bayesian interpretation: it corresponds to choosing a uniform prior for the word distribution on both classes, and to replace the maximum likelihood criterion by a maximum a posteriori *approach. This is a form of* regularization*, preventing the model from* overfitting *on the training data. See* e.g. *Manning and Schütze (1999); Manning et al. (2008) for more information. Report the new accuracies.*

1.4 Discriminative Classifiers

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

1.4.1 Features

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

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

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

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

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

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

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

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

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

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

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

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

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

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

where $\boldsymbol{g}(x) \in \mathbb{R}^J$ is a input feature vector, $\otimes$ is the Kronecker product ($[\boldsymbol{a} \otimes \boldsymbol{b}]_{ij} = a_i b_j$) and $\boldsymbol{e}_y \in \mathbb{R}^K$, with $[\boldsymbol{e}_y]_c = 1$ iff $y = c$, and 0 otherwise. Hence $\boldsymbol{f}(x, y) \in \mathbb{R}^{J \times K}$.

1.4.2 Inference

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

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

where

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

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

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

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

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

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

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

where

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

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

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

1.4.3 Online Discriminative Algorithms

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

Perceptron

The *perceptron* (Rosenblatt, 1958) is perhaps the oldest algorithm used to train a linear classifier. The perceptron works as follows: at each round, it takes an element x from the dataset, 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, it proceeds to the next round.

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

Alg. 2 shows the pseudo-code of the perceptron algorithm. As it can be seen, it is remarkably simple; yet it often reaches a very good performance, often better than the Naïve Bayes, and usually not much worse than maximum entropy models or SVMs (which will be described in the following section). [2]

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 if the data are separable, the perceptron will eventually find a separating hyperplane $\boldsymbol{w}$.

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

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

1. *Run the following commands to generate a simple dataset similar to the one plotted on Figure 1.1:*

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

2. *Run the perceptron algorithm on the simple dataset previously generated and report its train and test set accuracy:*

```
import lxmls.classifiers.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))
```

3. *Plot the decision boundary found:*

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

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

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

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

Margin Infused Relaxed Algorithm (MIRA)

The MIRA algorithm (Crammer and Singer, 2002; Crammer et al., 2006) has achieved very good performance in NLP problems. Recall that the Perceptron takes an input pattern and, if its prediction is wrong, adds the quantity $[\boldsymbol{f}(x^m, y^m) - \boldsymbol{f}(x^m, \hat{y})]$ to the weight vector. MIRA changes this by adding $\eta^t[\boldsymbol{f}(x^m, y^m) - \boldsymbol{f}(x^m, \hat{y})]$ to the weight vector. The difference is the step size η^t, which depends on the iteration t.

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

$$\boldsymbol{w}^{t+1} \leftarrow \arg\min_{\boldsymbol{w}, \xi} \quad \xi + \tfrac{\lambda}{2}\|\boldsymbol{w} - \boldsymbol{w}^t\|^2 \tag{1.22}$$

$$\text{s.t.} \quad \boldsymbol{w} \cdot \boldsymbol{f}(x^m, y^m) \geq \boldsymbol{w} \cdot \boldsymbol{f}(x^m, \hat{y}) + 1 - \xi \tag{1.23}$$

$$\xi \geq 0, \tag{1.24}$$

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. 1.22 we see that MIRA attempts to achieve a tradeoff between *conservativeness* (penalizing large changes from the previous weight vector via the term $\frac{\lambda}{2}\|\boldsymbol{w} - \boldsymbol{w}^t\|^2$) 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).[3] Note that, if the prediction is correct ($\hat{y} = y^m$) the solution of the problem Eq. 1.22 leaves the weight vector unchanged ($\boldsymbol{w}^{t+1} = \boldsymbol{w}^t$). This quadratic programming problem has a closed form solution:[4]

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

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):

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

MIRA is depicted in Alg. 3. For other variants of MIRA, see Crammer et al. (2006).

Exercise 1.3 *We provide an implementation of the MIRA algorithm. Compare it with the perceptron for various values*

[3] The intuition for this large margin separation is the same for support vector machines, which will be discussed in §1.4.4.
[4] Note that the perceptron updates are identical, except that we always have $\eta_t = 1$.

of λ

```
import lxmls.classifiers.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.

1.4.4 Batch Discriminative Classifiers

As we have mentioned, the perceptron and MIRA algorithms are called *online* or *stochastic* because they look at one data point at a time. We now describe two discriminative classifiers which look at all points at once; these are called *offline* or *batch* algorithms.

Maximum Entropy Classifiers

The notion of *entropy* in the context of Information Theory (Shannon, 1948) 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 (Jaynes, 1982). [5] The basic rationale is that choosing the model with the highest entropy (subject to constraints that depend on the observed data) corresponds to making the fewest possible assumptions regarding what was unobserved, making uncertainty about the model as large as possible.

For example, if we throw a die and want to estimate the probability of its outcomes, the distribution with the highest entropy would be the uniform distribution (each outcome having of probability a $1/6$). Now suppose that we are only told that outcomes $\{1,2,3\}$ occurred 10 times in total, and $\{4,5,6\}$ occurred 30 times in total, then the principle of maximum entropy would lead us to estimate $P(1) = P(2) = P(3) = 1/12$ and $P(4) = P(5) = P(6) = 1/4$ (i.e., outcomes would be uniform within each of the two groups).[6]

This example could be presented in a more formal way. Suppose that we want to use binary features to represent the outcome of the die throw. We use two features: $f_{123}(x,y) = 1$ if and only if $y \in \{1,2,3\}$, and $f_{456}(x,y) = 1$ if and only if $y \in \{4,5,6\}$. Our observations state that in 40 throws, we observed f_{123} 10 times (25%) and f_{456} 30 times (75%). The maximum entropy principle states that we want to find the parameters $\boldsymbol{w}$ of our model, and consequently the probability distribution $P_{\boldsymbol{w}}(Y|X)$, which makes f_{123} have an expected value of 0.25 and f_{456} have an expected value of 0.75. These constraints, $E[f_{123}] = 0.25$ and $E[f_{456}] = 0.75$, are known as *first moment matching constraints*.[7]

An important fundamental result, which we will not prove here, is that the maximum entropy distribution $P_{\boldsymbol{w}}(Y|X)$ under first moment matching constraints is a *log-linear model*. [8] It has the following parametric form:

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

[5] For an excellent textbook on Information Theory, we recommend Cover et al. (1991).

[6] For an introduction of maximum entropy models, along with pointers to the literature, see `http://www.cs.cmu.edu/~aberger/maxent.html`.

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

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

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

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

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

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

The average conditional log-likelihood is:

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

We try to find the parameters $\boldsymbol{w}$ that maximize the log-likelihood $\mathcal{L}(\boldsymbol{w}; \mathcal{D})$; to avoid overfitting, we add a regularization term that penalizes values of $\boldsymbol{w}$ that have a high magnitude. The optimization problem becomes:

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

Here we use the squared L_2-norm as the regularizer,[9] but other norms are possible. The scalar $\lambda \geq 0$ controls the amount of regularization. Unlike the naïve Bayes examples, this optimization problem does not have a closed form solution in general; hence we need to resort to numerical optimization (see section 0.7). Let $F_\lambda(\boldsymbol{w}; \mathcal{D}) = -\mathcal{L}(\boldsymbol{w}; \mathcal{D}) + \frac{\lambda}{2} \|\boldsymbol{w}\|^2$ be the objective function in Eq. 1.30. This function is convex, which implies that a local optimum of Eq. 1.30 is also a global optimum. $F_\lambda(\boldsymbol{w}; \mathcal{D})$ is also differentiable: its gradient is

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

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

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

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 (Nocedal and Wright, 1999). In practice, more sophisticated methods exist for optimizing Eq. 1.30, such as conjugate gradient or L-BFGS. The latter is an example of a quasi-Newton method, which only requires gradient information, but uses past gradients to try to construct second order (Hessian) approximations.

In large-scale problems (very large M) batch methods are slow. *Online* or *stochastic* optimization are at-

[9] 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 $y' \in \mathcal{Y}$:

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

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

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

Exercise 1.4 *We provide an implementation of the L-BFGS algorithm for training maximum entropy models in the class* `MaxEnt_batch`*, as well as an implementation of the SGD algorithm in the class* `MaxEnt_online`*.*

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 lxmls.classifiers.max_ent_batch as mebc

me_lbfgs = mebc.MaxEntBatch()
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?*

```
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)
```

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 $y' \in \mathcal{Y}$:

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

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:** $\hat{w} \leftarrow \boldsymbol{w}^{T+1}$

```
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 lxmls.classifiers.max_ent_online as meoc

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

Support Vector Machines

Support vector machines are also a discriminative approach, but they are not a probabilistic model at all. The basic idea is that, if the goal is to accurately predict outputs (according to some cost function), we should focus on that goal in the first place, rather than trying to estimate a probability distribution ($P(Y|X)$ or $P(X, Y)$), which is a more difficult problem. As Vapnik (1995) 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 (Crammer and Singer, 2002), which is of primary interest in natural language processing. There is a significant amount of literature about Kernel Methods (Schölkopf and Smola, 2002; Shawe-Taylor and Cristianini, 2004) mostly focused on the *dual* formulation. We will not discuss non-linear kernels or this dual formulation here.[10]

Consider $\rho(y', y)$ as a non-negative cost function, representing the cost of assigning a label y' when the correct label was y. For simplicity, we focus here on the 0/1-cost defined by Equation 1.25 (but keep in mind

[10]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.

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:** $\hat{\boldsymbol{w}} \leftarrow \boldsymbol{w}^{T+1}$

that other cost functions are possible). The *hinge loss*[11] is the function

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

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

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

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

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

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

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

Geometrically, we are trying to choose the linear classifier that yields the largest possible separation margin, while we allow some violations, penalizing the amount of slack via extra variables $\xi_1, \ldots, \xi_m$. There is now a trade-off: increasing the slack variables ξ_m makes it easier to satisfy the constraints, but it will also increase the value of the cost function.

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

A variant of this algorithm was proposed by Shalev-Shwartz et al. (2007) under the name *Pegasos*, with excellent properties in large-scale settings. Other algorithms and software packages for training SVMs that have become popular are SVMLight (`http://svmlight.joachims.org`) and LIBSVM (`http://www.csie.ntu.edu.tw/~cjlin/libsvm/`), which allow non-linear kernels. These will generally be more suitable for smaller datasets, where high accuracy optimization can be obtained without much computational effort.

[11] 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.

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

Exercise 1.5 *Run the SVM primal algorithm. Then, repeat the MaxEnt exercise now using SVMs, for several values of* λ*:*

```
import lxmls.classifiers.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.

1.5 Comparison

Table 1.1 provides a high-level comparison among the different algorithms discussed in this chapter.

	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 algorithms.

Exercise 1.6

- *Using the simple dataset run the different algorithms varying some characteristics of the data: like the 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?*

1.6 Final remarks

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

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

Day 2

Sequence Models

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

We start by defining the notation for this lecture in Section 2.1. Afterwards, in section 2.2, we focus on the well known Hidden Markov Models and in Section 2.3 we describe how to estimate its parameters from labeled data. In Section 2.4 we explain the 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. In Section 2.6 we describe the task of Part-of-Speech tagging, and how the Hidden Markov Models are suitable for this task. Finally, in Section 2.7 we address unsupervised learning of Hidden Markov Models through the Expectation Maximization algorithm.

Today's assignment

The assignment of today's class is to implement one inference algorithm for Hidden Markov Models, used to find the most likely hidden state sequence given an observation sequence.

2.1 Notation

In what follows, we assume a finite set of *observation labels*, $\Sigma := \{w_1, \ldots, w_J\}$, and a finite set of *state labels*, $\Lambda := \{c_1, \ldots, c_K\}$. We denote by Σ^*, Λ^* the two infinite sets of sequences obtained by grouping the elements of each label set including repetitions and the empty string ε[1]. Elements of Σ^* and Λ^* are *strings of observations* and *strings of states*, respectively. Throughout this class, we assume our input set is $\mathcal{X} = \Sigma^*$, and our output set is $\mathcal{Y} = \Lambda^*$. In other words, our inputs are observation sequences, $x = x_1 x_2 \ldots x_N$, for some $N \in \mathbb{N}$, where each $x_i \in \Sigma$; given such an x, we seek the corresponding state sequence, $y = y_1 y_2 \ldots y_N$, where each $y_i \in \Lambda$. We also consider two special states: the `start` symbol, which starts the sequence, and the `stop` symbol, which ends the sequence.

Moreover, in this lecture we will assume two scenarios:

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

Our notation is summarized in Table 2.1.

2.2 Hidden Markov Models

Hidden Markov Models (HMMs) are one of the most common sequence probabilistic models, and have been applied to a wide variety of tasks. HMMs are particular instances of directed probabilistic graphical models (or Bayesian networks) which have a chain topology. In a Bayesian network, every random variable is represented

[1] More formally, we say $\Sigma^* := \{\varepsilon\} \cup \Sigma \cup \Sigma^2 \cup \ldots$ and $\Lambda^* := \{\varepsilon\} \cup \Lambda \cup \Lambda^2 \cup \ldots$ is the Kleene closure of each of the two sets above.

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

Table 2.1: General notation used in this class

as a node in a graph, and the edges in the graph are directed and represent probabilistic dependencies between the random variables. For an HMM, the random variables are divided into two sets, the *observed variables*, $X = X_1 \dots X_N$, and the *hidden variables* $Y = Y_1 \dots Y_N$. In the HMM terminology, the observed variables are called *observations*, and the hidden variables are called *states*. The states are generated according to a first order Markov process, in which the i^{th} state Y_i depends only of the previous state Y_{i-1}. Two special states are the `start` symbol, which starts the sequence, and the `stop` symbol, which ends the sequence. In addition, states emit observation symbols. In an HMM, it is assumed that all observations are independent given the states that generated them.

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

Example 2.1 *Consider a person who is only interested in four activities: walking in the park (*`walk`*), shopping (*`shop`*), cleaning the apartment (*`clean`*) and playing tennis (*`tennis`*). Also, consider that the choice of what the person does on a given day is determined exclusively by the weather on that day, which can be either* `rainy` *or* `sunny`*. Now, supposing that we observe what the person did on a sequence of days, the question is: can we use that information to predict the weather on each of those days? To tackle this problem, we assume that the weather behaves as a discrete Markov chain: the weather on a given day depends only on the weather on the previous day. The entire system can be described as an HMM.*

For example, assume we are asked to predict the weather conditions on two different sequences of days. During these two sequences, we observed the person performing the following activities:

- "`walk walk shop clean`"
- "`clean walk tennis walk`"

This will be our test set.

Moreover, and in order to train our model, we are given access to three different sequences of days, containing both the activities performed by the person and the weather on those days, namely:

- "`walk/rainy walk/sunny shop/sunny clean/sunny`"
- "`walk/rainy walk/rainy shop/rainy clean/sunny`"
- "`walk/sunny shop/sunny shop/sunny clean/sunny`"

This will be our training set.

Figure 2.2 shows the HMM model for the first sequence of the training set, which already includes the `start` *and* `stop` *symbols. The notation is summarized in Table 2.2.*

Exercise 2.1 *Load the simple sequence dataset. From the ipython command line create a simple sequence object and look at the training and test set.*

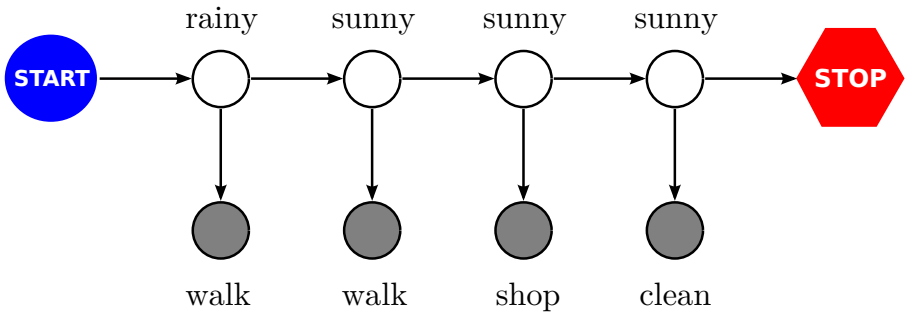

Figure 2.1: Diagram showing the conditional independence relations of the HMM. As an example, the variables are the values of the first sentence of the training set of the simple sequence.

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

Table 2.2: HMM notation for the simple example.

```
import lxmls.readers.simple_sequence as ssr
simple = ssr.SimpleSequence()
print(simple.train)

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

print(simple.test)

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

Get in touch with the classes used to store the sequences, you will need this for the next exercise. Note that each label is internally stored as a number. This number can be used as index of an array to store information regarding that label.

```
for sequence in simple.train.seq_list:
    print(sequence)

walk/rainy walk/sunny shop/sunny clean/sunny
walk/rainy walk/rainy shop/rainy clean/sunny
walk/sunny shop/sunny shop/sunny clean/sunny

for sequence in simple.train.seq_list:
    print(sequence.x)

[0, 0, 1, 2]
[0, 0, 1, 2]
[0, 1, 1, 2]

for sequence in simple.train.seq_list:
    print(sequence.y)

[0, 1, 1, 1]
```

```
[0, 0, 0, 1]
[1, 1, 1, 1]
```

The probability distributions $P(Y_i|Y_{i-1})$ are called *transition probabilities*; the distributions $P(Y_1|Y_0 = \texttt{start})$ are the *initial probabilities*, and $P(Y_{N+1} = \texttt{stop}|Y_N)$ the *final probabilities*.[2] Finally, the distributions $P(X_i|Y_i)$ are called *emission probabilities*.

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

- **Independence of previous states.** The probability of being in a given state at position i only depends on the state of the previous position $i - 1$. Formally,

$$P(Y_i = y_i|Y_{i-1} = y_{i-1}, Y_{i-2} = y_{i-2}, \ldots, Y_1 = y_1) = P(Y_i = y_i|Y_{i-1} = y_{i-1})$$

  defining a first order Markov chain.[3]

- **Homogeneous transition.** The probability of making a transition from state c_l to state c_k is independent of the particular position in the sequence. That is, for all $i, t \in \{1, \ldots, N\}$,

$$P(Y_i = c_k|Y_{i-1} = c_l) = P(Y_t = c_k|Y_{t-1} = c_l)$$

- **Observation independence.** The probability of observing $X_i = x_i$ at position i is fully determined by the state Y_i at that position. Formally,

$$P(X_i = x_i|Y_1 = y_1, \ldots, Y_i = y_i, \ldots, Y_N = y_N) = P(X_i = x_i|Y_i = y_i)$$

  This probability is independent of the particular position so, for every i and t, we can write:

$$P(X_i = w_j|Y_i = c_k) = P(X_t = w_j|Y_t = c_k)$$

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

The distributions that define the HMM model are summarized in Table 2.3. For each one of them we will use a short notation to simplify the exposition.

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

Table 2.3: HMM probability distributions.

