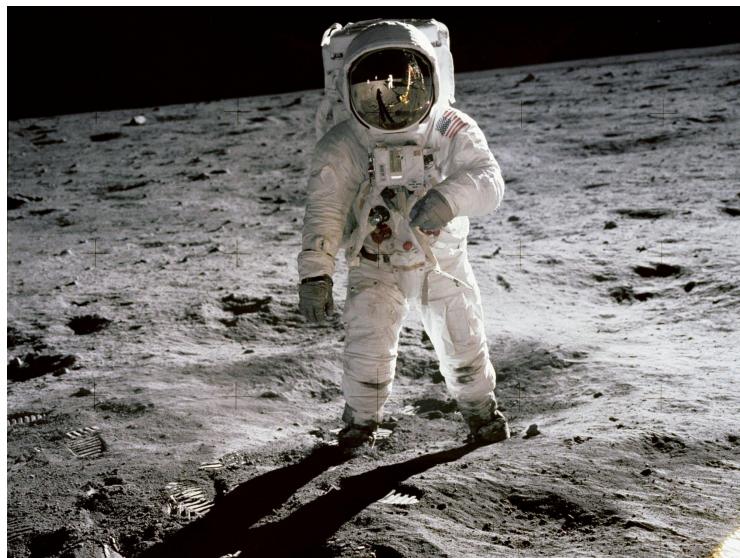


# Labs for Foundations of Applied Mathematics

Volume III  
Modeling with Uncertainty and Data





# List of Contributors

E. Evans

*Brigham Young University*

R. Evans

*Brigham Young University*

J. Grout

*Drake University*

J. Humpherys

*Brigham Young University*

T. Jarvis

*Brigham Young University*

J. Whitehead

*Brigham Young University*

J. Adams

*Brigham Young University*

J. Bejarano

*Brigham Young University*

Z. Boyd

*Brigham Young University*

M. Brown

*Brigham Young University*

A. Carr

*Brigham Young University*

T. Christensen

*Brigham Young University*

M. Cook

*Brigham Young University*

R. Dorff

*Brigham Young University*

B. Ehlert

*Brigham Young University*

M. Fabiano

*Brigham Young University*

A. Frandsen

*Brigham Young University*

K. Finlinson

*Brigham Young University*

J. Fisher

*Brigham Young University*

R. Fuhriman

*Brigham Young University*

S. Giddens

*Brigham Young University*

C. Gigena

*Brigham Young University*

M. Graham

*Brigham Young University*

F. Glines

*Brigham Young University*

M. Goodwin

*Brigham Young University*

R. Grout

*Brigham Young University*

D. Grundvig

*Brigham Young University*

J. Hendricks

*Brigham Young University*

A. Henriksen

*Brigham Young University*

I. Henriksen

*Brigham Young University*

C. Hettinger

*Brigham Young University*

S. Horst

*Brigham Young University*

K. Jacobson

*Brigham Young University*

J. Leete

*Brigham Young University*

J. Lytle	C. Robertson
<i>Brigham Young University</i>	<i>Brigham Young University</i>
R. McMurray	M. Russell
<i>Brigham Young University</i>	<i>Brigham Young University</i>
S. McQuarrie	R. Sandberg
<i>Brigham Young University</i>	<i>Brigham Young University</i>
D. Miller	M. Stauffer
<i>Brigham Young University</i>	<i>Brigham Young University</i>
J. Morrise	J. Stewart
<i>Brigham Young University</i>	<i>Brigham Young University</i>
M. Morrise	S. Suggs
<i>Brigham Young University</i>	<i>Brigham Young University</i>
A. Morrow	A. Tate
<i>Brigham Young University</i>	<i>Brigham Young University</i>
R. Murray	T. Thompson
<i>Brigham Young University</i>	<i>Brigham Young University</i>
J. Nelson	M. Victors
<i>Brigham Young University</i>	<i>Brigham Young University</i>
E. Parkinson	J. Webb
<i>Brigham Young University</i>	<i>Brigham Young University</i>
M. Probst	R. Webb
<i>Brigham Young University</i>	<i>Brigham Young University</i>
M. Proudfoot	J. West
<i>Brigham Young University</i>	<i>Brigham Young University</i>
D. Reber	A. Zaitzeff
<i>Brigham Young University</i>	<i>Brigham Young University</i>

# Preface

This lab manual is designed to accompany the textbook *Foundations of Applied Mathematics* by Humpherys and Jarvis.

©This work is licensed under the Creative Commons Attribution 3.0 United States License. You may copy, distribute, and display this copyrighted work only if you give credit to Dr. J. Humpherys. All derivative works must include an attribution to Dr. J. Humpherys as the owner of this work as well as the web address to

<https://github.com/Foundations-of-Applied-Mathematics/Labs>  
as the original source of this work.

To view a copy of the Creative Commons Attribution 3.0 License, visit

<http://creativecommons.org/licenses/by/3.0/us/>  
or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105,  
USA.





# Contents

Preface	iii
<b>I Data Science Technologies</b>	<b>1</b>
1 Introduction to the Unix Shell	3
2 More on the Unix Shell	15
3 Basic Regular Expressions	27
4 SQL	41
5 Advanced SQL	53
6 Pandas I: Introduction to Pandas	63
7 Pandas II: Plotting with Pandas	79
8 Pandas III: Grouping and Presenting Data	93
9 Pandas IV: Time Series	101
10 Web Technologies 1: Internet Protocols	115
11 Web Technologies 2: Data Serialization	123
12 BeautifulSoup	135
13 Advanced BeautifulSoup	147
14 MongoDB	155
15 Parallel Computing with ipyparallel	161
16 Parallel Programming with MPI	173

<b>II Machine Learning Algorithms</b>	<b>181</b>
<b>17 Kalman Filter</b>	<b>183</b>
<b>18 ARMA Models</b>	<b>193</b>
<b>19 Discrete Hidden Markov Models</b>	<b>201</b>
<b>20 Gaussian Mixture Models</b>	<b>213</b>
<b>21 Speech Recognition using CDHMMs</b>	<b>219</b>
<b>22 Gibbs Sampling and LDA</b>	<b>225</b>
<b>23 Metropolis Algorithm</b>	<b>233</b>
<b>24 PCA and LSI</b>	<b>243</b>
<b>25 Naive Bayes</b>	<b>255</b>
<b>26 Logistic Regression</b>	<b>263</b>
<b>27 Classification Trees</b>	<b>269</b>
<b>28 Random Forests</b>	<b>273</b>
<b>29 K-Nearest Neighbors and Support Vector Machines</b>	<b>275</b>
<b>30 Crime Mapping</b>	<b>279</b>
<b>31 Image Recognition Tasks</b>	<b>285</b>
<b>32 K-Means Clustering</b>	<b>289</b>
<b>33 Bayesian Search</b>	<b>299</b>

Part I

## Data Science Technologies



# 1

# Unix Shell 1: Introduction

**Lab Objective:** *Explore the basics of the Unix Shell. Understand how to navigate and manipulate file directories. Introduce the Vim text editor for easy writing and editing of text or other similar documents.*

Unix was first developed by AT&T Bell Labs in the 1970s. In the 1990s, Unix became the foundation of Linux and MacOSX. The Unix shell is an interface for executing commands to the operating system. The majority of servers are Linux based, so having a knowledge of Unix shell commands allows us to interact with these servers.

As you get into Unix, you will find it is easy to learn but difficult to master. We will build a foundation of simple file system management and a basic introduction to the Vim text editor. We will address some of the basics in detail and also include lists of commands that interested learners are encouraged to research further.

## NOTE

Windows is not built off of Unix, but it does come with a command line tool. We will not cover the equivalent commands in Windows command line, but you could download a Unix-based shell such as Git Bash or Cygwin to complete this lab (you will still lose out on certain commands).

## File System

### ACHTUNG!

In this lab you will work with files on your computer. Be careful as you go through each problem and as you experiment on your own. Be sure you are in the right directories and subfolders before you start creating and deleting files; some actions are irreversible.

## Navigation

Typically you have probably navigated your computer by clicking on icons to open directories and programs. In the terminal, instead of point and click we use typed commands to move from directory to directory.

Begin by opening the Terminal. The text you see in the upper left of the Terminal is called the *prompt*. As you navigate through the file system you will want to know *where* you are so that you know you aren't creating or deleting files in the wrong locations.

To see what directory you are currently working in, type `pwd` into the prompt. This command stands for **p**rint **w**orking **d**irectory, and as the name suggests it prints out the string of your current location.

Once you know where you are, you'll want to know where you can move. The `ls`, or **l**ist **s**egments, command will list all the files and directories in your current folder location. Try typing it in.

When you know what's around you, you'll want to navigate directories. The `cd`, or **c**hange **d**irectory, command allows you to move through directories. To change to a new directory, type the `cd` command followed by the name of the directory to which you want to move (if you `cd` into a file, you will get an error). You can move up one directory by typing `cd ...`

Two important directories are the root directory and the home directory. You can navigate to the home directory by typing `cd ~` or just `cd`. You can navigate to root by typing `cd /`.

**Problem 1.** Using these commands, navigate to the `Shell1/` directory provided with this lab. We will use this directory for the remainder of the lab. Use the `ls` command to list the contents of this directory. NOTE: You will find a directory within this directory called `Test/` that is available for you to experiment with the concepts and commands found in this lab. The other files and directories are necessary for the exercises we will be doing, so take care not to modify them.

## Getting Help

As you go through this lab, you will come across many commands with functionality beyond what is taught here. The Terminal has two nice commands to help you with these commands. The first is `man <command>`, which opens the manual page for the command following `man`. Try typing in `man ls`; you will see a list of the name and description of the `ls` command, among other things. If you forget how to use a command the manual page is the first place you should check to remember.

The `apropos <keyword>` command will list all Unix commands that have `<keyword>` contained somewhere in their manual page names and descriptions. For example, if you forget how to copy files, you can type in `apropos copy` and you'll get a list of all commands that have `copy` in their description.

Flags	Description
-a	Do not ignore hidden files and folders
-l	List files and folders in long format
-r	Reverse order while sorting
-R	Print files and subdirectories recursively
-s	Print item name and size
-S	Sort by size
-t	Sort output by date modified

Table 1.1: Common flags of the `ls` command.

## Flags

When you typed in `man ls` up above, you may have noticed several options listed in the description, such as `-a`, `-A`, `--author`. These are called flags and change the functionality of commands. Most commands will have flags that change their behavior. Table 1.1 contains some of the most common flags for the `ls` command.

Multiple flags can be combined as one flag. For example, if we wanted to list all the files in a directory in long format sorted by date modified, we would use `ls -a -l -t` or `ls -alt`.

## Manipulating Files and Directories

In this section we will learn how to create, copy, move, and delete files and folders. Before you begin, `cd` into the `Test/` directory in `Shell1/`.

To create a text file, use `touch <filename>`. To create a new directory, use `mkdir <dir_name>`.

To copy a file into a directory, use `cp <filename> <dir_name>`. When making a copy of a directory, the command is similar but must use the `-r` flag. This flag stands for recursively copying files in subdirectories. If you try to copy a file without the `-r` the command will return an error.

Moving files and directories follows a similar format, except no `-r` flag is used when moving one directory into another. The command `mv <filename> <dir_name>` will move a file to a folder and `mv <dir1> <dir2>` will move the first directory into the second. If you want to rename a file, use `mv <file_old> <file_new>`; the same goes for directories.

When deleting files, use `rm <filename>`, or `rm -r <dir_name>` when deleting a directory. Again, the `-r` flag tells the Terminal to recursively remove all the files and subfolders within the targeted directory.

If you want to make sure your command is doing what you intend, the `-v` flag tells `rm`, `cp`, or `mkdir` to have the Terminal print strings of what it is doing. When your Terminal gets too cluttered, use `clear` to clean it up.

Below is an example of all these commands in action.

```
$ cd Test
$ touch data.txt          # create new empty file data.txt
$ mkdir New               # create directory New
$ ls                      # list items in test directory
New      data.txt
$ cp data.txt New/        # copy data.txt to New directory
$ cd New/
$ ls                      # list items in New directory
```

Commands	Description
<code>clear</code>	Clear the terminal screen
<code>cp file1 dir1</code>	Create a copy of <code>file1</code> and move it to <code>dir1/</code>
<code>cp file1 file2</code>	Create a copy of <code>file1</code> and name it <code>file2</code>
<code>cp -r dir1 dir2</code>	Create a copy of <code>dir1/</code> and all its contents into <code>dir2/</code>
<code>mkdir dir1</code>	Create a new directory named <code>dir1/</code>
<code>mkdir -p path/to/new/dir1</code>	Create <code>dir1/</code> and all intermediate directories
<code>mv file1 dir1</code>	Move <code>file1</code> to <code>dir1/</code>
<code>mv file1 file2</code>	Rename <code>file1</code> as <code>file2</code>
<code>rm file1</code>	Delete <code>file1</code> [-i, -v]
<code>rm -r dir1</code>	Delete <code>dir1/</code> and all items within <code>dir1/</code> [-i, -v]
<code>touch file1</code>	Create an empty file named <code>file1</code>

Table 1.2: The commands discussed in this section.

```

data.txt
$ mv data.txt new_data.txt          # rename data.txt new_data.txt
$ ls                                # list items in New directory
new_data.txt
$ cd ..                             # Return to test directory
$ rm -rv New/                         # Remove New directory and its contents
removed 'New/data.txt'
removed directory: 'New/'
$ clear                            # Clear terminal screen

```

Table 1.2 contains all the commands we have discussed so far. Notice the common flags are contained in square brackets; use `man` to see what these mean.

**Problem 2.** Inside the `Shell1/` directory, delete the `Audio/` folder along with all its contents. Create `Documents/`, `Photos/`, and `Python/` directories.

## Wildcards

As we are working in the file system, there will be times that we want to perform the same command to a group of similar files. For example, if you needed to move all text files within a directory to a new directory. Rather than copy each file one at a time, we can apply one command to several files using *wildcards*. We will use the \* and ? wildcards. The \* wildcard represents any string and the ? wildcard represents any single character. Though these wildcards can be used in almost every Unix command, they are particularly useful when dealing with files.

```

$ ls
File1.txt  File2.txt  File3.jpg  text_files
$ mv -v *.txt text_files/
File1.txt -> text_files/File1.txt
File2.txt -> text_files/File2.txt
$ ls
File3.jpg  text_files

```

Command	Description
<code>*.txt</code>	All files that end with <code>.txt</code> .
<code>image*</code>	All files that have <code>image</code> as the first 5 characters.
<code>*py*</code>	All files that contain <code>py</code> in the name.
<code>doc*.txt</code>	All files of the form <code>doc1.txt</code> , <code>doc2.txt</code> , <code>docA.txt</code> , etc.

Table 1.3: Common uses for wildcards.

Command	Description
<code>cat</code>	Print the contents of a file in its entirety
<code>more</code>	Print the contents of a file one page at a time
<code>less</code>	Like more, but you can navigate forward and backward
<code>head</code>	Print the first 10 lines of a file
<code>head -nK</code>	Print the first K lines of a file
<code>tail</code>	Print just the last 10 lines of a file
<code>tail -nK</code>	Print the last K lines of a file

Table 1.4: Commands for printing contents of a file

See Table 1.3 for examples of common wildcard usage.

**Problem 3.** Within the `Shell1/` directory, there are many files. We will organize these files into directories. Using wildcards, move all the `.jpg` files to the `Photos/` directory, all the `.txt` files to the `Documents/` directory, and all the `.py` files to the `Python/` directory. You will see a few other folders in the `Shell1/` directory. Do not move any of the files within these folders at this point.

## Displaying File Contents

When using the file system, you may be interested in checking file content to be sure you're looking at the right file. Several commands are made available for ease in reading file content.

The `cat` command, followed by the filename will display all the contents of a file on the screen. If you are dealing with a large file, you may only want to view a certain number of lines at a time. Use `less <filename>` to restrict the number of lines that show up at a time. Use the arrow keys to navigate up and down. Press `q` to exit.

For other similar commands, look at table 1.4.

## Searching the File System

There are two commands we use for searching through our directories. The `find` command is used to find files or directories in a directory hierarchy. The `grep` command is used to find lines matching a string. More specifically, we can use `grep` to find words inside files. We will provide a basic template in Table 1.5 for using these two commands and leave it to you to explore the uses of the other flags. The `man` command can help you learn about them.

Command	Description
<code>find dir1 -type f -name "word"</code>	Find all files in <code>dir1/</code> (and its subdirectories) called <code>word</code> ( <code>-type f</code> is for files; <code>-type d</code> is for directories)
<code>grep "word" filename</code>	Find all occurrences of <code>word</code> within <code>filename</code>
<code>grep -nr "word" dir1</code>	Find all occurrences of <code>word</code> within the files inside <code>dir1/</code> ( <code>-n</code> lists the line number; <code>-r</code> performs a recursive search)

Table 1.5: Commands using `find` and `grep`.

**Problem 4.** In addition to the `.jpg` files you have already moved into the `Photot/` folder, there are a few other `.jpg` files in a few other folders within the `Shell1/` directory. Find where these files are using the `find` command and move them to the `Photos/` folder.

## Pipes and Redirects

Terminal commands can be combined using *pipes*. When combined, or *piped*, the output of one command is passed to the another. Two commands are piped together using the `|` operator. To demonstrate pipes we will first introduce commands that allow us to view the contents of a file in Table 1.4.

In the first example below, the `cat` command output is piped to `wc -l`. The `wc` command stands for `word count`. This command can be used to count words or lines. The `-l` flag tells the `wc` command to count lines. Therefore, this first example counts the number of lines in `assignments.txt`. In the second example below, the command lists the files in the current directory sorted by size in descending order. For details on what the flags in this command do, consult `man sort`.

```
$ cd Shell1/Files/Feb
$ cat assignments.txt | wc -l
9

$ ls -s | sort -nr
12 project3.py
12 project2.py
12 assignments.txt
 4 pics
total 40
```

In the previous example, we pipe the contents of `assignments.txt` to `wc -l` using `cat`. When working with files specifically, you can also use *redirects*. The `<` operator gives a file to a Terminal command. The same output from the first example above can be achieved by running the following command:

```
$ wc -l < assignments.txt
9
```

If you are wanting to save the resulting output of a command to a file, use `>` or `>>`. The `>` operator will overwrite anything that may exist in the output file whereas `>>` will append the output to the end of the output file. For example, if we want to append the number of lines in `assignments.txt` to `word_count.txt`, we would run the following command:

```
$ wc -l < assignements.txt >> word_count.txt
```

Since `grep` is used to print lines matching a pattern, it is also very useful to use in conjunction with piping. For example, `ls -l | grep root` prints all files associated with the root user.

**Problem 5.** The words.txt file in the Documents/ directory contains a list of words that are not in alphabetical order. Write the number of words in words.txt and an alphabetically sorted list of words to sortedwords.txt using pipes and redirects. Save this file in the Documents/ directory. Try to accomplish this with a total of two commands or fewer.

## Archiving and Compression

In file management, the terms archiving and compressing are commonly used interchangeably. However, these are quite different. To archive is to combine a certain number of files into one file. The resulting file will be the same size as the group of files that were archived. To compress is to take a file or group of files and shrink the file size as much as possible. The resulting compressed file will need to be extracted before being used.

The ZIP file format is the most popular for archiving and compressing files. If the `zip` Unix command is not installed on your system, you can download it by running `sudo apt-get install zip`. Note that you will need to have administrative rights to download this package. To unzip a file, use `unzip`.

```
$ cd Shell1/Documents  
$ zip zipfile.zip doc?.txt  
adding: doc1.txt (deflated 87%)  
adding: doc2.txt (deflated 90%)  
adding: doc3.txt (deflated 85%)  
adding: doc4.txt (deflated 97%)  
  
# use -l to view contents of zip file  
$ unzip -l zipfile.zip  
Archive: zipfile.zip  
Length      Date    Time     Name  
-----  
      5234  2015-08-26 21:21  doc1.txt  
      7213  2015-08-26 21:21  doc2.txt  
      3634  2015-08-26 21:21  doc3.txt  
      4516  2015-08-26 21:21  doc4.txt  
-----  
     16081                               3 files
```

```
inflating: doc1.txt
inflating: doc2.txt
inflating: doc3.txt
inflating: doc4.txt
```

While the zip file format is more popular on the Windows platform, the `tar` utility is more common in the Unix environment. The following commands use `tar` to archive the files and `gzip` to compress the archive.

Notice that all the commands below have the `-z`, `-v`, and `-f` flags. The `-z` flag calls for the `gzip` compression tool, the `-v` flag calls for a verbose output, and `-f` indicates the next parameter will be the name of the archive file.

```
$ ls
doc1.txt    doc2.txt    doc3.txt    doc4.txt

# use -c to create a new archive
$ tar -zcvf docs.tar.gz doc?.txt
doc1.txt
doc2.txt
doc3.txt
doc4.txt

$ ls
docs.tar.gz

# use -t to view contents
$ tar -ztvf <archive>
-rw-rw-r-- username/groupname 5119 2015-08-26 16:50 doc1.txt
-rw-rw-r-- username/groupname 7253 2015-08-26 16:50 doc2.txt
-rw-rw-r-- username/groupname 3524 2015-08-26 16:50 doc3.txt
-rw-rw-r-- username/groupname 4516 2015-08-26 16:50 doc4.txt

# use -x to extract
$ tar -zxvf <archive>
doc1.txt
doc2.txt
doc3.txt
doc4.txt
```

**Problem 6.** Archive and compress the files in the `Photos/` directory using `tar` and `gzip`. Name the archive `pics.tar.gz` and save it inside the `Photos/` directory. Use `ls -l` to see how much the files were compressed in the process.

## Vim: A Terminal Text Editor

Today many have become accustomed to having GUIs (Graphic User Interfaces) for all their applications. Before modern text editors (i.e. Microsoft Word, Pages for Mac, Google Docs) there were terminal text editors. Vim is one of the most popular terminal text editors. While vim may be intimidating at first, as you become familiar with vim it may become one of your preferred text editors for writing code.

One of the major philosophies of vim is to be able to keep your fingers on the keyboard at all times. Thus, vim has many keyboard shortcuts that allow you to navigate the file and execute commands without relying on a mouse, toolbars, or arrow keys.

In this section, we will go over the basics of navigation and a few of the most common commands. We will also provide a list of commands that interested readers are encouraged to research.

It has been said that at no point does somebody finish learning Vim. You will find that you will constantly be able to add something new to your arsenal.

### Getting Started

Start Vim with the following command:

```
$ vim my_file.txt
```

When executing this command, if `my_file.txt` already exists, vim will open the file and we may begin editing the existing file. If `my_file.txt` does not exist, it will be created and we may begin editing the file.

You may notice if you start typing the characters may or may not appear on your screen. This is because vim has multiple modes. When vim starts, we are placed in *command mode*. We want to be in *insert mode* to begin entering text. To enter insert mode from command mode, hit the `i` key. You should see `-- INSERT --` at the bottom of your terminal window. In insert mode vim act like a typical word processor. Letters will appear in the document as you type them. If you ever need to leave insert mode and return to command mode, hit the `Esc` key.

### Saving/Quitting Vim

To save or quit the current document, first enter last line mode by pressing the `:` key. To just save, type `w` and hit enter. To save and quit, type `wq`. To quit without saving, run `q!`

**Problem 7.** Using vim, create a new file in the `Documents/` directory named `first_vim.txt`. Write least multiple lines to this file. Save and exit the file you have created.

### Navigation

We are accustomed to navigating GUI text editors using a mouse and arrow keys. In vim, we navigate using keyboard shortcuts while in command mode.

Command	Description
a	append text after cursor
A	Append text to end of line
o	Begin a new line below the cursor
O	Begin a new line above the cursor
s	Substitute characters under cursor

Table 1.6: Commands for entering insert mode

**Problem 8.** Become accustomed to navigating in command mode using the following keys:

Command	Description
k	up
j	down
h	left
l	right
w	beginning of next word
e	end of next word
b	beginning of previous word
0	(zero) beginning of line
\$	end of line
gg	beginning of file
#gg	go to line #
G	end of file

## Alternative Ways to Enter Insert Mode

Hitting the `i` key is not the only way to enter insert mode. Alternative methods are described in Table 1.6.

## Visual Mode

Visual mode allows you to select multiple characters. Among other things, we can use this to replace words with the `s` command, and we can select text to cut or copy.

**Problem 9.** Open the document you created in the previous problem. While in command mode, enter visual mode by pressing the `v` key. Using the navigation keys discussed earlier, move the cursor to select a few words. Copy this text using the `y` key (stands for `yank`). Return to command mode by pressing `Esc`. Move the cursor to where you would like to paste the text and press the `p` key to paste. Similarly, select text in visual mode and hit `d` to delete the text and paste it somewhere else with the `p` key.

Command	Description
x	delete letter after cursor
X	delete letter before cursor
dd	delete line
d1	delete letter
d#l	delete # letters
dw	delete word
d#w	delete # words

Table 1.7: Commands for deleting in command mode

Command	Description
:map	customize
:help	view vim docs
cw	change word
u	undo
Ctrl-R	redo
.	Repeat the previous command
*	find next occurrence of word under cursor
#	find previous occurrence of word under cursor
/str	find str in file
n	find next match
N	find previous match

Table 1.8: Commands for entering insert mode

## Deleting Text in Command Mode

Insert mode should only be used for inserting text. Try to get in the habit of leaving insert mode as soon as you are done adding the text you want to add. Deleting text is much more efficient and versatile in command mode. The x and X commands are used to delete single characters. The d command is always accompanied by another navigational command. See Table 1.7 for a few examples.

## A Few Closing Remarks

In the next lab, we will introduce how to access another machine through the terminal. Vim will be essential in this situation since GUIs will not be an option.

If you are interested in continuing to use vim, you may be interested in checking out *gvim*. Gvim is a GUI that uses vim commands in a more traditional text editor window.

Also, in Table 1.8, we have listed a few more commands that are worth exploring. If you are interested in any of these features of vim, we encourage you to research these features further on the internet. Additionally, many people have published their *vimrc* file on the internet so other vim users can learn what options are worth exploring. It is also worth noting that we can use vim navigation commands in many other places in the shell. For example, try using the navigation commands when viewing the `man vim` page.



# 2

# More on the Unix Shell

**Lab Objective:** *Introduce system management, calling Unix Shell commands within Python, and other advanced topics.*

In this lab, we will build upon the foundation of the previous lab. As in the last lab, the majority of learning will not be had in finishing the problems, but in following the examples. By the end of this lab, you will have a solid foundation in Unix. You will be able to understand enough to learn any additional topics you want.

## File Security

To begin, run the following command while inside the `Shell2/Python/` directory (`Shell2/` is the end product of `Shell1/` from the previous lab). Notice your output will differ from that printed below; this is for learning purposes.

```
$ ls -l
-rw-rw-r-- 1 username groupname 194 Aug  5 20:20 calc.py
-rw-rw-r-- 1 username groupname 373 Aug  5 21:16 count_files.py
-rwxr-xr-x 1 username groupname   27 Aug  5 20:22 mult.py
-rw-rw-r-- 1 username groupname 721 Aug  5 20:23 project.py
```

Notice the first column of the output. The first character denotes the type of the item whether it be a normal file, a directory, a symbolic link, etc. The remaining nine characters denote the permissions associated with that file. Specifically, these permissions deal with reading, writing, and executing files. There are three categories of people associated with permissions. These are the user (the owner), group, and others. For example, look at the output for `mult.py`. The first character `-` denotes that `mult.py` is a normal file. The next three characters, `rwx` tell us the owner can read, write, and execute the file. The next three characters `r-x` tell us members of the same group can read and execute the file. The final three characters `--x` tell us other users can execute the file and nothing more.

Command	Description
<code>chmod u+x file1</code>	Add executing (x) permissions to user (u)
<code>chmod g-w file1</code>	Remove writing (w) permissions from group (g)
<code>chmod o-r file1</code>	Remove reading (r) permissions from other other users (o)
<code>chmod a+w file1</code>	Add writing permissions to everyone (a)

Table 2.1: Symbolic permissions notation

Command	Description
<code>chmod 760 file1</code>	Sets rwx to user, rw- to group, and --- to others
<code>chmod 640 file1</code>	Sets rw- to user, r-- to group, and --- to others
<code>chmod 775 file1</code>	Sets rwx to user, rwx to group, and r-x to others
<code>chmod 500 file1</code>	Sets r-x to user, --- to group, and --- to others

Table 2.2: Octal permissions notation

Permissions can be modified using the `chmod` command. There are two different ways to specify permissions, *symbolic permissions* notation and *octal permissions* notation. Symbolic permissions notation is easier to use when we want to make small modifications to a file's permissions. See Table 2.1.

Octal permissions notation is easier to use when we want to set all the permissions at once. The number 4 corresponds to reading, 2 corresponds to writing, and 1 corresponds to executing. See Table 2.2.

The commands in Table 2.3 are also helpful when working with permissions.

## Scripts

A shell script is a series of shell commands saved in a file. Scripts are useful when we have a process that we do over and over again. The following is a very simple script:

```
#!/bin/bash
echo "Hello World"
```

**Problem 1.** Using vim, create a file called `hello` that contains the previous text and save it. Note that no file type is necessary.

The first line starts with `"#!"`. This is called the *shebang* or *hashbang* character sequence. It is followed by the absolute path to the `bash` interpreter. If we were unsure where the `bash` interpreter is saved, we run `which bash`.

To execute a script, type the script name preceded by `./`

```
$ ./hello
bash: ./hello: Permission denied

# Notice you do not have permission to execute this file. This is by default.
$ ls -l hello
```

Command	Description
<code>chown</code>	change owner
<code>chgrp</code>	change group
<code>getfacl</code>	view all permissions of a file in a readable format.

Table 2.3: Other commands when working with permissions

Command	Description
<code>df dir1</code>	Display available disk space in file system containing <code>dir1</code>
<code>du dir1</code>	Display disk usage within <code>dir1</code> [-a, -h]
<code>free</code>	Display amount of free and used memory in the system
<code>ps</code>	Display a snapshot of current processes
<code>top</code>	Display interactive list of current processes

Table 2.4: Commands for resource management

```
-rw-r--r-- 1 username groupname 31 Jul 30 14:34 hello
```

**Problem 2.** Add executable permission to your `hello` script. Run the script again.

You can do this same thing with Python scripts. All you have to do is change the path following the shebang. To see where the Python interpreter is stored, run `which python`.

**Problem 3.** In the `Python/` directory you will find `count_files.py`. `count_files.py` is a python script that counts all the files within the `Shell2/` directory. Modify this file so it can be run as a script and change the permissions of this script so the user and group can execute the script.

Note: In the `subprocess.check_output` command, if `Shell2/` is not contained in your home directory (~), you will need to change ~ to the correct path to navigate there.

If you would like to learn how to run scripts on a set schedule, consider researching *cron jobs*.

## Resource Management

To be able to optimize performance, it is valuable to always be aware of the resources we are using. Hard drive space and computer memory are two resources we must constantly keep in mind. The commands found in table 2.4 are essential to managing resources.

Command	Description
COMMAND &	Adding an ampersand to the end of a command runs the command in the background
bg %N	Restarts the Nth interrupted job in the background
fg %N	Brings the Nth job into the foreground
jobs	Lists all the jobs currently running
kill %N	Terminates the Nth job
ps	Lists all the current processes
Ctrl-C	Terminates current job
Ctrl-Z	Interrupts current job
nohup	Run a command that will not be killed if the user logs out

Table 2.5: Job control commands

## Job Control

Let's say we had a series of scripts we wanted to run. If we knew that these would take a while to execute, we may want to start them all at the same time and let them run while we are working on something else. In table 2.5, we have listed some of the most common commands used in job control. We strongly encourage you to experiment with these commands. In the `Scripts/` directory, you will find a `five_secs` and a `ten_secs` script that takes five seconds and ten seconds to execute respectively. These will be particularly useful as you are experimenting with these commands.

```
# Don't forget to change permissions if needed
$ ./ten_secs &
$ ./five_secs &
$ jobs
[1]+  Running      ./ten_secs &
[2]-  Running      ./five_secs &
$ kill %2
[2]-  Terminated   ./five_secs &
$ jobs
[1]+  Running      ./ten_secs &
```

**Problem 4.** In addition to the `five_secs` and `ten_secs` scripts, the `Scripts/` folder contains three scripts that will each take about a forty-five seconds to execute. Execute each of these commands in the background so all three are running at the same time. To verify all scripts are running at the same time, write the output of `jobs` to a new file `log.txt` saved in the `Scripts/` directory.

## Python Integration

To this point, we have barely scratched the surface of all the functionality that Unix has to offer. However, the tools and commands we have addressed so far provide us with a foundation of the basics. Using the `subprocess` module in Python, we can call Unix commands. By combining Python and the Unix commands, our toolset is automatically broadened.

There are two functions in particular within the `subprocess` module we will use. When wanting to run a Unix command, use `subprocess.call()`. When wanting to run a Unix command and be able to store and manipulate the output, use `subprocess.check_output()`. These functions have a keyword argument `shell` that defaults to `False`. We want to set this argument to `True` to run the command in the Unix shell.

```
$ cd Shell-Lab/Documents
$ python
>>> import subprocess
>>> subprocess.call("ls -l", shell=True)
-rw-rw-r-- 1 username groupname 142 Aug  5 20:20 assignments.txt
-rw-rw-r-- 1 username groupname 427 Aug  5 20:21 doc1.txt
-rw-rw-r-- 1 username groupname 326 Aug  5 20:21 doc2.txt
-rw-rw-r-- 1 username groupname 612 Aug  5 20:21 doc3.txt
-rw-rw-r-- 1 username groupname 298 Aug  5 20:21 doc4.txt
-rw-rw-r-- 1 username groupname 1027 Aug  5 20:23 review.txt
-rw-rw-r-- 1 username groupname 920 Aug  5 23:50 words.txt
>>> files = subprocess.check_output("ls -l", shell=True)
>>> files
'-rw-rw-r-- 1 username groupname 142 Aug  5 20:20 assignments.txt\n-rw-rw-r-- 1 username groupname 427 Aug  5 20:21 doc1.txt\n-rw-rw-r-- 1 username groupname 326 Aug  5 20:21 doc2.txt\n-rw-rw-r-- 1 username groupname 612 Aug  5 20:21 doc3.txt\n-rw-rw-r-- 1 username groupname 298 Aug  5 20:21 doc4.txt\n-rw-rw-r-- 1 username groupname 1027 Aug  5 20:23 review.txt\n-rw-r-- 1 username groupname 920 Aug  5 23:50 words.txt\n'
>>> files.split('\n')
['-rw-rw-r-- 1 username groupname 142 Aug  5 20:20 assignments.txt',
 '-rw-rw-r-- 1 username groupname 427 Aug  5 20:21 doc1.txt',
 '-rw-rw-r-- 1 username groupname 326 Aug  5 20:21 doc2.txt',
 '-rw-rw-r-- 1 username groupname 612 Aug  5 20:21 doc3.txt',
 '-rw-rw-r-- 1 username groupname 298 Aug  5 20:21 doc4.txt',
 '-rw-rw-r-- 1 username groupname 1027 Aug  5 20:23 review.txt',
 '-rw-rw-r-- 1 username groupname 920 Aug  5 23:50 words.txt',
 '']
>>> files = files.split('\n')
# To get rid of the last empty string in the list
>>> files.pop()
''

# Now that we have a list object, we can manipulate and analyze this data in Python.
We can make it even more accessible by splitting the lines again
>>> files = [line.split() for line in files]
```

Command	Description
<code>passwd</code>	Change user password
<code>uname</code>	View operating system name
<code>uname -a</code>	Print all system information
<code>uname -m</code>	Print machine hardware
<code>w</code>	Show who is logged in and what they are doing
<code>whoami</code>	Print userID of current user

Table 2.6: Commands for system administration.

**Problem 5.** Create a `Shell` class in Python. Write a `find_file()` method that will search for a filename using the `find` command in the given directory. Write a `find_word()` method that finds a given word within the contents of the directory using the `grep` command. Both functions should accept a directory keyword as input which defaults to `None`. If no directory location is provided, then set it to be the current directory within the function. For both these functions, return a list of filepaths.

**Problem 6.** Write a method for the `Shell` class that recursively finds the  $n$  largest files within a directory. Have a keyword argument for the directory that defaults to the current directory. Be sure that your function only returns files. Hint: To view the size of a file `file1`, you can use `ls -s file1` or `du file1`

## System Management

In this section, we will address some of the basics of system management. As an introduction, the commands in table 2.6 are used to learn more about the computer system.

### Secure Shell

Let's say you are working for a company with a file server. Hundreds of people need to be able to access the content of this machine, but how is that possible? Or say you have a script to run that requires some serious computing power. How are you going to be able to access your company's super computer to run your script? We do this through *Secure Shell* (SSH).

SSH is a network protocol encrypted using public-key cryptography. The system we are connecting *to* is commonly referred to as the *host* and the system we are connecting *from* is commonly referred to as the *client*. Once this connection is established, there is a secure tunnel through which commands and files can be exchanged between the client and host. To end a secure connection, type `exit`.

As a warning, you cannot normally SSH into a Windows machine. If you want to do this, search on the web for available options.

```
$ whoami      # use this to see what your current login is
client_username
$ ssh my_host_username@my_hostname
```

```
# You will then be prompted to enter the password for my_host_username

$ whoami      # use this to verify that you are logged into the host
my_host_username

$ hostname
my_hostname

$ exit
logout
Connection to my_host_name closed.
```

Now that you are logged in on the host computer, all the commands you execute are as though you were executing them on the host computer.

## Secure Copy

When we want to copy files between the client and the host, we use the *secure copy* command, `scp`. The following commands are run when logged into the client computer.

```
# copy filename to the host's system at filepath
$ scp filename host_username@hostname:filepath

#copy a file found at filepath to the client's system as filename
$ scp host_username@hostname:filepath filename

# you will be prompted to enter host_username's password in both these ←
    instances
```

**Problem 7.** You will either need a partner for this problem or have access to a username on another computer. Experiment with SSH. Verify that you can connect from a client to a host. Copy a few files between the host and the client.

## Generating SSH Keys (Optional)

If there is a host that we access on a regular basis, typing in our password over and over again can get tedious. By setting up SSH keys, the host can identify if a client is a trusted user without needing to type in a password. If you are interested in experimenting with this setup, a Google search of "How to set up SSH keys" will lead you to many quality tutorials on how to do so.

## Web Related

Sometimes you will need to download files from the internet. `wget` and `curl` are both used to download content from the web, and in many applications they both perform the same tasks. Most of the differences between `wget` and `curl` are beyond the scope of this book. At its most basic, `curl` is the more robust tool of the two while `wget` can download recursively. The provided examples will use `wget`.

### Downloading files using Wget

When we want to download a single file, we just need the URL for the file we want to download. Running the command below will download a JPEG image of a person writing on a chalkboard. Similarly, you can download PDF files, HTML files, and other content simply by providing a different URL.

```
$ wget http://acme.byu.edu/wp-content/uploads/2013/07/0906-13-00903.jpg
```

The following are also useful commands using `wget`.

```
# Download files from URLs listed in urls.txt  
$ wget -i list_of_urls.txt  
  
# Download in the background  
$ wget -b URL  
  
# Download something recursively  
$ wget -r --no-parent URL
```

**Problem 8.** In the `Documents/` directory, you will find a file named `urls.txt` with a list of URLs. Download the files in this list using `wget`. Move the pictures that will be downloaded to the `Photos/` directory.

## Additional Material

### sed and awk

`sed` and `awk` are two different scripting languages in their own right. Like Unix, these languages are easy to learn but difficult to master. It is very common to combine Unix commands and `sed` and `awk` commands. We will address the basics, but if you would like more information see <<http://www.theunixschool.com/p/awk-sed.html>>

### Printing Specific Lines Using sed

We have already used the `head` and `tail` commands to print the beginning and end of a file respectively. What if we wanted to print lines 30 to 40, for example? We can accomplish this using `sed`. In the `Documents/` folder, you will find the `lines.txt` file. We will use this file for the following examples.

```
# Same output as head -n3
$ sed -n 1,3p lines.txt
line 1
line 2
line 3

# Same output as tail -n3
$ sed -n 3,5p lines.txt
line 3
line 4
line 5

# Print lines 2-4
$ sed -n 3,5p lines.txt
line 2
line 3
line 4

# Print lines 1,3,5
$ sed -n -e 1p -e 3p -e 5p lines.txt
line 1
line 3
line 5
```

### Find and Replace Using sed

Using `sed`, we can also perform find and replace. We can perform this function on the output of another command or we can perform this function in place on other files. The basic syntax of this `sed` command is the following.

```
sed s/str1/str2/g
```

This command will replace every instance of **str1** with **str2**. More specific examples follow.

```
$ sed s/line/LINE/g lines.txt
LINE 1
LINE 2
LINE 3
LINE 4
LINE 5

# Notice the file didn't change at all
$ cat lines.txt
line 1
line 2
line 3
line 4
line 5

# To save the changes, add the -i flag
$ sed -i s/line/LINE/g lines.txt
$ cat lines.txt
LINE 1
LINE 2
LINE 3
LINE 4
LINE 5
```

## Formatting output using awk

Earlier in this lab we mentioned `ls -l` and as we have seen, this outputs lots of information. Using `awk`, we can select which fields we wish to print. Suppose we only cared about the file name and the permissions. We can get this output by running the following command.

```
$ ls -l | awk ' {print $1, $9}'
```

Notice we pipe the output of `ls -l` to `awk`. When calling a command using `awk`, we use quotation marks. Note it is a common mistake to forget to add these quotation marks. Inside these quotation marks, commands always take the same format.

```
awk ' <options> {<actions>} '
```

In the remaining examples we will not be using any of the options, but we will address various actions. For those interested in learning what options are available see <<http://www.theunixschool.com/p/awk-sed.html>>.

In the `Documents/` directory, you will find a `people.txt` file that we will use for the following examples. In our first example, we use the `print` action. The `$1` and `$9` mean that we are going to print the first and ninth fields. Beyond specifying which fields we wish to print, we can also choose how many characters to allocate for each field.

```

# contents of people.txt
$ cat people.txt
male,John,23
female,Mary,31
female,Sally,37
male,Ted,19
male,Jeff,41
female,Cindy,25

# Change the field separator (FS) to "," at the beginning of execution (BEGIN)
# By printing each field individually proves we have successfully separated the←
# fields
$ awk ' BEGIN{ FS = "," }; {print $1,$2,$3} ' < people.txt
male John 23
female Mary 31
female Sally 37
male Ted 19
male Jeff 41
female Cindy 25

# Format columns using printf so everything is in neat columns in order (gender←
# ,age,name)
$ awk ' BEGIN{ FS = ","}; {printf "%-6s %2s %s\n", $1,$3,$2} ' < people.txt
male    23 John
female  31 Mary
female  37 Sally
male    19 Ted
male    41 Jeff
female  25 Cindy

```

The statement "`%-6s %2s %s\n`" formats the columns of the output. This says to set aside six characters left justified, then two characters right justified, then print the last field to its full length.

**Problem 9.** Inside the `Documents/` directory, you should find a file named `files.txt`. This file contains details on approximately one hundred files. The different fields in the file are separated by tabs. Using `awk`, `sort`, pipes, and redirects, write a file named `date_modified.txt` with the following specifications:

- in the first column, print the date the file was modified
- in the second column, print the name of the file
- sort the file from newest to oldest based on the date last modified

All this can be accomplished using one command.

We have barely scratched the surface of what `awk` can do. Performing an internet search for "awk one-liners" will give you many additional examples of useful commands you can run using `awk`.



# 3

# Basic Regular Expressions

**Lab Objective:** *Learn the basics of using regular expressions to find text*

Regular expressions allow for quick searching and replacing of general patterns of text. While nearly all text editors have a feature that will find and replace exact strings of text, regular expressions are used to find text in a much more general way. For example, using a single regular expression, you can find every email address in a text file without having to sift through it by hand.

## Terminology and Basics

A “regular expression” is basically just a string of characters that follow a certain syntax. Computer programs can then interpret these expressions as instructions to search for certain kinds of text. We will often call regular expressions “patterns”, and we will say that certain patterns “match” certain strings. The general idea is that a regular expression represents a large set of strings (for example, all valid email addresses), and if a specific string is in that set, we say that the regular expression matches that string.

### ACHTUNG!

Regular expression libraries have been implemented and are a part of the standard distribution of nearly every programming language, and many text editors have a find-and-replace mode that uses regular expressions. Unfortunately, the syntax for regular expressions may be slightly different in each implementation. There is no universal standard for all regular expressions across all platforms. However, the original syntax and a few variants are very widespread, so the basic regular expression techniques we learn in this lab should be virtually the same in almost every situation you will encounter them.

The simplest use of regular expressions is to match text literally. For example, the pattern `"cat"` matches the string `"cat"` but does not match the strings `"dog"` or `"bat"`.

Now that we have a general idea of what regular expressions are, we will see how to use them in Python.

## Regular Expressions in Python

The python package `re` contains the functionality for using regular expressions. To use it, simply run the command `import re`.

The following Python code demonstrates what we said earlier about the regular expression `"cat":`

```
>>> bool(re.match("cat", "cat"))
True
>>> bool(re.match("cat", "dog"))
False
>>> bool(re.match("cat", "bat"))
False
```

The main functions we will use are `re.match(pattern, string_to_test)` and `re.compile(pattern)`. You can think of `re.match` as returning a boolean value representing whether the given pattern matched the given string. The function `re.compile` returns a compiled object that represents a regular expression. You can then call the `match` function on this compiled object to get a boolean value. There is a similar function, `re.search`, which will match the regular expression anywhere inside a given string. We will see one example shortly where `re.search` is preferred in multiline matching.

The following code shows an example of how to use `re.compile`:

```
>>> pattern = re.compile("any regular expression")
>>> result = pattern.match("any string")
```

The above code is equivalent to the following:

```
>>> result = re.match("any regular expression", "any string")
```

Most programs use the compiled form (the first of the above two examples) for efficiency.

When constructing a regular expression, it is best to construct your pattern string using Python's syntax for raw strings by prefacing the string with the '`r`' character. This causes the constructed string to treat backslashes as actual backslash characters, rather than the start of an escape sequence.

For example:

```
>>> normal = "hello\nworld"
>>> raw = r"hello\nworld"
>>> print normal
hello
world
>>> print raw
hello\nworld
>>> type(normal), normal
(str, 'hello\nworld')
>>> type(raw), raw
(str, 'hello\\nworld')
```

Note that `raw` and `normal` are both python strings; one was just constructed differently. Also notice that when we constructed `raw`, it inserted an extra backslash before the existing backslash.

We use raw strings because the backslash character is a very important special character in regular expressions. If we wanted to use backslash characters as part of a normally-constructed Python string, we would need to either escape every single backslash by using two backslashes each time, or we could take the much easier and less confusing route of using Python's raw strings. To demonstrate this effect, suppose we wanted to know whether the regular expression "`\$3\.00`" matched the string "`">$3.00`". We could get our answer in either of the following ways:

```
>>> bool(re.match("\$3\.00", "$3.00"))
True
>>> bool(re.match(r"\$3\.00", "$3.00"))
True
```

(You will see why this pattern matches this string soon)

Remember, readability counts.

## Literal Characters and Metacharacters

The following characters are used as metacharacters in regular expressions:

```
. ^ $ * + ? { } [ ] \ | ( )
```

These characters mean special things when used in regular expressions, making the vast power of regular expressions possible. We will get to using these characters later. For now, what do we do if want to match these characters literally? We simply escape these characters using the metacharacter '`\`':

```
>>> pattern = re.compile(r"\$2\.95, please")
>>> bool(pattern.match("$2.95, please"))
True
>>> bool(pattern.match("$295, please"))
False
>>> bool(pattern.match("$2.95"))
False
```

**Problem 1.** Define the variable `pattern_string` using literal characters and escaped metacharacters in such a way that the following python program prints `True`:

```
import re
pattern_string = r"" # Edit this line
pattern = re.compile(pattern_string)
print bool(pattern.match("^{(!%.*_)}&"))
```

A little misleadingly, the `re.match` method isn't actually checking whether the given regular expression matches entire strings. Rather, it checks whether the regular expression matches *at the beginning* of the string, even if the string continues on afterward. For example:

```
>>> pattern = re.compile(r"x")
>>> bool(pattern.match("x"))
True
>>> bool(pattern.match("xabc"))
True
>>> bool(pattern.match("abcx"))
False
```

You might not expect the pattern '`x`' to match the string "`xabc`", but it does. This can cause confusion and headache, so we'll have to be a little more precise with the help of metacharacters.

The *line anchor* metacharacters, '`^`' and '`$`', are used to match the start and the end of a line of text, respectively. Let's see them in action:

```
>>> pattern = re.compile(r"^\$")
>>> bool(pattern.match("x"))
True
>>> bool(pattern.match("xabc"))
False
>>> bool(pattern.match("abcx"))
False
```

An added benefit of using '`^`' and '`$`' is that they allow you to search across multiple lines. For example, how would we match "`World`" in the string "`Hello\nWorld`"? Using `re.MULTILINE` in the `re.search` function will allow us to match at the beginning of each new line, instead of just the beginning of the string. Since we have seen two ways to match strings with regex expressions, the following shows two ways to implement multiline searching:

```
>>> bool(re.search("^W", "Hello\nWorld"))
False
>>> bool(re.search("^W", "Hello\nWorld", re.MULTILINE))
True
>>> pattern1 = re.compile("^W")
>>> pattern2 = re.compile("^W", re.MULTILINE)
>>> bool(pattern1.search("Hello\nWorld"))
False
>>> bool(pattern2.search("Hello\nWorld"))
True
```

For simplicity, the rest of the lab will focus on single line matching.

Let's move on to '`(`', '`)`', and '`|`'. The '`|`' character (the “pipe” character, usually found on the key below the backspace key) matches one of two or more regular expressions:

```
>>> pattern2 = re.compile(r"red$|^blue$")
>>> pattern3 = re.compile(r"red$|^blue$|^orange$")
>>> bool(pattern2.match("red")), bool(pattern3.match("red"))
```

```
(True, True)
>>> bool(pattern2.match("blue")),    bool(pattern3.match("blue"))
(True, True)
>>> bool(pattern2.match("orange")),  bool(pattern3.match("orange"))
(False, True)
>>> bool(pattern2.match("redblue")), bool(pattern3.match("redblue"))
(False, False)
```

You can think of '`|`' as doing an "or" operation. How would we create a regular expression that matched both "`one fish`" and "`two fish`"? Although the regular expression "`one fish|two fish`" works, there is a better way, by using both the pipe character and parentheses:

```
>>> pattern = re.compile(r"^(one|two) fish$")
>>> bool(pattern.match("one fish"))
True
>>> bool(pattern.match("two fish"))
True
>>> bool(pattern.match("three fish"))
False
>>> bool(pattern.match("one two fish"))
False
```

As the above example demonstrates, parentheses are used to group sequences of characters together and change the order of precedence of the metacharacters, much like how parentheses work in an arithmetic expression such as  $3*(4+5)$ . In regular expressions, the '`|`' metacharacter has the lowest precedence out of all the metacharacters.

Parentheses actually have more uses, which we will learn later. For now, note that parentheses aren't matched literally:

```
>>> bool(re.match(r"r(hi)no(c(e)ro)s", "rhinoceros"))
True
```

Parentheses help give regular expressions higher precedence. For example, "`^one|two fish$`" gives precedence to the invisible string concatenation between "`two`" and "`fish`" while "`^(one|two ) fish$`" gives precedence to the '`|`' metacharacter.

**Problem 2.** Define the variable `pattern_string` using the metacharacter '`|`' and parentheses in such a way that the following python program prints `True`:

```
import re
pattern_string = r"" # Edit this line
pattern = re.compile(pattern_string)
strings_to_match = [ "Book store", "Book supplier", "Mattress store", "←
    Mattress supplier", "Grocery store", "Grocery supplier"]
print all(pattern.match(string) for string in strings_to_match)
```

Your regular expression should not match any other string, including strings such as "Book store sale".

## Character Classes

The metacharacters '[' and ']' are used to create *character classes*. Here they are in action:

```
>>> pattern = re.compile(r"[xy]")
>>> bool(pattern.match("x"))
True
>>> bool(pattern.match("y"))
True
>>> bool(pattern.match("z"))
False
>>> bool(pattern.match("x: Why does this match? Were you paying attention?"))
True
```

In essence, a character class will match any one out of several characters.

Inside character classes, there are two additional metacharacters: '-' and '^'. Although we've already seen '^' as a metacharacter, it has a different meaning when used inside a character class. When '^' appears as *the first character* in a character class, the character class matches anything not specified instead. Think of '^' as performing a set complement operation on the character class. For example:

```
>>> pattern = re.compile(r"^[^ab]$")
>>> bool(pattern.match("x"))
True
>>> bool(pattern.match("#"))
True
>>> bool(pattern.match("a"))
False
>>> bool(pattern.match("b"))
False
```

Note that the two '^' characters mean completely different things; the first '^' anchors us at the beginning of the line, while the second '^' performs a set complement operation on the character class "[ab]".

The other character class metacharacter is '-'. This is used to specify a range of values. For example:

```
>>> pattern = re.compile(r"^[a-z][0-9][0-9]$")
>>> bool(pattern.match("a90"))
True
>>> bool(pattern.match("z73"))
True
>>> bool(pattern.match("A90"))
False
>>> bool(pattern.match("zs3"))
```

```
False
```

Multiple ranges or characters can be included in a single character class; in this case, the character class will match any character that fits either criterion:

```
>>> pattern = re.compile(r"^[abcA-C][0-27-9]$")
>>> bool(pattern.match("b8"))
True
>>> bool(pattern.match("B2"))
True
>>> bool(pattern.match("a9"))
True
>>> bool(pattern.match("a4"))
False
>>> bool(pattern.match("E1"))
False
```

Notice in the first line that `[abcA-C]` acts like `[a|b|c|(A-C)]` and `[0-27-9]` acts like `[(0-2)|(7-9)]`.

Finally, there are some built-in shorthands for certain character classes:

- '`\d`' (think “digit”) matches any digit. It is equivalent to "`[0-9]`".
- '`\w`' (think “word”) matches any alphanumeric character or underscore. It is equivalent to "`[a-zA-Z0-9_]`".
- '`\s`' (think “space”) matches any whitespace character. It is equivalent to "`[ \t\n\r\f\v]`".

The following character classes are the complements of those above:

- '`\D`' is equivalent to "`[\^0-9]`" or "`[\^D]`"
- '`\W`' is equivalent to "`[\^a-zA-Z0-9_]`" or "`[\^W]`"
- '`\S`' is equivalent to "`[\^ \t\n\r\f\v]`" or "`[\^S]`"

These character classes can be used in character classes; for example, "`[_A-Z\s]`" will match an underscore, any capital letter, or any whitespace character.

The '`.`' metacharacter, equivalent to "`[\^n]`" on UNIX and "`[\^r\n]`" on Windows, matches any character except for a line break. For example:

```
>>> pattern = re.compile(r"^\.\d$")
>>> bool(pattern.match("a0b"))
True
>>> bool(pattern.match("888"))
True
>>> bool(pattern.match("n2%"))
True
>>> bool(pattern.match("abc"))
False
>>> bool(pattern.match("m&m"))
```

```
False
>>> bool(pattern.match("cat"))
False
```

**Problem 3.** Define the variable `pattern_string` in such a way that the following python program prints `True`:

```
import re
pattern_string = r"" # Edit this line
pattern = re.compile(pattern_string)

strings_to_match = ["a", "b", "c", "x", "y", "z"]
uses_line_anchors = (pattern_string.startswith('^') and pattern_string.endswith('$'))
solution_is_clever = (len(pattern_string) == 8)
matches_list = all(pattern.match(string) for string in strings_to_match)

print uses_line_anchors and solution_is_clever and matches_list
```

**Problem 4.** A *valid python identifier* (aka a valid variable name) is defined as any string composed of an alphabetic character or underscore followed by any (possibly empty) sequence of alphanumeric characters and underscores.

Define the variable `identifier_pattern_string` that defines a regular expression that matches valid python identifiers that are exactly five characters long.

To help you test your pattern, the following program should print `True`. (This is necessary but not sufficient to show your regular expression is correct):

```
import re
identifier_pattern_string = r"" # Edit this line
identifier_pattern = re.compile(identifier_pattern_string)

valid = ["mouse", "HORSE", "_1234", "__x__", "while"]
not_valid = ["3rats", "err*r", "sq(x)", "too_long"]

print all(identifier_pattern.match(string) for string in valid) and not any(identifier_pattern.match(string) for string in not_valid)
```

Hint: Use the '`\w`' character class to keep your regular expression relatively short.

**NOTE**

As you might have noticed, using this definition, "`while`" is considered a valid python identifier, even though it really is a reserved word. In the following problems, we will make a few other simplifying assumptions about the python language.

## Repetition

Suppose in the last problem we wanted the string to be 20 characters long. You wouldn't want to write `\w` 20 times. In fact, what if you wanted to match at most one instance of a character or a number with at least three digits? The metacharacters '`*`', '`+`', '`{`', and '`}`' are very useful for repetition.

The '`*`' metacharacter means "Match zero or more times (as many as possible)" when it follows another regular expression. For instance:

```
>>> pattern = re.compile(r"^\w*\b$")
>>> bool(pattern.match("b"))
True
>>> bool(pattern.match("ab"))
True
>>> bool(pattern.match("aab"))
True
>>> bool(pattern.match("aaab"))
True
>>> bool(pattern.match("abab"))
False
>>> bool(pattern.match("abc"))
False
```

The '`+`' metacharacter means "Match one or more times (as many as possible)" when it follows another regular expression. As an example:

```
>>> pattern = re.compile(r"^\w{1,2}\b")
>>> bool(pattern.match("ha"))
True
>>> bool(pattern.match("hi"))
True
>>> bool(pattern.match("hiaiaa"))
True
>>> bool(pattern.match("h"))
False
>>> bool(pattern.match("hah"))
False
```

It's important to understand why "`hiaiaaa`" is a match here; matching multiple times means matching the preceding *expression* multiple times, not matching the *results* of the preceding expression multiple times. We haven't yet learned how to construct a regular expression with that behavior.

The '`?`' metacharacter means "Match one time (if possible) or do nothing (i.e. match zero times)" when it follows another regular expression:

```
>>> pattern = re.compile(r"^\abc?$")
>>> bool(pattern.match("abc"))
True
>>> bool(pattern.match("ab"))
True
>>> bool(pattern.match("abd"))
False
>>> bool(pattern.match("ac"))
False
```

The curly brace metacharacters are used to specify a more precise amount of repetition:

```
>>> pattern = re.compile(r"^\{2,4}\$")
>>> bool(pattern.match("a"))
False
>>> bool(pattern.match("aa"))
True
>>> bool(pattern.match("aaa"))
True
>>> bool(pattern.match("aaaa"))
True
>>> bool(pattern.match("aaaaa"))
False
```

If two arguments `x` and `y` are given to the curly braces (i.e., `{x, y}`), the preceding regular expression must appear between `x` and `y` times, inclusive, in order for the overall expression to match.

### ACHTUNG!

In this last example, line anchors can save us from a lot of confusion. Note the differences between the following example and the example immediately above:

```
>>> pattern = re.compile(r"\{2,4}")
>>> bool(pattern.match("a"))
False
>>> bool(pattern.match("aa"))
True
>>> bool(pattern.match("aaa"))
True
>>> bool(pattern.match("aaaa"))
True
```

```
>>> bool(pattern.match("aaaaa"))
True
```

If only one argument  $x$  is given and is followed by a comma, the preceding regular expression must match  $x$  or more times. If only one argument  $x$  is given without a comma, the preceding regular expression must match *exactly*  $x$  times. For example:

```
>>> exactly_three = re.compile(r"^\w{3}$")
>>> three_or_more = re.compile(r"^\w{3,}$")
>>> def test_both_patterns(string):
...     return bool(exactly_three.match(string)), bool(three_or_more.match(string))
>>> test_both_patterns("a")
(False, False)
>>> test_both_patterns("aa")
(False, False)
>>> test_both_patterns("aaa")
(True, True)
>>> test_both_patterns("aaaa")
(False, True)
>>> test_both_patterns("aaaaa")
(False, True)
```

You can also test  $\{\mathit{x}\}$  which will match the preceding regular expression up to  $x$  times.

**Problem 5.** Modify your definition of `identifier_pattern_string` from the previous problem to match valid python identifiers of any length.

## Cleaning Dirty Data with Regular Expressions

A common consensus among data scientists is that the majority of your time will be spent cleaning data. Throughout the remainder of this volume, you will have multiple opportunities to practice cleaning data.

Often times, cleaning data is as simple as changing the format of your data or filling missing values. However, with text-based data, additional work is often necessary. Using regular expressions to clean text-based data is often a good option.

**Problem 6.** The provided file `contacts.txt` contains poorly formatted contact data for 5000 (fictitious) individuals. This dataset contains birthdays, email addresses, and phone numbers for the individuals.

You will notice that much of this data is missing. To make things more complicated, the format of the data isn't consistent. For example, some birthdays are in the format 1/1/99, some in the format 01/01/1999, and some in the format 1/1/1999. The formatting for phone numbers is not consistent either. Some phone numbers are of the form (123)456-7890 while others are of the form 123-456-7890.

Using regular expressions, create a Python dictionary where the key is the name of the individual and the value is a dictionary of data. For example, the resulting dictionary should look something like this:

```
{"John Doe": {"bday": "1/1/1990",
               "email": "john_doe90@gmail.com",
               "phone": "(123)456-7890"}}
```

## Additional Material

### Regular Expressions in the Unix Shell

As we have seen thusfar, regular expressions are very useful when we want to match patterns. Regular expressions can be used when matching patterns in the Unix Shell. Though there are many Unix commands that take advantage of regular expressions, we will focus on `grep` and `awk`.

#### Regular Expressions and grep

Recall from Lab 1 that `grep` is used to match patterns in files or output. It turns out we can use regular expressions to define the pattern we wish to match.

In general, we use the following syntax:

```
$ grep 'regexp' filename
```

We can also use regular expressions when piping output to `grep`.

```
# List details of directories within current directory.
$ ls -l | grep ^d
```

#### Regular Expressions and awk

As in Lab 2, we will be using `awk` to format output. By incorporating regular expressions, `awk` becomes much more robust. Before GUI spreadsheet programs like Microsoft Excel, `awk` was commonly used to visualize and query data from a file.

Including `if` statements inside `awk` commands gives us the ability to perform actions on lines that match a given pattern. The following example prints the filenames of all files that are owned by `freddy`.

```
$ ls -l | awk ' {if ($3 ~ /freddy/) print $9} '
```

Because there is a lot going on in this command, we will break it down piece-by-piece. The output of `ls -l` is getting piped to `awk`. Then we have an `if` statement. The syntax here means if the condition inside the parenthesis holds, print field 9 (the field with the filename). The condition is where we use regular expressions. The `~` checks to see if the contents of field 3 (the field with the username) matches the regular expression found inside the forward slashes. To clarify, `freddy` is the regular expression in this example and the expression must be surrounded by forward slashes.

Consider a similar example. In this example, we will list the names of the directories inside the current directory. (This replicates the behavior of the Unix command `ls -d */`)

```
$ ls -l | awk ' {if ($1 ~ /^d/) print $9} '
```

Notice in this example, we printed the names of the directories, whereas in one of the example using `grep`, we printed all the details of the directories as well.

**ACHTUNG!**

Some of the definitions for character classes we used earlier in this lab will not work in the Unix Shell. For example, \w and \d are not defined. Instead of \w, use [:alnum:]]. Instead of \d, use [:digit:]]. For a complete list of similar character classes, search the internet for “POSIX Character Classes” or “Bracket Character Classes.”

**Problem 7.** You have been given a list of transactions from a fictional start-up company. In the `transactions.txt` file, each line represents a transaction. Transactions are represented as follows:

```
# Notice the semicolons delimiting the fields. Also, notice that in ←
     between the last and first name, that is a comma, not a semicolon.
<ORDER_ID>;<YEAR><MONTH><DAY>;<LAST>,<FIRST>;<ITEM_ID>
```

Using this set of transactions, produce the following information using regular expressions and the given command:

- Using `grep`, print all transactions by either Nicholas Ross or Zoey Ross.
- Using `awk`, print a sorted list of the names of individuals that bought item 3298.
- Using `awk`, print a sorted list of items purchased between June 13 and June 15 of 2014 (inclusive).

These queries can be produced using one command each.

We encourage the interested reader to research more about how regular expressions can be used with `sed`.

# 4

# SQL and Relational Databases

**Lab Objective:** *Understand concepts of a relational database and the fundamentals of the SQL language via SQLite.*

When working with large amounts of data, it is important to be able to quickly find and retrieve interesting information. Fortunately, there is a way to handle such massive amounts of data in a reasonably efficient way: a database. A database is simply a structured repository of data, and it allows us to store and retrieve information very quickly. It is managed by a *database management system*, or DBMS. The DBMS is software that allows users to interact directly with the database.

## Relational Databases

A *relational database* is a paradigm for organizing data inside of a database. In this paradigm, the data is broken down into tuples of information. These tuples are then grouped into tables, or *relations*, each of which is simply a set of tuples. Each table has a *schema* that defines the attributes of the tuples within the table. If we fix an order to the attributes in the schema, we can think of each attribute as a column of the table, and each tuple as a row of the table. See Figure 4.1 for an illustration of these ideas.

As an example, suppose we have demographic data for a large number of individuals. If we are interested in the gender and age of the individuals, we might make a table with schema (Name, Gender, Age). This table would consist of several 3-tuples, such as (Jane Doe, F, 20). Alternatively, we can view this table as having three columns and as many rows as there are individuals within our data set. We might also create a table with schema (Name, Employment Status, Income, Education).

In the relational paradigm, there must be at least one attribute in each schema that can act as a *primary key*. This can uniquely identify each tuple of the table. It is common to use an ID number or other such unique information for the primary key. In our example above, the "Name" attribute acted as a primary key. However, this attribute only works as a primary key provided no two individuals within the data set have the same name.

One important feature of a database is the *transaction*, which is a conceptual protocol for interacting with the database. Most relational databases are transactional databases. The best way to conceptualize this is imagine that your database is like a bank. Your connection to the database is analogous to the bank teller. When you make one or more deposits and withdrawals, you are making a transaction. A database transaction should have certain properties to protect the integrity of the data. These properties are described in detail in [en.wikipedia.org/wiki/ACID](https://en.wikipedia.org/wiki/ACID)

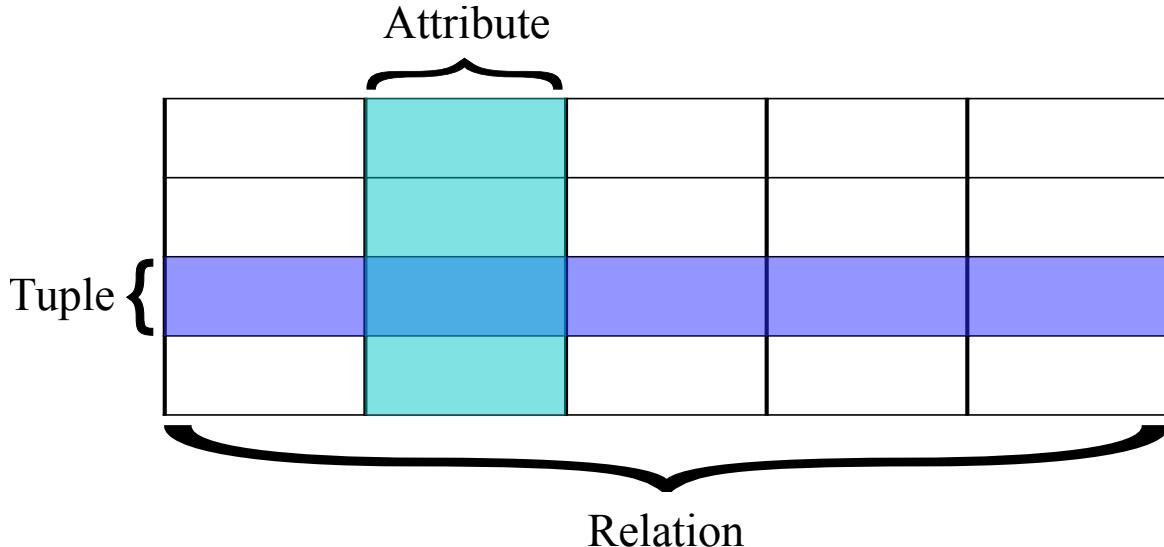


Figure 4.1: Elements of a relation.

## Introduction to SQL

Most common DBMSs use a variant of the SQL language to interact with the database. SQL is an acronym for *Structured Query Language*, and may be pronounced like the word “sequel” or by saying the letters “s”, “q”, and “l” separately. While SQL is not generally portable across different DBMSs, we will focus on the parts of SQL that are relatively common. In particular, we will base our discussion on the SQLite database management system, a very popular DBMS.

SQL consists of blocks of code called statements. Each statement is made up of clauses which may or may not require predicates. Predicates specify conditions that can limit the effect of a clause.

### NOTE

SQL commands are often written in all caps to help distinguish them from the other parts of the query. It is only a matter of style. SQLite, along with most other database managers, is case insensitive. In Python’s SQL interface, the semicolon is also not needed. However, most other database systems will require it, so it’s a good idea to conform in Python.

Let’s look at an example SQL statement:

```
SELECT * FROM table WHERE id=3+1 AND name='Bob';
```

This statement includes a SELECT clause and a WHERE clause. The WHERE clause contains two predicates: `id=3+1` and `name='Bob'`. These two predicates limit the effect of the SELECT clause because any resulting tuples in the table must satisfy both conditions. This entire statement is classified as a query since it does not modify the database in any way.

SQL has several classes of statements. The two main classes we will cover in this lab are schema (Table 4.1) and data manipulation (Table 4.2). We will give you a simplified description of each command and its syntax. You are encouraged to look up the full syntax outside of this lab.

Keyword	Syntax
<code>CREATE TABLE</code>	<code>CREATE TABLE &lt;table&gt; (&lt;col1&gt; &lt;type&gt;, &lt;col2&gt; &lt;type&gt;, ...);</code>
<code>DROP TABLE</code>	<code>DROP TABLE &lt;table&gt;;</code>
<code>CREATE INDEX</code>	<code>CREATE INDEX &lt;name&gt; ON &lt;table&gt; (&lt;col1&gt;);</code>
<code>DROP INDEX</code>	<code>DROP INDEX &lt;name&gt;;</code>

Table 4.1: The SQL Schema commands

Keyword	Syntax
<code>INSERT INTO</code>	<code>INSERT INTO &lt;table&gt; &lt;attributes&gt; VALUES (&lt;value1&gt;, &lt;value2&gt;, ...);</code>
<code>UPDATE</code>	<code>UPDATE &lt;table&gt; SET (&lt;col1&gt;=&lt;val1&gt;, &lt;col2&gt;=&lt;val2&gt;, ...) WHERE &lt;condition&gt;;</code>
<code>DELETE</code>	<code>DELETE FROM &lt;table&gt; WHERE &lt;condition&gt;;</code>
<code>SELECT</code>	<code>SELECT &lt;attributes&gt; FROM &lt;table&gt; WHERE &lt;condition&gt;;</code>

Table 4.2: The SQL Data Manipulation commands

## SQL in Python

Python has built-in support for SQLite databases using the standard library. Let's open a database called `test1`.

```
import sqlite3 as sql
db = sql.connect("test1")
```

The `connect()` function is used to connect to a database. If it does not already exist, then a new database will be created using the string passed as the argument for the name. The new database was created as a file in the current working directory.

### Ending the SQL Session

Once we are finished performing SQL statements and interacting with the database, we need to commit our changes and safely close the connection to the database. This can be done by calling methods on the database connection object.

```
db.commit()      #save changes made in the transaction
db.close()       #safely close the database
```

A database connection is automatically closed in Python when the connection object is garbage-collected. However, it is nice to be safe and explicit in closing a database connection using the `close()` method.

## Cursor

To execute SQL commands, we need to get a cursor object from the database.

```
cur = db.cursor()
```

Method	Description
<code>execute</code>	Execute a single SQL statement
<code>executemany</code>	Execute a single SQL statement over a sequence
<code>executescript</code>	Execute a SQL script (multiple SQL commands)
<code>close</code>	Closes the cursor object

Table 4.3: Cursor object methods

Python Type	SQLite Type
<code>None</code>	<code>NULL</code>
<code>int</code>	<code>INTEGER</code>
<code>long</code>	<code>INTEGER</code>
<code>float</code>	<code>REAL</code>
<code>str</code>	<code>TEXT</code>
<code>buffer</code>	<code>BLOB</code>

Table 4.4: Python and SQLite types mapping

The cursor object has several useful methods (Table 4.3). Through the cursor, we will execute all of our SQL commands.

Before creating a table, we need to understand how SQLite stores information in a database. SQLite uses five native data types (a simplified system from other SQL database managers). Table 4.4, gives a mapping between Python and SQLite native types.

## Creating and Dropping Tables

Let's create a table.

```
cur.execute('CREATE TABLE StudentInformation (StudentID INTEGER NOT NULL, Name ←
    TEXT, MajorCode INTEGER);')
```

This will create the empty table in Table 4.5.

The arguments in parentheses are the column names followed by the data type that entries in that column will be, and together these form the schema of the table. The `INTEGER` data type in SQLite is a 1, 2, 3, 4, 6, or 8 byte integer depending on the value. The `NOT NULL` command is a *constraint* on the `StudentID` column. It requires that all records in the table have a student ID.

### NOTE

SQLite does not enforce types on columns. Just like Python, SQLite is dynamically typed. However, most other database systems strictly enforce column types. It is a good idea to conform to the column types specified in the schema.

StudentID	Name	MajorCode
-----------	------	-----------

Table 4.5: StudentInformation

Note that each command we execute returns the same cursor object. This object is equipped with a method that allows us to look at any results of the previous command. The result is formally known as the *result set*. If you use `cur.fetchall()`, you will see an empty list. That is because the create table command does not return a result set.

Now we want to build a relation between students, the courses they've had, and their grades in those courses.

```
cur.execute('CREATE TABLE StudentGrades (StudentID INT NOT NULL, CourseID INT, ←
Grade TEXT);')
```

**Problem 1.** In this problem, you will create two new tables in the database “sql1”. The first table will be called MajorInfo and have a column called MajorID and MajorName. MajorID is an integer and MajorName is a string.

The second table will be called CourseInfo and have columns called CourseID and CourseName, also integers and strings, respectively.

Hint: In order to view information about the columns of the table, run the following command:

```
cur.execute("PRAGMA table_info('table_name')")
for info in cur:
    print info
```

For each column, this command will output (ID, name, type, notnull, default value, primary key). Also, don't forget to commit and close your database.

We can also destroy tables using the `DROP TABLE` command.

```
cur.execute("CREATE TABLE test_table (id INT, name TEXT);")
```

We can delete the table by dropping it.

```
cur.execute("DROP TABLE test_table;")
```

If a table doesn't exist, an exception will be raised. We can tell the database to drop the table only if it really exists by using `DROP TABLE IF EXISTS test_table;`.