The joint distribution can be expressed as:

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

which for the example of Figure 2.1 is:

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

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

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

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

2.3 Finding the Maximum Likelihood Parameters

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

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

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

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

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

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

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

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

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

Here y_i^m, the underscript denotes the state index position for a given sequence, and the superscript denotes the sequence index in the dataset, and the same applies for the observations. Note that $\mathbf{1}$ is an indicator function that has the value 1 when the particular event happens, and zero otherwise. In other words, the previous equations go through the training corpus and count how often each event occurs. For example, Eq. 2.5 counts how many times c_k follows state c_l. Therefore, $C_{\text{trans}}(\texttt{sunny}, \texttt{rainy})$ contains the number of times that a sunny day followed a rainy day.

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

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

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

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

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

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

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

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

```
import lxmls.sequences.hmm as hmmc
hmm = hmmc.HMM(simple.x_dict, simple.y_dict)
hmm.train_supervised(simple.train)
print("Initial Probabilities:", hmm.initial_probs)

[ 0.66666667  0.33333333]

print("Transition Probabilities:", hmm.transition_probs)

[[ 0.5    0.   ]
 [ 0.5    0.625]]

print("Final Probabilities:", hmm.final_probs)

[ 0.     0.375]

print("Emission Probabilities", hmm.emission_probs)

[[ 0.75   0.25 ]
 [ 0.25   0.375]
 [ 0.     0.375]
 [ 0.     0.   ]]
```

2.4 Decoding a Sequence

Given the learned parameters and a new observation sequence $x = x_1 \dots x_N$, we want to find the sequence of hidden states $y^* = y_1^* \dots y_N^*$ that "best" explains it. This is called the *decoding* problem. There are several ways to define what we mean by the "best" y^*, depending on our goal: for instance, we may want to minimize the probability of error on each hidden variable Y_i, or we may want to find the best assignment to the sequence $Y_1 \dots Y_N$ as a whole. Therefore, finding the best sequence can be accomplished through different approaches:

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

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

  Note, however, that this approach does not guarantee that the sequence $y^* = y_1^* \dots y_N^*$ will be a valid sequence of the model. For instance, there might be a transition between two of the best state posteriors with probability zero.

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

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

Both approaches (which will be explained in Sections 2.4.2 and 2.4.3, respectively) rely on dynamic programming and make use of the independence assumptions of the HMM model. Moreover, they use an alternative representation of the HMM called a *trellis*.

A trellis unfolds all possible states for each position and it makes explicit the independence assumption: each position only depends on the previous position. Here, each column represents a position in the sequence and each row represents a possible state. Figure 2.2 shows the trellis for the particular example in Figure 2.1.

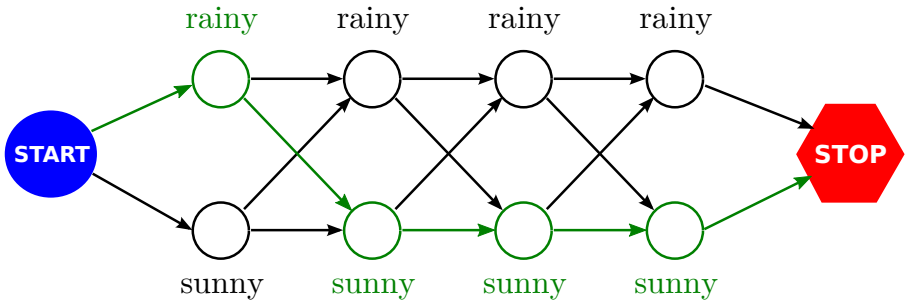

Figure 2.2: Trellis representation of the HMM in Figure 2.1, for the observation sequence "`walk walk shop clean`", where each hidden variable can take the values `rainy` or `sunny`.

Considering the trellis representation, note that we can include the following information:

- an *initial probability* to the arrows that depart from the start symbol;
- a *final probability* to the arrows that reach the stop symbol;
- a *transition probability* to the remaining arrows; and,
- an *emission probability* to each circle, which is the probability that the observed symbol is emitted by that particular state.

For convenience, we will be working with log-probabilities, rather than probabilities.[4] Therefore, if we associate to each circle and arrow in the trellis a score that corresponds to the log-probabilities above, and if we define the score of a path connecting the `start` and `stop` symbols as the sum of the scores of the circles and arrows it traverses, then the goal of finding the most likely sequence of states (Viterbi decoding) corresponds to finding the path with the highest score.

The trellis scores are given by the following expressions:

- For each state c_k:

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

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

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

- For each state c_l:

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

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

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

[4] This will be motivated further in Section 2.4.1, where we describe how operations can be performed efficiently in the log-domain.

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

Exercise 2.3 *Convince yourself that the score of a path in the trellis (summing over the scores above) is equivalent to the log-probability* $\log P(X = x, Y = y)$*, as defined in Eq. 2.2. Use the given function* compute_scores *on the first training sequence and confirm that the values are correct. You should get the same values as presented below.*

```
initial_scores, transition_scores, final_scores, emission_scores = hmm.compute_scores(
    simple.train.seq_list[0])
print(initial_scores)

[-0.40546511 -1.09861229]

print(transition_scores)

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

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

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

print(final_scores)

[       -inf -0.98082925]

print(emission_scores)

[[-0.28768207 -1.38629436]
 [-0.28768207 -1.38629436]
 [-1.38629436 -0.98082925]
 [       -inf -0.98082925]]
```

Note that scores which are $-\infty$ *(*`-inf`*) correspond to zero-probability events.*

2.4.1 Computing in log-domain

We will see that the decoding algorithms will need to multiply twice as many probability terms as the length N of the sequence. This may cause underflowing problems when N is large, since the nested multiplication of numbers smaller than 1 may easily become smaller than the machine precision. To avoid that problem, Rabiner (1989) presents a scaled version of the decoding algorithms that avoids this problem. An alternative, which is widely used, is computing in the log-domain. That is, instead of manipulating probabilities, manipulate log-probabilities (the scores presented above). Every time we need to multiply probabilities, we can sum their log-representations, since:

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

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

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

where we assume that a is smaller than b.

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

```
import numpy as np
a = np.random.rand(10)
np.log(sum(np.exp(a)))
2.8397172643228661
```

```
np.log(sum(np.exp(10*a)))
10.121099917705818

np.log(sum(np.exp(100*a)))
93.159220940569128

np.log(sum(np.exp(1000*a)))
inf

from lxmls.sequences.log_domain import *
logsum(a)
2.8397172643228665

logsum(10*a)
10.121099917705818

logsum(100*a)
93.159220940569114

logsum(1000*a)
925.88496219586864
```

2.4.2 Posterior Decoding

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

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

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

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

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

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

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

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

The number of possible hidden state sequences is exponential in the length of the sequence ($|\Lambda|^N$), which makes the sum over all of them hard. In our simple example, there are $2^4 = 16$ paths, which we can actually explicitly enumerate and calculate their probability using Equation 2.1. But this is as far as it goes: for example, for Part-of-Speech tagging with a small tagset of 12 tags and a medium size sentence of length 10, there are $12^{10} = 61917364224$ such paths. Yet, we must be able to compute this sum (sum over $y \in \Lambda^N$) to compute the above likelihood formula; this is called the inference problem. For sequence models, there is a well known dynamic programming algorithm, the **Forward-Backward** (FB) algorithm, which allows the computation to be performed in linear time,[5] 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 having arrows only between consecutive states. The FB algorithm defines two auxiliary probabilities, the forward probability and the backward probability.

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

Efficient forward probability computation

The forward probability represents the probability that in position i we are in state $Y_i = c_k$ and that we have observed $x_1, \ldots, x_i$ up to that position. Therefore, its mathematical expression is:

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

Using the independence assumptions of the HMM we can compute $\text{forward}(i, c_k)$ using all the forward computations $\{\text{forward}(i-1, c) \text{ for } c \in \Lambda\}$. In order to facilitate the notation of the following argument we will denote by $x_{i:j}$ the assignemnt $X_i = x_i, \ldots, X_j = x_j$. Therefore we can write $\text{forward}(i, y_i)$ as $P(y_i, x_{1:i})$ and rewrite the forward expression as follows:

$$P(y_i, x_{1:i}) = \sum_{y_{i-1} \in \Lambda} P(y_i, y_{i-1}, x_{1:i}) = \sum_{y_{i-1} \in \Lambda} P(x_i | y_i, y_{i-1}, x_{1:i-1}) \cdot P(y_i | y_{i-1}, x_{1:i-1}) \cdot P(y_{i-1}, x_{1:i-1}) \tag{2.26}$$

Using the **Observation independence** and the **Independence of previous states** properties of the first order HMM we have $P(x_i | y_i, y_{i-1}, x_{1:i-1}) = P(x_i | y_i)$ and $P(y_i | y_{i-1}, x_{1:i-1}) = P(y_i | y_{i-1})$. Therefore equation (2.26) can be written, for $i \in \{2, \ldots, N\}$ (where N is the length of the sequence), as

$$\text{forward}(i, y_i) = \sum_{y_{i-1} \in \Lambda} P(x_i | y_i,) \cdot P(y_i | y_{i-1}) \cdot \text{forward}(i-1, y_{i-1}) \tag{2.27}$$

Using equation (2.27) we have proved that the forward probability can be defined by the following recurrence rule:

$$\begin{aligned}
\text{forward}(1, c_k) &= P_{\text{init}}(c_k | \texttt{start}) \times P_{\text{emiss}}(x_1 | c_k) && (2.28)\\
\text{forward}(i, c_k) &= \left(\sum_{c_l \in \Lambda} P_{\text{trans}}(c_k | c_l) \times \text{forward}(i-1, c_l) \right) \times P_{\text{emiss}}(x_i | c_k) && (2.29)\\
\text{forward}(N+1, \texttt{stop}) &= \sum_{c_l \in \Lambda} P_{\text{final}}(\texttt{stop} | c_l) \times \text{forward}(N, c_l). && (2.30)
\end{aligned}$$

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

$$P(X = x) = \text{forward}(N+1, \texttt{stop}). \tag{2.31}$$

Although the forward probability is enough to calculate the likelihood of a given sequence, we will also need the backward probability to calculate the state posteriors.

Efficient backward probability computation

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

$$\textbf{Backward Probability}: \quad \text{backward}(i, c_l) = P(X_{i+1} = x_{i+1}, \ldots, X_N = x_N | Y_i = c_l). \tag{2.32}$$

Using the independence assumptions of the HMM we can compute $\text{backward}(i, c_k)$ using all the backward computations $\{\text{backward}(i+1, c) \text{ for } c \in \Lambda\}$. Therefore we can write $\text{backward}(i, y_i)$ as $P(x_{i+1:N} | y_i)$ and rewrite the forward expression as follows:

$$P(x_{i+1:N} | y_i) = \sum_{y_{i+1} \in \Lambda} P(x_{i+1:N}, y_{i+1} | y_i) = \sum_{y_{i+1} \in \Lambda} P(x_{i+2:N} | y_i, y_{i+1}, x_{i+1}) P(x_{i+1}, | y_{i+1}, y_i) P(y_{i+1} | y_i) \tag{2.33}$$

Using the previous equation we have proved that the backward probability can be defined by the following recurrence rule:

$$\begin{aligned}
\text{backward}(N, c_l) &= P_{\text{final}}(\texttt{stop}|c_l) && (2.34)\\
\text{backward}(i, c_l) &= \sum_{c_k \in \Lambda} P_{\text{trans}}(c_k|c_l) \times \text{backward}(i+1, c_k) \times P_{\text{emiss}}(x_{i+1}|c_k) && (2.35)\\
\text{backward}(0, \texttt{start}) &= \sum_{c_k \in \Lambda} P_{\text{init}}(c_k|\texttt{start}) \times \text{backward}(1, c_k) \times P_{\text{emiss}}(x_1|c_k). && (2.36)
\end{aligned}$$

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

$$P(X = x) = \text{backward}(0, \texttt{start}). \tag{2.37}$$

The forward backward algorithm

We have seen how we can compute the probability of a sequence x using the the forward and backward probabilities by computing forward$(N+1, \texttt{stop})$ and backward$(0, \texttt{start})$ respectively. Moreover, the probability of a sequence x can be computed with both forward and backward probabilities at a particular position i. The probability of a given sequence x at any position i in the sequence can be computed as follows [6]:

$$\begin{aligned}
P(X = x) &= \sum_{c_k \in \Lambda} P(X_1 = x_1, \ldots, X_N = x_N, Y_i = c_k)\\
&= \sum_{c_k \in \Lambda} \underbrace{P(X_1 = x_1, \ldots, X_i = x_i, Y_i = c_k)}_{\text{forward}(i, c_k)} \times \underbrace{P(X_{i+1} = x_{i+1}, \ldots, X_N = x_N | Y_i = c_k)}_{\text{backward}(i, c_k)}\\
&= \sum_{c_k \in \Lambda} \text{forward}(i, c_k) \times \text{backward}(i, c_k). && (2.38)
\end{aligned}$$

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

Algorithm 7 shows the pseudo code for the forward backward algorithm. The reader can notice that the *forward* and *backward* computations in the algorithm make use of P_{emiss} and P_{trans}. There are a couple of details that should be taken into account if the reader wants to understand the algorithm using scores instead of probabilities.

- $forward(i, \hat{c})$ is computed using $P_{emiss}(x_i|\hat{c})$ which does not depend on the sum over all possible states $c_k \in \Lambda$. Therefore when taking the logarithm of the sum over all possible states the recurrence of the forward computations can be split as a sum of two logarithms.
- $backward(i, \hat{c})$ is computed using $P_{\text{trans}}(c_k|\hat{c})$ and $P_{\text{emiss}}(x_{i+1}|c_k)$ both of which depend on c_k. Therefore when taking the logarithm of the sum the expression cannot be split as a sum of logarithms.

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

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

Log-Likelihood = -5.06823232601

log_likelihood, backward = hmm.decoder.run_backward(initial_scores, transition_scores,
    final_scores, emission_scores)
print('Log-Likelihood =', log_likelihood)

Log-Likelihood = -5.06823232601
```

[6] The second equality in 2.38 can be justified as follows. Since $Y_i = y_i$ is observed then $X_{i+1}, \ldots, X_N$ is conditionally independent from any X_k for $k \leq i$. Therefore $P(x_1, \ldots, x_N, y_i) = P(x_{i+1}, \ldots, x_N | yi, x_1, \ldots, x_i) \cdot P(yi, x_1, \ldots, x_i) = P(x_{i+1}, \ldots, x_N | yi) \cdot P(yi, x_1, \ldots, x_i) = backward(i, y_i) \cdot forward(i, y_i)$.

Algorithm 7 Forward-Backward algorithm

input: sequence $x_1, \ldots, x_N$, scores $P_{\text{init}}, P_{\text{trans}}, P_{\text{final}}, P_{\text{emiss}}$

Forward pass: Compute the forward probabilities
for $c_k \in \Lambda$ **do**
  $\text{forward}(1, c_k) = P_{\text{init}}(c_k|\texttt{start}) \times P_{\text{emiss}}(x_1|c_k)$
end for
for $i = 2$ **to** N **do**
  for $\hat{c} \in \Lambda$ **do**
    $$\text{forward}(i, \hat{c}) = \left(\sum_{c_k \in \Lambda} P_{\text{trans}}(\hat{c}|c_k) \times \text{forward}(i-1, c_k) \right) \times P_{\text{emiss}}(x_i|\hat{c})$$
  end for
end for

Backward pass: Compute the backward probabilities
for $c_k \in \Lambda$ **do**
  $\text{backward}(N, c_k) = P_{\text{final}}(\texttt{stop}|c_k)$
end for
for $i = N - 1$ **to** 1 **do**
  for $\hat{c} \in \Lambda$ **do**
    $$\text{backward}(i, \hat{c}) = \sum_{c_k \in \Lambda} P_{\text{trans}}(c_k|\hat{c}) \times \text{backward}(i+1, c_k) \times P_{\text{emiss}}(x_{i+1}|c_k)$$
  end for
end for

output: The forward and backward probabilities.

Given the forward and backward probabilities, one can compute both the state and transition posteriors (you can hint why by looking at the term inside the sum in Eq. 2.38).

$$\textbf{State Posterior}: \quad P(Y_i = y_i|X = x) = \frac{\text{forward}(i, y_i) \times \text{backward}(i, y_i)}{P(X = x)}; \tag{2.39}$$

$$\textbf{Transition Posterior}: \quad P(Y_i = y_i, Y_{i+1} = y_{i+1}|X = x) = \frac{\text{forward}(i, y_i) \times P_{\text{trans}}(y_{i+1}|y_i) \times P_{\text{emiss}}(x_{i+1}|y_{i+1}) \times \text{backward}(i+1, y_{i+1})}{P(X = x)}. \tag{2.40}$$

A graphical representation of these posteriors is illustrated in Figure 2.3. On the left it is shown that $\text{forward}(i, y_i) \times \text{backward}(i, y_i)$ returns the sum of all paths that contain the state y_i, weighted by $P(X = x)$; on the right we can see that $\text{forward}(i, y_i) \times P_{\text{trans}}(y_{i+1}|y_i) \times P_{\text{emiss}}(x_{i+1}|y_{i+1}) \times \text{backward}(i+1, y_{i+1})$ returns the same for all paths containing the edge from y_i to y_{i+1}.

As a practical example, given that the person performs the sequence of actions "`walk walk shop clean`", we want to know the probability of having been raining in the second day. The state posterior probability for this event can be seen as the probability that the sequence of actions above was generated by a sequence of weathers and where it was raining in the second day. In this case, the possible sequences would be all the sequences which have `rainy` in the second position.

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

$$\widehat{y}_i := \arg\max_{y_i \in \Lambda} P(Y_i = y_i|X = x). \tag{2.41}$$

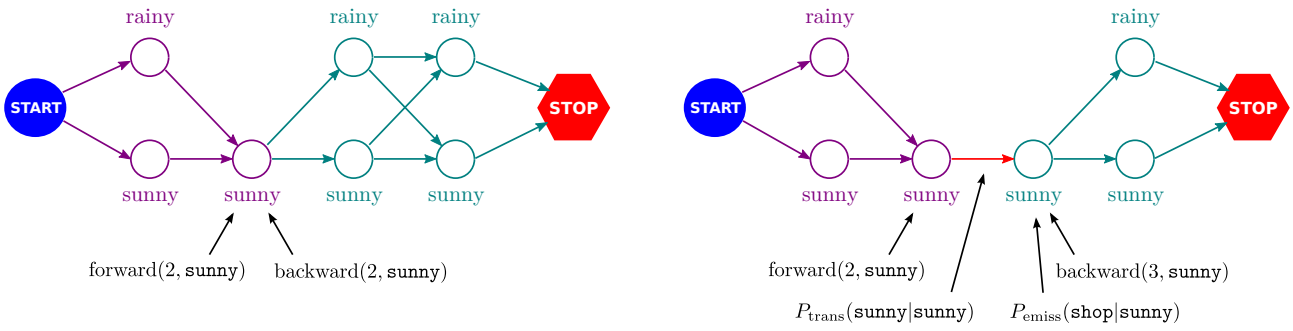

Figure 2.3: A graphical representation of the components in the state and transition posteriors. Recall that the observation sequence is "`walk walk shop clean`".

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

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

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

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

```
y_pred = hmm.posterior_decode(simple.test.seq_list[0])
print("Prediction test 0:", y_pred)

walk/rainy walk/rainy shop/sunny clean/sunny

print("Truth test 0:", simple.test.seq_list[0])

walk/rainy walk/sunny shop/sunny clean/sunny
```

Do the same for the second test sequence:

```
y_pred = hmm.posterior_decode(simple.test.seq_list[1])
print("Prediction test 1:", y_pred)

clean/rainy walk/rainy tennis/rainy walk/rainy

print("Truth test 1:", simple.test.seq_list[1])

clean/sunny walk/sunny tennis/sunny walk/sunny
```

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

Change the train_supervised *method to add smoothing:*

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

Try, for example, adding 0.1 to all the counts, and repeating this exercise with that smoothing. What do you observe?

```
hmm.train_supervised(simple.train, smoothing=0.1)
y_pred = hmm.posterior_decode(simple.test.seq_list[0])
print("Prediction test 0 with smoothing:", y_pred)

walk/rainy walk/rainy shop/sunny clean/sunny

print("Truth test 0:", simple.test.seq_list[0])

walk/rainy walk/sunny shop/sunny clean/sunny

y_pred = hmm.posterior_decode(simple.test.seq_list[1])
print("Prediction test 1 with smoothing:", y_pred)

clean/sunny walk/sunny tennis/sunny walk/sunny

print("Truth test 1:", simple.test.seq_list[1])

clean/sunny walk/sunny tennis/sunny walk/sunny
```

2.4.3 Viterbi Decoding

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

$$\widehat{y} = \arg\max_{y \in \Lambda^N} P(Y = y | X = x) = \arg\max_{y \in \Lambda^N} P(X = x, Y = y). \tag{2.42}$$

The Viterbi algorithm is very similar to the forward procedure of the FB algorithm, making use of the same trellis structure to efficiently represent the exponential number of sequences without prohibitive computation costs. In fact, the only difference from the forward-backward algorithm is in the recursion 2.29 where instead of *summing* over all possible hidden states, we take their *maximum*.

$$\textbf{Viterbi} \quad \text{viterbi}(i, y_i) = \max_{y_1 \dots y_{i-1}} P(Y_1 = y_1, \dots Y_i = y_i, X_1 = x_1, \dots, X_i = x_i) \tag{2.43}$$

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

$$\begin{aligned}
\text{viterbi}(1, c_k) &= P_{\text{init}}(c_k | \texttt{start}) \times P_{\text{emiss}}(x_1 | c_k) && (2.44)\\
\text{viterbi}(i, c_k) &= \left(\max_{c_l \in \Lambda} P_{\text{trans}}(c_k | c_l) \times \text{viterbi}(i-1, c_l) \right) \times P_{\text{emiss}}(x_i | c_k) && (2.45)\\
\text{backtrack}(i, c_k) &= \left(\arg\max_{c_l \in \Lambda} P_{\text{trans}}(c_k | c_l) \times \text{viterbi}(i-1, c_l) \right) && (2.46)\\
\text{viterbi}(N+1, \texttt{stop}) &= \max_{c_l \in \Lambda} P_{\text{final}}(\texttt{stop} | c_l) \times \text{viterbi}(N, c_l) && (2.47)\\
\text{backtrack}(N+1, \texttt{stop}) &= \arg\max_{c_l \in \Lambda} P_{\text{final}}(\texttt{stop} | c_l) \times \text{viterbi}(N, c_l). && (2.48)
\end{aligned}$$

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

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

Algorithm 8 Viterbi algorithm

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

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

2.5 Assignment

With the previous theoretical background, you have the necessary tools to solve today's assignment.

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

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

Hint: look at the implementation of `run_forward`. *Also check the help for the numpy methods max and argmax.*
This method will be called by

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

in the module `sequence_classifier.py`.
Test your method on both test sequences and compare the results with the ones given.

```
hmm.train_supervised(simple.train, smoothing=0.1)
y_pred, score = hmm.viterbi_decode(simple.test.seq_list[0])
print("Viterbi decoding Prediction test 0 with smoothing:", y_pred, score)

walk/rainy walk/rainy shop/sunny clean/sunny  -6.02050124698

print("Truth test 0:", simple.test.seq_list[0])

walk/rainy walk/sunny shop/sunny clean/sunny

y_pred, score = hmm.viterbi_decode(simple.test.seq_list[1])
print("Viterbi decoding Prediction test 1 with smoothing:", y_pred, score)

clean/sunny walk/sunny tennis/sunny walk/sunny  -11.713974074

print("Truth test 1:", simple.test.seq_list[1])
```

```
clean/sunny walk/sunny tennis/sunny walk/sunny
```

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

2.6 Part-of-Speech Tagging (POS)

Part-of-Speech (PoS) tagging is one of the most important NLP tasks. The task is to assign each word a grammatical category, or Part-of-Speech, *i.e.* noun, verb, adjective,... Recalling the defined notation, Σ is a vocabulary of word types, and Λ is the set of Part-of-Speech tags.

In English, using the Penn Treebank (PTB) corpus (Marcus et al., 1993), the current state of the art for part of speech tagging is around 97% for a variety of methods.

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

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

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

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

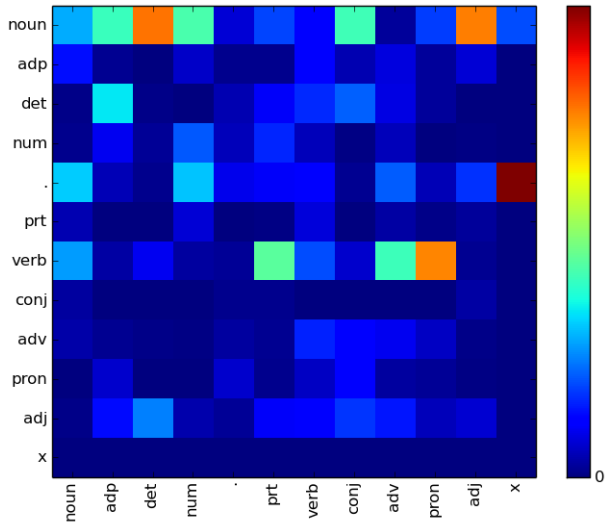

Figure 2.4: Transition probabilities of the trained model. Each column is the previous state and row is the current state. Note the high probability of having Noun after Determinant or Adjective, or of having Verb after Nouns or Pronouns, as expected.

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

```
viterbi_pred_train = hmm.viterbi_decode_corpus(train_seq)
posterior_pred_train = hmm.posterior_decode_corpus(train_seq)
eval_viterbi_train =   hmm.evaluate_corpus(train_seq, viterbi_pred_train)
eval_posterior_train =  hmm.evaluate_corpus(train_seq, posterior_pred_train)
print "Train Set Accuracy: Posterior Decode %.3f, Viterbi Decode: %.3f"%(
    eval_posterior_train,eval_viterbi_train)
Train Set Accuracy: Posterior Decode 0.985, Viterbi Decode: 0.985

viterbi_pred_test = hmm.viterbi_decode_corpus(test_seq)
posterior_pred_test = hmm.posterior_decode_corpus(test_seq)
eval_viterbi_test =   hmm.evaluate_corpus(test_seq,viterbi_pred_test)
eval_posterior_test = hmm.evaluate_corpus(test_seq,posterior_pred_test)
print "Test Set Accuracy: Posterior Decode %.3f, Viterbi Decode: %.3f"%(
    eval_posterior_test,eval_viterbi_test)
Test Set Accuracy: Posterior Decode 0.350, Viterbi Decode: 0.509
```

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

```
best_smoothing = hmm.pick_best_smoothing(train_seq, dev_seq, [10,1,0.1,0])

Smoothing 10.000000 --  Train Set Accuracy: Posterior Decode 0.731, Viterbi Decode: 0.691
Smoothing 10.000000 -- Test Set Accuracy: Posterior Decode 0.712, Viterbi Decode: 0.675
Smoothing 1.000000 --  Train Set Accuracy: Posterior Decode 0.887, Viterbi Decode: 0.865
Smoothing 1.000000 -- Test Set Accuracy: Posterior Decode 0.818, Viterbi Decode: 0.792
Smoothing 0.100000 --  Train Set Accuracy: Posterior Decode 0.968, Viterbi Decode: 0.965
Smoothing 0.100000 -- Test Set Accuracy: Posterior Decode 0.851, Viterbi Decode: 0.842
Smoothing 0.000000 --  Train Set Accuracy: Posterior Decode 0.985, Viterbi Decode: 0.985
Smoothing 0.000000 -- Test Set Accuracy: Posterior Decode 0.370, Viterbi Decode: 0.526

hmm.train_supervised(train_seq, smoothing=best_smoothing)
viterbi_pred_test = hmm.viterbi_decode_corpus(test_seq)
posterior_pred_test = hmm.posterior_decode_corpus(test_seq)
eval_viterbi_test =   hmm.evaluate_corpus(test_seq, viterbi_pred_test)
eval_posterior_test = hmm.evaluate_corpus(test_seq, posterior_pred_test)
print "Best Smoothing %f --  Test Set Accuracy: Posterior Decode %.3f, Viterbi Decode: %.
    3f"%(best_smoothing,eval_posterior_test,eval_viterbi_test)

Best Smoothing 0.100000 --  Test Set Accuracy: Posterior Decode 0.837, Viterbi Decode: 0.
    827
```

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

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

2.7 Unsupervised Learning of HMMs

We next address the problem of *unsupervised* learning. In this setting, we are not given any labeled data; all we get to see is a set of natural language sentences. The underlying question is:

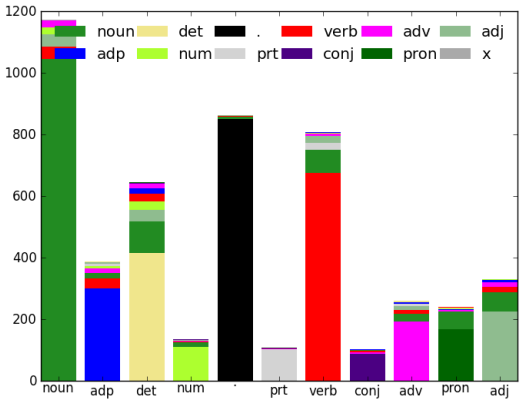

Figure 2.5: Confusion Matrix for the previous example. Predicted tags are columns and the true tags corresponds to the constituents of each column.

Can we learn something from raw text?

This task is challenging, since the process by which linguistic structures are generated is not always clear; and even when it is, it is typically too complex to be formally expressed. Nevertheless, unsupervised learning has been applied to a wide range of NLP tasks, such as: Part-of-Speech Induction (Schütze, 1995; Merialdo, 1994; Clark, 2003), Dependency Grammar Induction (Klein and Manning, 2004; Smith and Eisner, 2006), Constituency Grammar Induction (Klein and Manning, 2004), Statistical Word Alignments (Brown et al., 1993) and Anaphora Resolution (Charniak and Elsner, 2009), just to name a few.

Different motivations have pushed research in this area. From both a linguistic and cognitive point of view, unsupervised learning is useful as a tool to study language acquisition. From a machine learning point of view, unsupervised learning is a fertile ground for testing new learning methods, where significant improvements can yet be made. From a more pragmatic perspective, unsupervised learning is required since annotated corpora is a scarce resource for different reasons. Independently of the reason, unsupervised learning is an increasingly active field of research.

A first problem with unsupervised learning, since we don't observe any labeled data (i.e., the training set is now $\mathcal{D}_U = \{x^1, \ldots, x^M\}$), is that most of the methods studied so far (e.g., Perceptron, MIRA, SVMs) cannot be used since we cannot compare the true output with the predicted output. Note also that a direct minimization of the *complete negative log-likelihood* of the data, $\log P_\theta(X^1 = x^1, \ldots, X^M = x^m)$, is very challenging, since it would require marginalizing out (*i.e.*, summing over) all possible hidden variables:

$$\log P_\theta(X^1 = x^1, \ldots, X^M = x^m) = \sum_{m=1}^{M} \log \sum_{y \in \Lambda} P_\theta(X = x^m, Y = y). \tag{2.49}$$

Note also that the objective above is *non-convex* even for a linear model: hence, it may have local minima, which makes optimization much more difficult.

The most common optimization method in the presence of hidden (latent) variables is the Expectation Maximization (EM) algorithm, described in the next section. Note that this algorithm is a generic optimization routine that does not depend on a particular model. Later, in Section 2.7.2 we will apply the EM algorithm to the task of part-of-speech induction, where one is given raw text and a number of clusters and the task is to cluster words that behave similarly in a grammatical sense.

2.7.1 Expectation Maximization Algorithm

Given the training corpus $\mathcal{D}_U := \{x^1, \ldots, x^M\}$, training seeks model parameters θ that minimize the negative log-likelihood of the corpus:

$$\textbf{Negative Log Likelihood}: \quad \mathcal{L}(\theta) = \widehat{\mathbb{E}}[-\log P_\theta(X)] = -\frac{1}{M} \sum_{m=1}^{M} \log \left(\sum_{y^m \in \Lambda} P_\theta(X = x^m, Y = y^m) \right), \tag{2.50}$$

where $\widehat{\mathbb{E}}[f(X)] := \frac{1}{M}\sum_{m=1}^{M} f(x^m)$ denotes the empirical average of a function f over the training corpus. Because of the hidden variables $y^1, \ldots, y^M$, the likelihood term contains a sum over all possible hidden structures inside of a logarithm, which makes this quantity hard to compute.

The most common minimization algorithm to fit the model parameters in the presence of hidden variables is the Expectation-Maximization (EM) algorithm. The EM procedure can be thought of intuitively in the following way. If we observe the hidden variables' values for all sentences in the corpus, then we could easily compute the maximum likelihood value of the parameters as described in Section 2.3. On the other hand, if we had the model parameters we could label data using the model, and collect the sufficient statistics described in Section 2.3. However, since we are working in an unsupervised setting, we never get to observe the hidden state sequence. Instead, given the training set $\mathcal{D}_U = \{x^1 \ldots x^M\}$, we will need to collect *expected* sufficient statistics (in this case, *expected* counts) that represent the number of times that each hidden variable is expected to be used with the current parameters setting. These expected sufficient statistics will then be used during learning as "fake" observations of the hidden variables. Using the state and transition posterior distributions described in Equations 2.22 and 2.23, the expected sufficient statistics can be computed by the following formulas:

$$\textbf{Initial Counts}: \quad C_{\text{init}}(c_k) = \sum_{m=1}^{M} P_\theta(Y_1^m = c_k \mid X^m = x^m); \tag{2.51}$$

$$\textbf{Transition Counts}: \quad C_{\text{trans}}(c_k, c_l) = \sum_{m=1}^{M}\sum_{i=2}^{N} P_\theta(Y_i^m = c_k \wedge Y_{i-1}^m = c_l \mid X^m = x^m); \tag{2.52}$$

$$\textbf{Final Counts}: \quad C_{\text{final}}(c_k) = \sum_{m=1}^{M} P_\theta(Y_N^m = c_k \mid X^m = x^m); \tag{2.53}$$

$$\textbf{Emission Counts}: \quad C_{\text{emiss}}(w_j, c_k) = \sum_{m=1}^{M}\sum_{i=1}^{N} \mathbf{1}(x_i^m = w_j) P_\theta(Y_i^m = c_k \mid X^m = x^m); \tag{2.54}$$

Compare the previous equations with the ones described in Section 2.3 for the same quantities (Eqs. 2.4–2.7). The main difference is that while in the presence of supervised data you sum the observed events, when you have no label data you sum the posterior probabilities of each event. If these probabilities were such that the probability mass was around single events then both equations will produce the same result.

The EM procedure starts with an initial guess for the parameters θ^0 at time $t = 0$. The algorithm keeps iterating until it converges to a local minima of the negative log likelihood. Each iteration is divided into two steps:

- The **E-Step** (Expectation) computes the posteriors for the hidden variables $P_{\theta^t}(Y|X = x^m)$ given the current parameter values θ^t and the observed variables $X = x^m$ for the m-th sentence. For an HMM, this can be done through the Forward-Backward algorithm described in the previous sections.
- The **M-step** (Maximization) uses those posteriors $P_{\theta^t}(Y|X = x^m)$ to "softly fill in" the values of the hidden variables Y^m, and collects the sufficient statistics: initial counts (Eq: 2.51), transition counts (Eq: 2.52), final counts (Eq: 2.53), and emission counts (Eq: 2.54). These counts are then used to estimate maximum likelihood parameters θ^{t+1}, as described in Section 2.3.

The EM algorithm is guaranteed to converge to a local minimum of the negative log-likelihood $\mathcal{L}(\theta)$, under mild conditions. Note that we are not committing to the best assignment of the hidden variables, but summing the occurrences of each parameter weighed by the posterior probability of all possible assignments. This modular split into two intuitive and straightforward steps accounts for the vast popularity of EM.[7]

Algorithm 9 presents the pseudo code for the EM algorithm.

One important thing to note in Algorithm 9 is that for the HMM model we already have all the model pieces we require. In fact the only method we don't have yet implemented from previous classes is the method to update the counts given the posteriors.

Exercise 2.10 *Implement the method to update the counts given the state and transition posteriors.*

```
def update_counts(self, sequence, state_posteriors, transition_posteriors):
```

[7] More formally, EM minimizes an *upper bound* of $\mathcal{L}(\theta)$ via block-coordinate descent. See Neal and Hinton (1998) for details. Also, even though we are presenting EM in the context of HMMs, this algorithm can be more broadly applied to any probabilistic model with latent variables.

Algorithm 9 EM algorithm.

input: dataset $\mathcal{D}_U$, initial model θ
for $t = 1$ **to** T **do**

  E-Step:
  Clear counts: $C_{\text{init}}(.) = C_{\text{trans}}(.,.) = C_{\text{final}}(.) = C_{\text{emiss}}(.,.) = 0$
  for $x^m \in \mathcal{D}_U$ **do**
    Compute posterior expectations $P_\theta(Y_i|X = x^m)$ and $P_\theta(Y_i, Y_{i+1}|X = x^m)$ using the current model θ
    Update counts as shown in Eqs. 2.51–2.54.
  end for

  M-Step:
  Update the model parameters θ based on the counts.
end for

You now have all the pieces to implement the EM algorithm. Look at the code for EM algorithm in file sequences/hmm.py *and check it for yourself.*