## Inserting and Removing Data

Let's insert some data into our new tables. We can add rows to tables using the `INSERT INTO` command.

```
cur.execute("INSERT INTO StudentInformation VALUES(55, 'John Smith', 2);")
```

After running this statement, we will have the table in Table 4.6.

Note that SQLite will assume that values match sequentially with the schema of the table. We can also specify the schema of the table to use in the mapping of the values.

StudentID	Name	MajorCode
55	John Smith	2

Table 4.6: StudentInformation

```
cur.execute("INSERT INTO StudentInformation(MajorCode, Name, StudentID) VALUES←
(55, 'John Smith', 2);")
```

This will map the value 55 to MajorCode and the value 2 to StudentID. This may be useful sometimes.

It can quickly become tedious to insert large amounts of data into a table, one row at a time. We can automate the process somewhat by using the `executemany` method of the cursor object. To insert several rows into a table using a single command, we can do the following:

```
cur.executemany("INSERT INTO StudentInformation VALUES (?, ?, ?, ?);", rows)
```

In the code above, we assume that `rows` is a Python list of tuples, each tuple containing the data for one row.

We may remove rows from a table using the `DELETE FROM` command.

```
cur.execute("DELETE FROM StudentInformation WHERE MajorCode=55;")
```

### ACHTUNG!

**Never** use Python's string operations to construct a SQL query. It is extremely insecure and is an easy target for a well known type of database called a SQL injection attack.

Parameter substitution can be used to construct dynamic queries. In the simplest way, it involves using a '?' character whenever you want to use a value and providing a sequence of values as a second argument to `execute()`.

```
statement = "INSERT INTO StudentInformation VALUES(?, ?, ?, ?);"
values = (55, 'John Smith', 372897382, 2)
cur.execute(statement, values)
```

**Problem 2.** The ICD is a large collection of codes used to classify any diagnosis that a doctor would make. When someone goes to the hospital or doctors office, their visit will be recorded using these codes. Insurance companies, the government, and researchers find this data useful. The data file provided to you, `icd9.csv`, has simulated health histories for one million persons. Each line has columns for identification number, gender, and age, followed by ICD-9 codes of various quantities. Note that the codes for each individual are written in a single string, each code separated by semicolons. Create a new database with a single table to store all the simulated data. Call the database “sql2” and the table “ICD.” Your table should have four columns, one each for id number, gender, age, and codes.

Because of the volume of data, it is highly recommended you use the `executemany()` method of the cursor. It will be about twice as fast as using an `execute()` for each line of the CSV file. Recall the `csv` package in Python. To read a CSV file into a list of tuples, where each tuple consists of the delimited values of a particular line in the file, one can use the following code as a guideline:

```
import csv
with open('filename', 'rb') as csvfile:
    rows = [row for row in csv.reader(csvfile, delimiter=',')]
```

Hint: Don't forget to commit and close your database.

**Problem 3.** Create the following tables in the same database you created in Problem 1 (“sql1”). You may do so however you think is best.

StudentID	Name	MajorCode
401767594	Michelle Fernandez	1
678665086	Gilbert Chapman	1
553725811	Roberta Cook	2
886308195	Rene Cross	3
103066521	Cameron Kim	4
821568627	Mercedes Hall	3
206208438	Kristopher Tran	2
341324754	Cassandra Holland	1
262019426	Alfonso Phelps	3
622665098	Sammy Burke	2

Table 4.7: StudentInformation

ID	Name
1	Math
2	Science
3	Writing
4	Art

Table 4.8: MajorInfo

	StudentID	ClassID	Grade
	401767594	4	C
	401767594	3	B-
	678665086	4	A+
	678665086	3	A+
	553725811	2	C
	678665086	1	B
	886308195	1	A
	103066521	2	C
	103066521	3	C-
	821568627	4	D
	821568627	2	A+
	821568627	1	B
	206208438	2	A
	206208438	1	C+
	341324754	2	D-
	341324754	1	A-
	103066521	4	A
	262019426	2	B
	262019426	3	C
	622665098	1	A
	622665098	2	A-

Table 4.9: StudentGrades

ClassID	Name
1	Calculus
2	English
3	Pottery
4	History

Table 4.10: CourseInfo

## Updating Rows of a Table

We can modify records in a table by using the **UPDATE** command.

```
cur.execute("UPDATE StudentInformation SET MajorCode=2, StudentID=55, Name='←
    Jonathan Smith' WHERE StudentID=2;")
```

NOTE

When updating a table, having a sufficient `WHERE` clause is essential. Any record that matches the criteria will be modified. If we omitted the `WHERE` clause, every record in the table would be set to the values given in the example.

## Adding Columns to a Table

After you have created a table, you can add more columns to the table by using the `ALTER TABLE` command. For example, if we wanted to add the column "age" into our students table. We run this command.

```
cur.execute("ALTER TABLE StudentInformation ADD COLUMN age INTEGER;")
```

## Selecting Data From Tables

The process of retrieving data from a table in a database is accomplished by the `SELECT` statement. The `SELECT` statement can be thought of as a very high level set description. For example, to view the contents of an entire table, we simply need to unconditionally select its contents.

```
SELECT * FROM students;
```

This is equivalent to the following set (where  $x$  is a row).

$$\{x : x \in \text{classes}\}$$

We can also select specific columns.

```
SELECT StudentID, Name FROM students;
```

Or we can impose conditions on the selected rows.

```
SELECT StudentID, Name FROM students WHERE MajorCode=1;
```

This query results in the following table (Table 4.11) where the contents are all the students that are math majors.

You can also further refine which rows you select by using the `AND` or `OR` commands. These commands will connect expressions. For example, to get the math and science majors. One could run the command.

```
SELECT StudentID, Name FROM students WHERE MajorCode=1 OR MajorCode=2;
```

Select statements return a *result set*. This is an iterable object. Each row in the object is represented as a tuple of values.

```
cur.execute('SELECT StudentID, Name FROM students WHERE MajorCode=1;')
for student in cur:
    print student
```

StudentID	Name
401767594	Michelle Fernandez
678665086	Gilbert Chapman
341324754	Cassandra Holland

Table 4.11: Selected students who are math majors.

Method	Description
<code>fetchone()</code>	Return a single row from the result set
<code>fetchmany(n)</code>	Return the next $n$ rows from the result set
<code>fetchall()</code>	Return the entire result set

Table 4.12: Fetch methods of a cursor.

We can also use the fetch methods of the returned cursor to extract rows from the result set (Table 4.12).

**Problem 4.** From the ICD9 table you created in Problem 2, how many men between the ages of 25 and 35 are there? How many women between those same ages? Return your answers as a tuple.

## When an Error Occurs

It is important to be able to recover from errors gracefully, especially when working a database. Data integrity in a database is often a critical need. When an error occurs, we need to undo the changes that triggered the error. Fortunately, `sqlite3` reports a variety of errors. These errors and when they are raised is explained in PEP249 (<http://legacy.python.org/dev/peps/pep-0249/>).

**Error** The base class for errors thrown by `sqlite3`. All other errors inherit from this class. Catching this error will catch any error raised.

**InterfaceError** Raised when there is a problem with the interface to the database rather than the database itself.

**DatabaseError** Raised when there is an error with the database itself.

**DataError** Subclass of DatabaseError. Raised when there are errors in the processed data (division by zero, value out of range, etc.).

**OperationalError** Subclass of DatabaseError. Raised for errors related to the database that are not the fault of the programmer. For example, an unexpected disconnect, failure to process a transaction, a memory allocation error during a transaction, etc.

**IntegrityError** Subclass of DatabaseError. Raised when the relational integrity of the database is compromised.

**InternalError** Subclass of DatabaseError. Raised when there is an internal error such as an invalid cursor, out-of-sync transaction, etc.

**ProgrammingError** Subclass of DatabaseError. Raised for programming errors.

**NotSupportedError** Subclass of DatabaseError. Raised when a method is called that is not supported by the database.

The way to gracefully recover from errors is to catch them and handle them accordingly. For example, if any error occurs, with the interface or the database, we immediately rollback the transaction. If no error occurs, commit. We could use if-statements or we could use a try-except block.

```
try:  
    <code>  
    db.commit()  
except sql.Error:  
    db.rollback()
```

Note that rolling back is not needed if we are just performing queries. If we don't change any of the data in the database, there is no need to roll anything back. However, even with queries, there is the potential for errors. You must design your code to handle these errors gracefully.



# 5

# SQL 2 (The SQL Sequel)

**Lab Objective:** *Learn more of the advanced and specialized features of SQL.*

## Database Normalization

Normalizing a database is the process of organizing tables and columns to minimize the amount of redundant information in the database. For example, a non-normalized database might have a table that stores customer contact information and a table that contains all of the products a company has sold. However, they might want to track who buys what products in case they need to contact them later. To do so, they store all the contact information of a particular buyer along with every item they purchased. Now, two tables store the customer contact information. If we needed to update a customer's phone number, we have to update two tables. While that may not be bad for small databases, larger databases would be near impossible to update correctly. The idea of normalizing a database allows us to store all customer contact information in one place in the database. All other tables that might need a customer's name, phone number, or address would reference the contact information table. When an update needs to be performed, we only need to update the contact information table. Then any table that references this information is also automatically up to date.

To properly normalize a database, we need to discuss the types of relations tables might have.

### One to One

This is the simplest relation to model. A single table can be used to express this relation. The relation is between one record and at most one other record. An example of this relationship is an employee and their organization. One employee works at one organization. Another example would be that a driver's license belongs to only one person.

### One to Many

This relationship and its inverse must be modeled with at least two tables. The general approach is to use a unique ID. Note that a relationship that appears one to one may actually be a one to many relationship. Many people will, therefore, use the same unique ID approach on one to one relationships too in the case it turns out to be a one to many relationship. An example of a one to many relationship would be between an department and its employees. The department would receive a unique ID and then each employee in that department would be tagged with that ID.

StudentID	Name	MajorCode	MinorCode
401767594	Michelle Fernandez	1	NULL
678665086	Gilbert Chapman	NULL	NULL
553725811	Roberta Cook	2	1
886308195	Rene Cross	3	1
103066521	Cameron Kim	4	2
821568627	Mercedes Hall	NULL	3
206208438	Kristopher Tran	2	4
341324754	Cassandra Holland	1	NULL
262019426	Alfonso Phelps	NULL	NULL
622665098	Sammy Burke	2	3

Table 5.1: students

ID	Name
1	Math
2	Science
3	Writing
4	Art

Table 5.2: fields

## Many to Many

This relationship requires at least three tables. A many to many relationship can be visualized as two, separate one to many relationships. The records in each of the two tables receive a unique ID. A third table then serves as a map between IDs of table to IDs of the other table. An example of a many to many relationship is doctors and patients. One doctor can have several patients and one patient can have several doctors.

For the rest of the lab, we will be using the following tables: 5.1, 5.2, 5.3, and 5.4.

**Problem 1.** Classify the relations between the various records in these tables: 5.1, 5.2, 5.3, and 5.4.

Classify each relation as either one to one, one to many, or many to many. Identify the tables used in each relationship.

### NOTE

There are instances where you would not want a completely normalized database. Whether to normalize your database depends on your specific needs. Usually, though, the decision to denormalize a database is a last-resort attempt to improve performance.

StudentID	ClassID	Grade
401767594	4	C
401767594	3	B-
678665086	4	NULL
678665086	3	A+
553725811	2	C
678665086	1	NULL
886308195	1	A
103066521	2	C
103066521	3	C-
821568627	4	D
821568627	2	NULL
821568627	1	B
206208438	2	A
206208438	1	C+
341324754	2	D-
341324754	1	NULL
103066521	4	A
262019426	2	B
262019426	3	NULL
622665098	1	A
622665098	2	A-

Table 5.3: grades

ClassID	Name
1	Calculus
2	English
3	Pottery
4	History

Table 5.4: classes

## Joining Tables

We can use SQL to join two or more tables together for a query. This is a very powerful tool. SQLite supports three types of standard table joins.

Joining tables is a common practice to collect data from different parts of the database into a single table. Joins are absolutely essential in a normalized database since data is split between multiple tables.

### Inner Join

This is often the default join operation in SQL. An inner join can be depicted as an intersection of two or more tables. When performing an inner join on tables, the result will only be those records that match across all tables. For example, this join will select all the students' names along with their major as long as the `students.majorcode` matches the `majors.id`. If the `students.majorcode` is missing or null, then they won't be selected.

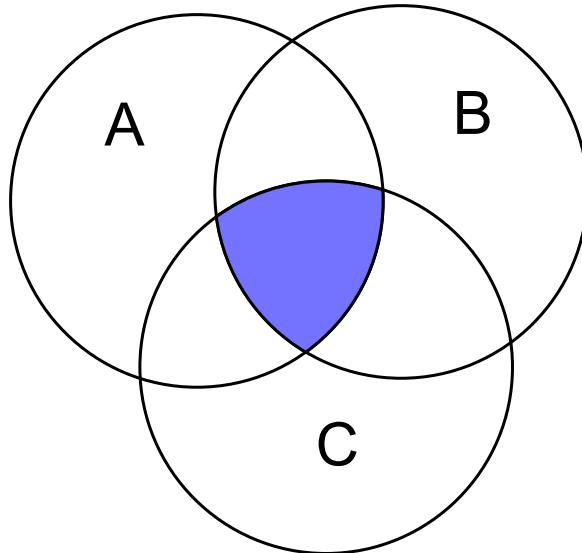


Figure 5.1: An inner joining of tables A, B, and C.

students.name	majors.name
Michelle Fernandez	Math
Roberta Cook	Science
Rene Cross	Writing
Cameron Kim	Art
Kristopher Tran	Science
Cassandra Holland	Math
Sammy Burke	Science

Table 5.5: An inner join of students and majors

```
SELECT students.name, majors.name FROM students JOIN majors ON students.←
majorcode=majors.id;
```

An inner join is equivalent to the following pseudo-loop in Python

```
for row_s in students:
    for row_m in majors:
        if predicates(row_s, row_m):
            yield columns(row_s, row_m)
```

## Left Outer Join

A left outer join will return all relations from the left table even if they don't match any relation on the joined tables. An illustration of a left outer join is given in figure 5.2.

A Pythonesque loop that illustrates how to perform a left outer join is

```
for row_s in students:
```

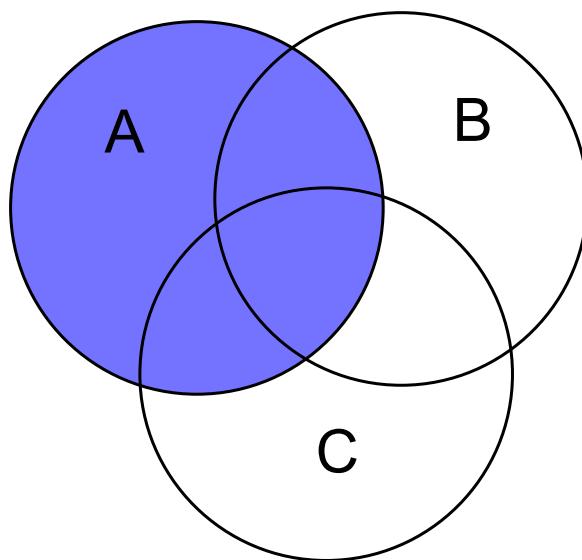


Figure 5.2: A left outer table join of A with tables B and C.

students.name	majors.name
Michelle Fernandez	Math
Gilbert Chapman	None
Roberta Cook	Science
Rene Cross	Writing
Cameron Kim	Art
Meredes Hall	None
Kristopher Tran	Science
Cassandra Holland	Math
Alfonso Phelps	None
Sammy Burke	Science

Table 5.6: A left outer join of students and majors

```

for row_m in majors:
    if predicates(row_s, row_m):
        yield columns(row_s, row_m)
    else:
        yield columns(row_s)

```

The following left outer join will result in the table shown in table 5.6.

```

SELECT students.name, majors.name FROM students LEFT OUTER JOIN majors ON ←
students.majorcode=majors.id;

```

Function	Description
MIN()	Retrieve the smallest numeric value of a column
MAX()	Retrieve the largest numeric value of a column
SUM()	Sum the numeric values of a column
AVG()	Retrieve the average numeric value of the column
COUNT()	Retrieve the total number of matching records in a column

Table 5.7: SQL aggregation functions

## Cross Join

Essentially a Cartesian product of tables. Care must be taken when using cross join because of the size of the joined table. A cross join should only be used on small tables. It matches each relation in one table with every other possible combination of relations in the joined tables. For example, if you were to run a cross join on the above join. You would get every student matched with every major. This doesn't really make much sense but can be useful for other applications. Use with caution.

## Advanced Selections

Aggregate functions are useful for summarizing the data in a column. The functions can be found in 5.7.

We can count the number of students by executing the following SQL statement.

```
SELECT COUNT(*) FROM students;
```

## Ordering and Grouping Relations

The `ORDER BY` keyword can be used to sort the result set by columns. We can sort in ascending order or descending order.

```
SELECT name FROM students ORDER BY name ASC;
SELECT name FROM students ORDER BY name DESC;
```

Another useful SQL keyword is the `GROUP BY` keyword. It is used along with an aggregating function to group the result set by columns.

```
SELECT grade, COUNT(studentid) FROM grades GROUP BY grade;
```

The result set is given in table 5.8.

**Problem 2.** Create a database containing tables 5.1, 5.2, 5.3, and 5.4. Write a SQL query to count how many students belong to each major, including students that don't have a major. Sort your results in ascending order by name. Your result set should be table 5.9

grade	COUNT(studentid)
None	5
A	4
A+	1
A-	1
B	2
B-	1
C	3
C+	1
C-	1
D	1
D-	1

Table 5.8: Grouping of students by grade.

None	3
Art	1
Math	2
Science	3
Writing	1

Table 5.9: Result set

Another important keyword is the `HAVING` keyword. This is necessary because the `WHERE` clause does not support aggregate functions. A `HAVING` clause requires a `GROUP BY` clause. The following will not work.

```
SELECT grade FROM grades GROUP BY grade WHERE COUNT(*)=1;
```

Since `COUNT` is an aggregating function, the following is required.

```
SELECT grade FROM grades GROUP BY grade HAVING COUNT(*)=1;
```

This SQL query returns all the grades that occur only once in the table. A simple way to remember the difference is *WHERE operates on individual records and HAVING operates on groups of records*.

**Problem 3.** Select all the students who have received grades (non-null grades) in more than two classes. How many grades did he receive?

## Case Expression

A case expression allows you to temporarily modify records from a select operation. There are two forms of the case expression; simple and searched. The simple form of the expression is a match and replace on a specified column. A simple case expression is demonstrated below. This code will return the name of the student along with their majorcode. But instead of integers, the names of the majors will be returned.

```

SELECT name,
CASE majorcode
    WHEN 1 THEN 'Math'
    WHEN 2 THEN 'Science'
    WHEN 3 THEN 'Writing'
    WHEN 4 THEN 'Art'
    ELSE 'Undeclared'
END AS major
FROM students;

```

A searched case expression using a boolean expression for the `WHEN` clauses.

```

SELECT name,
CASE
    WHEN majorcode IS NULL THEN 'Undeclared'
    ELSE majorcode
END AS major,
CASE
    WHEN minorcode IS NULL THEN 'Undeclared'
    ELSE minorcode
END AS minor
FROM students;

```

**Problem 4.** Find the overall GPA of all the students in the school. Use a regular 4.0 scale (A=4.0, B=3.0, C=2.0, D=1.0). Any pluses or minuses are dropped so an A- becomes an A.

Your result set should be one column and one row with average of all GPAs of all the students taking classes. Your solution should return a single floating point number.

Use the `ROUND()` function in SQL to round your result to the nearest hundredth.

## Like Command

The SQL keyword, `LIKE`, allows us to match patterns in a column. For example,

```
SELECT name, studentid FROM students WHERE studentid LIKE '\%4';
```

will return all the students that have a student ID that ends with the digit 4. Use '\%' before the '4' to signify that there can be any number of characters before the '4.' If you only want one character, you can use an underscore. For example, if you were to search a database of words in the english dictionary and you entered the following command:

```
SELECT word FROM englishDictionary WHERE word LIKE 'i_';
```

You would get words like 'is,' 'it,' or 'in.'

**Problem 5.** Write a SQL statement that will find all students with a last name that begins with the letter 'C' and return their names and majors. Your returned records should be

Gilbert Chapman	None
Roberta Cook	Science
Rene Cross	Writing



# 6

# Pandas I: Introduction to Pandas

**Lab Objective:** *Become acquainted with the data structures and tools that pandas offers for data analysis.*

In volumes 1 and 2, we solved data problems primarily using NumPy and SciPy. While extremely flexible and useful tools, these libraries lack some of the high-level data-analytic abstractions present in other popular data packages. In particular, *pandas* is a Python library that is more specifically built for data analysis. In this lab, we will give an overview of some of the functions which will prove very useful in this regard.

## Data Structures in pandas

Just as NumPy is built on the `ndarray` data structure suited for efficient scientific and numerical computation, pandas is centered around a handful of core data structures custom built for data analysis. These data structures include the `Series`, `DataFrame`, and `Panel`, which correspond roughly to one, two, and three-dimensional arrays. We will explore the first two data structures in some detail. The interested reader can learn more about the `Panel` data structure (the least-used one in pandas) in the online documentation; we will not delve into this data structure here.

### Series

The `Series` is a one-dimensional array with labeled entries. The values contained may be any data type, including integers, strings, or general Python objects. Further, unlike NumPy arrays, pandas `Series` need not be homogeneous. That is, they can hold values of different data types. Together, the values are referred to as the *data* of the `Series`. The labels must consist of hashable types, and are most commonly integers or strings. Together, the labels are referred to as the *index* of the `Series`. Thus, a `Series` consists of data and an index. The most basic way to initialize such an object is as follows:

```
>>> import pandas as pd  
>>> s = pd.Series(data, index=index)
```

We don't need to explicitly define an index. The default is `np.arange(len(data))`. For example, we can create a `Series` containing the integers from 9 to 0:

```
>>> s1 = pd.Series(np.arange(9, -1, -1))
>>> s1.values      #the data
array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])
>>> s1.index      #the labels
Int64Index([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype='int64')
>>> s1            #left column is index, right column is data
0    9
1    8
2    7
3    6
4    5
5    4
6    3
7    2
8    1
9    0
dtype: int64
```

Here is an example where we create customized labels:

```
>>> import numpy as np
>>> data = np.random.randn(3)
>>> index = ['first', 'second', 'third']
>>> s2 = pd.Series(data, index=index)
>>> s2
first    1.661255
second   -0.033570
third    -2.185991
dtype: float64
```

We can create a `Series` having constant values in the following manner:

```
>>> val = 4      #desired constant value of Series
>>> n = 6       #desired length of Series
>>> s3 = pd.Series(val, index=np.arange(n))
>>> s3
0    4
1    4
2    4
3    4
4    4
5    4
dtype: int64
```

It is also possible to use a Python dictionary to create a `Series`:

```
>>> d = {'e1': 93, 'e2': 95, 'e3': 87, 'e4': 82, 'e5': 94}
>>> s4 = pd.Series(d)
```

```
>>> s4
e1    93
e2    95
e3    87
e4    82
e5    94
dtype: int64
```

Note that we didn't need to specify the index; the keys of the dictionary are used as the index for the `Series`. There are many more ways to create `Series` objects. For a more complete discussion of how to create `Series` objects, see the online documentation.

**Problem 1.** Create the following pandas `Series`.

- Constant array with value -3, length 5. The index labels should be the first five positive even integers.
- Data is given by the dictionary:  
`{'Bill': 31, 'Sarah': 28, 'Jane': 34, 'Joe': 26}`.

## DataFrame

The `DataFrame` data structure is a two-dimensional generalization of the `Series`. It can be viewed as a tabular structure with labeled rows and columns. The row labels are collectively called the index, and the column labels are collectively called the columns. An individual column in a `DataFrame` object is a `Series`.

There are many ways to initialize a `DataFrame`. In the following code, we build a `DataFrame` out of a dictionary of `Series`:

```
>>> x = pd.Series(np.random.randn(4), ['a', 'b', 'c', 'd'])
>>> y = pd.Series(np.random.randn(5), ['a', 'b', 'd', 'e', 'f'])
>>> d = {1: x, 2: y}
>>> df1 = pd.DataFrame(d)
>>> df1
      1          2
a -0.924259 -0.708301
b  0.767422 -2.214516
c  0.399212      NaN
d  0.130365 -2.352364
e        NaN   0.789419
f        NaN  -0.859482
```

Note that the index of this `DataFrame` is the union of the index of `Series` `x` and that of `Series` `y`. The columns are given by the keys of the dictionary `d`. Since `x` doesn't have a label `e`, the value in row `e`, column 1 is `NaN`. This same reasoning explains the other missing values as well. Note that if we take the first column of the `DataFrame` and drop the missing values, we recover the `Series` `x`:

```
>>> x == df1[1].dropna()
a    True
b    True
c    True
d    True
dtype: bool
```

### ACHTUNG!

A Pandas `DataFrame` cannot be sliced the same way a NumPy array could. Notice how we just used `df1[1]` to access the first *column* of the the `DataFrame` `df1`. We will discuss this in more detail later on.

We can also initialize a `DataFrame` using a NumPy array, creating custom row and column labels:

```
>>> data = np.random.random((3, 4))
>>> pd.DataFrame(data, index=['A', 'B', 'C'], columns=np.arange(1, 5))

          1         2         3         4
A  0.065646  0.968593  0.593394  0.750110
B  0.803829  0.662237  0.200592  0.137713
C  0.288801  0.956662  0.817915  0.951016
3 rows    4 columns
```

As with Series, if we don't specify the index or columns, the default is `np.range(n)`, where `n` is either the number of rows or columns.

It is also possible to create multi-indexed arrays, for example:

```
>>> grade=['eighth', 'ninth', 'tenth']
>>> subject=['math', 'science', 'english']
>>> myindex = pd.MultiIndex.from_product([grade, subject], names=['grade', 'subject'])
>>> myseries = pd.Series(np.random.randn(9), index=myindex)
>>> myseries
grade   subject
eighth  math      1.706644
          science   -0.899587
          english   -1.009832
ninth   math      2.096838
          science   1.884932
          english   0.413266
tenth   math     -0.924962
          science   -0.851689
          english   1.053329
dtype: float64
```

Multi-indexing is visually convenient, and will be explored further in Lab 8, where we will discuss pandas pivot tables.

## Data I/O

Being able to import and export data is a fundamental skill in data science. Unfortunately, with the multitude of data formats and conventions out there, importing data can often be a tricky task. The pandas library seeks to reduce some of the difficulty by providing file readers for various types of formats, including CSV, Excel, HDF5, SQL, JSON, HTML, and pickle files.

The CSV (comma separated values) format is a simple way of storing tabular data in plain text. Because CSV files are one of the most popular file formats for exchanging data, we will explore the `read_csv()` function in more detail. To learn to read other types of file formats, see the online pandas documentation. To read a CSV data file into a `DataFrame`, call the `read_csv()` function with the path to the CSV file, along with the appropriate keyword arguments. Below we list some of the most important keyword arguments:

- **delimiter**: This argument specifies the character that separates data fields, often a comma or a whitespace character.
- **header**: The row number (starting at 0) in the CSV file that contains the column names.
- **index\_col**: If you want to use one of the columns in the CSV file as the index for the `DataFrame`, set this argument to the desired column number.
- **skiprows**: If an integer  $n$ , skip the first  $n$  rows of the file, and then start reading in the data. If a list of integers, skip the specified rows.
- **names**: If the CSV file does not contain the column names, or you wish to use other column names, specify them in a list assigned to this argument.

There are several other keyword arguments, but this should be enough to get you started.

When you need to save your data, pandas allows you to write to several different file formats. A typical example is the `to_csv()` function method attached to `Series` and `DataFrame` objects, which writes the data to a CSV file. Keyword arguments allow you to specify the separator character, omit writing the columns names or index, and specify many other options. The code below demonstrates its typical usage:

```
>>> df.to_csv("my_df.csv")
```

## Viewing and Accessing Data

Once we have our data ready to go in pandas, how can we interact with it? In this section we will explore some elementary accessing and querying techniques that enable us to maneuver through and gain insight into our data.

### Basic Data Access

Some of the basic slicing paradigms in NumPy carry over to pandas. For example, we can slice a `Series` using the usual syntax:

```
>>> s = pd.Series(np.random.randn(5))
>>> s[1:3]

1    3.188112
2    0.080191
dtype: float64
```

Notice that both the data and the index are sliced in this manner.

Likewise, we can slice the rows of a `DataFrame` much as with a NumPy array:

```
>>> df = pd.DataFrame(np.random.randn(4, 2), index=['a', 'b', 'c', 'd'],
                      columns = ['I', 'II'])
>>> df[:2]

           I      II
a  0.758867  1.231330
b  0.402484 -0.955039

[2 rows x 2 columns]
```

More generally, we can select subsets of the data using the `.iloc` and `.loc` indexers. The `.loc` index selects rows and columns based on their *labels*, while the `.iloc` method selects them based on their integer *position*. Accessing `Series` and `DataFrame` objects using these indexing operations is more efficient than using bracket indexing, because the bracket indexing has to check many cases before it can determine how to slice the data structure. By using `loc/iloc` explicitly, you bypass the extra checks.

```
>>> # select rows a and c, column II
>>> df.loc[['a','c'], 'II']

a    1.231330
c    0.556121
Name: II, dtype: float64

>>> # select last two rows, first column
>>> df.iloc[-2:, 0]

c   -0.475952
d   -0.518989
Name: I, dtype: float64
```

Finally, a column of a `DataFrame` may be accessed using simple square brackets and the name of the column, or alternatively by treating the label as an object:

```
>>> # get second column of df
>>> df['II']

a    1.231330
```

```

b    -0.955039
c     0.556121
d     0.173165
Name: II, dtype: float64

>>> # equivalent result to above
>>> df.II

a    1.231330
b    -0.955039
c     0.556121
d     0.173165
Name: II, dtype: float64

```

All of these techniques for getting subsets of the data may also be used to set subsets of the data:

```

>>> # set second columns to zeros
>>> df['II'] = 0
>>> df['II']

a    0
b    0
c    0
d    0
Name: II, dtype: int64

>>> # add additional column of ones
>>> df['III'] = 1
>>> df

      I   II   III
a  -0.460457   0     1
b   0.973422   0     1
c  -0.475952   0     1
d  -0.518989   0     1

```

## Plotting

Plotting is often a much more effective way to view and gain understanding of a dataset than simply viewing the raw numbers. Fortunately, pandas interfaces well with matplotlib, allowing relatively painless data visualization.

Most of the functionality of plotting using pandas will be discussed in Lab 7. In this lab, we will simply show how to plot a `Series`. Doing so is easy, as the `Series` object is equipped with its own plot function.

Let's start with visualizing a simple random walk. By way of background, a *random walk* is a stochastic process used to model a non-deterministic path through some space. It is a useful construct in many fields and can be used to explain things like the motion of a molecule as it travels through a liquid to modeling the fluctuations of stock prices. Here we will simulate a one-dimensional symmetric random walk on the integers, which can be described as follows.

1. Start at 0.
2. Flip a fair coin.
3. If heads, move one unit to the right. Otherwise, move one unit to the left.
4. Go to Step 2.

How can we simulate this random walk efficiently? Note that the walk is really characterized by the outcomes of the coin flip. If we represent heads by the number 1 and tails by  $-1$ , then our position at a given moment is just the cumulative sum of all previous outcomes. Below, we simulate a sequence of coin flips, build the resulting random walk, and plot the outcome.

```
>>> import matplotlib.pyplot as plt
>>> N = 1000      # length of random walk
>>> s = np.zeros(N)
>>> s[1:] = np.random.binomial(1, .5, size=(N-1,))*2-1 #coin flips
>>> s = pd.Series(s)
>>> s = s.cumsum() # random walk
>>> s.plot()
>>> plt.ylim([-50, 50])
>>> plt.show()
>>> plt.close()
```

The random walk is shown in Figure 6.1.

**Problem 2.** Create five random walks of length 100, and plot them together in the same plot.

Next, create a “biased” random walk by changing the coin flip probability of head from 0.5 to 0.51. Plot this biased walk with lengths 100, 10000, and then 100000. Notice the definite trend that emerges. Your results should be comparable to those in Figure 6.2.

## SQL Operations in pandas

The `DataFrame`, being a tabular data structure, bears an obvious resemblance to a typical relational database table. SQL is the standard for working with relational databases, and in this section we will explore how pandas accomplishes some of the same tasks as SQL. The SQL-like functionality of pandas is one of its biggest advantages, since it can eliminate the need to switch between programming languages for different tasks. Within pandas we can handle both the querying *and* data analysis.

For the following examples, we will use the following data:

```
>>> #build toy data for SQL operations
>>> name = ['Mylan', 'Regan', 'Justin', 'Jess', 'Jason', 'Remi', 'Matt', '←
    Alexander', 'JeanMarie']
```

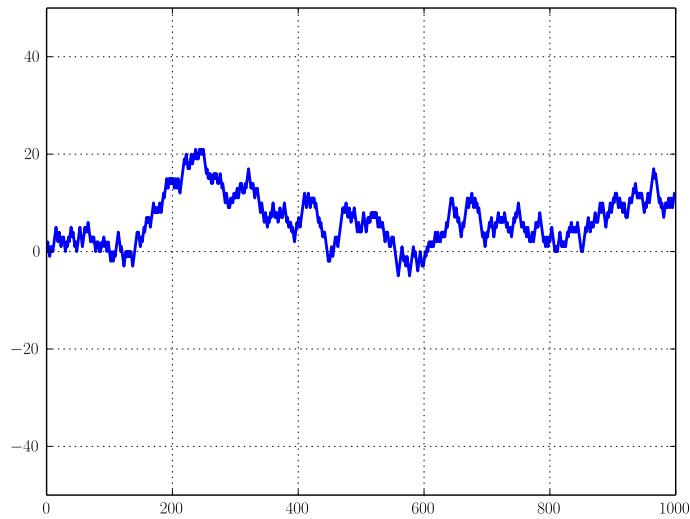


Figure 6.1: Random walk of length 1000.

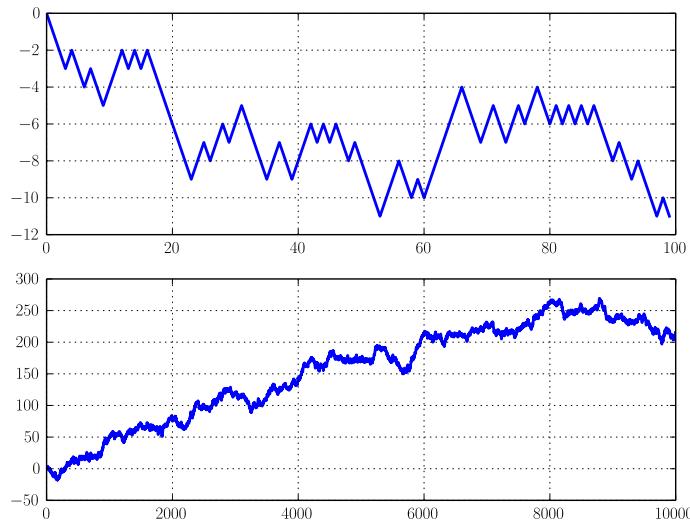


Figure 6.2: Biased random walk of length 100 (above) and 10000 (below).

```
>>> sex = ['M', 'F', 'M', 'F', 'M', 'F', 'M', 'M', 'F']
>>> age = [20, 21, 18, 22, 19, 20, 20, 19, 20]
>>> rank = ['Sp', 'Se', 'Fr', 'Se', 'Sp', 'J', 'J', 'J', 'Se']
>>> ID = range(9)
>>> aid = ['y', 'n', 'n', 'y', 'n', 'n', 'n', 'y', 'n']
>>> GPA = [3.8, 3.5, 3.0, 3.9, 2.8, 2.9, 3.8, 3.4, 3.7]
```

```
>>> mathID = [0, 1, 5, 6, 3]
>>> mathGd = [4.0, 3.0, 3.5, 3.0, 4.0]
>>> major = ['y', 'n', 'y', 'n', 'n']
>>> studentInfo = pd.DataFrame({'ID': ID, 'Name': name, 'Sex': sex, 'Age': age, 'Class': rank})
>>> otherInfo = pd.DataFrame({'ID': ID, 'GPA': GPA, 'Financial_Aid': aid})
>>> mathInfo = pd.DataFrame({'ID': mathID, 'Grade': mathGd, 'Math_Major': major})
```

Before querying our data, it is important to know some of its basic properties, such as number of columns, number of rows, and the datatypes of the columns. This can be done by simply calling the `info()` method on the desired `DataFrame`:

```
>>> mathInfo.info()
<class 'pandas.core.frame.DataFrame'>
Int64Index: 5 entries, 0 to 4
Data columns (total 3 columns):
Grade      5 non-null float64
ID         5 non-null int64
Math_Major 5 non-null object
dtypes: float64(1), int64(1), object(1)
```

We can also get some basic information about the structure of the `DataFrame` using the `head()` or `tail()` methods.

```
>>> mathInfo.head()
   Grade  ID Math_Major  ID  Age  GPA
0    4.0    0          y  0   20  3.8
1    3.0    1          n  2   18  3.0
2    3.5    5          y  4   19  2.8
3    3.0    6          n  6   20  3.8
4    4.0    3          n  7   19  3.4
```

The method `isin()` is a useful way to find certain values in a `DataFrame`. It compares the input (a list, dictionary, or `Series`) to the `DataFrame` and returns a boolean `Series` showing whether or not the values match. You can then use this boolean array to select appropriate locations. Now let's look at the pandas equivalent of some SQL SELECT statements.

```
>>> # SELECT ID, Age FROM studentInfo
>>> studentInfo[['ID', 'Age']]

>>> # SELECT ID, GPA FROM otherInfo WHERE Financial_Aid = 'y'
>>> otherInfo[otherInfo['Financial_Aid']=='y'][['ID', 'GPA']]

>>> # SELECT Name FROM studentInfo WHERE Class = 'J' OR Class = 'Sp'
>>> studentInfo[studentInfo['Class'].isin(['J', 'Sp'])]['Name']
```

**Problem 3.** The example above shows how to implement a simple WHERE condition, and it is easy to have a more complex expression. Simply enclose each condition by parentheses, and use the standard boolean operators & (AND), | (OR), and ~ (NOT) to connect the conditions appropriately. Use pandas to execute the following query:

```
SELECT ID, Name from studentInfo WHERE Age > 19 AND Sex = 'M'
```

Next, let's look at JOIN statements. In pandas, this is done with the `merge` function, which takes as arguments the two `DataFrame` objects to join, as well as keyword arguments specifying the column on which to join, along with the type (left, right, inner, outer).

```
>>> # SELECT * FROM studentInfo INNER JOIN mathInfo ON studentInfo.ID = ←
      mathInfo.ID
>>> pd.merge(studentInfo, mathInfo, on='ID') # INNER JOIN is the default
      Age Class ID Name Sex Grade Math_Major
0   20   Sp  0 Mylan M  4.0      y
1   21   Se  1 Regan F  3.0      n
2   22   Se  3 Jess  F  4.0      n
3   20     J  5 Remi  F  3.5      y
4   20     J  6 Matt  M  3.0      n
[5 rows x 7 columns]

>>> # SELECT GPA, Grade FROM otherInfo FULL OUTER JOIN mathInfo ON otherInfo.ID←
      = mathInfo.ID
>>> pd.merge(otherInfo, mathInfo, on='ID', how='outer')[['GPA', 'Grade']]
      GPA Grade
0  3.8  4.0
1  3.5  3.0
2  3.0  NaN
3  3.9  4.0
4  2.8  NaN
5  2.9  3.5
6  3.8  3.0
7  3.4  NaN
8  3.7  NaN
[9 rows x 2 columns]
```

**Problem 4.** Using a join operation, create a `DataFrame` containing the ID, age, and GPA of all male individuals. You ought to be able to accomplish this in one line of code.