```
    def train_EM(self, dataset, smoothing=0, num_epochs=10, evaluate=True):
        self.initialize_random()

        if evaluate:
            acc = self.evaluate_EM(dataset)
            print("Initial accuracy: %f"%(acc))

        for t in range(1, num_epochs):
            #E-Step
            total_log_likelihood = 0.0
            self.clear_counts(smoothing)
            for sequence in dataset.seq_list:
                # Compute scores given the observation sequence.
                initial_scores, transition_scores, final_scores, emission_scores = \
                    self.compute_scores(sequence)

                state_posteriors, transition_posteriors, log_likelihood = \
                    self.compute_posteriors(initial_scores,
                                            transition_scores,
                                            final_scores,
                                            emission_scores)

                self.update_counts(sequence, state_posteriors, transition_posteriors)
                total_log_likelihood += log_likelihood
            print("Iter: %i Log Likelihood: %f"%(t, total_log_likelihood))
            #M-Step
            self.compute_parameters()
            if evaluate:
                 ### Evaluate accuracy at this iteration
                acc = self.evaluate_EM(dataset)
                print("Iter: %i Accuracy: %f"%(t,acc))
```

2.7.2 Part of Speech Induction

In this section we present the Part-of-Speech induction task. Part-of-Speech tags are pre-requisite for many text applications. The task of Part-of-Speech tagging where one is given a labeled training set of words and respective tags is a well studied task with several methods achieving high prediction quality, as we saw in the first part of this chapter.

On the other hand the task of Part-of-Speech induction where one does not have access to a labeled corpus is a much harder task with a huge space for improvement. In this case, we are given only the raw text along with sentence boundaries and a predefined number of clusters we can use. This problem can be seen as a clustering problem. We want to cluster words that behave grammatically in the same way on the same cluster. This is a much harder problem.

Depending on the task at hand we can pick an arbitrary number of clusters. If the goal is to test how well our method can recover the true POS tags then we should use the same number of clusters as POS tags. On the other hand, if the task is to extract features to be used by other methods we can use a much bigger number of clusters (e.g., 200) to capture correlations not captured by POS tags, like lexical affinity.

Note, however that nothing is said about the identity of each cluster. The model has no preference in assigning cluster 1 to nouns vs cluster 2 to nouns. Given this non-identifiability several metrics have been proposed for evaluation (Reichart and Rappoport, 2009; Haghighi and Klein, 2006; Meilă, 2007; Rosenberg and Hirschberg, 2007). In this class we will use a common and simple metric called **1-Many**, which maps each cluster to majority pos tag that it contains (see Figure 2.6 for an example).

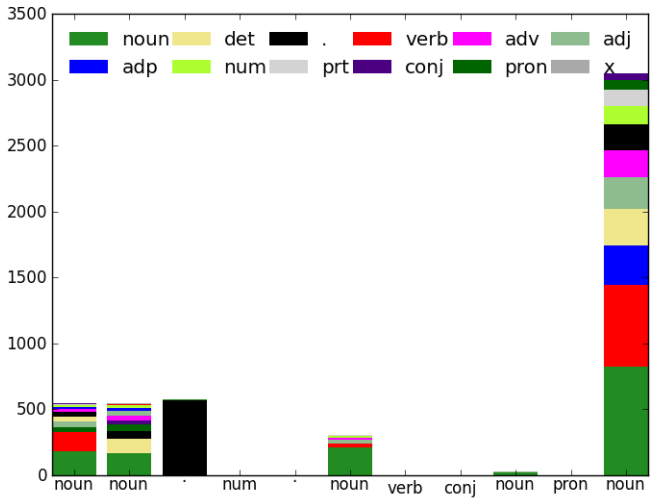

Figure 2.6: Confusion Matrix example. Each cluster is a column. The best tag in each column is represented under the column (1-many) mapping. Each color represents a true Pos Tag.

Exercise 2.11 *Run 20 epochs of the EM algorithm for part of speech induction:*

```
hmm.train_EM(train_seq, 0.1, 20, evaluate=True)
viterbi_pred_test = hmm.viterbi_decode_corpus(test_seq)
posterior_pred_test = hmm.posterior_decode_corpus(test_seq)
eval_viterbi_test =   hmm.evaluate_corpus(test_seq, viterbi_pred_test)
eval_posterior_test = hmm.evaluate_corpus(test_seq, posterior_pred_test)

Initial accuracy: 0.303638
Iter: 1 Log Likelihood: -101824.763927
Iter: 1 Accuracy: 0.305441
Iter: 2 Log Likelihood: -78057.108346
Iter: 2 Accuracy: 0.321976
Iter: 3 Log Likelihood: -77813.725501
Iter: 3 Accuracy: 0.357451
Iter: 4 Log Likelihood: -77192.947674
Iter: 4 Accuracy: 0.385109
Iter: 5 Log Likelihood: -76191.800849
Iter: 5 Accuracy: 0.392123
Iter: 6 Log Likelihood: -75242.572729
Iter: 6 Accuracy: 0.391121
Iter: 7 Log Likelihood: -74392.892496
Iter: 7 Accuracy: 0.404249
Iter: 8 Log Likelihood: -73357.542833
```

```
Iter: 8 Accuracy: 0.399940
Iter: 9 Log Likelihood: -72135.182778
Iter: 9 Accuracy: 0.399238
Iter: 10 Log Likelihood: -70924.246230
Iter: 10 Accuracy: 0.395430
Iter: 11 Log Likelihood: -69906.561800
Iter: 11 Accuracy: 0.394328
Iter: 12 Log Likelihood: -69140.228623
Iter: 12 Accuracy: 0.390821
Iter: 13 Log Likelihood: -68541.416423
Iter: 13 Accuracy: 0.391522
Iter: 14 Log Likelihood: -68053.456865
Iter: 14 Accuracy: 0.389117
Iter: 15 Log Likelihood: -67667.318961
Iter: 15 Accuracy: 0.386411
Iter: 16 Log Likelihood: -67337.685686
Iter: 16 Accuracy: 0.385409
Iter: 17 Log Likelihood: -67054.571821
Iter: 17 Accuracy: 0.385409
Iter: 18 Log Likelihood: -66769.973881
Iter: 18 Accuracy: 0.385409
Iter: 19 Log Likelihood: -66442.608458
Iter: 19 Accuracy: 0.385409

confusion_matrix = cm.build_confusion_matrix(test_seq.seq_list, viterbi_pred_test,
                                             len(corpus.tag_dict), hmm.get_num_states())
cm.plot_confusion_bar_graph(confusion_matrix, corpus.tag_dict,
                            xrange(hmm.get_num_states()), 'Confusion matrix')
```

Note: your results may not be the same as in this example since we are using a random start, but the trend should be the same, with log-likelihood increasing at every iteration.

In the previous exercise we used an HMM to do Part-of-Speech induction using 12 clusters (by omission the HMM uses as number of hidden states the one provided by the corpus). A first observation is that the log-likelihood is always increasing as expected. Another observation is that the accuracy goes up from 32% to 38%. Note that normally you will run this algorithm for 200 iterations, we stopped earlier for time constraints. Another observation is that the accuracy is not monotonic increasing, this is because the likelihood is not a perfect proxy for the accuracy. In fact all that the likelihood is measuring are co-occurrences of words in the corpus; it has no idea of pos tags. The fact that we are improving derives from the fact that language is not random but follows some specific hidden patterns. In fact this patterns are what true pos-tags try to capture. A final observation is that the performance is really bad compared to the supervised scenario, so there is a lot of space for improvement. The actual state of the art is around 71% for fully unsupervised (Graça, 2010; Berg-Kirkpatrick et al., 2010) and 80% (Das and Petrov, 2011) using parallel data and information from labels in the other language.

Looking at Figure 2.6, it shows the confusion matrix for this particular example. A first observation is that most clusters are mapped to nouns, verbs or punctuation. This is a known fact since there are many more nouns and verbs than any other tags. Since maximum likelihood prefers probabilities to be uniform (imagine two parameters: in one setting both have value 0.5 so the likelihood will be 0.5*0.5 = 0.25, while in the other case one as 0.1 and 0.9 so the maximum likelihood is 0.09). Several approaches have been proposed to address this problem, moving towards a Bayesian setting (Johnson, 2007), or using Posterior Regularization (Graça et al., 2009).

Day 3

Non-Linear Classifiers

Today's class will introduce modern neural network models, commonly known as deep learning models. We will learn the concept of *computation graph*, a general way of describing complex functions as composition of simpler functions. We will also learn about *Backpropagation*, a generic solution for gradient-descent based optimization in computation graphs.

3.1 Today's assignment

Your objective today should be to understand fully the concept of Backpropagation. For this, we will code Backpropagation in Numpy on a simple feed forward network. Then we will learn about the Pytorch module, which allows to easily create dynamic computation graphs and computes Backpropagation automatically for us. If you are new to the topic, you should aim to understand the concept of computation graph, finish the Backpropagation exercise and attain a basic understanding of Pytorch. If you already know Backpropagation well and have experience with normal Python, you should aim to complete the whole day.

3.2 Introduction to Deep Learning and Pytorch

Deep learning is the name behind the latest wave of neural network research. This is a very old topic, dating from the first half of the 20th century, that has attained formidable impact in the machine learning community recently. There is nothing particularly difficult in deep learning. You have already visited all the mathematical principles you need in the first days of the labs of this school. At their core, deep learning models are just functions mapping vector inputs $\mathbf{x}$ to vector outputs $\mathbf{y}$, constructed by composing linear and non-linear functions. This composition can be expressed in the form of a *computation graph*, where each node applies a function to its inputs and passes the result as its output. The parameters of the model are the weights given to the different inputs of nodes. This architecture vaguely resembles synapse strengths in human neural networks, hence the name artificial neural networks.

Since neural networks are just compositions of simple functions, we can apply the chain rule to derive gradients and learn the parameters of neural networks regardless of their complexity. See Section for a refresh on the basic concept. We will also refer to the gradient learning methods introduced in Section 1.4.4. Today we will focus on *feed-forward networks*. The topic of *recurrent neural networks* (RNNs) will be visited in a posterior chapter.

Some of the changes that led to the surge of deep learning are not only improvements on the existing neural network algorithms, but also the increase in the amount of data available and computing power. In particular, the use of Graphical Processing Units (GPUs) has allowed neural networks to be applied to very large datasets. Working with GPUs is not trivial as it requires dealing with specialized hardware. Luckily, as it is often the case, we are one Python import away from solving this problem.

For the particular case of deep learning, there is a growing number of toolboxes with python bindings that allow you to design custom computational graphs for GPUs some of the best known are Theano[1], TensorFlow[2] and Pytorch[3].

[1] http://deeplearning.net/software/theano/
[2] https://www.tensorflow.org/
[3] http://pytorch.org/

In these labs we will be working with Pytorch. Pytorch allows us to create computation graphs of arbitrary complexity and automatically compute the gradients of the cost with respect to any parameter. It will also produce CUDA-compatible code for use with GPUs. One salient property of Pytorch, shared with other toolkits such as Dynet or Chainer, is that its computation graphs are *dynamic*. This will be a key factor simplifying design once we start dealing with more complex models.

3.3 Computation Graphs

3.3.1 Example: The computation graph of a log-linear model

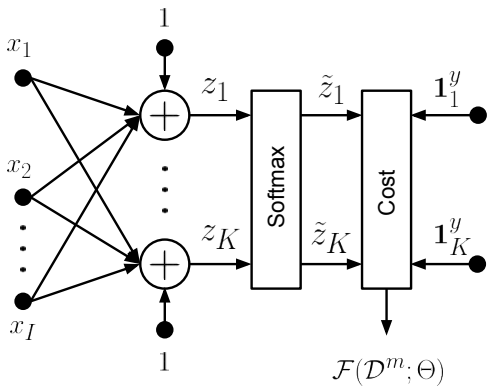

Figure 3.1: Representation of a log-linear model as a computation graph: a composition of linear and non-linear transformations. The classification cost for the m-th training example $\mathcal{D}^m = \{\mathbf{x}, y\}$ is also shown. Note $\mathbf{1}^y$ is an indicator vector of size K with a one in position y and zeros elsewhere.

A computation graph is just a way of expressing compositions of functions with a directed acyclic graph. Fig. 3.1 depicts a log-linear model. Each circle or box corresponds to a node generating one or more outputs by applying some operation over one or more inputs of the preceding nodes. Circles here denote linear transformations, that is weighted sums of the node input plus a bias. The k_{th} node output can be thus described as

$$z_k = \sum_{i=1}^{I} W_{ki} x_i + b_k, \tag{3.1}$$

where W_{ki} and b_k are weights and bias respectively. Squared boxes represent non-linear transformations. Applying a *softmax* function is a way of transforming a K dimensions real-valued vector into a vector of the same dimension where the sum of all components is one. This allows us to consider the output of this node as a probability distribution. The softmax in Fig. 3.1 can be expressed as

$$p(y = k|x) \equiv \tilde{z}_k = \frac{\exp(z_k)}{\sum_{k'=1}^{K} \exp(z_{k'})}. \tag{3.2}$$

Note that in the following sections we will also use $\mathbf{z}$ and $\tilde{\mathbf{z}}$ to denote the output of linear and non-linear functions respectively. By composing Eq. 3.1 and Eq. 3.2 we obtain a log-linear model similar to the one we saw on Chapter 1[4]. This is give by

$$p(y = k|x) = \frac{1}{Z(\mathbf{W}, \mathbf{b}, \mathbf{x})} \exp\left(\sum_{i=1}^{I} W_{ki} x_i + b_k\right), \tag{3.3}$$

[4]There are some differences with respect to Eq.1.26, like the use of a bias $\mathbf{b}$. Also, if we consider the binary joint feature mapping $\boldsymbol{f}(x, y) = \boldsymbol{g}(x) \otimes \boldsymbol{e}_y$ of Eq.1.16, the maximum entropy classifier in Eq.1.26 becomes a special case of Eq.3.3, in which the feature vector $\mathbf{x}$ only takes binary values and the bias parameters in $\mathbf{b}$ are all set to zero.

where

$$Z(\mathbf{W}, \mathbf{b}, \mathbf{x}) = \sum_{k'=1}^{K} \exp\left(\sum_{i=1}^{I} W_{k'i} x_i + b_{k'}\right) \tag{3.4}$$

is the partition function ensuring that all output values sum to one. The model thus receives a feature vector $\mathbf{x} \in \mathbb{R}^I$ and assigns a probability over $y \in 1 \cdots K$ possible class indices. It is parametrized by weights and bias $\Theta = \{\mathbf{W}, \mathbf{b}\}$, with $\mathbf{W} \in \mathbb{R}^{K \times I}$ and $\mathbf{b} \in \mathbb{R}^K$.

3.3.2 Stochastic Gradient Descent: a refresher

As we saw on day one, the parameters of a log linear model $\Theta = \{\mathbf{W}, \mathbf{b}\}$ can be learned with Stochastic Gradient Descent (SGD). To apply SGD we first need to define an error function that measures how good we are doing for any given parameter values. To remain close to the maximum entropy example, we will use as cost function the average minus posterior probability of the correct class, also known as the Cross-Entropy (CE) criterion. Bear in mind, however, that we could pick other non-linear functions and cost functions that do not have a probabilistic interpretation. For example, the same principle could be applied to a regression problem where the cost is the Mean Square Error (MSE). For a training data-set $\mathcal{D} = \{(\mathbf{x}^1, y^1), \ldots, (\mathbf{x}^M, y^M)\}$ of M examples, the CE cost function is given by

$$\mathcal{F}(\mathcal{D}; \Theta) = -\frac{1}{M} \sum_{m=1}^{M} \log p(y^m = k(m) | \mathbf{x}^m), \tag{3.5}$$

where $k(m)$ is the correct class index for the m-th example. To learn the parameters of this model with SGD, all we need to do is compute the gradient of the cost $\nabla\mathcal{F}$ with respect to the parameters of the model and iteratively update our parameter estimates as

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \nabla_{\mathbf{W}} \mathcal{F} \tag{3.6}$$

and

$$\mathbf{b} \leftarrow \mathbf{b} - \eta \nabla_{\mathbf{b}} \mathcal{F}, \tag{3.7}$$

where η is the learning rate. Note that in practice we will use a mini-batch of examples as opposed to the whole train set. Very often, more elaborated learning rules as e.g. momentum or Adagrad are used. Bear in mind that, in general, these still require the computation of the gradients as the main step. The reasoning here outlined will also be applicable to those.

3.3.3 Deriving Gradients in Computation Graphs using the Chain Rule

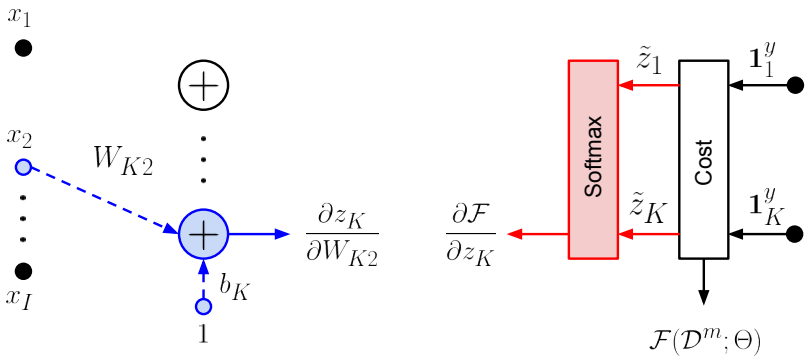

Figure 3.2: Forward-pass (blue) and Backpropagation (red) calculations to estimate the gradient of weight W_{K2} and bias b_K of a log-linear model.

The expressions for $\nabla\mathcal{F}$ are well known in the case of log-linear models. However, for the sake of the introduction to deep learning, we will show how they can be derived by exploiting the decomposition of the

cost function into the computational graph seen in the last section (and represented in Fig. 3.1). To simplify notation, and without loss of generality, we will work with the classification cost of an individual example

$$\mathcal{F}(\mathcal{D}^m;\Theta) = -\log p(y^m = k(m)|\mathbf{x}^m), \tag{3.8}$$

where $\mathcal{D}^m = \{(\mathbf{x}^m, y^m)\}$.

Lets start by computing the element (k,i) of the gradient matrix $\nabla_{\mathbf{W}}\mathcal{F}(\mathcal{D};\Theta)$, which contains the partial derivative with respect to the weight W_{ki}. To do this, we invoke the **chain rule** to split the derivative calculation into two terms at variable $z_{k'}$ (Eq.3.1) with $k' = 1 \cdots K$

$$\frac{\partial \mathcal{F}(\mathcal{D}^m;\Theta)}{\partial W_{ki}} = \sum_{k'=1}^{K} \frac{\partial \mathcal{F}(\mathcal{D}^m;\Theta)}{\partial z_{k'}} \frac{\partial z_{k'}}{\partial W_{ki}}. \tag{3.9}$$

We have thus transformed the problem of computing the derivative into computing two easier derivatives. We start by the right-most term. The relation between $z_{k'}$ and W_{ki} is given in Eq. 3.1. Since $z_{k'}$ only depends on the weight W_{ki} in a linear way, the second derivative in Eq.3.12 is given by

$$\frac{\partial z_{k'}}{\partial W_{ki}} = \frac{\partial}{\partial W_{ki}}\left(\sum_{i'=1}^{I} W_{k'i'}x_{i'}^m + b_{k'}\right) = \begin{cases} x_i^m & \text{if } k = k' \\ 0 & \text{otherwise .} \end{cases} \tag{3.10}$$

For the left-most term, the relation between $\mathcal{F}(\mathcal{D}^m;\Theta)$ and z_k is given by Eq. 3.2 together with 3.8

$$\frac{\partial \mathcal{F}(\mathcal{D}^m;\Theta)}{\partial z_k} = -\frac{\partial}{\partial z_k}\left(z_{k(m)} - \log\left(\sum_{k'=1}^{K}\exp(z_{k'})\right)\right) = \begin{cases} -(1-\tilde{z}_k) & \text{if } k = k(m) \\ -(-\tilde{z}_k) & \text{otherwise .} \end{cases} \tag{3.11}$$

Bringing the two parts together, we obtain

$$\frac{\partial \mathcal{F}(\mathcal{D}^m;\Theta)}{\partial W_{ki}} = \begin{cases} -(1-\tilde{z}_k)x_i^m & \text{if } k = k(m) \\ -(-\tilde{z}_k)x_i^m & \text{otherwise .} \end{cases}. \tag{3.12}$$

From the formula for each element and a single example, we can now obtain the gradient matrix for a batch of M examples by simply averaging and expressing the previous equations in vector form as follows

$$\nabla_{\mathbf{W}}\mathcal{F}(\mathcal{D};\Theta) = -\frac{1}{M}\sum_{m=1}^{M}\left(\mathbf{1}^{y^m} - \tilde{\mathbf{z}}^m\right)(\mathbf{x}^m)^T. \tag{3.13}$$

Here $\mathbf{1}^{y^m} \in \mathbb{R}^K$ is a vector of zeros with a one in $y^m = k(m)$, which is the index of the correct class for the example m.

In order to compute the derivatives of the cost function with respect to the bias parameters b_k, we only need to compute one additional derivative

$$\frac{\partial z_{k'}}{\partial b_k} = \begin{cases} 1 & \text{if } k = k' \\ 0 & \text{otherwise .} \end{cases} \tag{3.14}$$

This leads us to the last gradient expression

$$\nabla_{\mathbf{b}}\mathcal{F}(\mathcal{D};\Theta) = -\frac{1}{M}\sum_{m=1}^{M}\left(\mathbf{1}^{y^m} - \tilde{\mathbf{z}}^m\right). \tag{3.15}$$

An important consequence of the previous derivation is the fact that each gradient of the parameters $\nabla_{\mathbf{W}}\mathcal{F}(\mathcal{D};\Theta)$ and $\nabla_{\mathbf{b}}\mathcal{F}(\mathcal{D};\Theta)$ can be computed from two terms, displayed with corresponding colours in Fig. 3.2:

1. The derivative of the cost with respect to the k_{th} output of the linear transformation $\partial\mathcal{F}(\mathcal{D}^m;\Theta)/\partial z_k$, denoted in red. This is, in effect, the error that we propagate *backwards* from the cost layer.
2. The derivative of the forward-pass up to the linear transformation applying the weight $\partial z_k/\partial W_{ki}$, denoted in blue. This is always equal to the input multiplying that weight or one in the case of the bias.

This is the key to the Backpropagation algorithm as we will see in the next Section 3.4.

Exercise 3.1 *To ease-up the upcoming implementation exercise, examine and comment the following implementation of a log-linear model and its gradient update rule. Start by loading Amazon sentiment corpus used in day 1*