Be aware that other types of SQL-like operations are also possible, such as UNION. When you find yourself unsure of how to carry out a more involved SQL-like operation, the online pandas documentation will be of great service.

## Analyzing Data

Although pandas does not provide built-in support for heavy-duty statistical analysis of data, there are nevertheless many features and functions that facilitate basic data manipulation and computation, even when the data is in a somewhat messy state. We will now explore some of these features.

### Basic Data Manipulation

Because the primary pandas data structures are subclasses of the `ndarray`, they are valid input to most NumPy functions, and can often be treated simply as NumPy arrays. For example, basic vectorized operations work just fine:

```
>>> x = pd.Series(np.random.randn(4), index=['a', 'b', 'c', 'd'])
>>> y = pd.Series(np.random.randn(5), index=['a', 'b', 'd', 'e', 'f'])
>>> x**2
a    1.710289
b    0.157482
c    0.540136
d    0.202580
dtype: float64
>>> z = x + y
>>> z
a    0.123877
b    0.278435
c        NaN
d   -1.318713
e        NaN
f        NaN
dtype: float64
>>> np.log(z)
a    -2.088469
b   -1.278570
c        NaN
d        NaN
e        NaN
f        NaN
dtype: float64
```

Notice that pandas automatically aligns the indexes when adding two `Series` (or `DataFrames`), so that the the index of the output is simply the union of the indexes of the two inputs. The default missing value `NaN` is given for labels that are not shared by both inputs.

It may also be useful to transpose `DataFrames`, re-order the columns or rows, or sort according to a given column. Here we demonstrate these capabilities:

```
>>> df = pd.DataFrame(np.random.randn(4,2), index=['a', 'b', 'c', 'd'], columns=['I', 'II'])
>>> df
           I         II
a -0.154878 -1.097156
```

```

b -0.948226  0.585780
c  0.433197 -0.493048
d -0.168612  0.999194

[4 rows x 2 columns]

>>> df.transpose()
      a          b          c          d
I -0.154878 -0.948226  0.433197 -0.168612
II -1.097156  0.585780 -0.493048  0.999194

[2 rows x 4 columns]

>>> # switch order of columns, keep only rows 'a' and 'c'
>>> df.reindex(index=['a', 'c'], columns=['II', 'I'])
      II          I
a -1.097156 -0.154878
c -0.493048  0.433197

[2 rows x 2 columns]

>>> # sort descending according to column 'II'
>>> df.sort_values('II', ascending=False)
      I          II
d -0.168612  0.999194
b -0.948226  0.585780
c  0.433197 -0.493048
a -0.154878 -1.097156

[4 rows x 2 columns]

```

## Basic Statistical Functions

The pandas library allows us to easily calculate basic summary statistics of our data, useful when we want a quick description of the data. The `describe()` function outputs several such summary statistics for each column in a `DataFrame`:

```

>>> df.describe()
      I          II
count  4.000000  4.000000
mean   -0.209630 -0.001308
std    0.566696  0.964083
min   -0.948226 -1.097156
25%   -0.363516 -0.644075
50%   -0.161745  0.046366
75%   -0.007859  0.689133
max    0.433197  0.999194

```

```
[8 rows x 2 columns]
```

Functions for calculating means and variances, the covariance and correlation matrices, and other basic statistics are also available. Below, we calculate the means of each row, as well as the covariance matrix:

```
>>> df.mean(axis=1)
a    -0.626017
b    -0.181223
c    -0.029925
d     0.415291
dtype: float64

>>> df.cov()
          I          II
I  0.321144 -0.256229
II -0.256229  0.929456

[2 rows x 2 columns]
```

## Dealing with Missing Data

Missing data is a ubiquitous problem in data science. Fortunately, pandas is particularly well-suited to handling missing and anomalous data. As we have already seen, the pandas default for a missing value is `NaN`. In basic arithmetic operations, if one of the operands is `NaN`, then the output is also `NaN`. The following example illustrates this concept:

```
>>> x = pd.Series(np.arange(5))
>>> y = pd.Series(np.random.randn(5))
>>> x.iloc[3] = np.nan
>>> x + y
0    0.731521
1    0.623651
2    2.396344
3    NaN
4    3.351182
dtype: float64
```

If we are not interested in the missing values, we can simply drop them from the data altogether:

```
>>> (x + y).dropna()
0    0.731521
1    0.623651
2    2.396344
4    3.351182
dtype: float64
```

This is not always the desired behavior, however. It may well be the case that missing data actually corresponds to some default value, such as zero. In this case, we can replace all instances of `NaN` with a specified value:

```
>>> # fill missing data with 0, add  
>>> x.fillna(0) + y  
0    0.731521  
1    0.623651  
2    2.396344  
3    1.829400  
4    3.351182  
dtype: float64
```

Other functions, such as `sum()` and `mean()` treat `NaN` as zero by default. When dealing with missing data, make sure you are aware of the behavior of the pandas functions you are using.

**Problem 5.** Using the dataset contained in the file `crime_data.txt` and the techniques learned in this lab, use pandas to complete the following.

- Load the data into a pandas `DataFrame`, using the column names in the file and the column titled “Year” as the index. Make sure to skip lines that don’t contain data.
- Insert a new column into the data frame that contains the crime rate by year (the ratio of “Total” column to the “Population” column).
- Plot the crime rate as a function of the year.
- List the 5 years with the highest crime rate in descending order.
- Calculate the average number of total crimes as well as burglary crimes between 1960 and 2012.
- Find the years for which the total number of crimes was below average, but the number of burglaries was above average.
- Plot the number of murders as a function of the population.
- Select the Population, Violent, and Robbery columns for all years in the 1980s, and save this smaller data frame to a CSV file `crime_subset.txt`.



# 7

## Pandas II: Plotting with Pandas

**Lab Objective:** *Pandas has many built-in plotting methods that wrap around matplotlib. Since pandas provides tools for organizing and correlating large sets of data, it is important to be able to visualize these relationships. First, we will go over different types of plots offered by pandas, and then some techniques we can use to visualize data in useful ways.*

### Plotting Data Frames

Recall from Lab 6 that in Pandas, a *DataFrame* is an ordered collection of *Series*. A *Series* is similar to a dictionary, with values assigned to various labels, or indices. Each *Series* becomes a column in the data frame, with each row corresponding to an index. When several *Series* are combined into a single data frame, it becomes very easy to compare and visualize data. Each entry has an associated index and column.

*DataFrames* have several methods for easy plotting. Most are simple wrappers around *matplotlib* commands, but some produce styles of plots that we have not previously seen. In this lab, we will go over seven different types of plots offered by *pandas*.

For these examples, we will use the data found in the file `crime_data.txt`.

```
>>> import pandas as pd  
>>> crime = pd.read_csv("crime_data.txt", header=1, index_col=0)
```

### Line Plots

In Lab 6, we showed how to plot a *Series* against its index (the years, in this case) using *matplotlib*. With a few extra lines we modify the *x*-axis label and limits and create a legend.

```
>>> from matplotlib import pyplot as plt  
>>> plt.plot(crime["Population"], label="Population")  
>>> plt.xlabel("Year")  
>>> plt.xlim(min(crime.index), max(crime.index))  
>>> plt.legend(loc="best")  
>>> plt.show()
```

Equivalently, we can produce the exact same plot with a single line using the `DataFrame` method `plot()`. Specify the *y*-values as a keyword argument. The *x*-values default to the index of the Series. See Figure 7.1.

```
>>> crime.plot(y="Population")
>>> plt.show()
```

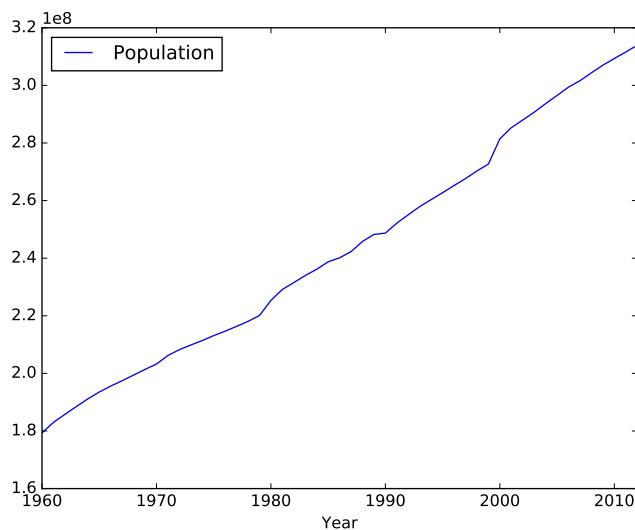


Figure 7.1: Population by Year.

We can also plot two Series against each other (ignoring the index). In matplotlib:

```
>>> plt.plot(crime["Population"], crime["Burglary"])
>>> plt.show()
```

With `DataFrame.plot()`, specify the *x*-values as a keyword argument:

```
>>> crime.plot(x="Population", y="Burglary")
>>> plt.show()
```

Both procedures produce the same line plot, but the `DataFrame` method automatically sets the limits and labels of each axis and includes a legend. See Figure 7.2.

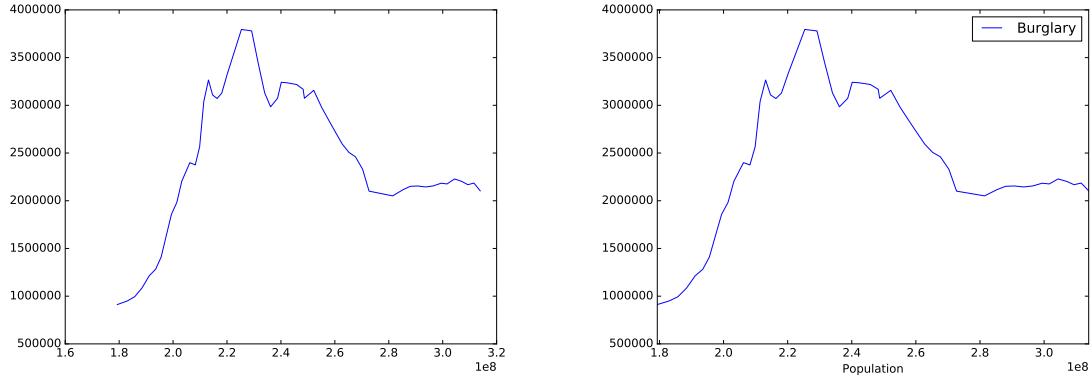


Figure 7.2: On the left, the result of `plt.plot()`. On the right, the result of `DataFrame.plot()`

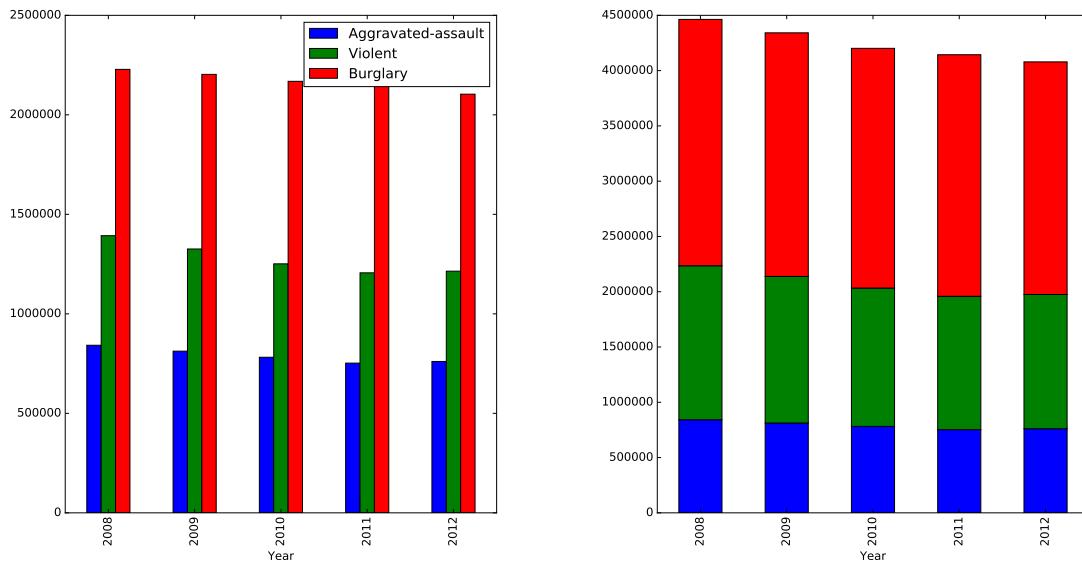
Standard matplotlib keyword arguments can be passed in as usual to `DataFrame.plot()`. This includes the ability to produce subplots quickly, modify the linestyle, and so on.

```
>>> crime.plot(subplots=True, layout=(4,3), linewidth=3, style="--", legend=False)
>>> plt.show()
```

## Bar Plots

By default, the `DataFrame`'s `plot()` function creates a line plot. We can create other types of plots easily by specifying the keyword `kind`. Bar plots are particularly useful for comparing several categories of data over time, or whenever there is a sense of progression in the index. Line plots are better suited to show a more continuous index (such as each year in a century), whereas bar plots are better for a discrete index (a few distinct years). Consider, for example, three different types of crime over the last five years. The argument `stacked` defaults to `False` (and `legend` to `True`). Stacking the bars can help to show the combined totals each year. Both types are shown below:

```
# Each call to plot() makes a separate figure automatically.
>>> crime.iloc[-5:][["Aggravated-assault", "Violent", "Burglary"]].plot(kind="bar")
>>> crime.iloc[-5:][["Aggravated-assault", "Violent", "Burglary"]].plot(kind="bar",
>>> stacked=True, legend=False)
>>> plt.show()
```



**Problem 1.** The `pydataset` module<sup>a</sup> contains numerous data sets stored as pandas DataFrames.

```
from pydataset import data
# "data" is a pandas DataFrame with IDs and descriptions.
# Call data() to see the entire list.
# To load a particular data set, enter its ID as an argument to data().
titanic_data = data("Titanic")
# To see the information about a data set, give data() the dataset_id with
# show_doc=True.
data("Titanic", show_doc=True)
```

Examine the data sets with the following `pydataset` IDs:

1. `nottem`: Average air temperatures at Nottingham Castle in Fahrenheit for 20 years.
2. `VADeaths`: Death rates per 1000 in Virginia in 1940.
3. `Arbuthnot`: Ratios of male to female births in London from 1629-1710.

Use line and bar plots to visualize each of these data sets. Decide which type of plot is more appropriate for each data set, and which columns to plot together. Write a short description of each data set based on the docstrings of the data and your visualizations.

<sup>a</sup>Run `pip install pydataset` if needed

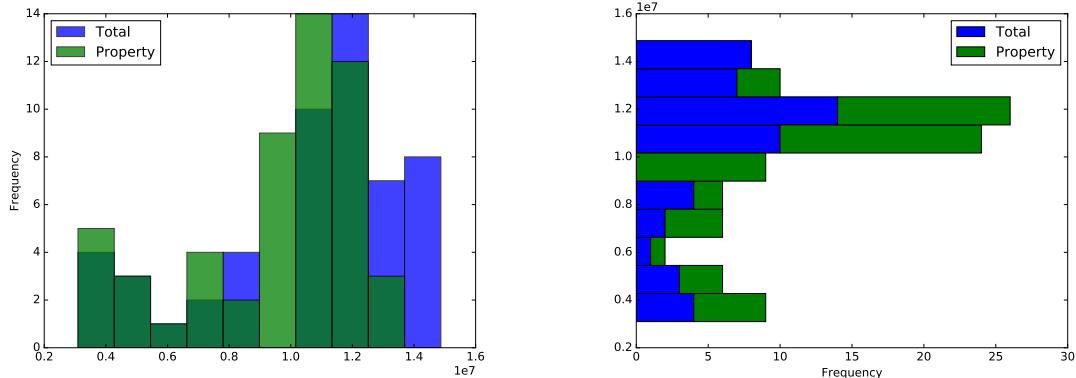
## Histograms

Line and bar plots work well when there is a logical progression in the index, such as time. However, when frequency of occurrence is more important than the location of the data, histograms and box plots can be more informative. Use `plot(kind="hist")` to produce a histogram. Standard histogram options, such as the number of bins, are also accepted as keyword arguments. The `alpha` keyword argument makes each bin slightly transparent.

```
>>> crime[["Total", "Property"]].plot(kind="hist", alpha=.75)
>>> plt.show()
```

Alternatively, the bins can be stacked on top of each other by setting the `stacked` keyword argument to `True`. We can also make the histogram horizontal by setting the keyword `orientation` to “horizontal”.

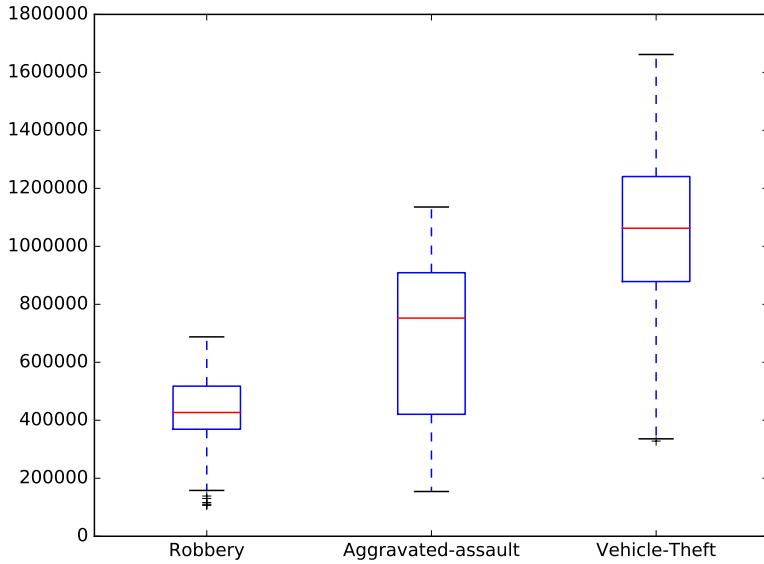
```
>>> crime[["Total", "Property"]].plot(kind="hist", stacked=True, orientation="horizontal")
>>> plt.show()
```



## Box Plots

Sometimes it is helpful to visualize a distribution of values using the box-and-whisker plot which displays the median, first and third quartiles, and outliers. Like the previous examples, select the columns to examine and plot them with `plot()`. To switch the orientation, use `vert=False`.

```
crime[["Robbery", "Aggravated-assault", "Vehicle-Theft"]].plot(kind="box")
plt.show()
```



**Problem 2.** Examine the data sets with the following pydataset IDs:

1. `trees`: Girth, height and volume for black cherry trees.
2. `road`: Road accident deaths in the United States.
3. `birthdeathrates`: Birth and death rates by country.

Use histograms and box plots to visualize each of these data sets. Decide which type of plot is more appropriate for each data set, and which columns to plot together. Write a short description of each data set based on the docstrings of the data and your visualizations.

## Scatter Plots

Scatter plots are commonly used in a myriad of areas and have a simple implementation in pandas. Unlike other plotting commands, `scatter` needs both an `x` and a `y` column as arguments.

The scatter plot option includes many features which can be used to make the plots easier to understand. For example, we can change the size of the point based on another column. Consider the pydataset `HairEyeColor`, which contains the hair and eye color of various individuals. A scatter plot of hair color vs eye color is relatively useless unless we can see the frequencies with which each combination occurs. Including the keyword argument `s` allows us to control the size of each point. This can be set to a fixed value or the value in another column. In the example below, the size of each point is set to the frequency with which each observation occurs.

```
>>> hec = data("HairEyeColor")
>>> X = np.unique(hec["Hair"], return_inverse=True)
>>> Y = np.unique(hec["Eye"], return_inverse=True)
```

```
>>> hec["Hair"] = X[1]
>>> hec["Eye"] = Y[1]
>>> hec.plot(kind="scatter", x="Hair", y="Eye", s=hec["Freq"]*20)
>>> plt.xticks([0,1,2,3], X[0])
>>> plt.yticks([0,1,2,3], Y[0])
>>> plt.show()
```

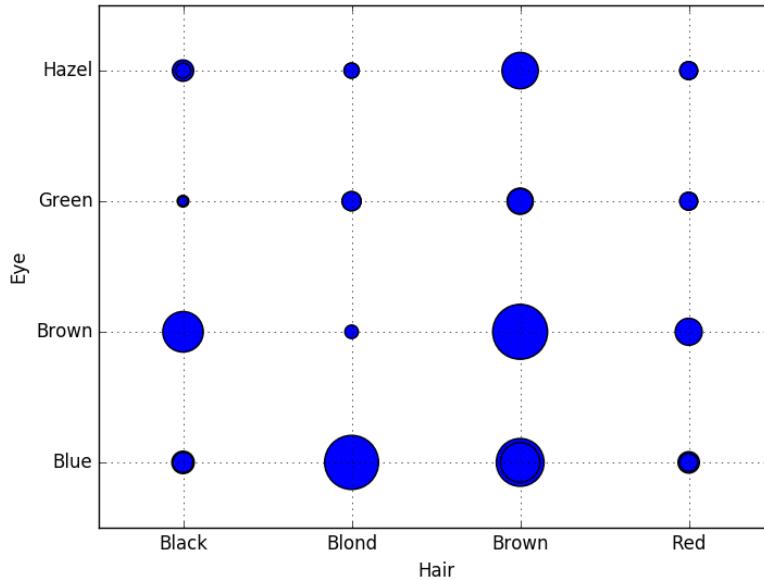


Figure 7.3: Frequency of Hair-Eye Color Combinations

## Hexbins

While scatter plots are a great visualization tool, they can be uninformative for large datasets. It is nearly impossible to tell what is going on in a large scatter plot, and the visualization is therefore of little value. Hexbin plots solve this problem by plotting point density in hexagonal bins. With hexbins, the structure of the data is easy to see despite the noise that is still present. Following is an example using pydataset's `sat.act` plotting the SAT Quantitative score vs ACT score of students

```
>>> satact = data("sat.act")
>>> satact.plot(kind="scatter", x="ACT", y="SATQ")
>>> satact.plot(kind="Hexbin", x="ACT", y="SATQ", gridsize=20)
>>> plt.show()
```

Note the scatter plot in Figure 7.4. While we can see the structure of the data, it is not easy to differentiate the densities of the areas with many data points. Compare this now to the hexbin plot in the same figure. The two plots are clearly similar, but with the hexbin plot we have the added dimension of density.

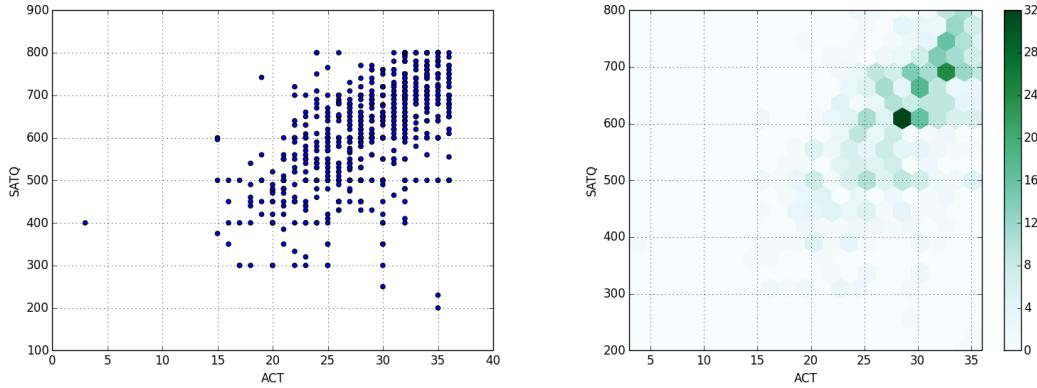


Figure 7.4: Comparing a scatter plot of SAT and ACT scores with a hexbin plot.

A key factor in creating an informative hexbin is choosing an appropriate `gridsize` parameter. This determines how large or small the bins will be. A large gridsize will give many small bins and a small gridsize gives a few large bins. Figure 7.5 shows the effect of changing the gridsize from 20 to 10.

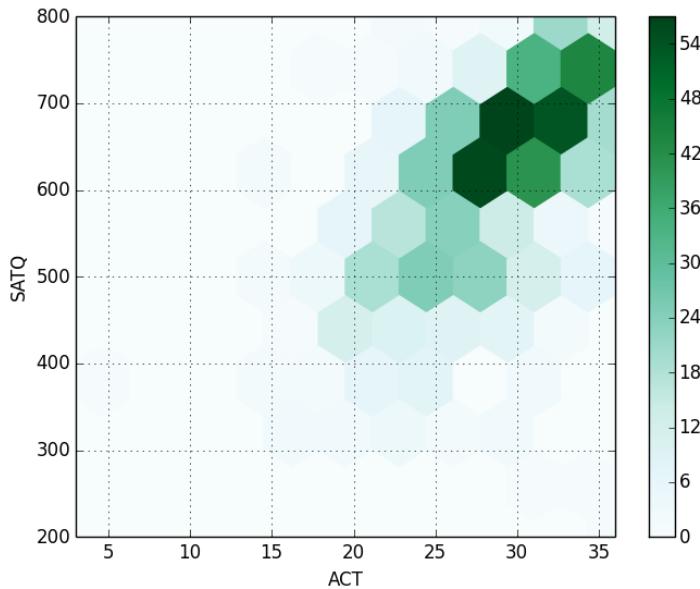


Figure 7.5: Hexbin Plot With Gridsize 10

## Lag Plot

We are frequently interested in whether or not data which we have collected is random. Lag plots are used to investigate the randomness of a dataset. If the data is in fact random, then the lag plot will exhibit no structure, while nonrandom data will exhibit some kind of structure. Unfortunately, this does not give us an idea of what exactly that structure may be, but it is a quick and effective way to investigate the randomness of a given dataset.

```
>>> from pandas.tools.plotting import lag_plot
>>> randomdata = pd.Series(np.random.rand(1000))
>>> lag_plot(randomdata)
>>> plt.show()

>>> structureddata = pd.Series(np.sin(np.linspace(-10*np.pi, 10*np.pi, num=100)←
    ))
>>> lag_plot(structureddata)
>>> plt.show()
```

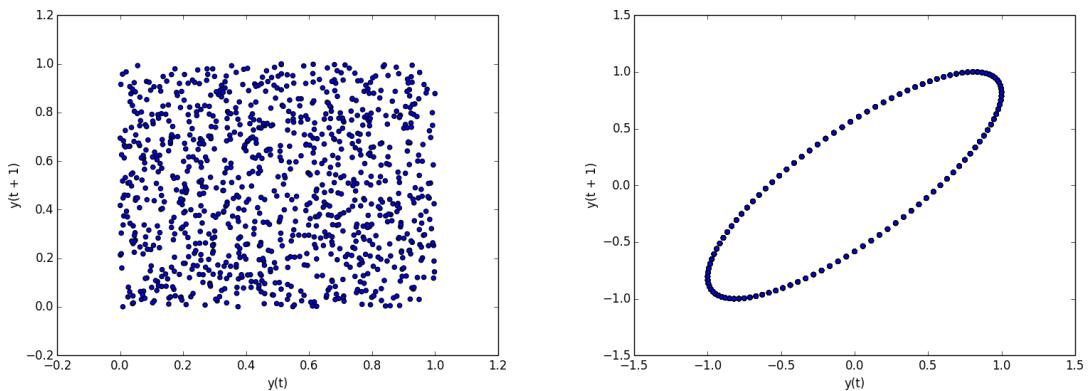


Figure 7.6: Lag plot of random data (left) compared to a lag plot of a sine wave (right).

**Problem 3.** Choose a dataset provided in the `pydataset` and produce scatter and hexbin plots demonstrating some characteristic of the data. A list of datasets in the `pydataset` module can be produced using the `data()` command.

For more types of plots available in Pandas and further examples, see <http://pandas.pydata.org/pandas-docs/stable/visualization.html>.

## Data Visualization

Visualization is much more than a set of pretty pictures scattered throughout a paper for the sole purpose of providing contrast to the text. When properly implemented, data visualization is a powerful tool for analysis and communication. When writing a paper or report, the author must make many decisions about how to use graphics effectively to convey useful information to the reader. Here we will go over a simple process for making deliberate, effective, and efficient design decisions.

### Catching all of the Details

Consider the plot in Figure 7.7. What does it depict? We can tell from a simple glance that it is a scatter plot of positively correlated data of some kind, with `temp`—likely temperature—on the *x* axis and `cons` on the *y* axis. However, the picture is not really communicating anything about the dataset. We have not specified the units for the *x* or the *y* axis, we have no idea what `cons` is, there is no title, and we don't even know where the data came from in the first place.

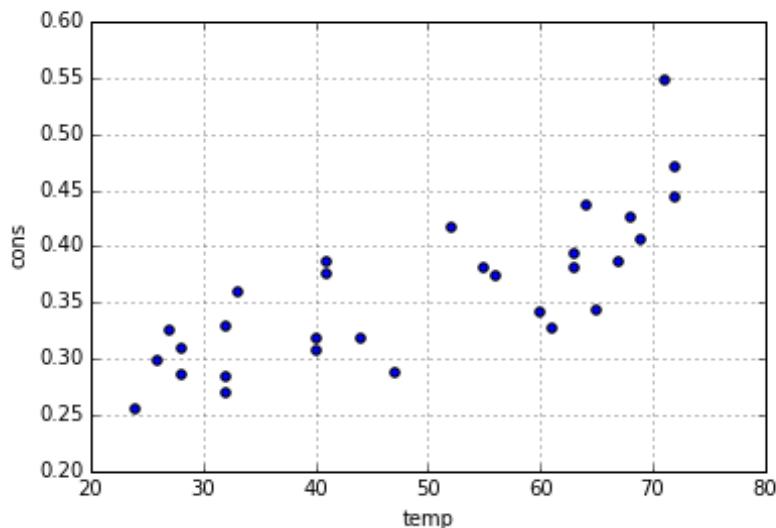


Figure 7.7: Non-specific data.

### Labels, Legends, and Titles

In a homework or lab setting, we sometimes (mistakenly) think that it is acceptable to leave off appropriate labels, legends, titles, and sourcing. In a published report or presentation, this kind of carelessness is confusing at best and, when the source is not included, even plagiaristic. Clearly, we need to explain our data in a useful manner that includes all of the vital information.

Consider again Figure 7.7. This figure comes from the `Icecream` dataset within the `pydataset` package, which we store here in a dataframe and then plot:

```
>>> icecream = data("Icecream")
>>> icecream.plot(kind="scatter", x="temp", y="cons")
```

We have at this point reproduced the rather substandard plot in Figure 7.7. Using `data('Icecream', show_doc=True)` we find the following information:

1. The dataset details ice cream consumption via four-weekly observations from March 1951 to July 1953 in the United States.
2. `cons` corresponds to “consumption of ice cream per head” and is measured in pints.
3. `temp` corresponds to temperature, degrees Fahrenheit.
4. The listed source is: “Hildreth, C. and J. Lu (1960) *Demand relations with autocorrelated disturbances*”, Technical Bulletin No 2765, Michigan State University.”

We add these important details using the following code. As we have seen in previous examples, pandas automatically generates legends when appropriate. However, although pandas also automatically labels the  $x$  and  $y$  axes, our data frame column titles may be insufficient. Appropriate titles for the  $x$  and  $y$  axes must also list appropriate units. For example, the  $y$  axis should specify that the consumption is in units of *pints per head*, in place of the ambiguous label `cons`.

```
>>> icecream = data("Icecream")
# Set title via the title keyword argument
>>> icecream.plot(kind="scatter", x="temp", y="cons", title="Ice Cream ←
    Consumption in the U.S., 1951-1953",)
# Override pandas automatic labelling using xlabel and ylabel
>>> plt.xlabel("Temp (Farenheit)")
>>> plt.ylabel("Consumption per head (pints)")
>>> plt.show()
```

Unfortunately, there is no explicit function call that allows us to add our source information. To arbitrarily add the necessary text to the figure, we may use either `plt.annotate` or `plt.text`.

```
>>> plt.text(20, .1, "Source: Hildreth, C. and J. Lu (1960) Demand relations ←
    with autocorrelated disturbances\nTechnical Bulletin No 2765, Michigan ←
    State University.", fontsize=7)
```

Both of these methods are imperfect, however, and can normally be just as easily replaced by a caption attached to the figure in your presentation or document setting. We again reiterate how important it is that you source any data you use. Failing to do so is plagiarism.

Finally, we have a clear and demonstrative graphic in Figure 7.8.

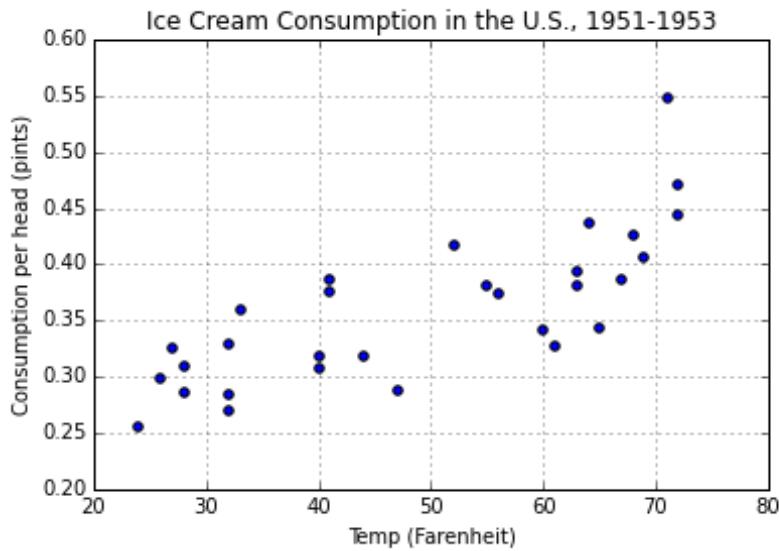


Figure 7.8: Source: Hildreth, C. and J. Lu (1960) *\_Demand relations with autocorrelated disturbances\_*, Technical Bulletin No 2765, Michigan State University.

**Problem 4.** Return to the plots you generated in problem 1 (datasets `nottem`, `VADeaths`, and `Arbuthnott`). Reproduce and modify these plots to include:

- A clear title, with relevant information for the period or region the data was collected in.
- Axes that specify units.
- A legend (for comparison data).
- The source. You may include the source information in your plot or print it after the plot at your discretion.

Note that in this problem, as well as for plots in subsequent labs, points will be taken off if any of these items are partially or fully missing from your graphs.

## Choosing the Right Plot

Now that we know how to add the appropriate details for remaining plots, we return to the fundamental question of data visualization: Which plot should you use for your data? In previous sections, we have already discussed the various strengths and weaknesses of available pandas plotting techniques. At this point, we know how to visualize data using line graphs, bar charts, histograms, box plots, scatter plots, hexbins, and lag plots.

However, perfectly plot-ready data sets—organized by a simple continuum or a convenient discrete set—are few and far between. In the real world, it is rare to find data that is perfectly ready to become a meaningful visual, even if it is already a `DataFrame`. As such, deciding how to organize and group your data is one of the most important parts of “choosing the right plot”.

In Lab 8 we will go over how to group data in meaningful ways. You should then be better able to choose the right type of plot for your data.