```
import lxmls.readers.sentiment_reader as srs
from lxmls.deep_learning.utils import AmazonData
corpus=srs.SentimentCorpus("books")
data = AmazonData(corpus=corpus)
```

Compare the following numpy implementation of a log-linear model with the derivations seen in the previous sections. Introduce comments on the blocks marked with # relating them to the corresponding algorithm steps.

```
from lxmls.deep_learning.utils import Model, glorot_weight_init, index2onehot
import numpy as np
from scipy.misc import logsumexp

class NumpyLogLinear(Model):

    def __init__(self, **config):

        # Initialize parameters
        weight_shape = (config['input_size'], config['num_classes'])
        # after Xavier Glorot et al
        self.weight = glorot_weight_init(weight_shape, 'softmax')
        self.bias = np.zeros((1, config['num_classes']))
        self.learning_rate = config['learning_rate']

    def log_forward(self, input=None):
        """Forward pass of the computation graph"""

        # Linear transformation
        z = np.dot(input, self.weight.T) + self.bias

        # Softmax implemented in log domain
        log_tilde_z = z - logsumexp(z, axis=1)[:, None]

        return log_tilde_z

    def predict(self, input=None):
        """Prediction: most probable class index"""
        return np.argmax(np.exp(self.log_forward(input)), axis=1)

    def update(self, input=None, output=None):
        """Stochastic Gradient Descent update"""

        # Probabilities of each class
        class_probabilities = np.exp(self.log_forward(input))
        batch_size, num_classes = class_probabilities.shape

        # Error derivative at softmax layer
        I = index2onehot(output, num_classes)
        error = (class_probabilities - I) / batch_size

        # Weight gradient
        gradient_weight = np.zeros(self.weight.shape)
        for l in range(batch_size):
            gradient_weight += np.outer(error[l, :], input[l, :])

        # Bias gradient
        gradient_bias = np.sum(error, axis=0, keepdims=True)

        # SGD update
        self.weight = self.weight - self.learning_rate * gradient_weight
        self.bias = self.bias - self.learning_rate * gradient_bias
```

Instantiate model and data classes. Check the initial accuracy of the model. This should be close to 50% since we are on a binary prediction task and the model is not trained yet.

```
# Instantiate model
model = NumpyLogLinear(
    input_size=corpus.nr_features,
    num_classes=2,
    learning_rate=0.05
)

# Define number of epochs and batch size
num_epochs = 10
batch_size = 30

# Instantiate data iterators
train_batches = data.batches('train', batch_size=batch_size)
test_set = data.batches('test', batch_size=None)[0]

# Check initial accuracy
hat_y = model.predict(input=test_set['input'])
accuracy = 100*np.mean(hat_y == test_set['output'])
print("Initial accuracy %2.2f %%" % accuracy)
```

Train the model with simple batch stochastic gradient descent. Be sure to understand each of the steps involved, including the code running inside of the model class. We will be wokring on a more complex version of the model in the upcoming exercise.

```
# Epoch loop
for epoch in range(num_epochs):

    # Batch loop
    for batch in train_batches:
        model.update(input=batch['input'], output=batch['output'])

    # Prediction for this epoch
    hat_y = model.predict(input=test_set['input'])

    # Evaluation
    accuracy = 100*np.mean(hat_y == test_set['output'])
    print("Epoch %d: accuracy %2.2f %%" % (epoch+1, accuracy))
```

3.4 Going Deeper than Log-linear by using Composition

3.4.1 The Multilayer Perceptron or Feed-Forward Network

We have seen that just by using the chain rule we can easily compute gradients for compositions of two functions (one non-linear and one linear). However, there was nothing in the derivation that would stop us from composing more than two functions. The algorithm in 10 describes the Multi-Layer Perceptron (MLP) or Feed-Forward (FF) network. In a similar fashion to the log-linear model, the MLP/FF can be expressed as a computation graph and is displayed in Fig. 3.3. Take into account the following:

- MLPs/FFs are characterized by applying functions in a set of layers subsequently to a single input. This characteristic is also shared by convolutional networks, although the latter also have parameter tying constraints.
- The non-linearities in the intermediate layers are usually one-to-one transformations. The most typical are the sigmoid, hyperbolic tangent and the rectified linear unit (ReLU).
- The output non-linearity is determined by the output to be estimated. In order to estimate probability distributions the softmax is typically used. For regression problems a last linear layer is used instead.

Algorithm 10 Forward pass of a Multi-Layer Perceptron (MLP) or Feed-Forward (FF) network

input: Initial parameters for an MLP of N layers $\Theta = \{\mathbf{W}^1, \mathbf{b}^1, \cdots \mathbf{W}^N, \mathbf{b}^N\}$
input: Input data vector $\tilde{\mathbf{z}}^0 \equiv \mathbf{x}$.
for $n = 1$ **to** $N - 1$ **do**
    Apply linear transformation

$$z_j^n = \sum_{i=1}^{I} W_{ji}^n \tilde{z}_i^{n-1} + b_j^n$$

    Apply non-linear transformation e.g. sigmoid (hereby denoted $\sigma()$)

$$\tilde{z}_j^n = \sigma(z_j^n) = \frac{1}{1 + \exp(-z_j^n)}$$

end for
Apply final linear transformation

$$z_k^N = \sum_{j=1}^{J} W_{kj}^N \tilde{z}_j^{N-1} + b_k^N$$

Apply final non-linear transformation e.g. softmax

$$p(y = k|x) \equiv \tilde{z}_k^N = \frac{\exp(z_k^N)}{\sum_{k'=1}^{K} \exp(z_{k'}^N)}$$

3.4.2 Backpropagation: an overview

For the examples in this chapter we will consider the case in which we are estimating a distribution over classes, thus we will use the CE cost function (Eq. 3.5).

To compute the gradient with respect the parameters of the n-th layer, we just need to apply the chain rule as in the previous section, consecutively. Fortunately, we do not need to repeat this procedure for each layer as it is easy to spot a recursive rule (the Backpropagation recursion) that is valid for many neural models, including feed-forward networks (such as MLPs) as well as recurrent neural networks (RNNs) with minor modifications. The Backpropagation method, which is given in Algorithm 11 for the case of an MLP, consists of the following steps:

- The forward pass step, where the input signal is injected though the network in a forward fashion (see Alg. 10)
- The Backpropagation step, where the derivative of the cost function (also called error) is injected back through the network and Backpropagated according to the derivative rules (see steps 8-17 in Alg. 11)

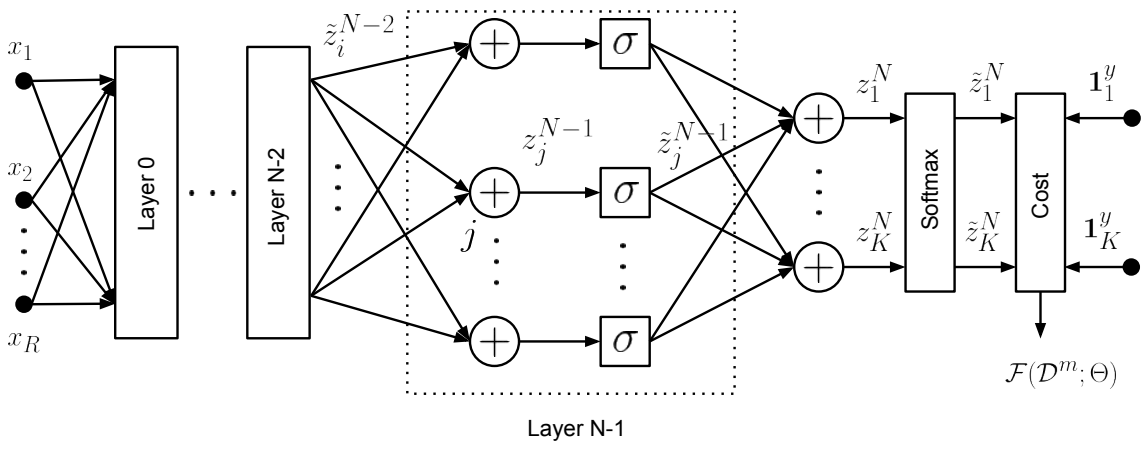

Figure 3.3: Representation of a Multi-Layer Perceptron (MLP) or Feed-Forward (FF) network as a computation graph. The classification cost for the m-th training example $\mathcal{D}^m = \{\mathbf{x}, y\}$ is also shown.

- Finally, the gradients with respect to the parameters are computed by multiplying the input signal from the forward pass and the Backpropagated error signal, at the corresponding places in the network (step 18 in Alg. 11)
- Given the gradients computed in the previous step, the model weights can then be easily update according to a specified learning rule (step 19 in Alg. 11 uses a mini-batch SGD update rule).

The main step of the method is the Backpropagation step, where one has to compute the Backpropagation recursion rules for a specific network. The next section presents a careful deduction of these recursion rules, for the present MLP model.

3.4.3 Backpropagation: deriving the rule for a feed forward network

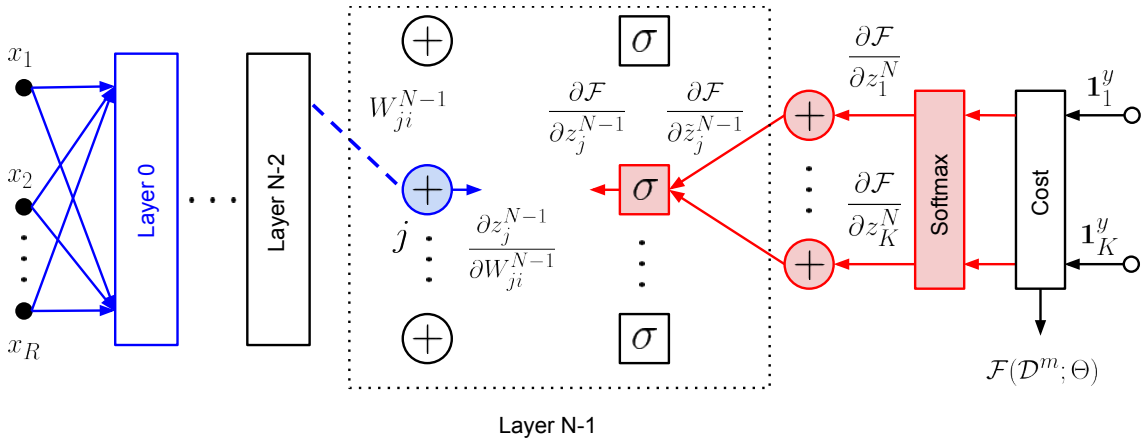

Figure 3.4: Forward-pass (blue) and Backpropagation (red) calculations to estimate the gradient of weight W_{ji} at layer $N-1$ of a MLP.

In a generic MLP we would like to compute the values of all parameters $\Theta = \{\mathbf{W}^1, \mathbf{b}^1, \cdots \mathbf{W}^N, \mathbf{b}^N\}$. As explained previously, we will thus need to compute the Backpropagated error at each node $\partial \mathcal{F}(\mathcal{D}^m;\Theta)/\partial z_k^n$, and the corresponding derivative for the forward-pass $\partial z_k^n/\partial W_{ki}$, for $n = 1 \cdots N$. Fortunately, it is easy to spot a recursion that will allow us to compute these values for each node, given all its child nodes. To spot it, we can start trying to compute the gradients one layer before the output layer (see Fig. 3.4), i.e. layer $N-1$. We start by splitting at with respect to the output of the linear layer at $N-1$

$$\frac{\partial \mathcal{F}(\mathcal{D}^m;\Theta)}{\partial W_{ji}^{N-1}} = \sum_{j'=1}^{J} \frac{\partial \mathcal{F}(\mathcal{D}^m;\Theta)}{\partial z_{j'}^{N-1}} \frac{\partial z_{j'}^{N-1}}{\partial W_{ji}^{N-1}} = \frac{\partial \mathcal{F}(\mathcal{D}^m;\Theta)}{\partial z_j^{N-1}} \frac{\partial z_j^{N-1}}{\partial W_{ji}^{N-1}} \tag{3.16}$$

where, as in the case of the log-linear model, we have used the fact that the ouput of the linear layer z_j^{N-1} only depends on W_{ji}^{N-1}. We now pick the left-most factor and apply the chain rule to split by the output of the non-linear layer $\tilde{z}_j^{N-1}$. Assuming that the non linear transformation is one-to-one, as e.g. a sigmoid, tanh, relu we have

$$\frac{\partial \mathcal{F}(\mathcal{D}^m;\Theta)}{\partial z_j^{N-1}} = \left(\sum_{j'=1}^{J} \frac{\partial \mathcal{F}(\mathcal{D}^m;\Theta)}{\partial \tilde{z}_{j'}^{N-1}} \frac{\partial \tilde{z}_{j'}^{N-1}}{\partial z_j^{N-1}} \right) = \frac{\partial \mathcal{F}(\mathcal{D}^m;\Theta)}{\partial \tilde{z}_j^{N-1}} \frac{\partial \tilde{z}_j^{N-1}}{\partial z_j^{N-1}} \tag{3.17}$$

To spot a recursion we only need to apply the chain rule a third time. The next variable to split by is the linear output of layer N, z_j^N. By looking at Fig. 3.4, it is clear that the derivatives at each node in layer $N-1$ will depend on all values of layer N. For the linear layer the summation wont go away yielding

$$\frac{\partial \mathcal{F}(\mathcal{D}^m;\Theta)}{\partial z_j^{N-1}} = \left(\sum_{k'=1}^{K} \frac{\partial \mathcal{F}(\mathcal{D}^m;\Theta)}{\partial z_{k'}^{N}} \frac{\partial z_{k'}^{N}}{\partial \tilde{z}_j^{N-1}} \right) \frac{\partial \tilde{z}_j^{N-1}}{\partial z_j^{N-1}} \tag{3.18}$$

If we call the derivative of the error with respect to the N_{th} linear layer output as

$$e_k^N = \frac{\partial \mathcal{F}(\mathcal{D}^m; \Theta)}{\partial z_k^N} \tag{3.19}$$

it is easy to deduce from Eqs. 3.17, 3.18 that

$$e_j^{N-1} = \left(\sum_{k'=1}^{K} e_{k'}^N \frac{\partial z_{k'}^N}{\partial \tilde{z}_j^{N-1}} \right) \frac{\partial \tilde{z}_j^{N-1}}{\partial z_j^{N-1}}. \tag{3.20}$$

coming back to the original 3.16 we obtain the formula for the update of each of the weights and bias

$$\frac{\partial \mathcal{F}(\mathcal{D}^m; \Theta)}{\partial W_{ji}^{N-1}} = e_j^{N-1} \tilde{z}_i, \quad \frac{\partial \mathcal{F}(\mathcal{D}^m; \Theta)}{\partial b_j^{N-1}} = e_j^{N-1} \tag{3.21}$$

These formulas are valid for any FF network with hidden layers using one-to-one non-linearities. For the network described in Algorithm 10 we have

$$\mathbf{e}^N = \mathbf{1}^y - \tilde{\mathbf{z}}^N, \quad \frac{\partial z_{k'}^N}{\partial \tilde{z}_j^{N-1}} = W_{k'j}^N \quad \text{and} \quad \frac{\partial \tilde{z}_j^n}{\partial z_j^n} = \tilde{z}_j^n \cdot (1 - \tilde{z}_j^n) \quad \text{with} \quad n \in \{1 \cdots N-2\} \tag{3.22}$$

A more detailed version can be seen in Algorithm 11

3.4.4 Backpropagation as a general rule

Once we get comfortable with the derivation of Backpropagation for the FF, it is simple to see that expanding to generic computations graphs is trivial. If we wanted to change the sigmoid non-linearity by a Rectified Linear Unit (ReLU) we would only need to change forward and Backpropagation derivative of the hidden non-linearities as

$$\tilde{z}_j = \begin{cases} z_j & \text{if } z_j >= 0 \\ 0 & \text{otherwise .} \end{cases} \qquad \frac{\partial \tilde{z}_j}{\partial z_j} = \begin{cases} 1 & \text{if } z_j > 0 \\ 0 & \text{otherwise .} \end{cases} \tag{3.23}$$

More importantly, Backpropagation can be always defined as a direct acyclic graph with the *reverse* direction of the forward-pass, where at each node we apply the transpose of the Jacobian of each linear or non linear transformation. Coming back to Eq. 3.22 we have the vector formula

$$\mathbf{e}^{N-1} = \left(\mathrm{J}_{\tilde{z}}^{N-1} \right)^T \left(\mathrm{J}_W^N \right)^T \mathbf{e}^N. \tag{3.24}$$

In other words, regardless of the topology of the network and as long as we can compute the forward-pass and the Jacobian of each individual node, we will be able to compute Backpropagation.

Algorithm 11 Mini-batch SGD with Back-Propagation

input: Data $\mathcal{D} = \{\mathcal{D}_1, \mathcal{D}_2, ..., \mathcal{D}_B\}$ split into B mini-batches of size M', MLP of N layers, with parameters $\Theta = \{\mathbf{W}^1, \mathbf{b}^1, \cdots \mathbf{W}^N, \mathbf{b}^N\}$, number of rounds T, learning rate η
initialize parameters Θ randomly
for $t = 1$ **to** T **do**
  for $b = 1$ **to** B **do**

    for $m = 1$ **to** M' **do**
      Compute the forward pass for each of the M' examples in batch b; keep not only $p(y^m|\mathbf{x}^m) \equiv \tilde{\mathbf{z}}^{m,N}$ but also all the intermediate non-linear outputs $\tilde{\mathbf{z}}^{m,1} \cdots \tilde{\mathbf{z}}^{m,N}$.
    end for

    for $n = N$ **to** 1 **do**
      if n==N **then**
        for $m = 1$ **to** M' **do**
          Initialize the error at last layer, for each example m. For the softmax with CE cost this is given by:

$$\mathbf{e}^{m,N} = \left(\mathbf{1}_{k(m)} - \tilde{\mathbf{z}}^{m,N}\right).$$

        end for
      else
        for $m = 1$ **to** M' **do**
          Backpropagate the error through the linear layer, for each example m:

$$\mathbf{e}^{m} = \left((\mathbf{W}^{n+1})^T \mathbf{e}^{m,n+1}\right)$$

          Backpropagate the error through the non-linearity, for the sigmoid this is:

$$\mathbf{e}^{m,n} = \mathbf{e}^{m} \odot \tilde{\mathbf{z}}^{m,n} \odot (1 - \tilde{\mathbf{z}}^{m,n}),$$

          where $\odot$ is the element-wise product and the 1 is replicated to match the size of $\tilde{\mathbf{z}}^n$.
        end for
      end if

      Compute the gradients using the backpropagated errors and the inputs from the forward pass

$$\nabla_{\mathbf{W}^n} \mathcal{F}(\mathcal{D}; \Theta) = -\frac{1}{M'} \sum_{m=1}^{M'} \mathbf{e}^{m,n} \cdot \left(\tilde{\mathbf{z}}^{m,n-1}\right)^T,$$

$$\nabla_{\mathbf{b}^n} \mathcal{F}(\mathcal{D}; \Theta) = -\frac{1}{M'} \sum_{m=1}^{M'} \mathbf{e}^{m,n}.$$

      Update the parameters

$$\mathbf{W}^n \leftarrow \mathbf{W}^n - \eta \nabla_{\mathbf{W}^n} \mathcal{F},$$
$$\mathbf{b}^n \leftarrow \mathbf{b}^n - \eta \nabla_{\mathbf{b}^n} \mathcal{F}.$$

    end for
  end for
end for

Exercise 3.2 *Instantiate the feed-forward model class and optimization parameters. This models follows the architecture described in Algorithm 10.*