# 8

# Pandas III: Grouping and Presenting Data

**Lab Objective:** *Learn about Pivot tables, groupby, etc.*

## Introduction

Pandas originated as a wrapper for numpy that was developed for purposes of data analysis. Data seldom comes in a format that is perfectly ready to use. We always need to be able to interpret what our data is telling us. Some data may be of little or no use, while other aspects of our data may be vital. *Pivoting* is an extremely useful way to sort through data and be able to present results clearly and compactly. Two central ways to accomplish this are by using `groupby` and by using *Pivot Tables*.

## Groupby

In Lab 7 we introduced `pydatasets`, and mentioned how, in their raw format, plotting would be nonsensical. Many datasets are simply composed of tables of individuals (represented as rows), with a list of classifiers associated with each one (columns).

For example, consider the `msleep` dataset. In order to view the columns present in this dataset, we make use of the function `head()`. This will show us the first five rows, and all of the columns.

```
>>> import pandas as pd
>>> from pydataset import data
>>> msleep = data("msleep")
>>> msleep.head()
      name      genus   vore      order  conservation
1    Cheetah  Acinonyx  carni  Carnivora          lc
2  Owl monkey     Aotus  omni   Primates         NaN
3 Mountain beaver  Aplodontia  herbi  Rodentia          nt
4 Greater short-tailed shrew    Blarina  omni Soricomorpha          lc
5           Cow       Bos  herbi Artiodactyla  domesticated

  sleep_total  sleep_rem  sleep_cycle   awake  brainwt   bodywt
1        12.1       NaN        NaN     11.9       NaN    50.000
2        17.0       1.8        NaN      7.0  0.01550     0.480
3        14.4       2.4        NaN      9.6       NaN    1.350
```

4	14.9	2.3	0.133333	9.1	0.00029	0.019
5	4.0	0.7	0.666667	20.0	0.42300	600.000

Each row consists of a single type of mammal and its corresponding identifiers, including genus and order, as well as sleep measurements such as total amount of sleep (in hours) and REM sleep, in hours. When we try to plot this data using plt.plot, the individual data points do not demonstrate overall trends.

```
>>> msleep.plot(y="sleep_total", title="Mammalian Sleep Data", legend=False)
>>> plt.xlabel("Animal Index")
>>> plt.ylabel("Sleep in Hours")
>>> plt.show()
```

The above code results in Figure 8.1.

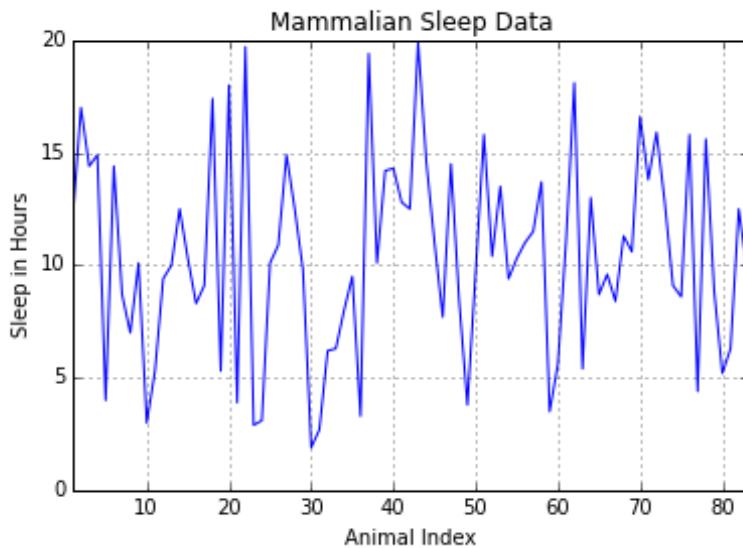


Figure 8.1: Source: Proceedings of the National Academy of Sciences, 104 (3):1051-1056, 2007. Updates from V. M. Savage and G. B. West, with additional variables supplemented by Wikipedia.

This set of connected data points is not particularly revealing, as it plots the first numerical column, `sleep_total`, as a function of the animal index, which is seemingly random.

The `DataFrame` contains information that will help us to make better sense of this data. Using `groupby()`, we can select the parts we want. As the name implies, `groupby()` takes a `DataFrame` and creates different groupings. For this dataset, let's consider the sleep differences between herbivores, omnivores, insectivores, and carnivores. To do so, we simply call the `groupby` method on the `vore` column to obtain a `groupby` object organized by diet classification.

```
>>> vore = msleep.groupby("vore")
```

You can also group the data by multiple columns, for example, both the `vore` and `order` classifications. To group and view this data, simply use:

```
>>> vorder = msleep.groupby(["vore", "order"])

# View groups within vorder
>>> vorder.describe()
```

This listing is much too long to view here, but you should be able to see that it first sorts all rows into the appropriate `vore` category, and then into the appropriate `order` category. It gives the count of rows in each section, along with the mean, standard deviation, and other potentially useful statistics.

Pandas `groupby` objects are not lists of new `DataFrames` associated with groupings. They are better thought of as a dictionary or generator-like object which can be *used* to produce the necessary groups. However, the `get_group()` method will do this, as follows:

```
# Get carnivore group
>>> Carni = vore.get_group("carni")
# Get herbivore group
>>> Herbi = vore.get_group("herbi")
```

The `groupby` object includes many useful methods that can help us make visual comparisons between groups. The `mean()` method, for example, returns a new `DataFrame` consisting of the mean values attached to each group. Using this method on our `vore` object returns a nicely organized `DataFrame` of the average sleep data for each mammalian diet pattern. We can similarly create a `DataFrame` of the standard deviations for each group.

At this point, we have a nicely organized dataset that can easily be turned into a bar chart. Here, we use the `DataFrame.loc` method to access three specific columns in the bar chart (`sleep_total`, `sleep_rem`, and `sleep_cycle`).

```
>>> means = vore.mean()
>>> errors = vore.std()
>>> means.loc[:,["sleep_total", "sleep_rem", "sleep_cycle"]].plot(kind="bar", ←
    yerr=errors, title="Mean Mammalian Sleep Data")
>>> plt.xlabel("Mammal diet classification (vore)")
>>> plt.ylabel("Hours")
>>> plt.show()
```

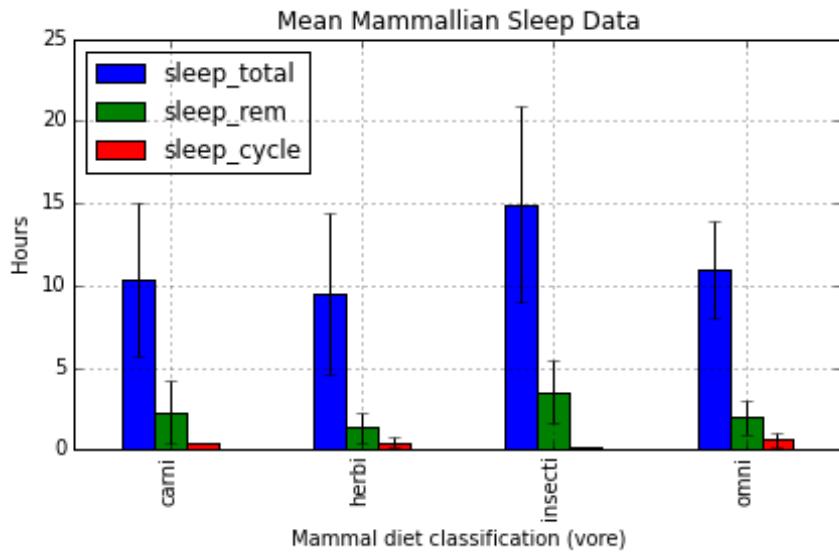


Figure 8.2: Source: Proceedings of the National Academy of Sciences, 104 (3):1051-1056, 2007. Updates from V. M. Savage and G. B. West, with additional variables supplemented by Wikipedia.

**Problem 1.** Examine the `diamonds` dataset found in the `pydataset` module. This dataset contains the identifiers and attributes of 53,940 individual round cut diamonds. Using the `groupby` method, create a visual highlighting and comparing different aspects of the data. This can be in the form of a single plot or comparative subplots. Use the plotting techniques from Lab 7.

Print a paragraph explaining what type of graph you used, why, and what we learn about the dataset from your plot. Don't forget to include titles, clear labels, and sourcing.

The following is an example of comparative subplots, which may *not* be used for your plot.

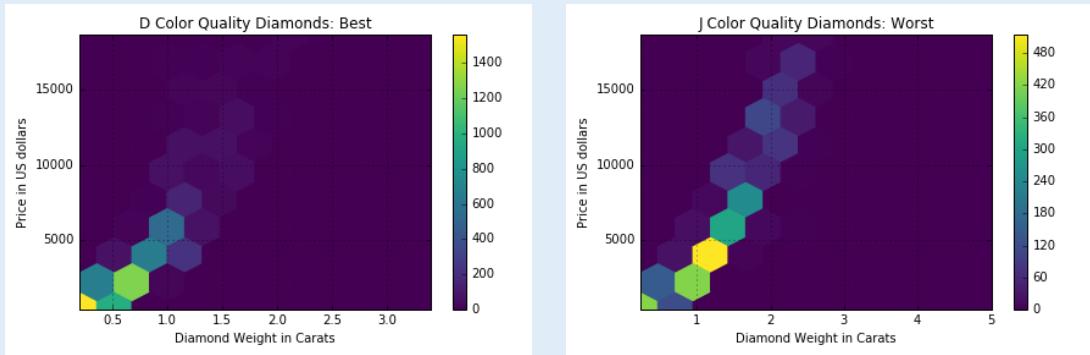


Figure 8.3: Source: Adopted from R Documentation

The following is an appropriate (if lengthy) description of our plots:

The above plots were created using `groupby` on the diamond colors and then using a hexbin comparing carats to price for the highest and lowest quality diamonds, respectively. This hexbin is particularly revealing for each set of thousands of diamonds because it meaningfully displays concentration of datapoints. Matplotlib's new `viridis` colorplot, with a dark background, reveals bins that would have been invisible with a white background. By comparing these plots, we note that the greatest number of J quality diamonds in the dataset are about 1.25 carats and \$4000 dollars in price, whereas the highest concentration of D quality diamonds are smaller and therefore cheaper. We may attribute this to D quality diamonds being rarer, but the colorbar on the side reveals that D diamond numbers are, in fact, far higher than those of the J color. Instead it is simply more likely that D quality diamonds of larger sizes are rarer than those of smaller sizes. Both hexbins reveal a linearity between diamond weight and diamond price, with D diamonds showing more variability and J diamonds displaying a strict linearity.

`Groupby` is a very useful method to order data. However, it is not the perfect tool for every situation. As we saw when sorting data by multiple columns, we can end up with a useful grouping, but too much information to display. If we want to see information in table format, we can make use Pivot Tables.

## Pivot Tables

With a given pandas `DataFrame`, we can visualize data easily with the method `pivot_table()`. The Titanic dataset (`titanic.csv`) is especially apt for this. It includes many columns, but we will use only `Survived`, `Pclass`, `Sex`, `Age`, `Fare`, and `Embarked` here. You will need to load in this dataset for upcoming problems, using principles taught in Lab 6. Note that many of the ages are missing values. For our purposes, simply fill these missing values with the average age. Then drop any rows that are missing data. Once we have a usable dataset, we can make Pivot Tables to view trends in the data.

```
>>> titanic.pivot_table('Survived', index='Sex', columns='Pclass')
Pclass      1          2          3
Sex
female    0.962406  0.893204  0.473684
male      0.350993  0.145570  0.169540
```

This simple command makes a table with rows `Sex` and columns `Pclass`, and averages the result of the column `Survived`, thereby giving the percentage of survivors in each grouping. Note that this is similar to `groupby`: it sorts all entries into their gender and class, and the "function" it performs is getting the percentage of survivors.

This is interesting, and we can clearly see how much more likely females were to survive than males. But how does age factor into survival rates? Were male children really that likely to die as compared to females in general?

We can investigate these results by *multi-indexing*. We can pivot based on more than just two variables, by adding in another index. Note that in the original dataset, the column `Age` has a floating point value for the age of each passenger. If we were to just add 'age' as an argument for index, then the table would create a new row for *each* age present. This wouldn't be a very useful or simple table to visualize, so we desire to partition the ages into 3 categories. We use the function `cut()` to do this.

```
>>> # Partition each of the passengers into 3 categories based on their age
>>> age = pd.cut(titanic['Age'], [0,12,18,80])
```

Now with this other partition of the column `age`, we can add this dimension to our pivot table, passing it along with `Sex` in a list to the parameter `index`.

```
>>> # Add a third dimension, age, to our pivot table
>>> titanic.pivot_table('Survived', index=['Sex', 'age'], columns='Pclass')
Pclass          1      2      3
Sex   Age
female (0, 12]  0.000000  1.000000  0.466667
           (12, 18]  1.000000  0.875000  0.607143
           (18, 80]  0.966667  0.878049  0.436170
male   (0, 12]  1.000000  1.000000  0.342857
           (12, 18]  0.500000  0.000000  0.081081
           (18, 80]  0.328671  0.087591  0.159420
```

What do we notice? First of all, male children (ages 0 to 12) in the 1st and 2nd class were very likely to survive, whereas those in 3rd class were much less likely to. However, look at the female children in first class. It says that zero percent survived. This might seem a little odd, but if we looked at our data set again to see how many passengers fell into this category of female, 1st class, and age 0 to 12, we would find only one passenger. Therefore, the statistic that 0% of female children in first class lived is misleading. We can see the number of people in each category by simply specifying the parameter `aggfunc='count'`.

```
>>> titanic.pivot_table('Survived', index=['Sex', 'age'], columns='Pclass', ←
    aggfunc='count')
Pclass          1      2      3
Sex   Age
female (0, 12]    1    13    30
           (12, 18]   12     8    28
           (18, 80]  120    82    94
male   (0, 12]     4    11    35
           (12, 18]     4    10    37
           (18, 80]  143   137   276
```

The parameter `aggfunc` defaults to ‘mean’, which is why we have seen the mean survival rate for each of the different categories. By specifying `aggfunc` to be ‘count’, we get how many passengers of each category are present in the table. In this case, specifying the value is ‘survived’ is redundant. We get the same table if we put in `Fare` in place of `Survived`.

This table brings up another point about partitioning datasets. This Titanic dataset includes data for only about 1000 passengers. This is not that large of a sample size, relatively speaking, and significant sample sizes aren’t guaranteed for each possible partitioning of the the data columns. It is a good practice to ask questions about the numbers you see in pivot tables before making any conclusions.

Now, for fun, let’s add a fourth dimension to our table—the cost of the passenger’s ticket. Let’s add this dimension to our columns of the pivot table. We now use the function `qcut()` to partition the fare column’s data into two different categories.

```
>>> # Partition fare column into 2 categories based on the values present in ←
     fare column
>>> fare = pd.qcut(titanic['fare'], 2)
>>> # Add the fare as a dimension of columns
>>> titanic.pivot_table('Survived', index=['Sex', age], columns=[fare, 'Pclass'←
     ])
Fare                               [0, 15.75]          (15.75, 512.329]
Pclass                         1       2       3           1       2       3
Sex      Age
female   (0, 12]    NaN  1.000000  0.600000  0.000000  1.000000  ←
     0.400000
          (12, 18]   NaN  0.750000  0.666667  1.000000  1.000000  ←
          0.250000
          (18, 80]   NaN  0.875000  0.434783  0.966667  0.880000  ←
          0.440000
male     (0, 12]    NaN  1.000000  0.636364  1.000000  1.000000  ←
     0.208333
          (12, 18]   NaN  0.000000  0.115385  0.500000  0.000000  ←
          0.000000
          (18, 80]   0.2   0.113636  0.153846  0.333333  0.040816  ←
          0.206897
```

We now see something else about our dataset—we get NaNs in some entries. The dataset has some invalid entries or incomplete data for the column fare. Let's investigate further.

By inspecting our dataset, we can see that there might not be any people in the categories given in the pivot table. We can fill in the NaN values in the pivot table with the argument `fill_value`. We show this below.

```
>>> # Specify fill_value = 0.0
>>> titanic.pivot_table('survived', index=['sex', age], columns=[fare, 'class'],←
     fill_value=0.0)
Fare                               [0, 15.75]          (15.75, 512.329]
Pclass                         1       2       3           1       2       3
Sex      Age
female   (0, 12]    0.0  1.000000  0.600000  0.000000  1.000000  ←
     0.400000
          (12, 18]   0.0  0.750000  0.666667  1.000000  1.000000  ←
          0.250000
          (18, 80]   0.0  0.875000  0.434783  0.966667  0.880000  ←
          0.440000
male     (0, 12]    0.0  1.000000  0.636364  1.000000  1.000000  ←
     0.208333
          (12, 18]   0.0  0.000000  0.115385  0.500000  0.000000  ←
          0.000000
          (18, 80]   0.2   0.113636  0.153846  0.333333  0.040816  ←
          0.206897
```

It should be noted though that with datasets where NaN's occur frequently, some preprocessing needs to be done so as to get the most accurate statistics and insights about your dataset. You should've worked on cleaning datasets in previous lab, so we don't go further into detail about the topic here.

**Problem 2.** Suppose that someone claims that the city from which a passenger embarked had a strong influence on the passenger's survival rate. Investigate this claim.

1. Using a `groupby()` call, see what the survival rates of the passengers were based on where they embarked from (`embark_town`).
2. Next, create a pivot table to further look at survival rates based on both place of embarkment and gender.
3. What do these tables suggest to you about the significance of where people embarked in influencing their survival rate? Examine the context of the problem, and explain what you think this really means. Find 2 more pivot tables that investigate the claim further, looking at other criterion (e.g., class, age, etc.). Include explanations.

As you have learned in the pandas labs, pandas is a powerful tool for data processing, and has a plethora of built-in functions that greatly simplify data analysis. Let's now put the skills you have acquired to practical use.

**Problem 3.** Search through the `pydatasets` and find one which you would like to present. Make sure there is enough data to produce a proper presentation. Process the data appropriately, and create your own presentation, as though you were presenting to a superior in the appropriate line of work. Your presentation can be in any format you like, but as a bare minimum, it must include the following:

- An appropriate title.
- 3 charts or visuals with captions.
- 3 tables with captions.
- 2 paragraphs of textual explanations.
- Appropriate data sourcing.

# 9

## Pandas IV: Time Series

**Lab Objective:** *Learn how to manipulate and prepare time series in pandas in preparation for analysis*

### Introduction: What is time series data?

Time series data is ubiquitous in the real world. Time series data is any form of data that comes attached to a timestamp (i.e. Sept 28, 2016 20:32:24) or a period of time (i.e. Q3 2012). Some examples of time series data include:

- stock market data
- ocean tide levels
- number of sales over a period of time
- website traffic
- concentrations of a certain compound in a solution over time
- audio signals
- seismograph data
- and more...

Notice that a common feature of all these types of data is that the values can be tied to a specific time or period of time.

In this lab, we will not go into depth on the analysis of such data. Rather, we will discuss some of the tools provided in pandas for cleaning and preparing time series data for further analysis.

### Initializing Time Series in pandas

To take advantage of all the time series tools available to us in pandas, we need to make a few adjustments to our normal DataFrame.

## The `datetime` Module and Initializing a DatetimeIndex

For pandas to know to treat a DataFrame or Series object as time series data, the index must be a `DatetimeIndex`. pandas utilizes the `datetime.datetime` object from the `datetime` module to standardize the format in which dates or timestamps are represented.

```
>>> from datetime import datetime

>>> datetime(2016, 9, 28) # 9/28/2016
datetime.datetime(2016, 9, 28, 0, 0)

>>> datetime(2016, 9, 28, 21, 12, 48) # 9/28/2016 9:12:48 PM
datetime.datetime(2016, 9, 28, 21, 12, 48)
```

Unsurprisingly, the format for dates varies greatly from dataset to dataset. The `datetime` module comes with a string parser (`datetime.strptime()`) flexible enough to translate nearly any format into a `datetime.datetime` object. This method accepts the string representation of the date, and a string representing the format of the string. See Table 9.1 for all the options.

%Y	4-digit year
%y	2-digit year
%m	2-digit month
%d	2-digit day
%H	Hour (24-hour)
%I	Hour (12-hour)
%M	2-digit minute
%S	2-digit second

Table 9.1: Formats recognized by `datetime.strptime()`

Here are some examples of using `datetime.strptime()` to parse the same date from different formats.

```
>>> datetime.strptime("2016-9-28", "%Y-%m-%d")
datetime.datetime(2016, 9, 28, 0, 0)

>>> datetime.strptime("9/28/16", "%m/%d/%y")
datetime.datetime(2016, 9, 28, 0, 0)

>>> datetime.strptime("2016-9-28 9:12:48", "%Y-%m-%d %I:%M:%S")
datetime.datetime(2016, 9, 28, 9, 12, 48)
```

If the dates are in an easily parsable format, pandas has a method `pd.to_datetime()` that can turn a whole pandas `Series` into `datetime.datetime` objects. In the case of the index, the index is automatically converted to a `DatetimeIndex`. This index type is what distinguishes a regular `Series` or `DataFrame` from a time series.

```
>>> dates = ["2010-1-1", "2010-2-1", "2010-3-1", "2011-1-1",
... "2012-1-1", "2012-1-2", "2012-1-3"]
```

```
>>> values = np.random.randn(7,2)
>>> df = pd.DataFrame(values, index=dates)
>>> df
          0      1
2010-1-1  0.566694  1.093125
2010-2-1 -0.219856  0.852917
2010-3-1  1.511347 -1.324036
2011-1-1  0.300766  0.934895
2012-1-1  0.212141  0.859555
2012-1-2  1.483123 -0.520873
2012-1-3  1.436843  0.596143

>>> df.index = pd.to_datetime(df.index)
```

#### NOTE

In earlier versions of pandas, there was a dedicated TimeSeries data type. This has since been deprecated, however the functionality remains. Therefore, if you happen to read any materials that reference the TimeSeries data type, know that the corresponding functionality is likely still in place as long as you have a `DatetimeIndex` associated with your Series or DataFrame.

**Problem 1.** The provided dataset, “DJIA.csv” is the daily closing value of the Dow Jones Industrial Average for every day over the past 10 years. Read this dataset into a Series with a DatetimeIndex. Replace any missing values with `np.nan`. Lastly, cast all the values in the Series to floats. We will use this dataset for many problems in this lab.

## Handling Data Without Marked Timestamps

There will be times you will need to analyze time series data that does not come marked with an index. For example, you may have a list of bank account balances at the beginning of every month for the last 5 years. You may have heart rate readings every 10 minutes for the past week. pandas provides efficient tools for generating indices for these kinds of situations.

### The `pd.date_range()` Method

The `pd.date_range()` method is analogous to `np.arange()`. The parameters we will use most are described in Table 9.2.

Exactly two of the parameters `start`, `end`, and `periods` must be defined to generate a range of dates. The `freqs` parameter accepts a variety of string representations. The accepted strings are referred to as *offset aliases* in the documentation. See Table 9.3 for a sampling of some of the options. For a complete list of the options, see <http://pandas.pydata.org/pandas-docs/stable/timeseries.html#offset-aliases>.

```
>>> pd.date_range(start='9/28/2016 16:00', periods=5)
```

<code>start</code>	start of date range
<code>end</code>	end of date range
<code>periods</code>	the number of dates to include in the date range
<code>freq</code>	the amount of time between dates (similar to "step")
<code>normalize</code>	trim the time of the date to midnight

Table 9.2: Parameters for `datetime.strptime()`

D	calendar daily (default)
B	business daily
H	hourly
T	minutely
S	secondly
MS	first day of the month
BMS	first weekday of the month
W-MON	every Monday
WOM-3FRI	every 3rd Friday of the month

Table 9.3: Parameters for `datetime.strptime()`

```

DatetimeIndex(['2016-09-28 16:00:00', '2016-09-29 16:00:00',
               '2016-09-30 16:00:00', '2016-10-01 16:00:00',
               '2016-10-02 16:00:00'],
              dtype='datetime64[ns]', freq='D')

>>> pd.date_range(start='1/1/2016', end='1/1/2017', freq="2BMS" )
DatetimeIndex(['2016-01-01', '2016-03-01', '2016-05-02', '2016-07-01',
               '2016-09-01', '2016-11-01'],
              dtype='datetime64[ns]', freq='2BMS')

>>> pd.date_range(start='9/28/2016 16:00',
                  end='9/29/2016 16:30', freq="10T")
DatetimeIndex(['2016-09-28 16:00:00', '2016-09-28 16:10:00',
               '2016-09-28 16:20:00', '2016-09-28 16:30:00'],
              dtype='datetime64[ns]', freq='10T')

```

The `freq` parameter also supports more flexible string representations.

```

>>> pd.date_range(start='9/28/2016 16:30', periods=5, freq="2h30min")
DatetimeIndex(['2016-09-28 16:30:00', '2016-09-28 19:00:00',
               '2016-09-28 21:30:00', '2016-09-29 00:00:00',
               '2016-09-29 02:30:00'],
              dtype='datetime64[ns]', freq='150T')

```

**Problem 2.** The “paychecks.csv” dataset has values of an hourly employee’s last 93 paychecks. He started working March 13, 2008. This company hands out paychecks on the first and third Fridays of the month. However, “paychecks.csv” is not indexed explicitly as such. To be able to manipulate it as a time series in pandas, we will need to add a `DatetimeIndex` to it. Read in the data and use `pd.date_range()` to generate the `DatetimeIndex`.

Hint: to combine two `DatetimeIndex` objects, you can use the `.union()` method of `DatetimeIndex` objects.

## Plotting Time Series

The process for plotting a time series is identical to plotting any other Series or DataFrame. For more information and examples, refer back to Lab 7.

**Problem 3.** Plot the DJIA dataset that you read in as part of Problem 1. Label your axes and title the plot.

## Dealing with Periods Instead of Timestamps

It is often important to distinguish whether a given figure corresponds to a single point in time or to a whole month, quarter, year, decade, etc. A `Period` object is better suited for the summary over a *period* of time rather than the timestamp of a specific event.

Some example of time series that would merit the use of periods would include,

- The number of steps a given person walks in a day.
- The box office results per week for a summer blockbuster.
- The population changes of a given city per year.
- etc.

### The `Period` Object

The principle parameters of the `Period` are “`value`” and “`freq`”. The “`value`” parameter indicates the label for a given `Period`. This label is tied to the *end* of the defined `Period`. The “`freq`” indicates the length of the `Period` and also (in some cases) indicates the offset of the `Period`. The “`freq`” parameter accepts the majority, but not all, of frequencies listed in Table 9.3.

These nuances are best clarified through examples.

```
# The default value for `freq` is "M" meaning month.
>>> p1 = pd.Period("2016-10")
>>> p1.start_time
Timestamp('2016-10-01 00:00:00')

>>> p1.end_time
Timestamp('2016-10-31 23:59:59.999999999')
```

```
# A `freq` value of 'A-DEC' indicates a annual period
#   with the end of the period occurring in December.
>>> p2 = pd.Period("2016-10-03", freq="A-DEC")
>>> p2.start_time
Timestamp('2007-01-01 00:00:00')

>>> p2.end_time
Timestamp('2007-12-31 23:59:59.999999999')

# Notice that the Period object that is created is the
#   week of the year when the given date occurs. Also
#   note that "W-SAT" indicates we wish to treat
#   Saturday as the last day of the week (and therefore
#   Sunday as the first day of the week.)
>>> p3 = pd.Period("2016-10-03", freq="W-SAT")
>>> p3
Period('2016-10-02/2016-10-08', 'W-SAT')
```

## The `pd.period_range()` Method

Like the `pd.date_range()` method, the `pd.period_range()` method is useful for generating a `PeriodIndex` for unindexed data. The syntax is essentially identical to that of `pd.date_range()`. When using `pd.period_range()`, remember that the `"freq"` parameter marks the end of the period.

```
# a 'freq' value of 'Q-DEC' means that Q4 ends in December.
>>> pd.period_range("2008", "2010-12", freq="Q-DEC")
PeriodIndex(['2009Q1', '2009Q2', '2009Q3', '2009Q4', '2010Q1', '2010Q2',
             '2010Q3', '2010Q4'], dtype='int64', freq='Q-DEC')
```

## Other Useful Functionality and Methods

After creating a `PeriodIndex`, you have the option to redefine the `"freq"` parameter using the `asfreq()` method.

```
>>> p = pd.period_range("2010-03", "2011", freq="3M")
PeriodIndex(['2010-03', '2010-06', '2010-09', '2010-12'],
            dtype='int64', freq='3M')

# Change frequency to be "Q-DEC" instead of "3M"
>>> p = p.asfreq("Q-DEC")
>>> p
PeriodIndex(['2010Q2', '2010Q3', '2010Q4', '2011Q1'],
            dtype='int64', freq='Q-DEC')
```

Say you have created a `PeriodIndex`, but the bounds are not exactly where you expected they would be. You can actually shift `PeriodIndex` objects by adding or subtracting an integer,  $n$ . The resulting `PeriodIndex` will be shifted by  $n \times \text{freq}$ .

```
# Shift index by 1
>>> p -= 1
>>> p
PeriodIndex(['2010Q1', '2010Q2', '2010Q3', '2010Q4'],
             dtype='int64', freq='Q-DEC')
```

If for any reason you need to switch from periods to timestamps, pandas provides a very simple method to do so.

```
# Convert to timestamp (last day of each quarter)
>>> p = p.to_timestamp(how='end')
>>> p
DatetimeIndex(['2010-03-31', '2010-06-30', '2010-09-30', '2010-12-31'],
               dtype='datetime64[ns]', freq='Q-DEC')
```

Similarly, you can switch from timestamps to periods.

```
>>> p.to_period("Q-DEC")
PeriodIndex(['2010Q1', '2010Q2', '2010Q3', '2010Q4'],
            dtype='int64', freq='Q-DEC')
```

**Problem 4.** The “finances.csv” dataset has a list of simulated quarterly earnings and expense totals from a fictional company. Load these values into a DataFrame with a `PeriodIndex` with a quarterly frequency. Assume the fiscal year starts at the beginning of September and that the dataset goes back to September 1978.

## Operations on Time Series

There are certain operations only available to Series and DataFrames that have a `DatetimeIndex`. A sampling of this functionality is described throughout the remainder of this lab.

### Slicing

Slicing is much more flexible in pandas for time series. We can slice by year, by month, or even use traditional slicing syntax to select a range of dates.

```
# Select all rows in a given year
>>> df["2010"]
              0          1
2010-01-01  0.566694  1.093125
2010-02-01 -0.219856  0.852917
2010-03-01  1.511347 -1.324036
```

```
# Select all rows in a given month of a given year
>>> df["2012-01"]
      0      1
2012-01-01  0.212141  0.859555
2012-01-02  1.483123 -0.520873
2012-01-03  1.436843  0.596143

# Select a range of dates using traditional slicing syntax
>>> df["2010-1-2":"2011-12-31"]
      0      1
2010-02-01 -0.219856  0.852917
2010-03-01  1.511347 -1.324036
2011-01-01  0.300766  0.934895
```

## Resampling a Time Series

Imagine you have a dataset that does not have datapoints at a fixed frequency. For example, a dataset of website traffic would take on this form. Because the datapoints occur at irregular intervals, it may be more difficult to procure any meaningful insight. In situations like these, resampling your data is worth considering.

The two main forms of resampling are downsampling (aggregating data into fewer intervals) and upsampling (adding more intervals). We will only address downsampling in detail.

### Downsampling

When downsampling data, we aggregate our time series data into fewer intervals. Let's further consider the website traffic examples given above.

Say we have a data set of website traffic indexed by timestamp. Each entry in the dataset contains the IP address of the user, the time the user entered the website, and the time the user left the website.

If we were interested in having information regarding the average visit duration for any given day, we could downsample the data to a daily timescale. To aggregate the data, we would take the average across all the datapoints for the day.

Instead, if we were interested in the number of visits to the website per hour, we could downsample to a weekly timescale. To aggregate the data in this case, we would count up the number of visits in a hour.

The task of downsampling is handled using the `resample()` method. The parameters we will use are `rule`, `how`, `closed`, and `label`. For explanations of these parameters, see Table 9.4.

**Problem 5.** Using the “website\_traffic.csv” dataset, solve the problem described above. Namely, downsample the dataset to show daily average visit duration. In a different DataFrame, also downsample the dataset to show total number of visits per hour. Your resulting time series should use a `PeriodIndex`.

<code>rule</code>	the offset string (see Table 9.3)
<code>how</code>	the data aggregation method (i.e. “sum” or “mean”)
<code>closed</code>	the boundary of the interval that is closed/inclusive (default “right”)
<code>label</code>	the boundary of the interval that is assigned to the label (default “right”)

Table 9.4: Parameters for `DataFrame.resample()`

## Basic Time Series Analysis

As mentioned in the beginning of this lab, the focus of this lab is not meant to be on Time Series Analysis. However, we would like to address a few very basic methods that are built into pandas.

### Shifting

DataFrame and Series objects have a `shift()` method that allows you to move data up or down relative to the index. When dealing with time series data, we can also shift the DatetimeIndex relative to a time offset. See the following example code for clarification:

```
>>> df = pd.DataFrame(dict(VALUE=np.random.rand(5)),
                     index=pd.date_range("2016-10-7", periods=5, freq='D'))
>>> df
          VALUE
2016-10-07  0.127895
2016-10-08  0.811226
2016-10-09  0.656711
2016-10-10  0.351431
2016-10-11  0.608767

>>> df.shift(1)
          VALUE
2016-10-07      NaN
2016-10-08  0.127895
2016-10-09  0.811226
2016-10-10  0.656711
2016-10-11  0.351431

>>> df.shift(-2)
          VALUE
2016-10-07  0.656711
2016-10-08  0.351431
2016-10-09  0.608767
2016-10-10      NaN
2016-10-11      NaN

>>> df.shift(14, freq="D")
          VALUE
2016-10-21  0.127895
2016-10-22  0.811226
```

```
2016-10-23  0.656711
2016-10-24  0.351431
2016-10-25  0.608767
```

Shifting data has a particularly useful application. We can easily gather statistics about changes from one timestamp or period to the next.

```
# find the changes from one period/timestamp to the next
>>> df - df.shift(1)
      VALUE
2016-10-07      NaN
2016-10-08  0.683331
2016-10-09 -0.154516
2016-10-10 -0.305279
2016-10-11  0.257336
```

**Problem 6.** From the Dow Jones Industrial Average dataset referenced in Problem 1, use shifting to find the following information:

- The single day with the largest gain
- The single day with the largest loss
- The month with the largest gain
- The month with the largest loss

Hint: Downsample or use `groupby()` to answer the questions regarding months.

## Rolling Functions and Exponentially-Weighted Moving Functions

In many situations, your data will be noisy. You will often be more interested in general trends. We can accomplish this through *rolling functions* and *exponentially-weighted moving (EWM)* functions.

Rolling functions, or intuitively called *moving window functions*, perform some kind of calculation on just a window of data. There are a few rolling functions that come standard with pandas. However, we will only cover a few of these. To visualize the effects these operations have on our time series, we will plot the results.

We first begin by defining the time series we will use throughout the remaining example code.

```
# Generate a time series using random walk from a uniform distribution.
N = 10000
bias = 0.01
s = np.zeros(N)
s[1:] = np.random.uniform(low=-1, high=1, size=N-1) + bias
s = pd.Series(s, index=pd.date_range("2015-10-20", freq='H', periods=N))
s = s.cumsum()
```

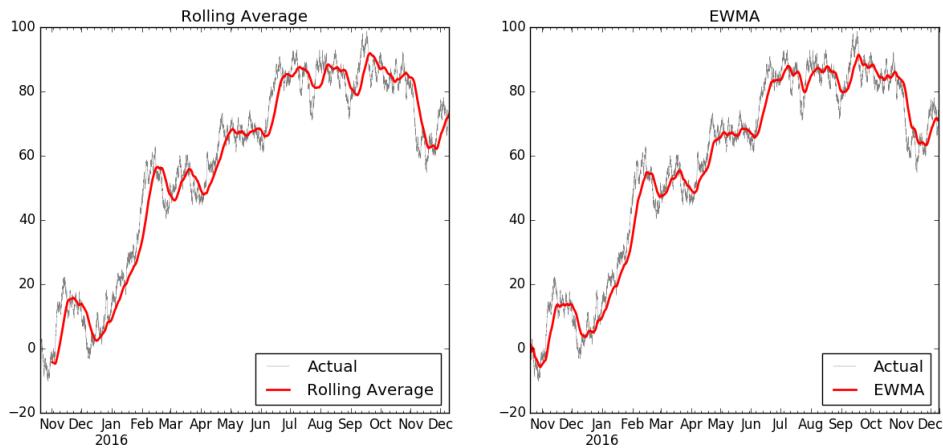


Figure 9.1: Rolling average and EWMA

### Rolling Functions (Moving Window Functions)

One of the most commonly used rolling functions is the rolling average.

```
s.plot(lw=.3, color='grey', label="Actual")
s.rolling(window=200).mean().plot(color='r', lw=2, label="Moving Average")
plt.legend(loc="lower right")
plt.show()
```

Now let's break apart the syntax. The function call, `s.rolling(window=200)` creates a `pd.core.rolling.Window` object. You can then call any functions you would traditionally call on a DataFrame. For example, you can call `mean()`, `std()`, `var()`, `min()`, `max()`, etc.. As your intuition would suggest, rather than executing these functions across the whole time series, these functions are executed across each window and then aggregated into a new time series object.

### Exponentially-Weighted Moving (EWM) Functions

Whereas a moving window function gives equal weight to the whole window, an exponentially-weighted moving function gives more weight to the most recent data points.

In the case of a exponentially-weighted moving average (EWMA), each data point is calculated as,

$$z_i = \alpha \bar{x}_i + (1 - \alpha) z_{i-1}$$

where  $z_i$  is the value of the EWMA at time  $i$ ,  $\bar{x}_i$  is the average for the  $i$ -th window, and  $\alpha$  is the decay factor that controls the importance of previous data points. Notice that  $\alpha = 1$  reduces to the rolling average.

More commonly, the decay is expressed as a function of the window size. In fact, the `span` for an EWMA is nearly analogous to `window size` for a rolling average.

Notice the syntax for EWM functions is very similar to that of rolling functions.

```
s.plot(lw=.3, color='grey', label="Actual")
s.ewm(span=200).mean().plot(color='r', lw=2, label="EWMA")
plt.legend(loc="lower right")
plt.show()
```

**Problem 7.** In this problem, we will explore the differences between rolling functions and EWM functions.

Generate a time series plot with the following information from the DJIA dataset:

- The original data points.
- Rolling average with window 30.
- Rolling average with window 365.
- Exponential average with span 30.
- Exponential average with span 365.

Include a legend, axis labels, and title. Your plot should look like Figure ??.

**Problem 8.** Plot the rolling minimum and rolling maximum values over the DJIA dataset with a window size of 30. This will create a moving bound for all the data that is easy to visualize.

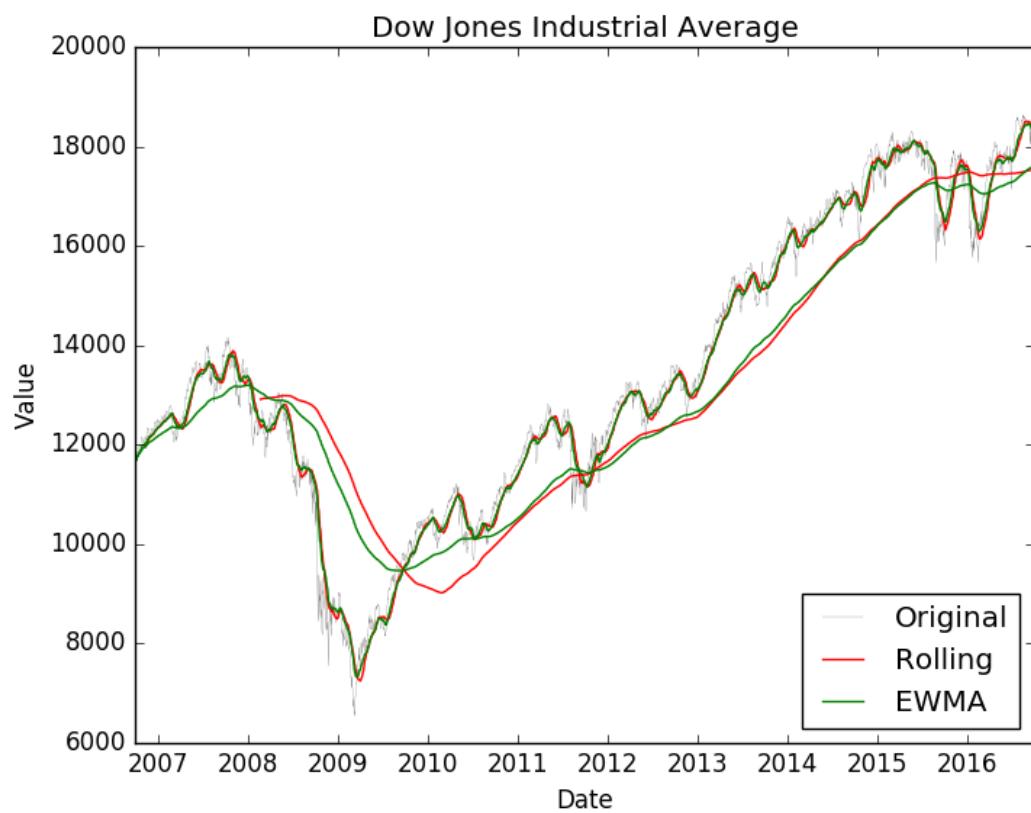


Figure 9.2: Rolling average and EWMA with windows 30 and 365.



# 10

## Web Technologies 1: Internet Protocols

**Lab Objective:** *The internet has strict protocols for managing communications between machines. In this lab, we introduce the basics of TCP, IP, and HTTP, and create a server and client program. We then apply this understanding to navigation of online infrastructure and to data collection.*

The internet is comparable to a network of roads connecting the buildings of a city. Each building represents a computer and the roads are the physical wires or wireless pathways that allow for intercommunication. In order to properly navigate the road, you must use the correct kind of vehicle and follow the established instructions for travel. If these requirements are not met, you are not allowed to navigate the road to another building. There are also vehicles of different capabilities and sizes which can retrieve items from particular buildings. In a similar fashion, the internet has specific rules and procedures, called protocols, which allow for standardized communication within and between computers. By understanding these protocols, we can more easily navigate the world wide web to collect data, create web dependent programs, and interact with other network connected computers.

The most common communication protocols in computer networks are contained in the Internet Protocol Suite. Among these are TCP (Transmission Control Protocol) and IP (Internet Protocol), which are two of the oldest and most important protocols. In fact, they are so important that the entire suite is sometimes referred to as TCP/IP. However, there are many other protocols in the suite, all of which are divided into four abstraction layers:

1. **Application:** Software that utilizes transport protocols to move information between computers. This layer includes protocols important for email, file transfers, and browsing the web.
2. **Transport:** Protocols that define basic high level communication between two computers. The two most common protocols in this layer are TCP and UDP. TCP is by far the most widely used due to its reliability. UDP, however, trades reliability for low latency.
3. **Internet:** Protocols that handle routing and movement of data on a network.
4. **Link:** Protocols that deal with local networking hardware such as routers and switches.

For this lab, we will focus on understanding and utilizing TCP/IP and HTTP, which are the most common Transport and Application protocols, respectively.

## TCP

TCP is specifically used to establish a connection between computers, exchange bits of information called *packets*, and then close their connection. Built for host computers on an IP network, TCP allows for a very reliable, ordered, and error-checked stream of data bytes (eight binary bits).

Specifically, TCP creates network *sockets* that are used to send and receive data packets. While we would normally think of sending data between two different machines, two sockets on the same machine can communicate via TCP as well. Using the Python `socket` module, we will demonstrate how to create a network socket (a *server* to listen for incoming data, and how to create a second socket (a *client*) to send data.

### Creating a Server

A *server* is a program that provides functionality to *client* programs. Oftentimes, these server programs run on dedicated computer hardware that we also refer to as servers. These servers are fundamental to the modern networks we work on and provide services such as file sharing, authentication, webpage information, databases, etc. To create a server, we first create a new socket object. The socket object will be able to listen for and accept incoming connections from other sockets.

The two input arguments specify the socket type. Further description of socket types can be found in the python documentation.

```
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

We then define an address and a port for the socket to listen on. A port is analogous to a mailbox on the computer. There are 65535 available ports. Of those, about 250 are commonly used. Certain ports have pre-defined uses. For example:

- 0 to 1023: Special reserved ports
- 80, 443: Commonly used for web traffic
- 25, 110, 143, 465: Commonly used for email servers

We also specify an address for the host, which is analogous to the “mailing address” of the machine on which the server is running. The address may be set to the computer’s IP address, to “`localhost`”, or to an empty string. We bind the socket to the port and address, then call `listen()` to tell it to listen for connections.

```
address = '0.0.0.0' # Default address that specifies the local machine and ←
                    # allows connection on all interfaces
s.bind((address, 33498)) # Bind the socket to a port 33498, which was ←
                        # arbitrarily chosen
s.listen(1) # Tell the socket to listen for incoming connections
```

Next we tell the socket what to do with incoming connections. Once a connection is made, the `accept()` method returns the connection, which is itself a socket object. The connection object receives data (as a string) in blocks, so we specify a block size in bytes.

The connection object can also send back data. In the code below, the connection simply echoes back whatever data it receives. After all the data has been received, we close the connection.

```

size = 2048 # Block size of 20 bytes
conn, addr = s.accept() # conn is our new socket object for receiving/sending ←
    data
print "Accepting connection from:", addr
while True:
    data = conn.recv(size) # Read 20 bytes from the incoming connection
    if not data: # Terminate the connection if data stops arriving (no more ←
        blocks to receive)
        break
    conn.send(data) # Send the data back to the client
conn.close()

```

We can also close the server by using the KeyBoardInterrupt (**ctrl+c**).

### ACHTUNG!

When running the code above, you will see the program hang on the line,

```
conn, addr = s.accept()
```

As mentioned above, the `accept()` method does not return until a connection is made. Therefore for the code above to execute in its entirety, a client needs to connect to the server. Creating a client is addressed in the next section.

## Creating a Client

A *client* is a program that contacts a server in order to receive data or functionality. We use many client programs which include web browsers, mail programs, online video games, etc. We will create a new client socket to connect to our server. We follow a similar process as we did for the server socket:

```

import socket
client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

```

We specify the address of the server, and the port (this needs to be the same port on which the server is listening). We then connect to the server.

```

ip = '0.0.0.0'
port = 33498
client.connect((ip, port))

```

Once connected, the client can send and receive data. Unlike the server, the client sends and reads the data itself instead of creating a new connection socket. When we are done with the client, we close it.

```

size = 2048 # Block size of 20 bytes
msg = "Trololololo, lolololololo, ahahahaha."

```

```

client.send(msg)

print "Waiting for the server to echo back the data..."
data = client.recv(size)
print "Server echoed back:", data

client.close()

```

To see the client and server communicate, open a terminal and run the server. Then run the client in a separate terminal.

Command	Description
<code>bind((address, port))</code>	Binds to a port and an address.
<code>listen()</code>	Starts listening for requests.
<code>accept()</code>	Accepts a connection from a client, and returns a new socket object and a connection address.
<code>recv(size)</code>	Reads and returns a block of incoming data.
<code>send(data)</code>	Sends data to the client.
<code>close()</code>	Closes the socket.
<code>gethostname()</code>	Returns the host name of the machine.
<code>getsockname()</code>	Returns the socket's own address.

Table 10.1: Table of Socket Commands

**Problem 1.** Write a file called `simple_server.py`. When run, this file should create a server socket, accept a connection and then read incoming data. The server should append the current time onto each data block, then send it back to the client.

(Hint: use `time.strftime('%H:%M:%S')` to format the current time nicely.)

Also write a file called `simple_client.py`. In this file, create a client socket and connect to the server created in `simple_server.py`. The client should send a message to the server and print the server's response.

**Problem 2.** Write a file called `rps_server.py`, which plays rock-paper-scissors with a client. The server should accept a connection, and while the connection is open, cycle through the following loop:

- Receive a move ("`rock`", "`paper`", or "`scissors`").
- Generate a random move of its own. Print both moves.
- Determine who won the round.
- Send "`you win`", "`you lose`", or "`draw`" back to the client, depending on the outcome of the round. If the move is invalid, send back "`invalid move`".
- If the client won, break the loop and close the connection.

Also write a file called `rps_client.py`. The client should connect to the server and then enter a while loop. In the loop, the client sends the server a move and prints the server's response. The client should break the loop and close once it receives a "`you win`" back from the server.

Although these examples are simple, we use a similar pattern for every transfer of data over TCP. For simple connections, the amount of work the programmer has to do can be minimal. However, requesting a complicated webpage would require us to manage possibly hundreds of connections. Naturally, we would want to use a higher level protocol that takes care of the smaller details for us.

## HTTP

HTTP stands for Hypertext Transfer Protocol, which is an application layer networking protocol. It is a higher level protocol than TCP and takes care of many of the small details of TCP for us. It also relies on underlying TCP protocol to provide network capabilities. The protocol is centered around a request and response paradigm. A client makes a request to a server and the server replies with response. There are several methods, or requests, defined for HTTP servers. The three most common of which are GET, POST, and PUT. GET requests are typically used to request information from a server. POST requests are sent to the server with the intent of modifying the state of the server. PUT requests are used to add new pieces of data to the server.

Every HTTP request consists of two parts: a header and a body. The headers contain important information about the request including: the type of request, encoding, and a timestamp. We can add custom headers to any request to provide additional information. The body of the request contains the requested data and may or may not be empty.

We can setup an HTTP connection in Python as demonstrated below. We will encourage you to use the `requests` library instead of the modules in the standard library. However, the code below is illustrative of the steps in making an HTTP connection using `httpplib`.

```
import httpplib
conn = httpplib.HTTPConnection("www.example.net")
conn.request("GET", "/")
resp = conn.getresponse() # Server response message
if resp.status == 200:
    headers = resp.getheaders()
    data = resp.read()
conn.close() # After collecting the information we need, we close the ←
            connection
print headers
print data      # Long string full of HTML code from the webpage
```

The above codes starts by creating a connection to specific host. We then make a request, which in this case was a GET request. The host we are connected to then responds and we retrieve the response. In order to know if our request was successfully processed, we need to check the response message from the server. A status code of 200 means that everything went alright. We can now read the data of the response and then explicitly close the connection.

This exchange is greatly simplified by the `requests` library:

```
import requests
```

```
r = requests.get("http://www.example.net")
r.close()
print r.headers
print r.content
```

We will now examine an example of a GET request with a list of parameters in the form of a Python dictionary. We will use a web service called HTTPBin which is very helpful in developing applications that make HTTP requests.

```
>>> data = {'key1': 0, 'key2': 1}
>>> r = requests.get("http://httpbin.org/get", params=data)
>>> print r.content # Our parameters now show up in the args() of the get ←
    request
```

For more information on the `requests` library, see the documentation at <http://docs.python-requests.org/>.

**Problem 3.** The file `nameserver.py` is an example of a simple HTTP server using the Python module `BaseHTTPServer`. It allows clients to send the last name of a famous computer scientist with GET and then returns the corresponding first name. For example, running the following code results in the message "Grace" from the server. The `?` signifies the start of a *query* in the URL.

```
requests.get("http://localhost:8000?lastname=Hopper").text
```

Expand the functionality of the server to do the following:

- Obtain a list of everybody whose last name starts with a given string. For example, a query of `lastname=B` would return all the names whose last name starts with the letter `"B"`.
- Similarly, obtain a list of everybody in the list of names with `lastname>AllNames`.
- Then add a method `do_PUT()` so that clients can add a person to the dictionary (or modify an existing entry) using PUT with a query of `firstname` and `lastname`. Multiple query fields separated using the `"&"` character.

The format of the list of names returns from the server should be:

```
LASTNAME, FIRSTNAME
LASTNAME, FIRSTNAME
...
...
```

Though simple, this HTTP server was the original idea for Instant Messaging. The instant messaging client on your computer sends your name, IP address, and port number to the web server.

This server then advises your contacts that you are online. If they wish to message you, all communication then happens between your port and IP address and the port and IP address of the other user without passing through the main name server.

**Problem 4.** We will now practice chatting through HTTP. To do so, create a new HTTP server in a file called `chatserver.py` using the same method as Problem 3. This new server should store a dictionary with keys of names and values as lists of messages. The GET method should return in this format:

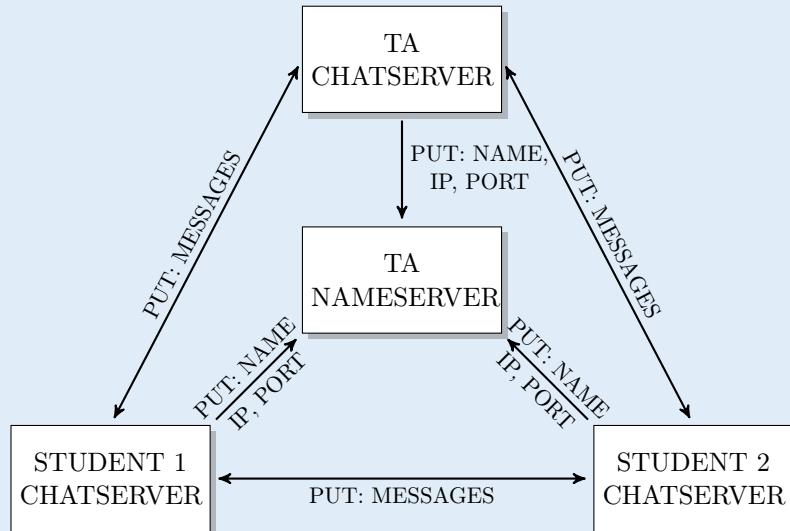
```
Name1:  
    Message 1  
    Message 2  
Name2:  
    Message 1  
    ...  
    ...
```

The PUT method should accept a `name` and a `message` and store them in your dictionary.

Once you have successfully created your server, run it through a terminal at your localhost IP address. You may then PUT your `name` and `ipaddressport` on the HTTP server provided by your instructor. To find your own IP address, run in a separate terminal, `ifconfig` (for Mac or Linux) or `ipconfig` on Windows command prompt. You may use GET requests of `AllNames` to the nameserver to retrieve the names of other active users. To send them a message, use PUT requests to their respective IP address and port.

Sample requests are as follows:

```
# Request names and IP addresses from nameserver.  
r = requests.get("http://ipaddress:portnumber?name=AllNames")  
# Send a message to another persons chat server.  
r = requests.put("http://ipaddress:portnumber?name=Thomas&message=Hello ←  
World")
```



To receive credit for this problem, please do the following:

1. PUT your name, IP address, and port number on the provided nameserver.
2. Send a message with your name to the *TA* user IP address and port.
3. Accomplish any additional tasks specified by your instructor.

When you have finished chatting, update your IP address to an empty string in the name server.

**ACHTUNG!**

This problem requires a level of remote network access. If the network doesn't allow you to connect, use SSH to remote into a network location where this can be done.

# 11

## Web Technologies 2: Data Serialization

**Lab Objective:** *Understand how serialization is used in web technologies. Practice using serialization to pack, transport, unpack, and navigate data structures in order to more easily utilize web based data sources.*

In order to more easily store and exchange data structures, standardized metalanguages have been developed to *serialize* data. Serialization is the process of packaging data with special properties in a form that can be easily unpacked and reconstructed as an identical copy on any computer. For example, if you wanted to share a k-d tree filled with data, you can easily send and replicate it on a colleague's computer by using serialization without having to send raw data and run the k-d sorting algorithm again. This can be used to transport exact data structures, or just to send data in an organized fashion, even between different programming languages. There are many different kinds of serialization methods for different languages and with different purposes, however, we will focus on serialization with the XML and JSON languages. These are two of the most prevalent serialization languages used for web communication and web design, but are commonly used to transport all programming languages. Their main application in web communication is in the transportation of organized data.

### JSON

JSON stands for *JavaScript Object Notation*. This serialization method stores information about the objects as a specially formatted string that is easy for both humans and machines to read and write. When JSON is deserialized, this special string is parsed and the objects are recreated. Despite its name, it is a completely language independent format. JSON is built on top of two types of data structures: a collection of key/value pairs and an ordered list of values. In Python, these data structures are more familiarly called dictionaries and lists respectively.

The following is a very simple example of the characteristics of a family expressed in JSON:

```
{  
    "lastname": "Smith"  
    "father": "John",  
    "mother": "Mary",  
    "children": [  
        {  
            "name": "Timmy",  
            "age": 10  
        },  
        {  
            "name": "Sarah",  
            "age": 12  
        }  
    ]  
}
```

```

        "age": 8
    },
    {
        "name": "Missy",
        "age": 5
    }
]
}

```

**NOTE**

You have likely been working very closely with JSON without even knowing it! Jupyter Notebooks are actually stored as JSON. To see this, open one of your `.ipynb` files in a basic text editor.

In general, the JSON libraries of various languages have a fairly standard interface. Though the Python standard library has modules for JSON, if performance is critical, there are additional modules for JSON that are written in C such as `ujson` and `simplejson`.

## Serialization using JSON

Let's begin by serializing some common Python data structures.

```

>>> import json
>>> ex1 = [0, 1, 2, 3, 4]
>>> json.dumps(ex1)
'[0, 1, 2, 3, 4]'
>>> ex2 = {'a': 34, 'b': 483, 'c':"Hello JSON"}
>>> json.dumps(ex2)
'{"a": 34, "c": "Hello JSON", "b": 483}'

```

The JSON representation of a Python list and dictionary are very similar to their respective string representations. Each of these generated strings is called a JSON *message*. Since JSON is based on a dictionary-like structure, you can nest multiple messages by distributing them appropriately within curly braces.

```

>>> aJSONstring = """{"car": {
    "make": "Ford",
    "model": "Focus",
    "year": 2010,
    "color": [255, 30, 30]
  }
}"""
>>> t = json.loads(aJSONstring)
>>> print t
{u'car': {u'color': [255, 30, 30], u'make': u'Ford', u'model': u'Focus', u'year': 2010}}

```

```
>>> print t['car']['color']
[255, 30, 30]
```

To generate a JSON message, use `dump()`. This method accepts a Python object, generate the message, and writes it to a file. The method `dumps()` does the same, but returns the string instead of writing it to a file. To perform the inverse operation, use `load()` or `loads()` to read a file or string, respectively.

The built-in JSON encoder/decoder only has support for the basic Python data structures such as lists and dictionaries. Trying to serialize an object which is not recognized will result in an error. Below is an example trying to serialize a set.

```
>>> a = set('abcdefg')
>>> json.dumps(a)
-----
TypeError: set(['a', 'c', 'b', 'e', 'd', 'g', 'f']) is not JSON serializable
```

The serialization fails because the JSON encoder doesn't know how it should represent the set. However, we can extend the JSON encoder by subclassing it and adding support for sets. Since JSON has support for sequences and maps, one easy way would be to express the set as a map with one key that tells us the data structure type, and the other containing the data in a string. Now, we can encode our set.

```
class CustomEncoder(json.JSONEncoder):
    def default(self, obj):
        if isinstance(obj, set):
            return {'dtype': 'set',
                    'data': list(obj)}
        return json.JSONEncoder.default(self, obj)

>>> a = set('abcdefg')
>>> json.dumps(a, cls=CustomEncoder)
'{"dtype": "set", "data": ["a", "c", "b", "e", "d", "g", "f"]}'
```

Though this is helpful, decoding this string would give us all of our data in a list. To allow for this string to be decoded as a Python set, we must build a custom decoder. Notice that we don't need to subclass anything.

```
>>> accepted_dtotypes = {'set': set}
>>> def CustomDecoder(item):
...     type = accepted_dtotypes.get(item['dtype'], None)
...     if type is not None and 'data' in item:
...         return type(item['data'])
...     return item

>>> c = json.loads(s, object_hook=CustomDecoder)
{u'a', u'b', u'c', u'd', u'e', u'f', u'g'}
# The 'u' is a prefix that signifies that the string value is Unicode.
# You can test this with:
>>> print c[0]
```

a

**Problem 1.** Python has a module in the standard library that allows easy manipulation of times and dates. The functionality is built around a `datetime` object.

```
>>> import datetime
>>> now = datetime.datetime.now()
>>> print now
2016-10-04 11:52:33.513885

# We can also extract the individual time units
>>> now.year
2016
>>> now.minute
33
>>> now.microsecond
513885
```

However, `datetime` objects are not JSON serializable. Determine how best to serialize and deserialize a `datetime` object, then write a custom encoder and decoder. The `datetime` object you serialize should be equal to the `datetime` object you get after deserializing.

## APIs and JSON

Many websites and web APIs (Application Program Interface) make extensive use of JSON. For example, almost any programs that utilize Twitter, Facebook, Google Maps, or YouTube communicate with their APIs using JSON. This means that any website that uses an embedded version of Google Maps will receive JSON strings with data to display Google Maps interface on a portion of the webpage. It also allows developers to receive information and update the embedded portions of their page without needing to reload the webpage file. An example of this is available at <https://developers.google.com/maps/documentation/javascript/examples/map-simple>. There are also web APIs which allow developers to retrieve the website data in JSON strings. A list of APIs for public data collection can be found at [http://catalog.data.gov/dataset?q=-aapi+api+OR+res\\_format%3Aapi#topic=developers\\_navigation](http://catalog.data.gov/dataset?q=-aapi+api+OR+res_format%3Aapi#topic=developers_navigation)

### ACHTUNG!

Each website has a policy about data usage and automated retrieval that requires certain behavior. If data is scraped without complying with these requirements, there can be legal consequences.

**Problem 2.** JSON files are often used by APIs to respond to data requests. To demonstrate an example of this, we will do a brief data examination of water usage in Los Angeles, California. The City of Los Angeles publishes some water usage data for the public. The API *endpoint* at which this data is available is at the address <https://data.lacity.org/resource/v87k-wgde.json>.

We can use the `requests` library from the previous lab to GET the data from the page.

```
requests.get("https://data.lacity.org/resource/v87k-wgde.json").json()
```

This code will load the object from JSON format into a Python list. Once the list has been created, gather the water usage data from 2012 to 2013 and the longitude and latitude of each point.

Use Bokeh to plot these points on a map of Los Angeles (refer to Lab 10 for additional help on Bokeh).

To draw a map centered on Los Angeles, you may use the following code:

```
from bokeh.plotting import figure
from bokeh.models import WMTSTileSource

fig = figure(title="Los Angeles Water Usage 2012-2013", plot_width=600, ←
    plot_height=600, tools=["wheel_zoom", "pan", "hover"],
    x_range=(-13209490, -13155375), y_range=(3992960, 4069860),
    webgl=True, active_scroll="wheel_zoom")
fig.axis.visible = False

STAMEN_TONER_BACKGROUND = WMTSTileSource(
    url='http://tile.stamen.com/toner-background/{Z}/{X}/{Y}.png',
    attribution=(
        'Map tiles by <a href="http://stamen.com">Stamen Design</a>, ' +
        'under <a href="http://creativecommons.org/licenses/by/3.0">CC BY ←
            3.0</a>.' +
        'Data by <a href="http://openstreetmap.org">OpenStreetMap</a>, ' +
        'under <a href="http://www.openstreetmap.org/copyright">ODbL</a>' +
    )
)

background = fig.add_tile(STAMEN_TONER_BACKGROUND)
```

To convert the longitude and latitude locations to be compatible with the map, you may the following code:

```
from pyproj import Proj, transform

from_proj = Proj(init="epsg:4326")
to_proj = Proj(init="epsg:3857")
```

```
def convert(longitudes, latitudes):
    x_vals = []
    y_vals = []
    for lon, lat in zip(longitudes, latitudes):
        x, y = transform(from_proj, to_proj, lon, lat)
        x_vals.append(x)
        y_vals.append(y)
    return x_vals, y_vals
```

More information about this dataset is available at <http://catalog.data.gov/dataset/residential-water-usage-zip-code-on-top-cb2ac>.

## XML

XML is another data interchange metalanguage. However, it is a markup language rather than a object notation language. This means that it utilizes tags to distinguish the formatting of data rather than just encoding them. So, broadly speaking, XML is somewhat more robust and versatile than JSON, but slightly less efficient. Due to this minor difference, it is utilized in different ways.

To understand XML, we need to further explore what tags are. A tag is a special command enclosed in angled brackets (< and >) that describes properties of the data enclosed. For example, we can represent a car's properties in the XML below. Notice that tags are used to both open and close a property.

```
<car>
    <make>Ford</make>
    <model>Focus</model>
    <year>2010</year>
    <color model='rgb'>255,30,30</color>
</car>
```

XML data can be read as a tree or as a stream.

Since XML is a hierarchical storage format, it is very easy to build a tree of the data. The advantage of a tree format is random access to any part of the document at any time. However, this requires all of the XML to be loaded into memory for construction of the tree.

Reading a document as a stream means reading each piece sequentially or only reading a small portion of the file at a time. Because memory usage is constant, there is no limit to size of XML document that we can process this way, however, we do not have the random access of a tree.

The following will explore two APIs that parse XML formatted files and strings.

## DOM

The DOM (Document Object Model) API allows you to work with an XML document as a tree in a hierarchy of elements. In order to sort the tree, the XML tags are read from the file and distributed accordingly. The DOM tree of the car from the XML code above would have `<car>` at the root element. This root element would have four children, `<make>`, `<model>`, `<year>`, and `<color>`. After a DOM tree has been sorted, we can traverse it like any other tree structure or search it by tag. Python's XML module includes two version of DOM: `xml.dom` and `xml.minidom`. MiniDOM is a minimal, more simple implementation of the DOM API.

## SAX

SAX, Simple API for XML, is a very fast and efficient way to read an XML file. The main advantage of this method is memory conservation. A SAX parser will read XML sequentially instead of all at once. As the SAX parser iterates through the file, it emits events at either the start or the end of tags. You can provide functions to handle these events.

## ElementTree

ElementTree is Python's unification of DOM and SAX APIs into a single, high-level API for parsing and creating XML trees. ElementTree provides a SAX-like interface for reading XML files via its `iterparse()` method. This will have all the benefits of reading XML via SAX. In addition to stream processing the XML, it will build the DOM tree as it iterates through each line of the XML input. ElementTree provides a DOM-like interface for reading XML files via its `parse()` method. This will create the tag tree that DOM creates.

We will demonstrate ElementTree using the following XML file.

```

1  <?xml version="1.0"?>
2  <contacts>
3      <person>
4          <firstname>John</firstname>
5          <lastname>Doe</lastname>
6          <phone type="mobile">1234567890</phone>
7          <phone type="home">5432229875</phone>
8          <email type="home">doughboy@bakery.com</email>
9          <address type="home">34 South Street, Jonesville</address>
10         <groups>personal,work</groups>
11     </person>
12     <person>
13         <firstname>Sally</firstname>
14         <lastname>Sue</lastname>
15         <phone type="mobile">8372289491</phone>
16         <groups>personal</groups>
17     </person>
18     <person>
19         <firstname>Thor</firstname>
20         <lastname></lastname>
21         <phone type="mobile"></phone>
22         <email type="home"></email>

```

```

24      <address type="home"></address>
25      <groups>work</groups>
26  </person>
27 </contacts>
```

contacts.xml

First, we will look at viewing an XML document as a tree similar to the DOM model described above.

```

import xml.etree.ElementTree as et

f = et.parse('contacts.xml')

# To manually traverse the tree:
# We iterate through the element directly
# Note: getchildren() is old and deprecated (not supported), so we instead use ←
#       list() to gather children.
root = f.getroot()
children = list(root) # root has three children
person0 = children[0]
fields = list(person0) # The children elements of person0
```

We can search the entire tree for specific elements by:

```

# Searching for all tags equal to firstname
for n in root.iter('firstname'):
    print n.text
```

We can also filter with multiple tags by:

```

seek = {'firstname', 'lastname', 'phone'}
fi = (x for x in root.iter() if x.tag in seek)
for n in fi:
    print n.text
```

We can even modify the document tree in place.

```

# To remove Thor:
for n in root.findall("person"):
    if n.find("firstname").text == 'Thor':
        root.remove(n)

# Verify that Thor is really gone
for n in root.iter('firstname'):
    print n.text
```

Next, we will look at ElementTree's `iterparse()` method. This method is very similar to the SAX method for parsing XML. There is one important difference. ElementTree will still build the document tree in the background as it is parsing. We can prevent this by clearing each element by calling its `clear()` method when are finished processing it.

```
f = et.iterparse('contacts.xml') # This is an iterator that you can use to go ←
    forward and backward in the tree
for event, tag in f:
    print "{}: {}".format(tag.tag, tag.text)
    tag.clear()
```

We can also get both start and end events, however, start events are mostly useful for looking at attributes or to trigger some other action on element starts. The element is not guaranteed to be complete until the end event.

```
for event, tag in et.iterparse('contacts.xml', events=('start', 'end')):
    print "{} {}<{}>: {}".format(event, tag.tag, tag.attrib, tag.text)
```

**Problem 3.** Using ElementTree to parse `books.xml`, which represents a small dataset of books, and answer the following questions. Include the code you used with your answers.

- 1) Who is the author of the most expensive book in the list?
- 2) How many of the books were published before Dec 1, 2000?
- 3) Which books reference Microsoft in their descriptions?

HINT: To answer these queries, it may be most convenient to populate a pandas DataFrame with all the XML data.

**Problem 4.** The City of New York makes publicly available some data concerning the location of publicly available recycling bins. The XML endpoint is located at <https://data.cityofnewyork.us/api/views/sxx4-xhzg/rows.xml?accessType=DOWNLOAD>. Using the `requests` library, GET the XML file from that address and build it into an Element Tree.

Then, using Monte Carlo approximation, estimate the average distance from a given point in New York City to the nearest recycling bin. One efficient way of solving this problem is to perform a nearest neighbor search using a KDTree. We recommend you use scipy's `cKDTree` and its `query` method to find the nearest neighbor for each point.

Here's a quick example of how to use `cKDTree`.

```
>>> import numpy as np
>>> from scipy.spatial import cKDTree

>>> pts = np.random.rand(10,2)
>>> kdtree = cKDTree(pts)
>>> query_pts = [[0.5, 0.5], [0.25, 0.25]]
# k=1 corresponds to finding the nearest neighbor
>>> distance, index = kdtree.query(query_pts, k=1)
```

```
# 'distance' returns the distance to the nearest neighbor  
# 'index' returns the index in 'pts' that corresponds to the  
#     nearest neighbor
```

The `random_newyork_locations.csv` file contains about 35,000 uniformly distributed points throughout New York City measured in longitude and latitude.

We could measure the distance between the longitude and latitude points, however the units would not have a very interpretable and relatable value. Therefore, use the `convert()` function in Problem 2 to transform your longitude and latitude points to epsg:3857. These units very closely approximately meters.

Return your final answer in miles.

More information on this dataset is available at <https://catalog.data.gov/dataset/public-recycling-bins-eb48e>.

(Hint: If you get the file from requests, you may want to remove '<response>' and '</response>' from the beginning and end of your content string to make parsing a little easier.)

## Additional Material

### Pickle

Aside from the serialization methods we have demonstrated, Python has a serialization library called *pickle*. This library makes it very easy to serialize and share your python objects. The following code is a simple example of how to create a pickled object and then unpack it.

```
>>> import pickle

>>> listobject = [1, 2, 3, 4, 5]
>>> item = pickle.dumps(listobject)
>>> item
'(lp0\nI1\nI2\nI3\nI4\nI5\n.' 

>>> pickle.loads(item)
[1, 2, 3, 4, 5]
```

You can also write these pickled objects to a text file as strings. The following code demonstrates this.

```
>>> a = open('list.txt', 'w')

>>> listobject = [1,2,3,4,5]
>>> pickle.dump(listobject, a)
>>> a.close()

>>> a = open('list.txt')
>>> pickle.load(a)
[1, 2, 3, 4, 5]
```

Pickle has many powerful applications such as these for communication between Python users. See <https://docs.python.org/2.7/library/pickle.html> for more documentation.



# 12

## BeautifulSoup

**Lab Objective:** Virtually everything rendered by an internet browser as a web page uses HTML. As a result, learning HTML is key to any kind of internet programming. BeautifulSoup is a Python package that helps navigate HTML documents. In this lab, we learn how to load HTML documents into BeautifulSoup and navigate the resulting BeautifulSoup object.

### HTML

HTML, or Hyper Text Markup Language is the standard markup language to create webpages. Just like XML, HTML tags describe different document content and are surrounded by angle brackets. Most tags can be combined with attributes such as `id` or `class` to help identify individual tags and make navigating the HTML tree much more simple. A list of all the current HTML tags can be found at <http://htmldog.com/reference/htmltags>. Here is an example:

```
<html>
  <body>
    <p>
      Click <a id='info' href='http://www.example.com'>here</a> for more ↵
      information.
    </p>
  </body>
</html>
```

The above example would output a single line

```
Click here for more information.
```

with ‘here’ being a clickable link to the website <http://www.example.com>.

While less readable, this HTML code can also be written as a single line as follows:

```
<html><body><p>Click <a id='info' href='http://www.example.com/info'>here</a> ↵
  for more information.</p></body></html>
```

If a given tag doesn’t contain any text or other tags, it can be written in a single pair of brackets as

```
<*tag_name*  ... *attributes*/>
```

The HTML of a website is very easy to view. Go to any website, such as `http://www.example.com`, in whatever browser is most convenient. Once on the website, right click with the mouse pointer and look for ‘View Page Source’ or a similarly worded option. Click it and the browser will open a new browser with the HTML code for your site. Some code is easy to follow, other code not so much.

**Problem 1.** Go to the website `http://www.example.com` and open the source code. What are all the tags used? What is the value of the `type` attribute associated with the `style` tag? Write a function that returns a list of all the tags used and the value of the `type` attribute.

## Loading HTML into BeautifulSoup

Now that we know what HTML is, we can use BeautifulSoup to create a `BeautifulSoup` object. BeautifulSoup is a library capable of pulling data out of HTML scripts and files. BeautifulSoup works with a parser to provide commands to navigate and search the resulting HTML tree. Make sure the module `bs4` is installed in your Python packages. This section takes most of its material from `http://www.crummy.com/software/BeautifulSoup/bs4/doc/index.html`.

First we want a variable to store our HTML code as a string.

```
>>> doc = """
...     <html><body><p>
...     Click <a id='info' href='http://www.example.com/info'>here</a> for more<-
...     information.
...     </p></body></html>
...     """
```

Next, we import `BeautifulSoup` from the `bs4` module. We call `BeautifulSoup()`, which takes as parameters the HTML string and an HTML parser. It returns a `BeautifulSoup` object, which represents the document as a nested data structure.

```
>>> from bs4 import BeautifulSoup
>>> soup = BeautifulSoup(doc, 'html.parser')
```

Including an HTML parser is optional, but a warning is given if one is not included. If that’s the case, BeautifulSoup uses the HTML parser included in Python’s standard library. Although other parsers are permitted, we have no need for them in our examples.

Once the document is stored, we can use `prettyify()` to view the HTML. The `prettyify()` method returns a string that can be printed to represent the `BeautifulSoup` object in a readable format. This will be useful later to make sure we are getting the correct HTML from websites.