```
# Model
geometry = [corpus.nr_features, 20, 2]
activation_functions = ['sigmoid', 'softmax']

# Optimization
learning_rate = 0.05
num_epochs = 10
batch_size = 30

# Instantiate model
from lxmls.deep_learning.numpy_models.mlp import NumpyMLP
model = NumpyMLP(
    geometry=geometry,
    activation_functions=activation_functions,
    learning_rate=learning_rate
)
```

Open the code for this model. This is located in lxmls/deep learning/numpy models/mlp.py. Implement the method 'back-propagation()' in the class 'NumpyMLP' using Backpropagation recursion that we just saw.

As a first step focus on getting the gradients of each layer, one at a time. Use the code below to plot the loss values for the study weight and perturbed versions.

```
from lxmls.deep_learning.mlp import get_mlp_parameter_handlers, get_mlp_loss_range

# Get functions to get and set values of a particular weight of the model
get_parameter, set_parameter = get_mlp_parameter_handlers(
    layer_index=1,
    is_bias=False,
    row=0,
    column=0
)

# Get batch of data
batch = data.batches('train', batch_size=batch_size)[0]

# Get loss and weight value
current_loss = model.cross_entropy_loss(batch['input'], batch['output'])
current_weight = get_parameter(model.parameters)

# Get range of values of the weight and loss around current parameters values
weight_range, loss_range = get_mlp_loss_range(model, get_parameter, set_parameter, batch)
```

Once you have implemented at least the gradient of the last layer. You can start checking if the values match

```
gradients = model.backpropagation(batch['input'], batch['output'])
current_gradient = get_parameter(gradients)
```

Now you can plot the values of the loss around a given parameters value versus the gradient. If you have implemented this correctly the gradient should be tangent to the loss at the current weight value, see Figure 3.5. Once you have completed the exercise, you should be able to plot also the gradients of the other layers. Take into account that the gradients for the first layer will only be non zero for the indices of words present in the batch. You can locate this using.

```
batch['input'][0].nonzero()
```

Copy the following code for plotting

```
%matplotlib inline  # for jupyter notebooks
import matplotlib.pyplot as plt
# Plot empirical
plt.plot(weight_range, loss_range)
plt.plot(current_weight, current_loss, 'xr')
plt.ylabel('loss value')
plt.xlabel('weight value')
# Plot real
h = plt.plot(
    weight_range,
    current_gradient*(weight_range - current_weight) + current_loss,
    'r--'
)
```

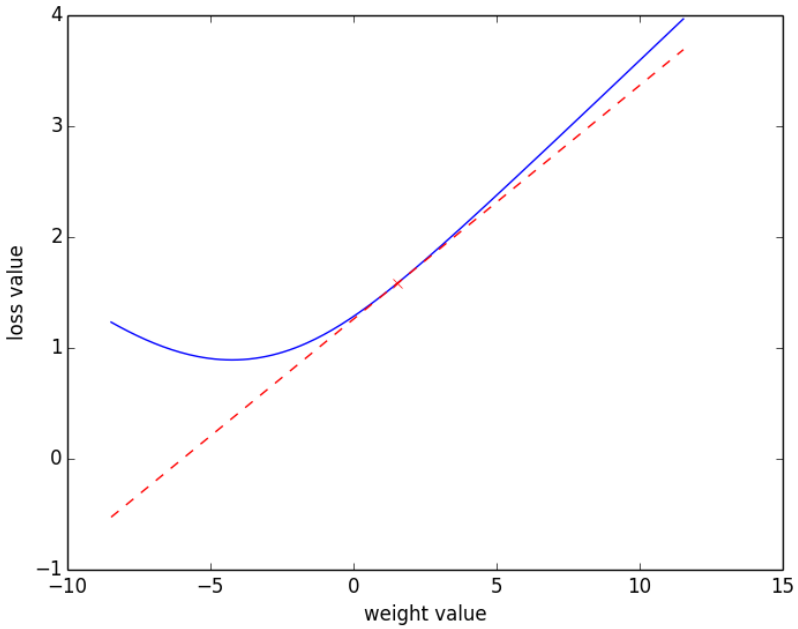

Figure 3.5: Values of the loss (blue) and gradient (dashed red) for a given weight of the network, as well loss values for small perturbations of the weight.

After you have ensured that your Backpropagation algorithm is correct, you can train a model with the data we have.

```
# Get batch iterators for train and test
train_batches = data.batches('train', batch_size=batch_size)
test_set = data.batches('test', batch_size=None)[0]

# Epoch loop
for epoch in range(num_epochs):

    # Batch loop
    for batch in train_batches:
        model.update(input=batch['input'], output=batch['output'])

    # Prediction for this epoch
    hat_y = model.predict(input=test_set['input'])

    # Evaluation
    accuracy = 100*np.mean(hat_y == test_set['output'])
    print("Epoch %d: accuracy %2.2f %%" % (epoch+1, accuracy))
```

3.4.5 Some final reflections on Backpropagation

If you are new to the neural network topic, this is about the most important piece of theory you should learn about deep learning. Here are some reflections that you should keep in mind.

- Backpropagation allows us in principle to compute the gradients for any differentiable computation graph.
- We only need to know the forward-pass and Jacobian of each individual node in the network to implement Backpropagation.
- Learning guarantees are however weaker than for expectation maximization or convex optimization algorithms.
- In practice optimization will often get trapped on local minima and exhibit high variance in performance for small changes.

3.5 Deriving gradients and GPU code with Pytorch

3.5.1 An Introduction to Pytorch and Computation Graph Toolkits

As you may have observed, the speed of SGD training for MLPs slows down considerably when we increase the number of layers. One reason for this is that the code that we use here is not very optimized. It is thought for you to learn the basic principles. Even if the code was more optimized, it would still be very slow for reasonable network sizes. The cost of computing each linear layer is proportional to the dimensionality of the previous and current layers, which in most cases will be rather large.

For this reason most deep learning applications use Graphics Processing Units (GPU) in their computations. This specialized hardware is normally used to accelerate computer graphics, but can also be used for some computation intensive tasks like matrix multiplications. However, we need to deal with specific interfaces and operations in order to use a GPU. This is where Pytorch comes in. Pytorch is a computation graph toolkit with following nice features

- Automatic differentiation. We only need to express the computation graph of the forward pass. Pytorch will compute the gradients for us.
- GPU integration: The code will be ready to work on a GPU.
- An active community focused on the application of Pytorch to Deep Learning.
- Dynamic computation graphs. This allows us to change the computation graph within each update.

Note that all of these properties are separately available in other toolkits. Dynet has very good dynamic graph functionality and cpu performance, Tensor Flow is backed by Google and has a large community, Amazon's MXNet of Microsoft's CNTK are also competing to play a central role. It is hard to say at this point which toolkit will be the best option in the future. At this point we chose Pytorch because strikes a balance between a strong community, ease of use and dynamic computation graphs. Also take into account that transiting from a toolkit to another is not very complicated, as the primitives are relatively similar across them.

In general, and compared to numpy, computation graph toolkits less easy to use. In the case of Pytorch we will have to consider following aspects

- Pytorch types are less flexible than numpy arrays since they have to act on data stored on the GPU. Casting of all variables to Pytorch types will be often a source of errors.
- Not all operations available in numpy are available on Pytorch. Also the semantics of the function may differ.
- Despite being a big improvement compared to the early toolkits like Theano, Pytorch errors can still be difficult to track sometimes.
- As we will see, Pytorch has a good GPU performance, but its CPU performance is not great. Particularly at small sizes.

Exercise 3.3 *In order to learn the differences between a numpy and a Pytorch implementation, explore the reimplementation of Ex. 3.1 in Pytorch. Compare the content of each of the functions, in particular the forward() and update methods(). The comments indicated as IMPORTANT will highlight common sources of errors.*

```
import torch
from torch.autograd import Variable

class PytorchLogLinear(Model):

    def __init__(self, **config):

        # Initialize parameters
        weight_shape = (config['input_size'], config['num_classes'])
        # after Xavier Glorot et al
        self.weight = glorot_weight_init(weight_shape, 'softmax')
        self.bias = np.zeros((1, config['num_classes']))
        self.learning_rate = config['learning_rate']

        # IMPORTANT: Cast to pytorch format
        self.weight = Variable(torch.from_numpy(self.weight).float(), requires_grad=True)
        self.bias = Variable(torch.from_numpy(self.bias).float(), requires_grad=True)

        # Instantiate softmax and negative logkelihood in log domain
        self.logsoftmax = torch.nn.LogSoftmax(dim=1)
        self.loss = torch.nn.NLLLoss()

    def _log_forward(self, input=None):
        """Forward pass of the computation graph in logarithm domain (pytorch)"""

        # IMPORTANT: Cast to pytorch format
        input = Variable(torch.from_numpy(input).float(), requires_grad=False)

        # Linear transformation
        z =  torch.matmul(input, torch.t(self.weight)) + self.bias

        # Softmax implemented in log domain
        log_tilde_z = self.logsoftmax(z)

        # NOTE that this is a pytorch class!
        return log_tilde_z

    def predict(self, input=None):
        """Most probably class index"""
        log_forward = self._log_forward(input).data.numpy()
        return np.argmax(np.exp(log_forward), axis=1)

    def update(self, input=None, output=None):
        """Stochastic Gradient Descent update"""

        # IMPORTANT: Class indices need to be casted to LONG
        true_class = Variable(torch.from_numpy(output).long(), requires_grad=False)

        # Compute negative log-likelihood loss
        loss = self.loss(self._log_forward(input), true_class)
        # Use autograd to compute the backward pass.
        loss.backward()

        # SGD update and zero gradients
        self.weight.data -= self.learning_rate * self.weight.grad.data
        self.weight.grad.data.zero_()
        self.bias.data -= self.learning_rate * self.bias.grad.data
        self.bias.grad.data.zero_()

        return loss.data.numpy()
```

Once you understand the model you can instantiate it and run it using the standard training loop we have used on previous exercises.

```
# Instantiate model
model = PytorchLogLinear(
    input_size=corpus.nr_features,
    num_classes=2,
    learning_rate=0.05
)
```

Exercise 3.4 *As the final exercise today implement the log_forward() method in lxmls/deep_learning/pytorch_models/mlp.py. Use the previous exercise as reference. After you have completed this you can run both systems for comparison.*

```
# Model
geometry = [corpus.nr_features, 20, 2]
activation_functions = ['sigmoid', 'softmax']