```
>>> print(soup.prettify())
<html>
  <body>
    <p>
```

```
Click <a id='info' href='http://www.example.com/info'>here</a> for more ↵
      information.
  </p>
</body>
</html>
```

**Problem 2.** Write a function that loads the following string into BeautifulSoup, then prettifies it. Print the prettified string.

```
html_doc = """
<html><head><title>The Three Stooges</title></head>
<body>
<p class="title"><b>The Three Stooges</b></p>

<p class="story">Have you ever met the three stooges? Their names are
<a href="http://example.com/larry" class="stooge" id="link1">Larry</a>,
<a href="http://example.com/mo" class="stooge" id="link2">Mo</a> and
<a href="http://example.com/curly" class="stooge" id="link3">Curly</a>;
and they are really hilarious.</p>

<p class="story">...</p>
"""

```

### NOTE

Note that the `<html>` and `<body>` tags are never actually closed. The parser used with `bs4` will automatically close these hanging tags, so don't get too stressed out by this example.

## Navigating the HTML Tree

Since `BeautifulSoup()` returns an object which acts like a nested data structure, navigating it is very intuitive. We will use the following for the rest of the section, unless otherwise specified.

```
>>> soup = BeautifulSoup(html_doc, 'html.parser')
```

where `html_doc` is the document defined in problem 2.

### By Tag Name

Because of the way BeautifulSoup objects store HTML tags, accessing them is as simple as calling the tag name. The output will be the called tag plus any nested tags or text. Below are some examples.

```
>>> soup.head
```

```
<head><title>The Three Stooges</title></head>

>>> soup.title
<title>The Three Stooges</title>
```

It is even possible to continue navigation down the tree through tags contained within tags.

```
>>> soup.body.b
<b>The Three Stooges</b>
```

Notice there are three `<a>` tags. When there are two or more tags of the same name, calling that tag only returns the *first* tag by that name.

```
>>> soup.a
<a class="stooge" href="http://example.com/larry" id="link1">Larry</a>
```

## Tag Properties

In addition to viewing the entire tag, we can choose to access only certain properties. These properties include its name, attributes and strings (if applicable).

A tag's name is found using `.name`.

```
>>> tag = soup.a
>>> tag.name
u'a'
```

The attributes of a tag, if it has them, are stored in a dictionary and can be accessed as such. Accessing all the tags at once can be done through `.attrs`. Individual tags values can be reached by calling the key associated with it. If we try to access a key that is not an attribute, we get a `KeyError` in return.

```
>>> tag.attrs
{u'class': [u'stooge'], u'href': u'http://example.com/larry', u'id': u'link1'}

>>> tag['class']
[u'stooge']

>>> tag['href']
u'http://example.com/larry'

>>> tag['id']
u'link1'
```

Note that `class` returns a list. This is because the `class` attribute can have more than one value. This may show up in some HTML trees, but is not very common.

If a tag contains any text, it can accessed with `.string`.

```
>>> tag.string
u'Larry'
```

**Problem 3.** Using what you have just learned, write a function that returns the following from the Three Stooges example:

```
[u'title']
```

(Hint: This is the attribute in the eighth line of the prettified string from the previous problem.)

## By Family Relations

Once we have selected a tag, we have several options available to navigate up, down, and sideways through an HTML tree.

### Going Down

As mentioned before, a tag may contain other nested tags or text. These are called its children. Calling `.contents` returns the children of the parent tag in a list.

```
>>> head_tag = soup.head
>>> head_tag
<head><title>The Three Stooges</title></head>

>>> head_tag.contents
[<title>The Three Stooges</title>]

>>> title_tag = head_tag.contents[0]
>>> title_tag
<title>The Three Stooges</title>

>>> title_tag.contents
[u'The Three Stooges']
```

Note that the child of `title_tag` is the string '`The Three Stooges`'. Since strings cannot have children, calling `.contents` on this string will return an error.

### NOTE

One thing to note is the following:

```
>>> children_doc = """
...     <html><head>The Three Little Pigs</head>
...     <body>
```

```

...
<p>The first little piggy</p>
...
<p>The second little piggy</p>
...
<p>The third little piggy</p>
...
</body>
...
</html>"""
>>> pig_soup = BeautifulSoup(children_doc, 'html.parser')
>>> pig_soup.body.contents
[u'\n',
 <p>The first little piggy</p>,
 u'\n',
 <p>The second little piggy</p>,
 u'\n',
 <p>The third little piggy</p>,
 u'\n']

```

In this example, each new line character in the `<body>` tag is counted as a child of `<body>`. This will be very important when trying to navigate between *siblings*, or children of a common tag.

In addition to creating a list of children with `.contents`, we can use `.children` to create a generator of children tags. Using the previous example, we get the following (remember the extra carriage returns):

```

>>> for pig in pig_soup.body.children:
...     print(repr(pig)) #use repr() to ignore escape sequences
u'\n'
<p>The first little piggy</p>
u'\n'
<p>The second little piggy</p>
u'\n'
<p>The third little piggy</p>
u'\n'

```

There is a `.descendants` attribute which will recursively go through a tag's children, then the children's children, etc. It is left to the student to look at the online documentation for this attribute.

If a tag has only one child, and that child is a string, the child is available using `.string`. If a tag has one tag, and that tag has a single string as a child, then the parent tag can use `.string` to access the string as well.

```

>>> head_tag = soup.head
>>> print(head_tag)
<head><title>The Three Stooges</head></title>
>>> title_tag = head_tag.contents[0]
>>> print(title_tag)
<title>The Three Stooges</title>
>>> head_tag.string
u'The Three Stooges'
>>> title_tag.string
u'The Three Stooges'

```

If a tag contains more than one string, `.string` return `None`. However, `.strings` returns a generator that iterates through all strings contained within a tag. Check the online documentation for examples.

## Going Up

Just as tags can have *children*, tags can also have a *parent*. To access a tag's parent, we use the `.parent` attribute.

```
>>> title_tag = soup.title
>>> title_tag
<title>The Three Stooges</title>
>>> title_tag.parent
<head><title>The Three Stooges</title></head>
```

The parent of a string is the tag that contains it.

```
>>> tag = title_tag.string
>>> print(tag)
The Three Stooges

>>> tag.parent
<title>The Three Stooges</title>
```

Calling `.parents` iterates through all parents of a given tag. Examples can be found in the online documentation.

## Going Sideways

Consider the following document, taken from the online documentation:

```
>>> sibling_soup = BeautifulSoup("<a><b>text 1</b><c>text 2</c></a>")
>>> print(sibling_soup.prettify())
<html>
  <body>
    <a>
      <b>
        text 1
      </b>
      <c>
        text 2
      </c>
    </a>
  </body>
</html>
```

Note the `<b>` and `<c>` tags are on the same level, underneath the `<a>` tag. These tags are considered *siblings*. Siblings in an HTML tree will always appear with the same indentation underneath a parent tag.

We use the attributes `.next_sibling` and `.previous_sibling` to navigate between these sibling elements. If a sibling has no next or previous sibling, calling these attributes returns `None`.

```
>>> sibling_soup.b
<b>text 1</b>

>>> sibling_soup.b.next_sibling
<c>text 2</c>

>>> sibling_soup.c.previous_sibling
<b>text 1</b>

>>> sibling_soup.c.next_sibling #<c> has no next sibling
None

>>> sibling_soup.b.string
u'text 1'

>>> print(sibling_soup.b.string.next_sibling) #text 1 and text 2 are not ←
      siblings
None
```

Recall that in the `pig_soup` example we saw extra carriage returns between the `<p>` tags.

```
>>> pig_soup.body.contents
[u'\n',
 <p>The first little piggy</p>,
u'\n',
<p>The second little piggy</p>,
u'\n',
<p>The third little piggy</p>,
u'\n']
```

What do you expect `pig_soup.body.p.next_sibling` to return?

```
>>> pig_soup.body.p.next_sibling
u'\n'

>>> pig_soup.body.p.next_sibling.next_sibling
<p>The second little piggy</p>
```

We need to make two calls to `.next_sibling` in order to get the next `<p>` tag. Keep this in mind for future questions as you navigate across siblings.

Just as with parents and children, there are also sibling generators `.next_siblings` and `.previous_siblings` to iterate through all the siblings of a given tag. These generators can be useful when multiple calls to `.next_sibling` must be made. As before, check the online documentation for more information.

**Problem 4.** Using the Three Stooges example and navigation by family relations, write a function that returns the following:

```
u'Mo'
```

**Problem 5.** Download the 'example.htm' file associated with the lab into your working directory (you can go to <http://example.com> to see the site this originates from). The following code opens and loads a file into a BeautifulSoup object:

```
>>> example_soup = BeautifulSoup(open('example.htm'), 'html.parser')
```

Write a function that returns the following line using two different methods.

```
u'More information...'
```

Have the function accept an integer `method`. If `method` is 1, return the line using your first method. If `method` is 2, return the line using the other.

## By `find()`

In actual website HTML, often there are many tags that share a common name. It would be nice to find characteristics that might be unique for a given tag. Look back at our previous examples and think about what characteristics differentiate tags with the same name. BeautifulSoup uses `.find()` to allow you to search for a tag not only by name, but also by a specific attribute value or strings. The following examples refer back to the "Three Stooges" HTML document in problem 2.

Search by name:

```
>>> soup.find('b') #Pass in tag names, just like soup.b
<b>The Three Stooges</b>

>>> #or use the name parameter
>>> soup.find(name='a') #Still only returns the first instance
<a class="stooge" href="http://example.com/larry" id="link1">Larry</a>
```

Search by attribute:

```
>>> soup.find(id='link3') #Search by unique id attribute
<a class="stooge" href="http://example.com/curly" id="link3">Curly</a>

>>> #class is a Python keyword. Use 'class_' for the attribute key.
>>> soup.find(class_='title')
<p class="title"><b>The Three Stooges</b></p>

>>> #use the attrs parameter
```

```
>>> soup.find(attrs={'id':'link3'})
<a class="stooge" href="http://example.com/curly" id="link3">Curly</a>

>>> #combine attributes
>>> soup.find(attrs={'class':'stooge', 'href':'http://example.com/curly'})
<a class="stooge" href="http://example.com/curly" id="link3">Curly</a>

>>> soup.find(class_='stooge', href='http://example.com/curly')
<a class="stooge" href="http://example.com/curly" id="link3">Curly</a>
```

Search by string:

```
>>> soup.find(string='Mo') #Recall strings act as individual units
u'Mo'

>>> soup.find(string='Mo').parent #access the tag through the parent
<a class="stooge" href="http://example.com/mo" id="link2">Mo</a>
```

Search by combining parameters:

```
>>> soup.find('a', attrs={'id':'link2', 'class':'stooge'})
<a class="stooge" href="http://example.com/mo" id="link2">Mo</a>
```

**Problem 6.** Refer to the `example.htm` file. Load it using BeautifulSoup. Write a function that returns the tag associated with the "More information..." link using two different methods. At least one of these methods should use the `.find()` function. As before, have the function accept an integer `method`. If `method` is 1, use the first method. If `method` is 2, use the second method.

**Problem 7.** Download 'SanDiegoWeather.htm' and load it into BeautifulSoup. You can find the corresponding website at [http://www.wunderground.com/history/airport/KSAN/2015/1/1/DailyHistory.html?req\\_city=San+Diego&req\\_state=CA&req\\_statename=California&reqdb.zip=92101&reqdb.magic=1&reqdb.wmo=99999&MR=1](http://www.wunderground.com/history/airport/KSAN/2015/1/1/DailyHistory.html?req_city=San+Diego&req_state=CA&req_statename=California&reqdb.zip=92101&reqdb.magic=1&reqdb.wmo=99999&MR=1). Using the `.find()` method, write a function that prints the tags referred to in the following questions:

1. What is the tag which contains the date, Thursday, January 1, 2015?
2. What are the tags which contain the links 'Previous Day' and 'Next Day'?
3. What is the tag which contains the number associated with the Actual Max Temperature?

(Hint: You can do a `ctrl+f` to find where the text is in the HTML, then study the tags around it.)

## By `find_all()`

Recall that when a tag appeared multiple times, calling that tag name would return the first tag of that name.

```
>>> soup.a
<a class="stooge" href="http://example.com/larry" id="link1">Larry</a>
```

To get all instances of a certain tag, use the `find_all()` command.

```
>>> soup.find_all('a')
[<a class="stooge" href="http://example.com/larry" id="link1">Larry</a>,
 <a class="stooge" href="http://example.com/mo" id="link2">Mo</a>,
 <a class="stooge" href="http://example.com/curly" id="link3">Curly</a>]
```

This works with all the same arguments as the `.find()` function. You may refer to the online documentation for explicit examples.

## Advanced Techniques

The following examples are techniques that can aid you in your search for specific tags. Consider the “Three Stooges” example from before. Suppose you want to find the tag that includes the url `http://example.com/curly`. You could search for it using the following:

```
>>> soup.find(href='http://example.com/curly')
<a class="stooge" href="http://example.com/curly" id="link3">Curly</a>
```

This could be annoying to type out the whole website. Instead you could use regular expressions as follows:

```
>>> import re
>>> soup.find(href=re.compile('curly')) #find href containing 'curly' in it.
<a class="stooge" href="http://example.com/curly" id="link3">Curly</a>
```

This method can be used for tag names, attributes, and strings as well.

```
>>> soup.find(string=re.compile('Cu')).parent #find tag with string that starts←
      with 'Cu'.
<a class="stooge" href="http://example.com/curly" id="link3">Curly</a>
```

To find a tag that has an attribute with a value, regardless of what the value is, we can use `True` and `False` in place of actual values. The following returns all tags that have some value associated with their `href` attribute:

```
>>> soup.find_all(href=True)
[<a class="stooge" href="http://example.com/larry" id="link1">Larry</a>,
 <a class="stooge" href="http://example.com/mo" id="link2">Mo</a>,
 <a class="stooge" href="http://example.com/curly" id="link3">Curly</a>]
```

**Problem 8.** Use BeautifulSoup to load the 'Big Data dates' file. This page can be found at <https://www.federalreserve.gov/releases/lbr/>. Note that the actual website may include more dates than the file provided. Notice all the release dates of bank data, ranging from 2003 to 2014 in the file you downloaded. Using `find_all()` and `re`, find all the links to bank data from September 30, 2003 to December 31, 2014. Write a function that loads the file into BeautifulSoup and returns a list of all of the tags containing these links.

# 13

## Advanced BeautifulSoup

**Lab Objective:** *Learn how to use BeautifulSoup to scrape information from the internet and put it into easy-to-access data tables*

The internet is full of information. Sometimes this information is easy to read, sometimes it's not. No matter the case, web scraping is a useful tool used to transform unstructured data into structured data that is easier to analyze.

### ACHTUNG!

Web scraping has legal implications. Do not scrape copyrighted information without the consent of the copyright owner. Many websites, in their terms and conditions agreement, prohibit the practice of crawling parts or all of their website. Be careful and considerate when doing any sort of crawling. Be careful when writing and testing code so that there is no unintended behavior.

## Scrapers and Crawlers

If you are on a website and all the information you want is contained on one page, you simply use a web scraper to read through the html code and pick out what you want. An example of this might be your professor's webpage, and you just need to scrape his phone number and email. Suppose instead that your information is found only after clicking through various links. For example, you want to find the email addresses and phone numbers for all the professors in the math department, but you can only get this information by clicking through an online directory, similar to <http://math.byu.edu/people/research/faculty>. This would require *crawling* through various links to open up each professor's home page, and then scraping their sites. So scrapers are good for getting the information you need from a page, while crawlers will get you to the various pages from which you want to scrape information.

## What Can You Scrape?

There are some website that permit the practice of web scraping. To ensure that scraping is well-behaved, most websites will tell a crawler where they can and cannot go, and scrapers what they can and cannot scrape. All of this information is included in a text file in the root domain of a website. The file is always titled `robots.txt` and defines considerate behaviors for web crawlers. Each robots file has a set of rules that label parts of a website as disallowed. Parts of a website that are not disallowed are implied to allow access by web crawlers. Many websites will limit crawlers to parts of the sites that will not place a large load on the website's server. It is your duty to honor the rules in `robots.txt` if they exist.

## Simple Scraping

In lab 12, BeautifulSoup was used to read short bits of HTML code or a file using the `open()` command. Once the file is loaded, you can navigate through the HTML tree and pick out the data that you want.

**Problem 1.** Go to the url <http://www.federalreserve.gov/releases/lbr/20030930/default.htm>. Download the page source as an htm file and use BeautifulSoup to load the file. Using the methods taught, write a function that returns a 2-D NumPy array for bank information. Each row in the array represents a different bank, and each column represents a different piece of bank information. In your array, include the Bank Name, Rank, ID, Domestic Assets, and number of Domestic Branches for the following banks: JPMORGAN, CAPITAL ONE, and DISCOVER.

This is a very slow way to access information from a website. Remember this was only one in a list of over 20 different links to bank information. What if we wanted to get information from every link? Imagine trying to go through HTML trees for a hundred different websites only by copying and pasting HTML code!

Use the `urllib2` library in conjunction with BeautifulSoup to load HTML code from any website. We will go through some simple examples. First import `urllib2`. Use `urllib2`'s `urlopen()` function in conjunction with `read()` to load the HTML code into Python. Then simply use `BeautifulSoup()` to turn the HTML code into a navigable HTML tree. Note `urllib2` may need to be installed or updated before use.

```
>>> from bs4 import BeautifulSoup
>>> import urllib2

>>> url = "http://csb.stanford.edu/class/public/pages/sykes_webdesign/05_simple.html"

>>> content = urllib2.urlopen(url).read()
>>> soup = BeautifulSoup(content)

>>> print(soup.prettify())
<html>
  <head>
    <title>
```

```
A very simple webpage
</title>
<basefront size="4">
</basefront>
</head>
<body bgcolor="FFFFFF">
...
</body>
</html>
```

### ACHTUNG!

Since `urllib2` accesses a website server, you may run into problems where you cannot establish a connection to the server. You can create a `try-except` clause to account for this, or just rerun your program. One possible way is as follows.

```
while True:
    try:
        content = urllib2.urlopen(url).read()
        break
    except:
        pass
```

**Problem 2.** Using the website `http://www.wunderground.com/history/airport/KSAN/2015/1/1/DailyHistory.html?req_city=San+Diego&req_state=CA&req_statename=California&reqdb.zip=92101&reqdb.magic=1&reqdb.wmo=99999&MR=1` and BeautifulSoup, return the Actual Max Temperature. Return the tag which contains the link for the 'Next Day' button. Also, return the url attached to the link.

Sometimes, data we are interested in is contained over several different web urls. For example, what if we wanted a graph of the temperature highs over a period of time? For `www.wunderground.com`, the temperature history for a given city will be contained on separate pages for each day, just as in problem 2. In order to access information over several websites, we just need to load each new website into BeautifulSoup and locate the information we're interested in. Consider the following example.

In this example we look at the actual maximum temperature over the year 2014 in San Diego.

```
from bs4 import BeautifulSoup
import urllib2
import re

weather_url = 'https://www.wunderground.com/history/airport/KSAN/2014/1/1/←
DailyHistory.html'
```

```

weather_content = urllib2.urlopen(weather_url).read()
weather_soup = BeautifulSoup(weather_content)

actual = []

while('2015' not in weather_soup.find(class_='history-date').string):
    while(len(weather_soup.find_all(string='Actual')) != 1):
        weather_content = urllib2.urlopen(weather_url).read()
        weather_soup = BeautifulSoup(weather_content)
    actual_temp = weather_soup.find(string='Max Temperature').parent.parent.←
        next_sibling.next_sibling.span.span.text
    actual.append(int(actual_temp))
    next_url = weather_soup.find(string=re.compile('Next Day')).parent['href']
    weather_url = 'https://www.wunderground.com'+next_url
    weather_content = urllib2.urlopen(weather_url).read()
    weather_soup = BeautifulSoup(weather_content)

```

Let's examine the code to see how it works.

After importing the necessary modules and opening up the url in BeautifulSoup, we define the variable `actual` to store the max temperatures in a list. Notice in the url that our dates start in 2014. Since we only want to go through the year 2014 and stop in 2015, we need a way to end our search once we get into 2015. Using `class_='history-date'`, we can find a tag which has the year in the string.

The next while loop is an error check. Sometimes this website does not load properly, so we check that all the information we need is located in the HTML code. In this case, we are looking for the actual max temp for a day, so we need to make sure the 'Actual' column shows up.

The next line of code defines `actual_temp`, which first directs us to the row of 'Max Temperature' and then navigates to the temperature column. This value is then turned to an `int` and stored in the list of temperatures.

Lastly, we find the url reference that is associated with the 'Next Day' link. We manually create the new url, keeping in mind that the new url found in the link is only an extension of the server website. This means that if the server is found at `http://www.wunderground.com` and the '`href`' value for the link is `/history/airport/KSAN/...`, then the new url we load into BeautifulSoup is `http://www.wunderground.com/history/airport/KSAN/...`. Therefore, we add the string representing the base url with the string representing the extension part of the url.

The output is the list of actual max temps over a year, which we can graph.

**Problem 3.** Adjust the above code to write a function that makes a list of the average max temperatures over a year. Graph this list, then return it. Note that because this code goes through a whole year's worth of urls, the function will take a long time to run.

Let's do another example before we turn you lose on the internet. This next example will look at the Commercial Bank data found at `http://www.federalreserve.gov/releases/lbr`. We will look primarily at the Consolidated Assets for JPMorgan over the years from 2003 to 2014.

```

from bs4 import BeautifulSoup
import urllib2

```

```

import re

bank_url = 'http://www.federalreserve.gov/releases/lbr/'
bank_content = urllib2.urlopen(bank_url).read()
bank_soup = BeautifulSoup(bank_content)

assets = []

dates_list = bank_soup.find_all(href=re.compile('((200[3-9])|(201[0-4])).*/'←
    default.htm'))

for url in dates_list:
    link_url = str('http://www.federalreserve.gov/releases/lbr/' + url['href'])
    link_content = urllib2.urlopen(link_url).read()
    link_soup = BeautifulSoup(link_content)
    amt = link_soup.find(string=re.compile('JPMORGAN')).parent.next_sibling.←
        next_sibling.next_sibling.next_sibling.next_sibling.next_sibling.←
        next_sibling.next_sibling.next_sibling.next_sibling
    assets.append(amt.string)

```

Now we will examine this code and see how it works.

We first import the necessary modules and load the url into BeautifulSoup. We create the variable `assets` to store the list of asset values.

To get the links we need, we use `find_all()` and select the unique identifiers of the link, namely that the links start with 20\*\* and end in /`default.htm`. Next we run the `for` loop to iterate through each link.

Just as in the previous example, we need to concatenate the server url `http://www.federalreserve.gov/release/lbr/` with the extension urls we stored in `dates_list`. This link is loaded into BeautifulSoup.

Now that we are on the webpage desired, we find the row we want by looking for the tag containing '`JPMORGAN`'. Once we have the tag, we navigate to the appropriate column. Notice we need to use 10 calls to `.next_sibling` in order to get to the correct information. The info is then appended into the data list.

**Problem 4.** Choose 1 of 3 options.

1. Load `http://www.google.com/finance` into BeautifulSoup. Towards the bottom of the web page, there is a Sector Summary. Go through each sector and locate the top five Gainers. In a SQL table, store the Name, abbreviation, % Change, and Mkt Cap of the top Gainer for each Sector.
2. Load `http://www.espn.go.com/nba/statistics` into BeautifulSoup. Go through the top five offensive leaders. In a SQL table, store the name, career games played, career mins per game, career points per game, and career FG% for each player.

3. Load <http://www.foxsports.com/soccer/united-states-women-team-stats> into BeautifulSoup. Go through each player on the World Cup US women's team. In a SQL table, store the name, hometown, position, and # of games played in the World Cup.

## Advanced Scraping

The examples we have looked at so far have been very basic since the HTML is stored in the source code for the web pages. However, we will look at some examples where the HTML is written dynamically. This means the HTML is brought in from a separate source through javascript or ajax as a .php or .aspx table. These tables can be difficult to grab data from.

Go to the website <http://www.simplesoccerstats.com/stats/teamstats.php?lge=14&type=goals&season=0>. Open up the page source by right clicking and choosing the option for the page source. There is HTML code, but is it correct? Hit **ctrl+f** and search for 'Chicago,' one of the teams that appears on the actual webpage. It's not there! You can even try the following:

```
>>> from bs4 import BeautifulSoup
>>> import urllib2
>>> import re

>>> soccer_url = 'http://www.simplesoccerstats.com/stats/teamstats.php?lge=14&←
    type=goals&season=0'
>>> soccer_content = urllib2.urlopen(soccer_url).read()
>>> soccer_soup = BeautifulSoup(soccer_content)

>>> print(soccer_soup.find(string='Chicago'))
None
```

Still nothing. This means the actual table of information is stored somewhere else.

## Selenium

Selenium is a great tool to use on simple websites as well as websites with dynamic HTML source code. Basically Selenium will open up a browser and you can see what it is looking at. You can look at the source code of the actual website, and you can have some limited control over navigation, such as clicking links, clicking dropdown menus, pressing the back or forward buttons, etc. To use Selenium, import the following:

```
from selenium import webdriver
from selenium.webdriver.common.keys import Keys
```

The `webdriver` allows you to use website functionality, while `Keys` allows you to use special keyboard keys like RETURN (i.e. when you want to send information through text boxes). Next, you want to open up a web browser and a website. For these exercises we will use Firefox. You can also use Chrome or IE if those are more familiar; all the commands will be similar.

```
driver = webdriver.Firefox()
example_url = 'http://www.example.com'
driver.get(example_url)
```

---

### NOTE

If not already installed, you will have to download both Selenium and the driver associated with the browser you would like you use. More information and links to download can be found at <https://pypi.python.org/pypi/selenium>. In addition, the browser will need to be up to date.

The `.get()` command acts like `urllib2`'s `.urlopen()` and `.read()` combination. We can print out the HTML code using Selenium's `.page_source` attribute.

```
>>> print(driver.page_source)
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"><head>
    <title>Example Domain</title>
    ...
</body></html>
```

Once we have the source code, we can read it into BeautifulSoup.

Remember how we said Selenium reads HTML from a webpage differently than BeautifulSoup? Take a look again at the soccer example.

```
>>> from selenium import webdriver
>>> from bs4 import BeautifulSoup

>>> browser = webdriver.Firefox()
>>> soccer_url = "http://www.simplesoccerstats.com/stats/teamstats.php?lge=14&←
    type=goals&season=0"
>>> browser.get(soccer_url)
>>> soccer_soup = BeautifulSoup(browser.page_source)
>>> browser.quit() #closes the web browser
>>> print(soccer_soup.find(string='Chicago').parent)
<td>Chicago</td>
```

This time there is a tag with 'Chicago' contained as text!

**Problem 5.** Consider the url <http://stats.nba.com/league/team/#!/?sort=W&dir=1>. Use Selenium to return a list of the `a` tags containing each of the 30 NBA teams. Use `.find_all()` in conjunction with whatever unique identifiers get you the correct tags. Hint: class and tag name are a good start.

**Problem 6.** Use the website from problem 5. Create a SQL table which stores the following information:

- The column titles are Name, HW%, AW%, where Name is each team name, HW% is the Home Win %, and AW% is the Away Win %.
- Each row represents a different basketball team, with its home and away win percentages.

Hint: You will need to use Selenium to access each teams website using the links from the tags found in problem 5. If the websites do not load properly, consider a `try-except` clause like the one suggested previously.

# 14

## MongoDB

**Lab Objective:** *In this lab we introduce MongoDB, a non-relational database management system that is well suited to handling expanding datasets and dynamic storage. MongoDB communication is formatted almost entirely as JSON strings, and includes many of the same properties. We will use some of the more common MongoDB commands to investigate the architecture of a Mongo database and to explore similarities and differences between different documents.*

### NoSQL Databases

Relational databases, such as SQL, were the most popular databases of the last decade. These databases rely on the data having relational attributes, meaning that each item in the database has the same attributes. We can visualize these databases as tables. As time passed and needs changed, relational databases became too structured for sets of data which involved unique attributes or were rapidly changing and expanding. In databases of this kind, each item may not have the same attributes. For example, in a database of wildlife, a salamander and an apple both have attributes of size and color, but an apple does not need a gender attribute and a salamander does not need a ripeness attribute. Relational databases store items with the same attributes, but if we want to store a salamander and an apple in the same database, we need a different type of database. Non-relational databases are built to allow for these types of databases and have opened the door to massively scalable databases that can store data in many forms and physical locations.

With these new databases, a new family of database managers arose to properly interface with them. Instead of designing a new relational database to meet every need, non-relational database managers were created that can adapt the different items for specific scenarios. MongoDB is such a manager. Several other managers, such as Cassandra, Redis, and Neo4j serve similar purposes. In this lab, we will focus on MongoDB.

### MongoDB

MongoDB is a document database. It is best suited for storing data that does not have a fixed schema. Each MongoDB database is made up of collections of one or more documents. These documents are a special type of JSON object called BSON (Binary JSON). For the most part, BSON objects, JSON objects, and Python dictionaries can be used in much the same manner. However, there are a few subtle differences, such as with special characters. Trying to use a Python dictionary that contains the ‘\$’ character will often throw errors if it is used as though it were a BSON object.

MongoDB, being a database server manager, needs to have a location for the target database where it can store its data. Thus, we will need to run a database on a dedicated terminal window. If you do not have the MongoDB service running on your machine with a target database, please refer to the additional material section of this lab and follow the instructions to run an isolated test database server on your machine.

MongoDB has both a command line interface and Python bindings. This lab will use the official supported Python bindings, Pymongo. To install, you may use a package manager such as pip or download the binaries from the Pymongo website. More information for installation may be found at <http://api.mongodb.com/python/current/installation.html>.

After installation, Pymongo can be imported as with other standard libraries as follows:

```
from pymongo import MongoClient
...
Create an instance of a client connected to a database running
at the default host ip and port.
...
mc = MongoClient()
```

The following example illustrates a good use for MongoDB: Suppose you are running a general store. You have all sorts of inventory: food, clothing, tools, toys, etc. There are some attributes that every item has: name, price, and producer. Then there are attributes held by only some items: color, weight, gluten-freedom. A SQL database would have to be full of mostly-blank rows, which is extremely inefficient. More importantly, as you add new inventory, you will run across new attributes. With SQL, you would have to restructure and rebuild the whole database each time this happens. For MongoDB, this isn't a problem because it doesn't use the same relation tables. Instead, each item is a JSON-like object (similar to a Python dictionary), and thus can contain whatever attributes are relevant to the specific item, without including any meaningless attributes.

## Creating and Removing Collections and Documents

A MongoDB database stores collections, and a collection stores documents. Each database can have several collections, each with its own documents. To visualize this, imagine we have a set of paper documents. We put the documents into folders (collections), and the folders into a filing cabinet (the database). When we need to add another collection, we simply create a reference to it and put it in the database. The new collection will not actually be created until we add documents to it, just as we would not file away a folder into the filing cabinet with all the rest until we have a document to be put into the folder. You can create a database and collection as follows:

```
# Create a new database
db = mc.db1

# Create a new collection
col = db.collection1
```

Documents in MongoDB are represented as JSON-like objects, and so do not adhere to a set schema. Each document can have its own fields.

```
col.insert({'name': 'Jack', 'age': 23})
```

```
col.insert({'name': 'Jack', 'age': 22, 'student': True, 'classes': ['Math', '←
    Geography', 'English']})
x = col.insert({'name': 'Jill', 'age': 24, 'student': False})
```

We can check to see if the insert was successful by calling `x.is_valid(x)`.

**Problem 1.** Create a MongoDB database called `mydb` and a collection in `mydb` called `rest`. The file `restaurants.json` contains thousands of JSON objects, each describing a single restaurant. Load these into `rest`. The `json.loads` method should be helpful in doing this.

## Querying for Documents

MongoDB uses a *query by example* paradigm for querying. This means that when you query, you provide an example that the database uses to match with other documents.

```
# Querying methods return a Cursor object which iterates through the result set←
.
r = col.find({'name': 'Jack'})
```

This query will return all documents in the collection that have the value ‘Jack’ in the ‘name’ field. You can also use the `count` method to return the number of documents that match your desired criteria.

```
# Find how many 'students' are in the database
col.find({'student':True}).count()
```

We can update documents in a collection using `update`. Note that a simple update acts like a replace.

```
col.update({'name': 'Jack', 'student': True})
```

**Problem 2.** The file `mylans_bistro.json` contains a json object describing one additional restaurant. Insert it into the collection. Note that this entry contains an additional key value not present in any other. A SQL database would have to be entirely rebuilt to support this insertion, but with MongoDB this is not an issue.

After this insert, use a query to list every restaurant that closes at eighteen o’clock (Mylan’s Bistro should be one of these).

## Query Operators

There are several special operators that we can use to define conditions in a query. These query operators are used as keys and the queries are values.

Operator	Description
\$lt, \$gt	<, >
\$lte	$\leq$ , $\geq$
\$in, \$nin	Match any value in, not in an array, respectively
\$or	Logical OR
\$and	Logical AND
\$not	Logical negation
\$nor	Logical NOR (condition fails for all clauses)
\$exists	Match documents with specific field
\$type	Match documents with values of a specific type
\$all	Match arrays that contain all queried elements

Table 14.1: MongoDB query operators

```
f = list(col.find({'age': {'$lt': 24}, 'classes': {'$in': ['Art', 'English']}})←
    )
```

**Problem 3.** Query your new collection to answer the following questions:

- How many of the restaurants are in Manhattan?
- How many restaurants have gotten a grade other than an “A” on a health inspection?
- Which are the ten northernmost restaurants?
- Which restaurants have “grill” (case-insensitive) in their names?

Understand that MongoDB is not a relational database, therefore there is no concept of a join. This also means that we cannot define database relationships between documents. We can associate two documents by including a field that contains the unique ObjectId of the other document. When we request one document, we see it has an ObjectId, and then we run a second query to get the other document. Any “relational” things must be handled by the developer. This means that a document needs to contain all the information needed to find or retrieve it again.

**Problem 4.** Use update operators to perform the following tasks:

- Whenever a restaurant has “grill” in its name, replace “grill” with “Magical Fire Table”.
- Increase all of the restaurant IDs by 1000.
- Delete the entries of every restaurant that has ever gotten a “C” health inspection grade.

## Additional Material

### Installation of MongoDB

MongoDB runs as an isolated program with a path directed to its database storage. To run a practice MongoDB server on your machine, complete the following steps:

#### Create Database Directory

To begin, navigate to an appropriate directory on your machine and create a folder called **data**. Within that folder, create another folder called **db**. Make sure that you have read, write, and execute permissions for both folders.

#### Retrieve Shell Files

To run a server on your machine, you will need the proper executable files from MongoDB. The following instructions are individualized by operating system. For all of them, download your binary files from <https://www.mongodb.com/download-center?jmp=nav#community>.

##### 1. For Linux/Mac:

Extract the necessary files from the downloaded package. In the terminal, navigate into the **bin** directory of the extracted folder. You may then start a Mongo server by running in a terminal: `mongod --dbpath /path to your data folder`.

##### 2. For Windows:

Go into your Downloads folder and run the Mongo **.msi** file. Follow the installation instructions. You may install the program at any location on your machine, but do not forget where you have installed it. You may then start a Mongo server by running in command prompt: `C:\locationofmongoprogram\mongod.exe --dbpath C:\path to data folder\data\db`.

MongoDB servers are set by default to run at address:port `127.0.0.1:27107` on your machine.

You can also run Mongo commands through a mongo terminal shell. More information on this can be found at <https://docs.mongodb.com/getting-started/shell/introduction/>.



# 15

## Parallel Computing with ipyparallel

**Lab Objective:** *Most computers today have multiple processors or multiple processor cores which allow various processes to run simultaneously. To perform enormous computations, "supercomputers" or computer clusters that combine many processors and a great deal of memory are commonly used. In this lab, we will explore the basic principles of designing code that fully utilizes available resources for parallel computing using the `iPyParallel` python package.*

### Why Parallel Computing?

When a single processor takes too long to perform a computationally intensive task, there are two simple solutions: build a faster processor or use multiple processors to work on the same task. Unfortunately, as processors have become smaller, it has become increasingly difficult to dissipate the heat they produce. This problem is called the "heat wall" and has presented a currently insurmountable barrier. Therefore, the second solution has taken precedence and seen incredible growth in the past two decades. Though there are many different architectures for parallel computing, essentially, a 'supercomputer' is made up of many normal computers which share or use their own memory.

In the majority of circumstances, these processors communicate with each other and coordinate their tasks with a message passing system. The details of this message passing system, MPI, will be the topic of the next lab.

In this lab, we will become familiar with some of the basic ideas behind parallel computing.

### Serial Execution vs. Parallel Execution

Up to this point, all the programs you have written are executed one line at a time, or in *serial*. The following exercise will help visualize the serial process of a program.

**Problem 1.** If you are working on a Linux computer, open a terminal and execute the `htop` command. (If `htop` is not on your system, install it using your default package manager). When opening this program, your terminal should see an interface similar to Figure 15.1. The numbered bars at the top represent each of the cores of your processor and the workload on each of these cores.

Now, run the following python code with your terminal running `htop` still visible. The sole purpose of the following code is to create a computationally intensive function that runs for about 15 seconds.