# Instantiate model
import numpy as np
from lxmls.deep_learning.pytorch_models.mlp import PytorchMLP
model = PytorchMLP(
    geometry=geometry,
    activation_functions=activation_functions,
    learning_rate=0.05
)
```

Day 4

Learning Structured Predictors

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

Today's Assignment

The assignment of today's class is to implement the structured version of the perceptron algorithm.

4.1 Classification vs Sequential Classification

Table 4.1 shows how the models for classification have counterparts in *sequential* classification. In fact, in the last chapter we discussed the Hidden Markov model, which can be seen as a generalization of the Naïve Bayes model for sequences. In this chapter, we will see a generalization of the Perceptron algorithm for sequence problems (yielding the Structured Perceptron, Collins 2002) and a generalization of Maximum Entropy model for sequences (yielding Conditional Random Fields, Lafferty et al. 2001). Note that both these generalizations are not specific for sequences and can be applied to a wide range of models (we will see in tomorrow's lecture how these methods can be applied to parsing). Although we will not cover all the methods described in Chapter 1, bear in mind that all of those have a structured counterpart. It should be intuitive after this lecture how those methods could be extended to structured problems, given the perceptron example.

Classification	**Sequences**
Generative	
Naïve Bayes (Sec. 1.2)	Hidden Markov Models (Sec. 2.2)
Discriminative	
Perceptron (Sec. 1.4.3)	Structured Perceptron (Sec. 4.5)
Maximum Entropy (Sec. 1.4.4)	Conditional Random Fields (Sec. 4.4)

Table 4.1: Summary of the methods used for classification and sequential classification covered in this guide.

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

$$\arg\max_{y \in \Lambda^N} P(Y = y | X = x) = \arg\max_{y \in \Lambda^N} \boldsymbol{w} \cdot \boldsymbol{f}(x, y). \tag{4.1}$$

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

As in the previous section, $y = y_1 \dots y_N$ is a sequence so the maximization is over an exponential number of objects, making it intractable for brute force methods. Again we will make an assumption analogous to the

"first order Markov independence property," so that the features may decompose as a sum over nodes and edges in a trellis. This is done by assuming that expression 4.1 can be written as:

$$\arg\max_{y \in \Lambda^N} \sum_{i=1}^{N} \underbrace{\boldsymbol{w} \cdot \boldsymbol{f}_{\text{emiss}}(i, x, y_i)}_{\text{score}_{\text{emiss}}} + \underbrace{\boldsymbol{w} \cdot \boldsymbol{f}_{\text{init}}(x, y_1)}_{\text{score}_{\text{init}}} + \sum_{i=1}^{N-1} \underbrace{\boldsymbol{w} \cdot \boldsymbol{f}_{\text{trans}}(i, x, y_i, y_{i+1})}_{\text{score}_{\text{trans}}} + \underbrace{\boldsymbol{w} \cdot \boldsymbol{f}_{\text{final}}(x, y_N)}_{\text{score}_{\text{final}}} \tag{4.2}$$

In other words, the scores $\text{score}_{\text{emiss}}$, $\text{score}_{\text{init}}$, $\text{score}_{\text{trans}}$ and $\text{score}_{\text{final}}$ are now computed as inner products between weight vectors and feature vectors rather than log-probabilities. The feature vectors depend locally on the output variable (*i.e.*, they only look at a single y_i or a pair y_i, y_{i+1}); however they may depend globally on the observed input $x = x_1, \ldots, x_N$.

4.2 Features

In this section we will define two simple feature sets.[1] The first feature set 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. Table 4.2 depicts the features that are implicit in the HMM, which was the subject of the previous chapter. These features are indicators of initial, transition, final, and emission events.

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

Table 4.2: `IDFeatures` feature set. This set replicates the features used by the HMM model.

Note that the fact that we were using a generative model has forced us (in some sense) to make strong independence assumptions. However, since we now move to a discriminative approach, where we model $P(Y|X)$ rather than $P(X, Y)$, we are not tied anymore to some of these assumptions. In particular:

- We may use "overlapping" features, *e.g.*, features that fire simultaneously for many instances. For example, we can use a feature for a word, such as a feature which fires for the word "brilliantly", and another for prefixes and suffixes of that word, such as one which fires if the last two letters of the word are "ly". This would lead to an awkward model if we wanted to insist on a generative approach.
- We may use features that depend arbitrarily on the *entire input sequence* x. On the other hand, we still need to resort to "local" features with respect to the *outputs* (*e.g.* looking only at consecutive state pairs), otherwise decoding algorithms will become more expensive.

This leads us to the second feature set, composed of features that are traditionally used in POS tagging with discriminative models. See Table 4.3 for some examples. Of course, we could have much more complex features, looking arbitrarily to the input sequence. We are not going to have them in this exercise only for performance reasons (to have less features and smaller caches). State-of-the-art sequence classifiers can easily reach over one million features!

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

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

$$\text{score}_{\text{init}}(x, y_1) = \boldsymbol{w} \cdot \boldsymbol{f}_{\text{init}}(x, y_1). \tag{4.3}$$

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

$$\text{score}_{\text{trans}}(i, x, y_i, y_{i+1}) = \boldsymbol{w} \cdot \boldsymbol{f}_{\text{trans}}(i, x, y_i, y_{i+1}). \tag{4.4}$$

[1] Although not required, all the features we will use are binary features, indicating whether a given condition is satisfied or not.

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

Table 4.3: `Extended` feature set. Some features in this set could not be included in the HMM model. The features included in the bottom row are all considered emission features for the purpose of our implementation, since they all depend on i, x and y_i.

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

$$\text{score}_{\text{final}}(x, y_N) = \boldsymbol{w} \cdot \boldsymbol{f}_{\text{final}}(x, y_N). \tag{4.5}$$

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

$$\text{score}_{\text{emiss}}(i, x, y_i) = \boldsymbol{w} \cdot \boldsymbol{f}_{\text{emiss}}(i, x, y_i). \tag{4.6}$$

4.3 Discriminative Sequential Classifiers

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

$$P_{\boldsymbol{w}}(Y = y|X = x) = \frac{1}{Z(\boldsymbol{w}, x)} \exp\left(\boldsymbol{w} \cdot \boldsymbol{f}_{\text{init}}(x, y_1) + \sum_{i=1}^{N-1} \boldsymbol{w} \cdot \boldsymbol{f}_{\text{trans}}(i, x, y_i, y_{i+1}) + \boldsymbol{w} \cdot \boldsymbol{f}_{\text{final}}(x, y_N) + \sum_{i=1}^{N} \boldsymbol{w} \cdot \boldsymbol{f}_{\text{emiss}}(i, x, y_i)\right) \tag{4.7}$$

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

$$Z(\boldsymbol{w}, x) = \sum_{y \in \Lambda^N} \exp\left(\boldsymbol{w} \cdot \boldsymbol{f}_{\text{init}}(x, y_1) + \sum_{i=1}^{N-1} \boldsymbol{w} \cdot \boldsymbol{f}_{\text{trans}}(i, x, y_i, y_{i+1}) + \boldsymbol{w} \cdot \boldsymbol{f}_{\text{final}}(x, y_N) + \sum_{i=1}^{N} \boldsymbol{w} \cdot \boldsymbol{f}_{\text{emiss}}(i, x, y_i)\right) \tag{4.8}$$

4.3.1 Training

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

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

$$\widehat{\boldsymbol{w}} = \arg\max_{\boldsymbol{w}} \sum_{m=1}^{M} \log P_{\boldsymbol{w}}(Y = y^m|X = x^m). \tag{4.9}$$

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

$$\widehat{\boldsymbol{w}} = \arg\max_{\boldsymbol{w}} \sum_{m=1}^{M} \log P_{\boldsymbol{w}}(Y = y^m|X = x^m) - \frac{\lambda}{2}\|\boldsymbol{w}\|^2. \tag{4.10}$$

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

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

4.3.2 Decoding

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

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

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

4.4 Conditional Random Fields

Conditional Random Fields (CRF) (Lafferty et al., 2001) can be seen as an extension of the Maximum Entropy (ME) models to structured problems.[2] They are *globally* normalized models: the probability of a given sentence is given by Equation 4.7. There are only two differences with respect to the standard ME model described a couple of days ago for multi-class classification:

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

Algorithm 12 shows the pseudo code to optimize a CRF with the stochastic gradient descent (SGD) algorithm (our toolkit also includes an implementation of a quasi-Newton method, L-BFGS, which converges faster, but for the purpose of this exercise, we will stick with SGD).

Exercise 4.1 *In this exercise you will train a CRF using different feature sets for Part-of-Speech Tagging. Start with the code below, which uses the ID feature set from table 4.2.*

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

print("CRF Exercise")

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

[2] An earlier, less successful, attempt to perform such an extension was via Maximum Entropy Markov models (MEMM) (McCallum et al., 2000). There each factor (a node or edge) is a *locally* normalized maximum entropy model. A shortcoming of MEMMs is its so-called *labeling bias* (Bottou, 1991), which makes them biased towards states with few successor states (see Lafferty et al. (2001) for more information).

Algorithm 12 SGD for Conditional Random Fields

input: $\mathcal{D}$, λ, number of rounds T, learning rate sequence $(\eta_t)_{t=1,\ldots,T}$
initialize $\boldsymbol{w}^1 = \mathbf{0}$
for $t = 1$ **to** T **do**
  choose $m = m(t)$ randomly
  take training pair (x^m, y^m) and compute the probability $P_{\boldsymbol{w}}(Y = y^m | X = x^m)$ using the current model $\boldsymbol{w}$ and Eq. 4.7.
  for every i, y'_i, and y'_{i+1}, compute marginal probabilities $P(Y_i = y'_i | X = x)$ and $P(Y_i = y'_i, Y_{i+1} = y'_{i+1} | X = x^m)$ using the current model $\boldsymbol{w}$, for each node and edge, through the forward-backward algorithm.
  compute the feature vector expectation:

$$\begin{aligned} E_{\boldsymbol{w}^t}[\boldsymbol{f}(x^m, Y)] &= \sum_{y_1} P(y_1|x^m) \boldsymbol{f}_{\text{init}}(x^m, y_1) + \\ &\quad \sum_{i=1}^{N-1} \sum_{y_i, y_{i+1}} P(y_i, y_{i+1}|x^m) \boldsymbol{f}_{\text{trans}}(i, x^m, y_i, y_{i+1}) + \\ &\quad \sum_{y_N} P(y_N|x^m) \boldsymbol{f}_{\text{final}}(x^m, y_N) + \\ &\quad \sum_{i=1}^{N} \sum_{y_i} P(y_i|x^m) \boldsymbol{f}_{\text{emiss}}(i, x^m, y_i). \end{aligned} \tag{4.11}$$

  update the model:

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

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

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

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

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

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

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

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

You should get values similar to these:

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

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

Exercise 4.2 *Repeat the previous exercise using the extended feature set. Compare the results.*

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

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

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

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

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

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

You should get values close to the following:

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

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

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

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

4.5 Structured Perceptron

The structured perceptron (Collins, 2002), 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 about the perceptron algorithm (Section 1.4.3) and compare it against the structured perceptron (Algorithm 13).

Algorithm 13 Averaged Structured perceptron

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

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

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

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

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

4.6 Assignment

Exercise 4.3 *Implement the structured perceptron algorithm.*
To do this, edit file `sequences/structured_perceptron.py` *and implement the function*

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

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

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

defined in file `sequences/crf_online.py`. *You will need to replace the computation of posterior marginals by the Viterbi algorithm, and to change the parameter updates according to Algorithm 13. Note the role of the functions*

```
self.feature_mapper.get_*_features()
```

in providing the indices for the features obtained for $\boldsymbol{f}(x^m, y^m)$ *or* $\boldsymbol{f}(x^m, \widehat{y})$

Exercise 4.4 *Repeat Exercises 4.1–4.2 using the structured perceptron algorithm instead of a CRF. Report the results.*
Here is the code for the simple feature set:

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

print("Perceptron Exercise")

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

Epoch: 0 Accuracy: 0.656806
Epoch: 1 Accuracy: 0.820898
Epoch: 2 Accuracy: 0.879176
Epoch: 3 Accuracy: 0.907432
Epoch: 4 Accuracy: 0.925239
Epoch: 5 Accuracy: 0.939956
Epoch: 6 Accuracy: 0.946284
Epoch: 7 Accuracy: 0.953790
Epoch: 8 Accuracy: 0.958499
Epoch: 9 Accuracy: 0.955114
Epoch: 10 Accuracy: 0.959235
Epoch: 11 Accuracy: 0.968065
Epoch: 12 Accuracy: 0.968212
Epoch: 13 Accuracy: 0.966740
Epoch: 14 Accuracy: 0.971302
Epoch: 15 Accuracy: 0.968653
Epoch: 16 Accuracy: 0.970419
Epoch: 17 Accuracy: 0.971891
Epoch: 18 Accuracy: 0.971744
Epoch: 19 Accuracy: 0.973510

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



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

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

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

Here is the code for the extended feature set:

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

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

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

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

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

And here are the expected results:

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

Day 5

Non-Linear Sequence Models

5.1 Today's assignment

Today's class will be focused on advanced deep learning concepts, mainly Recurrent Neural Networks (RNNs). In the first day we saw how the chain-rule allowed us to compute gradients for arbitrary computation graphs. Today we will see that we can still do this for more complex models like Recurrent Neural Networks (RNNs). In these models we will input data in different points of the graph, which will correspond to different time instants. The key factor to consider is that, for a fixed number of time steps, this is still a computation graph and all what we saw on the first day applies with no need for extra math.

If you managed to finish the previous day completely you should aim at finishing this as well. If you still have pending exercises from the first day e.g. the Pytorch part. It is recommended that you try to solve them first and then continue with this day.

5.2 Recurrent Neural Networks: Backpropagation Through Time

5.2.1 Feed Forward Networks Unfolded in Time

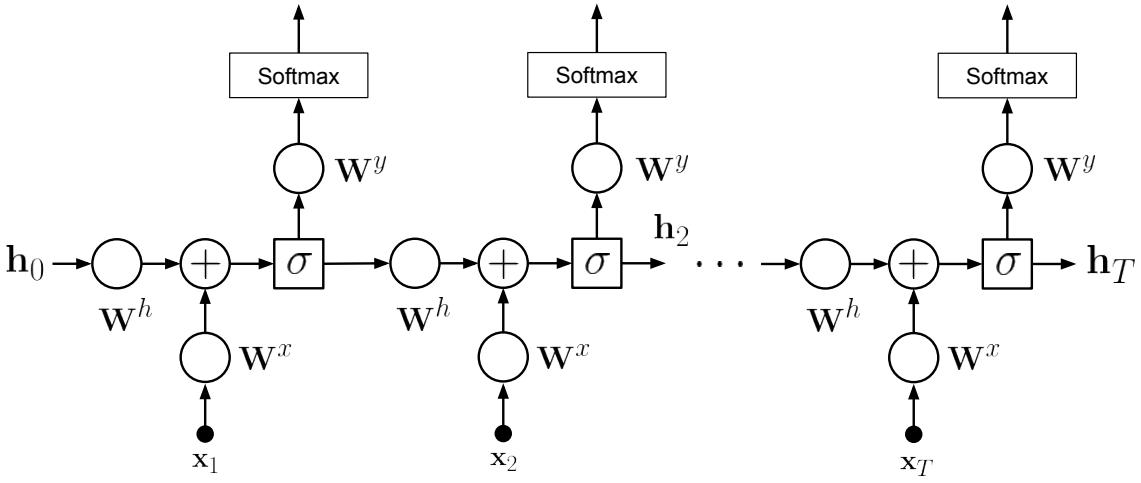

Figure 5.1: The simplest RNN can be seen as replicating a single hidden-layer FF network T times and passing the intermediate hidden variable $\mathbf{h}_t$ across different steps. Note that all nodes operate over vector inputs e.g,. $\mathbf{x}_t \in \mathbb{R}^I$. Circles indicate matrix multiplications.

We have seen already Feed Forward (FF) networks. These networks are ill suited to learn variable length patterns since they only accept inputs of a fixed size. In order to learn sequences using neural networks, we need therefore to define some architecture that is able to process variable length inputs. Recurrent Neural Networks (RNNs) solve this problem by unfolding the computation graph in time. In other words, the network is replicated as many times as it is necessary to cover the sequence to be modeled. In order to model the sequence one or more connections across different time instants are created. This allows the network to have a memory in time and thus capture complex patterns in sequences. In the simplest model, depicted in Fig. 5.2,

and detailed in Algorithm 5.2.2, a RNN is created by replicating a single hidden-layer FF network T times and passing the intermediate hidden variable across different steps. The strength of the connection is determined by the weight matrix $\mathbf{W}_h$

5.2.2 Backpropagating through Unfolded Networks

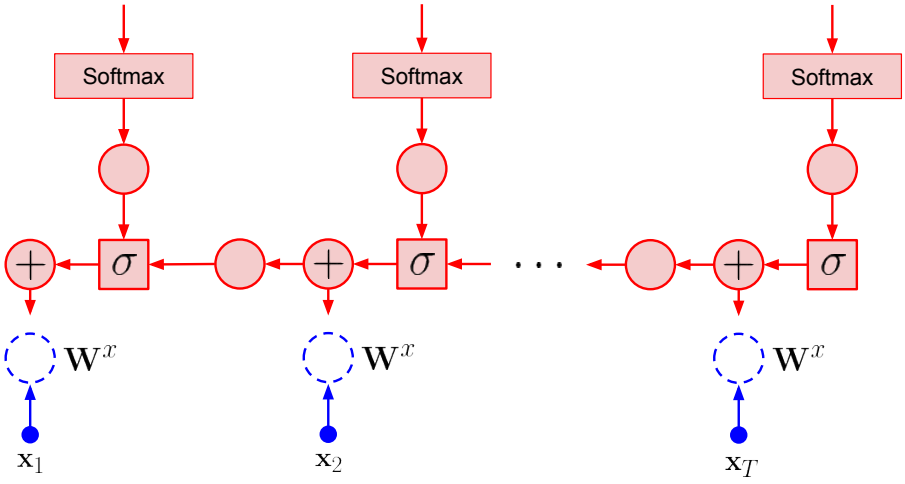

Figure 5.2: Forward-pass (blue) and backpropagated error (red) to the input layer of an RNN. Note that a copy of the error is sent to each output of each sum node $(+)$

It is important to note that there is no formal changes needed to apply backpropagation to RNNs. It concerns applying the chain rule just as it happened with FFs. It is however useful to consider the following properties of derivatives, which are not relevant when dealing with FFs

- When two variables are summed up in the forward-pass, the error is backpropagated to each of the summand sub-graphs
- When unfolding in T steps the same parameters will be copied T times. All updates for each copy are summed up to compute the total gradient.

Despite the lack of formal changes, the fact that we backpropagate an error over the length of the entire sequence often leads to numerical problems. The problem of *vanishing* and *exploding* gradients are a well know limitation. A number of solutions are used to mitigate this issue. One simple, yet inelegant, method is clipping the gradients to a fixed threshold. Another solution is to resort to more complex RNN models that are able to better handle long range dependencies and are less sensitive to this phenomena. It is important to bear in mind, however, that all RNNs still use backpropagation as seen in the previous day, although it is often referred as *Backpropagation through time*.

Exercise 5.1 *Convince yourself that a RNN is just an FF unfolded in time. Complete the backpropagation() method in NumpyRNN class in lxmls.deep_learning_numpy_models.rnn.py and compare it with lxmls.deep_learning_numpy_models.mlp.py.*

To work with RNNs we will use the Part-of-speech data-set seen in the sequence models day.

```
# Load Part-of-Speech data
from lxmls.readers.pos_corpus import PostagCorpusData
data = PostagCorpusData()
```

Algorithm 14 Forward pass of a Recurrent Neural Network (RNN)

1: **input:** Initial parameters for an RNN Input $\Theta = \{\mathbf{W}^x \in \mathbb{R}^{H \times I}, \mathbf{W}^h \in \mathbb{R}^{H \times H}, \mathbf{W}^y \in \mathbb{R}^{K \times H}\}$ input, recurrent and output transformations respectively.
2: **input:** Input data matrix $\mathbf{x} \in \mathbb{R}^{I \times T}$ of size T. Initial recurrent variable $\mathbf{h}_0$.
3: **for** $t = 1$ **to** $T - 1$ **do**
4: Apply linear transformation combining input and recurrent signals

$$z_{jt}^h = \sum_{i=1}^{I} W_{ji}^x x_{it} + \sum_{j'=1}^{J} W_{jj'}^h h_{j't-1}$$

5: Apply non-linear transformation e.g. sigmoid (hereby denoted $\sigma()$)

$$h_{jt} = \sigma(z_{jt}^h) = \frac{1}{1 + \exp(-z_{jt}^h)}$$

6: **end for**
7: Apply final linear transformation to each of the recurrent variables $\mathbf{h}_1 \cdots \mathbf{h}_T$

$$z_{kt}^y = \sum_{j=1}^{J} W_{kj}^y h_{jt}$$

8: Apply final non-linear transformation e.g. softmax

$$p(y_t = k | \mathbf{x}_{t:}) = \frac{\exp(z_{kt}^y)}{\sum_{k'=1}^{K} \exp(z_{k't}^y)}$$

Load and configure the NumpyRNN. Remember to use reload if you want to modify the code inside the rnns module

```
# Instantiate RNN
from lxmls.deep_learning.numpy_models.rnn import NumpyRNN
model = NumpyRNN(
    input_size=data.input_size,
    embedding_size=50,
    hidden_size=20,
    output_size=data.output_size,
    learning_rate=0.1
)
```

As in the case of the feed-forward networks you can use the following setup to test step by step the implementation of the gradients. First compute the cost variation for the variation of a single weight

```
from lxmls.deep_learning.rnn import get_rnn_parameter_handlers, get_rnn_loss_range

# Get functions to get and set values of a particular weight of the model
get_parameter, set_parameter = get_rnn_parameter_handlers(
    layer_index=-1,
    row=0,
    column=0
)

# Get batch of data
batch = data.batches('train', batch_size=1)[0]

# Get loss and weight value
current_loss = model.cross_entropy_loss(batch['input'], batch['output'])
current_weight = get_parameter(model.parameters)

# Get range of values of the weight and loss around current parameters values
weight_range, loss_range = get_rnn_loss_range(model, get_parameter, set_parameter, batch)
```

then conmpute the desired gradient from your implementation

```
# Get the gradient value for that weight
gradients = model.backpropagation(batch['input'], batch['output'])
current_gradient = get_parameter(gradients)
```

and finally call matlplotlib to plot the loss variation versus the gradient

```
%matplotlib inline
import matplotlib.pyplot as plt
# Plot empirical
plt.plot(weight_range, loss_range)
plt.plot(current_weight, current_loss, 'xr')
plt.ylabel('loss value')
plt.xlabel('weight value')
# Plot real
h = plt.plot(
    weight_range,
    current_gradient*(weight_range - current_weight) + current_loss,
    'r--'
)
```

After you have completed the gradients you can run the model in the POS task

```
import numpy as np
import time

# Hyper-parameters
num_epochs = 20

# Get batch iterators for train and test
train_batches = data.batches('train', batch_size=1)
dev_set = data.batches('dev', batch_size=1)
test_set = data.batches('test', batch_size=1)

# Epoch loop
start = time.time()
for epoch in range(num_epochs):

    # Batch loop
    for batch in train_batches:
        model.update(input=batch['input'], output=batch['output'])

    # Evaluation dev
    is_hit = []
    for batch in dev_set:
        is_hit.extend(model.predict(input=batch['input']) == batch['output'])
    accuracy = 100*np.mean(is_hit)
    print("Epoch %d: dev accuracy %2.2f %%" % (epoch+1, accuracy))

print("Training took %2.2f seconds per epoch" % ((time.time() - start)/num_epochs))

# Evaluation test
is_hit = []
for batch in test_set:
    is_hit.extend(model.predict(input=batch['input']) == batch['output'])
accuracy = 100*np.mean(is_hit)

# Inform user
print("Test accuracy %2.2f %%" % accuracy)
```

5.3 Implementing your own RNN in Pytorch

One of the big advantages of toolkits like Pytorch or Dynet is that creating computation graphs that dynamically change size is very simple. In many other tookits it is directly not possible to use a Python for loop with a variable length to define a computation graph. Again, as in other toolkits we will only need to create the forward pass of the RNN and the gradients will be computed automatically for us.

Exercise 5.2 *As we did with the feed-forward network, we will now implement a Recurrent Neural Network (RNN) in Pytorch. For this complete the log_forward() method in lxmls/deep_learning/pytorch_models/rnn.py.*
Load the RNN model in numpy and Python for comparison

```
# Numpy version
from lxmls.deep_learning.numpy_models.rnn import NumpyRNN
numpy_model = NumpyRNN(
    input_size=data.input_size,
    embedding_size=50,
    hidden_size=20,
    output_size=data.output_size,
    learning_rate=0.1
)

# Pytorch version
from lxmls.deep_learning.pytorch_models.rnn import PytorchRNN
model = PytorchRNN(
    input_size=data.input_size,
    embedding_size=embedding_size,
    hidden_size=hidden_size,
    output_size=data.output_size,
    learning_rate=learning_rate
)
```

To debug your code you can compare the numpy and Pytorch gradients using

```
# Get gradients for both models
batch = data.batches('train', batch_size=1)[0]
gradient_numpy = numpy_model.backpropagation(batch['input'], batch['output'])
gradient = model.backpropagation(batch['input'], batch['output'])
```

and then plotting them with matplotlib

```
%matplotlib inline
import matplotlib.pyplot as plt
# Gradient for  word embeddings in the example
plt.subplot(2,2,1)
plt.imshow(gradient_numpy[0][batch['input'], :], aspect='auto', interpolation='nearest')
plt.colorbar()
plt.subplot(2,2,2)
plt.imshow(gradient[0].numpy()[batch['input'], :], aspect='auto', interpolation='nearest'
    )
plt.colorbar()
# Gradient for  word embeddings in the example
plt.subplot(2,2,3)
plt.imshow(gradient_numpy[1], aspect='auto', interpolation='nearest')
plt.colorbar()
plt.subplot(2,2,4)
plt.imshow(gradient[1].numpy(), aspect='auto', interpolation='nearest')
plt.colorbar()
plt.show()
```

Once you are confident that your implementation is working correctly you can run it on the POS task using the Pytorch code from the Exercise 5.1.

Day 6

Reinforcement Learning

Today's class will introduce reinforcement learning. The previous classes mainly focused on machine learning in a full supervision scenario, i.e., the learning signal consisted of gold standard structures, and the learning goal was to optimize the system parameters so as to maximize the likelihood of the gold standard outputs. Reinforcement Learning (RL) assumes an *interactive learning* scenario where a system learns by trial and error, using the consequences of its own actions to learn to maximize the expected reward from the environment. The learning scenario can also be seen as learning under *weak supervision* in the sense that no gold standard examples are available, only rewards for system predictions.

6.1 Today's assignment

Your objective today should be to understand fully the concept of RL, including the standard formalization of the environment as a Markov Decision Process (MDP), and of the system as a stochastic policy. You will learn about algorithms that evaluate the expected reward for a given policy (policy evaluation/prediction) and algorithms that find the optimal policy parameters (policy optimization/control).

6.2 Markov Decision Processes

In order to define RL rigorously lets define an agent, for example a robot, that is surrounded by an environment, for example a room full of objects. The agent interacts with the environment in the following way: At each of a sequence of time steps $t = 0, 1, 2, \ldots$,

- the agent receives a **state** representation S_t,
- on which basis it selects an **action** A_t,
- and as a consequence, it receives a **reward** R_{t+1},
- after which it finds itself in a new state S_{t+1}.

The Goal of RL is then to teach the agent to maximize the total amount of reward it receives in such interactions in the long run. The standard formalization of the environment is given by a **Markov Decision Process (MDP)**, defined as a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R} \rangle$ where

- $\mathcal{S}$ is a set of states,
- $\mathcal{A}$ is a finite set of actions,
- $\mathcal{P}$ is a state transition probability matrix s.t. $\mathcal{P}^a_{ss'} = P[S_{t+1} = s' | S_t = s, A_t = a]$,
- $\mathcal{R}$ is a reward function s.t. $\mathcal{R}^a_s = \mathbb{E}[R_{t+1} | S_t = s, A_t = a]$.

6.3 Policies and Rewards

The system is formalized as a **stochastic policy**, which is defined as a distribution over actions given states s.t. $\pi(a|s) = P[A_t = a|S_t = s]$. The two central tasks in RL consist of policy evaluation (a.k.a. prediction), i.e., evaluation of the expected reward for a given policy; and policy optimization (a.k.a. learning/control), i.e., finding the optimal policy under the expected reward criterion. The reward criterion is formalized using the concepts of *return* and *value functions*:
The **total discounted return** from time-step t for discount $\gamma \in [0, 1]$ is

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}. \tag{6.1}$$

The **action-value function** $q_\pi(s, a)$ on an MDP is the expected return starting from state s, taking action a, and following policy π s.t.

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a]. \tag{6.2}$$

The **state-value function** $v_\pi(s)$ of an MDP is the expected return starting from state s and following policy π s.t.

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_{a\sim\pi}[q_\pi(s, a)]. \tag{6.3}$$

6.4 Dynamic Programming Solutions

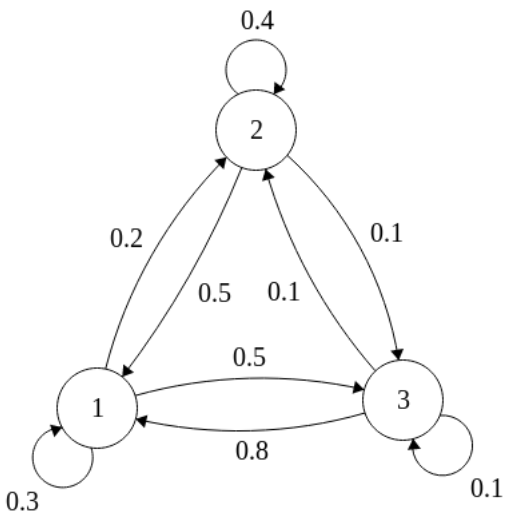

Figure 6.1: Markov Decision Process (MDP) with three states $\mathcal{S} = \{s_1, s_2, s_3\}$ and three possible actions $\mathcal{A} = \{a_1, a_2, a_3\}$, moving to state s_1, moving to state s_2 and moving to state s_3.

Consider the (simplified) MDP represented in Figure 6.1: it consists of three states (1, 2, and 3) that are connected by three actions (move to state 1, 2, or 3) that determine state transitions (our MDP is thus actually a Markov Reward Process (MRP) with deterministic state transitions). The rewards $\mathcal{R}_s^a$ are 1.5, -1.8333... and 19.8333... for a transition into state 1, 2, and 3, respectively. So, in our example $\mathcal{R}_s^a$ depends only on action (i.e. destination state) but it doesn't depend on source state.

Such reward values are chosen to get nice expected values of the next reward given policy π:

$$\mathcal{R}^\pi(s) = \sum_{a\in\mathcal{A}} \pi(a|s)\mathcal{R}_s^a$$

For the policy we will use below expected values of the next reward $\mathcal{R}^\pi(s)$ are 10.0, 2.0, and 3.0 for state 1, 2, and 3, respectively.

Given full information about states and rewards, we want to evaluate a given policy. A central tool is the **Bellman Expectation Equation**, which tells us how to decompose the state-value function into immediate reward plus discounted value of successor state s.t.

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s] \\ &= \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s') \right). \end{aligned}$$

A straightforward solution is to solve the linear equation directly:

$$\boldsymbol{v}_\pi = (\boldsymbol{I} - \gamma \boldsymbol{\mathcal{P}}^\pi)^{-1} \boldsymbol{\mathcal{R}}^\pi.$$

Because of the complexity of matrix inversion, this solution is applicable only to small MDPs. For larger MDPs, we can use iterative policy evaluation on the Bellman Expectation Equation.

Exercise 6.1 *Implement policy evaluation by dynamic programming and linear programming. Check that you obtain the same value.*

```
import numpy as np

policy=np.array([[0.3, 0.2, 0.5], [0.5, 0.4, 0.1], [0.8, 0.1, 0.1]])
# 'raw_rewards' variable contains rewards obtained after transition to each state
# In our example it doesn't depend on source state
raw_rewards = np.array([1.5, -1.833333333, 19.833333333])
# 'rewards' variable contains expected values of the next reward for each state
rewards = np.matmul(policy, raw_rewards)

state_value_function=np.array([0 for i in range(3)])

for i in range(20):
    print(state_value_function)
    state_value_function=# TODO: Implement the Policy Evaluation Update with a Discount
        Rate of 0.1
print(state_value_function)

solution=# TODO: Implement the linear programming solution
print(solution)
```

6.5 Monte Carlo and Q-Learning Solutions

While Dynamic Programming techniques allow to iteratively compute policy evaluation and optimization problems, a disadvantage of these techniques is the fact that we need to know a full MDP model and touch all transitions and rewards in learning. Monte Carlo techniques circumvent this problem by learning from episodes that are sampled under a given policy. An algorithm for Monte Carlo Policy Evaluation looks as follows:

- Sample episodes $S_0, A_0, R_1, \ldots, R_T \sim \pi$.
- For each sampled episode:
  - Increment state counter $N(s) \leftarrow N(s) + 1$.
  - Increment total return $S(s) \leftarrow S(s) + G_t$.
- Estimate mean return $V(s) = S(s)/N(s)$.

A more sophisticated technique is the so-called Q-Learning algorithm. In contrast to the simple Monte Carlo Policy Evaluation algorithm given above, it updates not towards the actual return G_t, but towards an estimate $(R_{t+1} + \gamma \max_{a'} Q(S_{t+1}, a')$ (the so-called Temporal Difference (TD) target). It thus combines Dynamic Programming (in form of the Bellman Optimality Equation) and Monte Carlo (by sampling episodes).

Algorithm 15 Q-Learning

for sampled episodes **do**
  Initialize S_t
  for steps t **do**
    Sample A_t, observe R_{t+1}, S_{t+1}.
    $Q(S_t, A_t) \leftarrow (1-\alpha)Q(S_t, A_t) + \alpha(R_{t+1} + \gamma \max_{a'} Q(S_{t+1}, a'))$
    $S_t \leftarrow S_{t+1}$.
  end for
end for

Exercise 6.2 *Implement Monte Carlo policy evaluation. See how similar results can be obtained by using sampling*

```
import numpy as np

policy=np.array([[0.3, 0.2, 0.5], [0.5, 0.4, 0.1], [0.8, 0.1, 0.1]])
rewards=np.array([10., 2., 3.])

import random
from collections import defaultdict
reward_counter=np.array([0., 0., 0.])
visit_counter=np.array([0., 0., 0.])

def gt(rewardlist, gamma=0.1):
    '''
    Function to calculate the total discounted reward
    >>> gt([10, 2, 3], gamma=0.1)
    10.23
    '''
    #TODO: Implement the total discounted reward
    return 0

for i in range(400):
    start_state=random.randint(0, 2)
    next_state=start_state
    rewardlist=[]
    occurence=defaultdict(list)
    for i in range(250):
        rewardlist.append(rewards[next_state])
        occurence[next_state].append(len(rewardlist)-1)
        action=np.random.choice(np.arange(0, 3), p=policy[next_state])
        next_state=action

    for state in occurence:
        for value in occurence[state]:
            rew=gt(rewardlist[value:])
            reward_counter[state]+=rew
            visit_counter[state]+=1
            #break #if break: return following only the first visit

print(reward_counter/visit_counter)
```

Implement Q-Learning policiy optimization in a simple exercise. This samples a state-action pair randomly, so that all the state-action pairs can be seen.

```
q_table=np.zeros((3, 3))
for i in range(1001):
    state=random.randint(0, 2)
    action=random.randint(0, 2)
    next_state=action
    reward=rewards[next_state]
    next_q=max(q_table[next_state])
    q_table[state, action]= #TODO: Implement the Q-Table update
    if i%100==0:
        print(q_table)
```

6.6 Policy Gradient Techniques

The Dynamic Programming and Monte Carlo techniques discussed above can be summarized under the header of *value-based* techniques since they focus on value functions. They are thus inherently indirect. Direct solutions to policy optimization that use gradient-based techniques to optimize parametric (stochastic) policies are called *policy gradient* techniques. Given an action-value function $q_{\pi_\theta}(S_t, A_t)$, the objective is to maximize the expected value

$$\mathbb{E}_{\pi_\theta}[q_{\pi_\theta}(S_t, A_t)].$$

The gradient of this objective is

$$\mathbb{E}_{\pi_\theta}[q_{\pi_\theta}(S_t, A_t)\nabla_\theta \log \pi_\theta(A_t|S_t)].$$

The well-known REINFORCE algorithm optimizes this objective by using stochastic gradient ascent updates and by replacing $q_{\pi_\theta}(S_t, A_t)$ by the actual return G_t.

Algorithm 16 REINFORCE

1: **for** sampled episodes $y = S_0, A_0, R_1, \ldots, R_T \sim \pi_\theta$ **do**
2: **for** time steps t **do**
3: Obtain reward G_t
4: Update $\theta \leftarrow \theta + \alpha(G_t \nabla_\theta \log \pi_\theta(A_t|S_t))$
5: **end for**
6: **end for**

Exercise 6.3 *Implement the score function gradient estimator in lxmls/reinforcement_learning/score_function_estimator.py. Check it is correct by calling the train() function.*

```
% matplotlib inline
from lxmls.reinforcement_learning.score_function_estimator import train
train()
```

A last exercise is to apply REINFORCE to the cart pole task: A pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The system is controlled by applying a force of +1 or -1 to the cart. The pendulum starts upright, and the goal is to prevent it from falling over. A reward of +1 is provided for every timestep that the pole remains upright. The episode ends when the pole is more than 15 degrees from vertical, or the cart moves more than 2.4 units from the center. See `https://gym.openai.com/envs/CartPole-v1/`.

Exercise 6.4 *Implement policy gradient for the cartpole task by coding the forward pass of Model() in lxmls/reinforcement_learning/policy_gradient.py. Check it is correct by calling the train() function.*

```
from lxmls.reinforcement_learning.policy_gradient import train
train()
```

Exercise 6.5 *As a last exercise, apply what you have learned to the RNN model seen in previous days. Implement REINFORCE to replace the maximum likelihood loss used on the RNN day. For this you can modify the PolicyRNN class in lxmls/deep_learning/pytorch_models/rnn.py*

```
# Load Part-of-Speech data
from lxmls.readers.pos_corpus import PostagCorpusData
data = PostagCorpusData()

# Get batch iterators for train and test
batch_size = 1
train_batches = data.batches('train', batch_size=batch_size)
dev_set = data.batches('dev', batch_size=batch_size)
test_set = data.batches('test', batch_size=batch_size)
```

```
reinforce = True

# Load version of the RNN
from lxmls.deep_learning.pytorch_models.rnn import PolicyRNN
model = PolicyRNN(
    input_size=data.input_size,
    embedding_size=50,
    hidden_size=100,
    output_size=data.output_size,
    learning_rate=0.05,
    gamma=0.8,
    RL=reinforce,
    maxL=data.maxL
)
```

After finishing the implementation, the following code shoul run

```
import numpy as np
import time

# Run training
num_epochs = 15

batch_size = 1
# Get batch iterators for train and test
train_batches = data.sample('train', batch_size=batch_size)
dev_set = data.sample('dev', batch_size=batch_size)
test_set = data.sample('test', batch_size=batch_size)

# Epoch loop
start = time.time()
for epoch in range(num_epochs):
    # Batch loop
    for batch in train_batches:
        model.update(input=batch['input'], output=batch['output'])
    # Evaluation dev
    is_hit = []
    for batch in dev_set:
        loss = model.predict_loss(batch['input'], batch['output'])
        is_hit.extend(loss)
    accuracy = 100 * np.mean(is_hit)
    # Inform user
    print("Epoch %d: dev accuracy %2.2f %%" % (epoch + 1, accuracy))

print("Training took %2.2f seconds per epoch" % ((time.time() - start) / num_epochs))
# Evaluation test
is_hit = []
for batch in test_set:
```

```
    is_hit.extend(model.predict_loss(batch['input'],batch['output']))
accuracy = 100 * np.mean(is_hit)

# Inform user
print("Test accuracy %2.2f %%" % accuracy)
```

Bibliography

Berg-Kirkpatrick, T., Bouchard-Côté, A., DeNero, J., and Klein, D. (2010). Painless unsupervised learning with features. In *Proc. NAACL*.

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

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

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

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

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

Brown, P. F., Pietra, S. A. D., Pietra, V. J. D., and Mercer, R. L. (1993). The mathematics of statistical machine translation: Parameter estimation. *Computational linguistics*, 19(2):263–311.

Charniak, E. and Elsner, M. (2009). EM works for pronoun anaphora resolution. In *Proceedings of the 12th Conference of the European Chapter of the Association for Computational Linguistics*, pages 148–156. Association for Computational Linguistics.

Clark, A. (2003). Combining distributional and morphological information for part of speech induction. In *Proc. EACL*.

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

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

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

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

Das, D. and Petrov, S. (2011). Unsupervised part-of-speech tagging with bilingual graph-based projections. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 600–609, Portland, Oregon, USA. Association for Computational Linguistics.

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

Graça, J. (2010). *Posterior Regularization Framework: Learning Tractable Models with Intractable Constraints*. PhD thesis, Universidade Técnica de Lisboa, Instituto Superior Técnico.

Graça, J., Ganchev, K., Pereira, F., and Taskar, B. (2009). Parameter vs. posterior sparisty in latent variable models. In *Proc. NIPS*.

Haghighi, A. and Klein, D. (2006). Prototype-driven learning for sequence models. In *Proc. HTL-NAACL*. ACL.

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

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

Johnson, M. (2007). Why doesn't EM find good HMM POS-taggers. In *In Proc. EMNLP-CoNLL*.

Klein, D. and Manning, C. (2004). Corpus-based induction of syntactic structure: Models of dependency and constituency. In *Proc. ACL*.

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

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

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

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

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

Meilă, M. (2007). Comparing clusterings—an information based distance. *J. Multivar. Anal.*, 98(5):873–895.

Merialdo, B. (1994). Tagging English text with a probabilistic model. *Computational linguistics*, 20(2):155–171.

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

Neal, R. M. and Hinton, G. E. (1998). A view of the em algorithm that justifies incremental, sparse, and other variants. In *Learning in graphical models*, pages 355–368. Springer.

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

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

Reichart, R. and Rappoport, A. (2009). The NVI clustering evaluation measure. In *Proc. CONLL*.

Rosenberg, A. and Hirschberg, J. (2007). V-measure: A conditional entropy-based external cluster evaluation measure. In *EMNLP-CoNLL*, pages 410–420.

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

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

Schütze, H. (1995). Distributional part-of-speech tagging. In *Proceedings of the seventh conference on European chapter of the Association for Computational Linguistics*, pages 141–148. Morgan Kaufmann Publishers Inc.

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

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

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

Smith, N. A. and Eisner, J. (2006). Annealing structural bias in multilingual weighted grammar induction. In *ACL-44: Proceedings of the 21st International Conference on Computational Linguistics and the 44th annual meeting of the Association for Computational Linguistics*, pages 569–576, Morristown, NJ, USA. Association for Computational Linguistics.

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