```
import numpy as np
for i in xrange(10000):
    np.random.random(100000)
```

You should have seen one of the cores get maxed out at 100%. It is also possible that you saw the load-carrying core switch midway through the execution of the file. This is evidence one indicator that our script is being executed in serial – one line at a time, one core at a time.

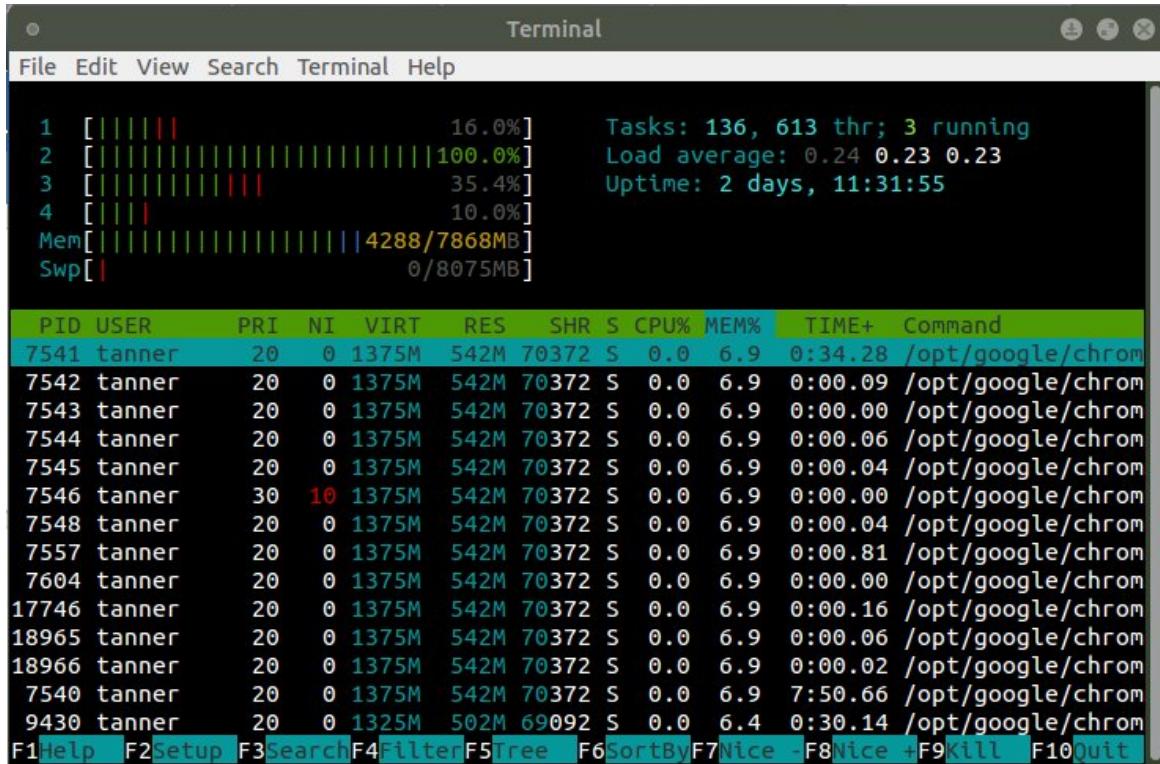


Figure 15.1: An example of `htop` with a computationally intense python script running.

As you saw in the exercise above, only one of the cores was carrying the load at a time (if it was more than one, your computer has default code to try distributing some of the process). This means that we are only using a fraction of the computer's resources. When working on a personal computer, this would often be to your benefit because dividing jobs among multiple cores is part of what makes smooth multitasking possible. However, in the event you wish to devote all the computer's resources to executing a certain program, we can employ the help of the `ipyparallel` module. In theory, you can make your code run  $N$  times faster when executing in parallel where  $N$  is the number of cores you have access to.

## The ipyparallel Module

We will begin our discussion on parallel computing by learning about the `ipyparallel` module. Since Python is a relatively slow scripting language, and since the main purpose of parallel computing is to speed up run time, most parallel computing in application is done in a language other than Python. Though this may not be the fastest parallel computing framework available, it is still fairly easy to take advantage of all the cores available to speed up run time and to test parallel code logic. This is done by specifying what happens on each core, which is core principle of parallel computing.

### Installation and Initialization of ipyparallel

If you have not already installed `ipyparallel`, you may do so using the conda package manager.

```
$ conda update conda
$ conda update anaconda
$ conda install ipyparallel
```

With `ipyparallel` installed, we can now initialize an IPython cluster that is comprised of iPyParallel *engines*. By default, an engine will be started on each of your machines processor cores and a controller will be started to communicate with each of the engines. The controller can be accessed through the `Client` object, which has two classes, `DirectView`, and `LoadBalancedView`. We will discuss these classes in further detail later.

We won't go too much into the architecture of the IPython cluster, but if you are interested in learning more, visit <https://ipyparallel.readthedocs.io/en/latest/intro.html#architecture-overview>.

Now to initialize an IPython cluster, run the following code:

```
$ ipcluster start
```

If you would like to specify the number of engines to initialize, run the following:

```
# Start a cluster with 8 engines.
$ ipcluster start --n 8
```

If you choose to explicitly specify the number of engines, it is not optimal to initialize more engines than you have processors. Doing so would require multitasking on each processor instead of having each processor dedicated to one task.

If you are more accustomed to using Jupyter Notebooks, you may have noticed the "Clusters" tab. You can start an IPython cluster in this tab after enabling the `ipcluster` notebook extension.

```
$ ipcluster nbextension enable
```

**Problem 2.** Initialize an IPython cluster with an engine for each processor. As you did in the previous problem, open `htop`. Run the following code and examine what happens in `htop`.

```
from ipyparallel import Client
client = Client()
dview = client[:]
```

```
dview.execute("""
import numpy as np
for i in xrange(10000):
    np.random.random(100000)
""")
```

The output of `htop` should appear similar to Figure 15.2. Notice that all of the processors are being utilized to run the script.

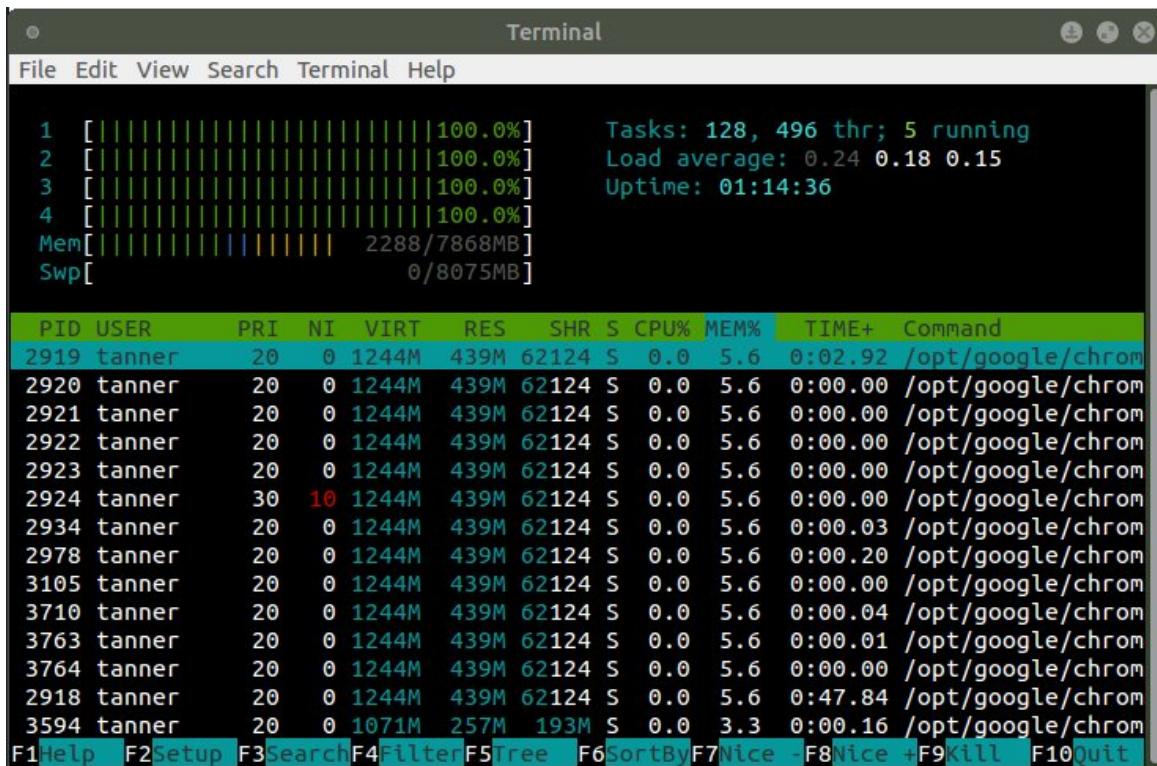


Figure 15.2: An example of `htop` with a computationally intense python script running in parallel.

This result ensures that our IPython cluster is successfully executing on all engines. We can now dive into the details of what syntax we can use to utilize the cluster.

## Syntax for ipyparallel

The basic framework for `ipyparallel` revolves around a `DirectView` or a `LoadBalancedView`. A `DirectView` is the object through which we can communicate with each of the engines individually and gives us control over which variables are pushed to each engine and what functions are performed. A `LoadBalancedView` takes the commands that are being executed and does its best to distribute the load evenly across all engines.

For the purposes of learning how each engine works, we will focus on the `DirectView` in this lab. To initialize a `DirectView`, run the following code:

```
>>> from ipyparallel import Client
>>> client = Client()

# Verify the dview has been generated correctly.
# If you had four processors, the output would be as follows.
>>> client.ids
[0, 1, 2, 3]

# Initialize DirectView
>>> dview = client[:]
```

## Variables on Different Engines

When using multiple processors, you can imagine each engine running its own iPython terminal with its own namespace. This implies that any variable we want to use must be initialized on each engine. There are a few different ways to do this.

```
# To share the variable 'a' across all engines
>>> a = 10
>>> b = 5
>>> dview["a"] = a
>>> dview["b"] = b

# Or alternatively,
>>> dview.push({'a':a, 'b':b})

# To ensure the variables are on engine 0
>>> client[0]['a']
10

>>> client[0]['b']
5
```

The code you just ran is the easiest way to get individual values on each of the engines. We will discuss a couple of other methods further on. We will now move on to simple computations in parallel.

## The apply() and apply\_sync() Methods

To execute functions on each of the engines, we can use the `apply()` or the `apply_sync()` methods which are differentiated by the presence of *blocking*. If a function is executed with blocking, you will be unable to send any other commands to the engine until the function has finished. If a function is run without blocking, you can still execute additional commands on that engine while the function is still computing its result. Though seemingly unimportant, blocking is an important principle if your computing processors or *nodes* are communicating one with another during a parallel process. The `apply()` method executes without blocking and the `apply_sync()` method executes with blocking. The following code box describes how this is done:

```
>>> def add():
...     return a+b

# Runs add() without blocking (in background)
>>> result = dview.apply(add)

# Checks to see if the output is ready
>>> result.ready()
True

# Retrieves output
>>> result.get()
[15, 15, 15, 15]

# Runs add() with blocking
>>> dview.apply_sync(add)
[15, 15, 15, 15]
```

You can also pass variables into your function as part of call to `apply_sync()`. Consider the following example:

```
>>> def add(x, y):
...     return x+y

>>> dview.apply_sync(add, 3, 6)
[9, 9, 9, 9]
```

To this point, the examples of what you can do with parallel computing may not seem too interesting since each engine is producing the same result. There are, however, circumstances in which the engines return different results. In these situations, parallel computing can drastically speed up the result.

```
# A function that draws four samples from a standard normal distribution
>>> def draw():
...     import numpy as np
...     a = np.random.randn(4)
...     return a

# Runs draw() on all engines simultaneously and returns the results
```

```
>>> dview.apply_sync(draw)
[array([-0.7277754 , -0.39273127,  0.05636817, -0.26855806]),
 array([ 0.46569263,  0.63911368, -0.02812979,  1.63223456]),
 array([ 0.92278649, -1.42868485,  0.32370856, -0.2386319 ]),
 array([-0.93787564,  1.16286507, -0.0388443 , -1.10649599])]
```

### NOTE

In the code above, notice that NumPy is imported within the function. Since each engine has its own namespace, we must ensure that the desired modules are imported in each engine. There is more than one way to do this.

For example, the following command imports NumPy to all engines simultaneously.

```
>>> dview.execute("import numpy as np")
```

**Problem 3.** Using `apply_sync()`, draw  $n$  samples from a standard normal distribution where  $n$  is default to 1,000,000. Print the mean, max, and min for draws on each individual engine. For example if you have four engines running, your output should look like:

```
means = [0.0031776784, -0.0058112042, 0.0012574772, -0.0059655951]
maxs = [4.0388107, 4.3664958, 4.2060184, 4.3391623]
mins = [-4.1508589, -4.3848019, -4.1313324, -4.2826519]
```

In theory, using parallel computing for this problem should be approximately  $N$  times faster where  $N$  is the number of engines you are using. In practice, however, the scaling is not quite linear. This is due in part to the controller running on one of the engines, your computers standard processes still running, and the overhead of communication from the controller with the engines. We test this in the following problem.

**Problem 4.** Using the function you wrote and passed into `apply_sync()` in the previous problem, compare the time it takes to run the function with parallel computing to the time it takes to run the function serially. That is, time how long it takes to run the function on all of your machine's engines simultaneously using `apply_sync`, and how long it takes to run the function in a `for` loop  $n$  times, where  $n$  is the number of engines on your machine. Print the results for 1,000,000, 5,000,000, 10,000,000, and 15,000,000 samples. You should notice an increase in efficiency as the problem size increases.

**Problem 5.** Now let's do a problem that is a bit more computationally intensive. Define the random variable  $X$  to be the maximum out of  $N$  draws from the standard normal distribution. For example, one draw from  $X$  when  $N = 10$  would be the maximum out of 10 draws from the normal distribution. Write a function that accepts an integer  $N$ , takes 500,000 draws from this distribution ( $X$ ), and plots the draws in a histogram. The resulting histogram will approximate the p.d.f. of  $X$ .

Write your function in such a way that each engine will carry an equal load. Also write your function in such a way that it is flexible to the number of engines that are running. HINT: Remember that you can get a list of all available engines using `clients.ids`.

## The `scatter()` and `gather()` Methods

There are many situations where we would want to spread a dataset across all the available engines. This way, we can have a function work on each of these portions of the dataset. In its simplest form, this is the basis of the MapReduce program which will address in more detail in a future lab.

We will first introduce an example of `scatter()` and explain the proper usage in more detail throughout the example.

```
# Initialize the dataset to scatter
>>> a = np.arange(10)

# Scatter the data. The pieces of the data will be
#   named "a_partition" on each of the engines.
>>> dview.scatter("a_partition", a)

# Verify that the data has been successfully scattered.
#   Notice that the data has been scattered as
#   equally as possible.
>>> client[0]["a_partition"]
array([0, 1, 2])

>>> client[1]["a_partition"]
array([3, 4, 5])

>>> client[2]["a_partition"]
array([6, 7])

>>> client[3]["a_partition"]
array([8, 9])
```

Now that the `a_partition` variable has been initialized on each engine, we can now execute functions that depend on this variable. Consider the following simple example using the `execute()` method.

```
# Pass a string with the Python code that we wish to run
#   on each engine. This code will simply sum the entries
#   in "a_partition"
```

```
>>> dview.execute("""
... b = a_partition.sum()
... """)
```

### NOTE

If you are using a Jupyter Notebook, there is a built in magic function that is analogous to `dview.execute()`. If you put the `%%px` magic at the beginning of a cell of code, that cell of code will be executed on each engine. This tool is very useful for designing and debugging parallel algorithms.

We have now computed the sum of each of these pieces of the data and stored the result in the variable `b`. To gain access to these results, we use the `gather()` method.

```
# Gather all the 'b' values into a list.
>>> b_list = dview.gather("b", block=True)
[3, 12, 13, 17]

>>> sum(b_list)
45
```

To summarize, this example has taken a piece of data, scattered it to all available engines, performed a computation on each of these pieces of data, then gathered the results back to the controller.

## Applications

Parallel computing, when used correctly, is one of the best ways to speed up the run time of an algorithm. As a result, it is very commonly used today and has many applications, such as the following:

- Graphic rendering
- Facial recognition with large databases
- Numerical integration
- Calculating Discrete Fourier Transforms
- Simulation of various natural processes (weather, genetics, etc.)
- Natural language processing

In fact, there are many problems that are only possible to solve through parallel computing because solving them serially would take too long. In these types of problems, even the parallel solution could take years. Some brute-force algorithms, like those used to crack well designed encryptions, are examples of this type of problem.

The problems mentioned above are well suited to parallel computing because they can be manipulated in a way such that running them on multiple processors results in a significant run time improvement. Manipulating an algorithm to be run with parallel computing is called *parallelizing* the algorithm. When a problem only requires very minor manipulations to parallelize, it is often called *embarrassingly parallel*. Typically, an algorithm is embarrassingly parallel when there is little to no dependency between results. Algorithms that do not meet this criteria can still technically be parallelized, but there is not a significant enough improvement in run time to make this worthwhile. For example, calculating the Fibonacci sequence using the usual formula,  $F(n) = F(n-1) + F(n-2)$ , is poorly suited to parallel computing because each element of the sequence is dependent on the previous two elements.

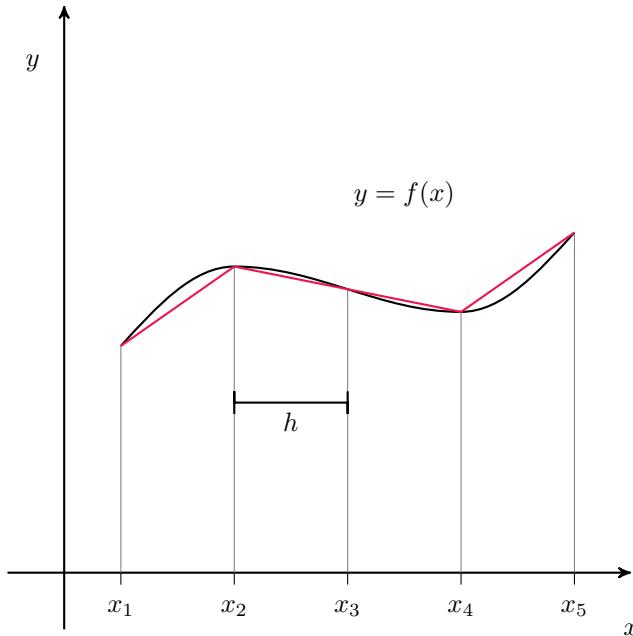


Figure 15.3: A depiction of the trapezoidal rule with uniform partitioning.

**Problem 6.** Consider the problem of numerical integration using the trapezoidal rule, depicted in Figure ???. Recall the following formula for estimating an integral using the trapezoidal rule,

$$\int_a^b f(x)dx \approx \frac{h}{2} \sum_{k=1}^N (f(x_{k+1}) + f(x_k)),$$

where  $x_k$  is the  $k$ th point, and  $h$  is the distance between any two points (note they are evenly spaced).

Note that estimation of the area of each interval is independent of all other intervals. As a result, this problem is considered to be embarrassingly parallel.

Write a function called `parallel_trapezoidal_rule()` that accepts a function handle to integrate, bounds of integration, and the number of points to use for the approximation. Utilize what you have learned about parallel computing to parallelize the trapezoidal rule in order to estimate the integral of  $f$ . That is, evenly divide the points among all available processors and run the trapezoidal rule on each portion simultaneously. The sum of the results of all the processors will be the estimation of the integral over the entire interval of integration. Return this sum.



# 16

# Parallel Programming with MPI

**Lab Objective:** *In the world of parallel computing, MPI is the most widespread and standardized message passing library. As such, it is used in the majority of parallel computing programs. In this lab, we explore and practice the basic principles and commands of MPI to further recognize when and how parallelization can occur.*

## MPI: the Message Passing Interface

At its most basic, the Message Passing Interface (MPI) provides functions for sending and receiving messages between different processes. MPI was developed to provide a standard framework for parallel computing in any language. Rather, MPI specifies a library of functions — the syntax and semantics of message passing routines — that can be called from programming languages such as Fortran and C.

MPI can be thought of as "the assembly language of parallel computing," because of this generality.<sup>1</sup> MPI is important because it was the first portable and universally available standard for programming parallel systems and continues to be the *de facto* standard today.

### NOTE

Most modern personal computers now have multicore processors. Programs that are designed for these multicore processors are "parallel" programs and are typically written using OpenMP or POSIX threads. MPI, on the other hand, is designed with a broader architecture in mind.

## Why MPI for Python?

In general, programming in parallel is more difficult than programming in serial because it requires managing multiple processors and their interactions. Python, however, is an excellent language for simplifying algorithm design because it allows for problem solving without too much detail. Unfortunately, Python is not designed for high performance computing and is a notably slower scripted language. It is best practice to prototype in Python and then to write production code in fast compiled languages such as C or Fortran.

<sup>1</sup>Parallel Programming with MPI, by Peter S. Pacheco, p. 7

In this lab, we will explore the Python library `mpi4py` which retains most of the functionality of C implementations of MPI and is a good learning tool. If you do not have the MPI library and `mpi4py` installed on your machine, please refer to the Additional Material at the end of this lab. There are three main differences to keep in mind between `mpi4py` and MPI in C:

- Python is array-based while C is not.
- `mpi4py` is object oriented but MPI in C is not.
- `mpi4py` supports two methods of communication to implement each of the basic MPI commands. They are the upper and lower case commands (e.g. `Bcast(...)` and `bcast(...)`). The uppercase implementations use traditional MPI datatypes while the lower case use Python's pickling method. Pickling offers extra convenience to using `mpi4py`, but the traditional method is faster. In these labs, we will only use the uppercase functions.

## Using MPI

We will start with a Hello World program.

```

1 #hello.py
2 from mpi4py import MPI

4 COMM = MPI.COMM_WORLD
5 RANK = COMM.Get_rank()

6 print "Hello world! I'm process number {}".format(RANK)

```

hello.py

Save this program as `hello.py` and execute it from the command line as follows:

```
$ mpirun -n 5 python hello.py
```

The program should output something like this:

```
Hello world! I'm process number 3.
Hello world! I'm process number 2.
Hello world! I'm process number 0.
Hello world! I'm process number 4.
Hello world! I'm process number 1.
```

Notice that when you try this on your own, the lines will not necessarily print in order. This is because there will be five separate processes running autonomously, and we cannot know beforehand which one will execute its `print` statement first.

**ACHTUNG!**

It is usually bad practice to perform I/O (e.g., call `print`) from any process besides the root process, though it can oftentimes be a useful tool for debugging.

How does this program work? First, the `mpirun` program is launched. This is the program which starts MPI, a wrapper around whatever program you want to pass into it. The `-n 5` option specifies the desired number of processes. In our case, 5 processes are run, with each one being an instance of the program "python". To each of the 5 instances of python, we pass the argument `hello.py` which is the name of our program's text file, located in the current directory. Each of the five instances of python then opens the `hello.py` file and runs the same program. The difference in each process's execution environment is that the processes are given different ranks in the communicator. Because of this, each process prints a different number when it executes.

MPI and Python combine to make wonderfully succinct source code. In the above program, the line `from mpi4py import MPI` loads the MPI module from the `mpi4py` package. The line `COMM = MPI.COMM_WORLD` accesses a static communicator object, which represents a group of processes which can communicate with each other via MPI commands. The next line, `RANK = COMM.Get_rank()`, accesses the processes *rank* number. A rank is the process's unique ID within a communicator, and they are essential to learning about other processes. When the program `mpirun` is first executed, it creates a global communicator and stores it in the variable `MPI.COMM_WORLD`. One of the main purposes of this communicator is to give each of the five processes a unique identifier, or rank. When each process calls `COMM.Get_rank()`, the communicator returns the rank of that process. `RANK` points to a local variable, which is unique for every calling process because each process has its own separate copy of local variables. This gives us a way to distinguish different processes while writing all of the source code for the five processes in a single file.

Here is the syntax for `Get_size()` and `Get_rank()`, where `Comm` is a communicator object:

**Comm.Get\_size()** Returns the number of processes in the communicator. It will return the same number to every process. Parameters:

**Return value** - the number of processes in the communicator

**Return type** - integer

Example:

```

1 #Get_size_example.py
2 from mpi4py import MPI
3 SIZE = MPI.COMM_WORLD.Get_size()
4 print "The number of processes is {}".format(SIZE)

```

Get\_size\_example.py

**Comm.Get\_rank()** Determines the rank of the calling process in the communicator. Parameters:

**Return value** - rank of the calling process in the communicator

**Return type** - integer

Example:

```

1 #Get_rank_example.py
2 from mpi4py import MPI

```

```

1 RANK = MPI.COMM_WORLD.Get_rank()
4 print "My rank is {}".format(RANK)

```

Get\_rank\_example.py

## The Communicator

A communicator is a logical unit that defines which processes are allowed to send and receive messages. In most of our programs we will only deal with the `MPI.COMM_WORLD` communicator, which contains all of the running processes. In more advanced MPI programs, you can create custom communicators to group only a small subset of the processes together. This allows processes to be part of multiple communicators at any given time. By organizing processes this way, MPI can physically rearrange which processes are assigned to which CPUs and optimize your program for speed. Note that within two different communicators, the same process will most likely have a different rank.

Note that one of the main differences between `mpi4py` and MPI in C or Fortran, besides being array-based, is that `mpi4py` is largely object oriented. Because of this, there are some minor changes between the `mpi4py` implementation of MPI and the official MPI specification.

For instance, the MPI Communicator in `mpi4py` is a Python class and MPI functions like `Get_size()` or `Get_rank()` are instance methods of the communicator class. Throughout these MPI labs, you will see functions like `Get_rank()` presented as `Comm.Get_rank()` where it is implied that `Comm` is a communicator object.

## Separate Codes in One File

When an MPI program is run, each process receives the same code. However, each process is assigned a different rank, allowing us to specify separate behaviors for each process. In the following code, the three processes perform different operations on the same pair of numbers.

```

1 #separateCode.py
2 from mpi4py import MPI
RANK = MPI.COMM_WORLD.Get_rank()

4
5 a = 2
6 b = 3
7 if RANK == 0:
8     print a + b
9 elif RANK == 1:
10    print a*b
11 elif RANK == 2:
12    print max(a, b)

```

separateCode.py

**Problem 1.** Write a program in which the processes with an even rank print "Hello" and process with an odd rank print "Goodbye." Print the process number along with the "Hello" or "Goodbye" (for example, "Goodbye from process 3").

## Message Passing between Processes

Let us begin by demonstrating a program designed for two processes. One will draw a random number and then send it to the other. We will do this using the routines `Comm.Send` and `Comm.Recv`.

```

1 #passValue.py
2 import numpy as np
3 from mpi4py import MPI
4
5 COMM = MPI.COMM_WORLD
6 RANK = COMM.Get_rank()
7
8 if RANK == 1: # This process chooses and sends a random value
9     num_buffer = np.random.rand(1)
10    print "Process 1: Sending: {}".format(num_buffer)
11    COMM.Send(num_buffer, dest=0)
12    print "Process 1: Message sent."
13 if RANK == 0: # This process receives a value from process 1
14    num_buffer = np.zeros(1)
15    print "Process 0: Waiting for the message... current num_buffer={}".format(
16        num_buffer)
17    COMM.Recv(num_buffer, source=1)
18    print "Process 0: Message received! num_buffer={}".format(num_buffer)

```

passValue.py

To illustrate simple message passing, we have one process choose a random number and then pass it to the other. Inside the receiving process, we have it print out the value of the variable `num_buffer` before it calls `Recv` to prove that it really is receiving the variable through the message passing interface.

Here is the syntax for `Send` and `Recv`, where `Comm` is a communicator object:

**Comm.Send(buf, dest=0, tag=0)** Performs a basic send from one process to another. Parameters:

- buf (array-like)** → data to send
- dest (integer)** → rank of destination
- tag (integer)** → message tag

The `buf` object is not as simple as it appears. It must contain a pointer to a Numpy array. It cannot, for example, simply pass a string. The string would have to be packaged inside an array first.

**Comm.Recv(buf, source=0, tag=0, Status status=None)** Basic point-to-point receive of data. Parameters:

**buf (array-like)** — initial address of receive buffer (choose receipt location)  
**source (integer)** — rank of source  
**tag (integer)** — message tag  
**status (Status)** — status of object

Example:

```

1 #Send_example.py
2 from mpi4py import MPI
3 import numpy as np
4
5 RANK = MPI.COMM_WORLD.Get_rank()
6
7 a = np.zeros(1, dtype=int) # This must be an array
8 if RANK == 0:
9     a[0] = 10110100
10    MPI.COMM_WORLD.Send(a, dest=1)
11 elif RANK == 1:
12    MPI.COMM_WORLD.Recv(a, source=0)
13    print a[0]
```

Send\_example.py

**Problem 2.** Write a script `passVector.py` that runs on two processes and passes an  $n$  by 1 vector of random values from one process to the other. Write it so that the user passes the value of  $n$  in as a command-line argument. Hint: This code will be useful in remembering how to pass command-line arguments.

```

from sys import argv

# Pass in the first command line argument as n
n = int(argv[1])
```

```
mpirun -n 2 python passVector.py 3
```

NOTE

`Send` and `Recv` are referred to as *blocking* functions. That is, if a process calls `Recv`, it will sit idle until it has received a message from a corresponding `Send` before it will proceed. (However, in Python the process that calls `Comm.Send` will *not* necessarily block until the message is received, though in C, `MPI_Send` does block) There are corresponding *non-blocking* functions `Irecv` and `Irecv` (The *I* stands for immediate). In essence, `Irecv` will return immediately. If a process calls `Irecv` and doesn't find a message ready to be picked up, it will indicate to the system that it is expecting a message, proceed beyond the `Irecv` to do other useful work, and then check back later to see if the message has arrived. This can be used to dramatically improve performance.

**Problem 3.** Write a script `passCircular.py` in which the process with rank  $i$  sends a random value to the process with rank  $i + 1$  in the global communicator. The process with the highest rank will send its random value to the root process. Notice that we are communicating in a ring. For communication, only use `Send` and `Recv`. The program should work for any number of processes. Hint: Remember that `Send` and `Recv` are blocking functions but that `Send`. Does the order in which `Send` and `Recv` are called matter?

#### NOTE

When calling `Comm.Recv`, you can allow the calling process to accept a message from any process that happened to be sending to the receiving process. This is done by setting `source` to a predefined MPI constant, `source=ANY_SOURCE` (note that you would first need to import this with `from mpi4py.MPI import ANY_SOURCE` or use the syntax `source=MPI.ANY_SOURCE`).

## Application: Monte Carlo Integration

Monte Carlo integration uses random sampling to approximate volumes (whereas most numerical integration methods employ some sort of regular grid). It is a useful technique, especially when working with higher-dimensional integrals. It is also well-suited to parallelization because it involves a large number of independent operations. In fact, Monte Carlo algorithms can be made “embarrassingly parallel” — the processes don’t need to communicate with one another during execution, simply reporting results to the root process upon completion.

In a simple example, the following code calculates the value of  $\pi$  by plotting random points inside a square. The probability of a given point landing inside the inscribed circle should be  $\pi/4$ .

```

1 import random
2
3 n = 100000
4 s = 0
5
6 for i in range(n):
7     x = random.uniform(-1.0,1.0)
8     y = random.uniform(-1.0,1.0)
```

```
10     if x**2 + y**2 <= 1:  
11         s += 1  
12  
print 4.0*s/n
```

pi.py

**Problem 4.** Write a script using  $n$  processes to find the volume of a unit  $m$ -sphere testing  $p$  random points in the unit  $m$  dimensional box. All of these variables should be passed from the command line (the  $n$  processes are passed in the mpirun command as in previous problems). The  $n$  processes should pass their individual results up to the root process, which then calculates an overall average.

#### NOTE

Good parallel code should pass as little data as possible between processes. Sending large or frequent messages requires a level of synchronization and causes some processes to pause as they wait to receive or send messages, negating the advantages of parallelism. It is also important to divide work evenly between simultaneous processes, as a program can only be as fast as its slowest process. This is called load balancing, and can be difficult in more complex algorithms.

## Additional Material

### Installing mpi4py

1. For All Systems: The easiest installation is using `conda install mpi4py`. You may also run `pip install mpi4py`

Part II

## Machine Learning Algorithms



# 17 Kalman Filter

**Lab Objective:** *Understand how to implement the standard Kalman Filter. Apply to the problem of projectile tracking.*

Measured observations are often prone to significant noise, due to restrictions on measurement accuracy. For example, most commercial GPS devices can provide a good estimate of geolocation, but only within a dozen meters or so. A Kalman filter is an algorithm that takes a sequence of noisy observations made over time and attempts to get rid of the noise, producing more accurate estimates than the original observations. To do this, the algorithm needs information about the system being observed.

Consider the problem of tracking a projectile as it travels through the air. Short-range projectiles approximately trace out parabolas, but a sensor that is recording measurements of the projectile's position over time will likely show a path that is much less smooth. Because we know something about the laws of physics, we can filter out the noise in the measurements using basic Newtonian mechanics, recovering a more accurate estimate of the projectile's trajectory. In this lab, we will simulate measurements of a projectile and implement a Kalman filter to estimate the complete trajectory of the projectile.

## Linear Dynamical Systems

The standard Kalman filter assumes that: (1) we have a linear dynamical system, (2) the state of the system evolves over time with some noise, and (3) we receive noisy measurements about the state of the system at each iteration. More formally, letting  $\mathbf{x}_k$  denote the state of the system at time  $k$ , we have

$$\mathbf{x}_{k+1} = F_k \mathbf{x}_k + B_k \mathbf{u}_k + \boldsymbol{\varepsilon}_k \quad (17.1)$$

where  $F_k$  is a state-transition model,  $B_k$  is a control-input model,  $\mathbf{u}_k$  is a control vector, and  $\boldsymbol{\varepsilon}_k$  is the noise present in state  $k$ . This noise is assumed to be drawn from a multivariate Gaussian distribution with zero mean and covariance matrix  $Q_k$ . The control-input model and control vector allow the assumption that the state can be additionally influenced by some other factor than the linear state-transition model.

We further assume that the states are “hidden,” and we only get the noisy observations

$$\mathbf{z}_k = H_k \mathbf{x}_k + \boldsymbol{\delta}_k \quad (17.2)$$

where  $H_k$  is the observation model mapping the state space to the observation space, and  $\delta_k$  is the observation noise present at iteration  $k$ . As with the aforementioned error, we assume that this noise is drawn from a multivariate Gaussian distribution with zero mean and covariance matrix  $R_k$ .

The dynamics stated above are all taken to be linear. Thus, for our purposes, the operators  $F_k$ ,  $B_k$ , and  $H_k$  are all matrices, and  $\mathbf{x}_k$ ,  $\mathbf{u}_k$ ,  $\mathbf{z}_k$ , and  $\delta_k$  are all vectors.

We will assume that the transition and observation models, the control vector, and the noise covariances are constant, i.e. for each  $k$ , we will replace  $F_k$ ,  $H_k$ ,  $\mathbf{u}_k$ ,  $Q_k$ , and  $R_k$  with  $F$ ,  $H$ ,  $\mathbf{u}$ ,  $Q$ , and  $R$ . We will also assume that  $B = I$  is the identity matrix, so it can safely be ignored.

**Problem 1.** Begin implementing a `KalmanFilter` class by writing an initialization method that stores the transition and observation models, noise covariances, and control vector. We provide an interface below:

```
class KalmanFilter(object):
    def __init__(self,F,Q,H,R,u):
        """
        Initialize the dynamical system models.

        Parameters
        -----
        F : ndarray of shape (n,n)
            The state transition model.
        Q : ndarray of shape (n,n)
            The covariance matrix for the state noise.
        H : ndarray of shape (m,n)
            The observation model.
        R : ndarray of shape (m,m)
            The covariance matrix for observation noise.
        u : ndarray of shape (n,)
            The control vector.
        """
        pass
```

We now derive the linear dynamical system parameters for a projectile traveling through  $\mathbb{R}^2$  undergoing a constant downward gravitational force of  $9.8 \text{ m/s}^2$ . The relevant information needed to describe how the projectile moves through space is its position and velocity. Thus, our state vector has the form

$$\mathbf{x} = \begin{pmatrix} s_x \\ s_y \\ V_x \\ V_y \end{pmatrix},$$

where  $s_x$  and  $s_y$  give the  $x$  and  $y$  coordinates of the position (in meters), and  $V_x$  and  $V_y$  give the horizontal and vertical components of the velocity (in meters per second), respectively.

How does the system evolve from one time step to the next? Assuming each time step is 0.1 seconds, it is easy enough to calculate the new position:

$$\begin{aligned}s'_x &= s_x + 0.1V_x \\ s'_y &= s_y + 0.1V_y.\end{aligned}$$

Further, since the only force acting on the projectile is gravity (we are ignoring things like wind resistance), the horizontal velocity remains constant:

$$V'_x = V_x.$$

The vertical velocity, however, does change due to the effects of gravity. From basic Newtonian mechanics, we have

$$V'_y = V_y - 0.1 \cdot 9.8.$$

In summary, over one time step, the state evolves from  $\mathbf{x}$  to  $\mathbf{x}'$ , where

$$\mathbf{x}' = \begin{pmatrix} s_x + 0.1V_x \\ s_y + 0.1V_y \\ V_x \\ V_y - 0.98 \end{pmatrix}.$$

From this equation, you can extract the state transition model  $F$  and the control vector  $u$ .

We now turn our attention to the observation model. Imagine that a radar sensor captures (noisy) measurements of the projectile's position as it travels through the air. At each time step, the radar transmits the observation  $z = (z_x, z_y)$  given by

$$\begin{aligned}z_x &= s_x + \delta_x \\ z_y &= s_y + \delta_y,\end{aligned}$$

where  $(\delta_x, \delta_y)$  is a noise vector assumed to be drawn from a multivariate Gaussian with mean zero and some known covariance. These equations indicate the appropriate choice of observation model.

**Problem 2.** Work out the transition and observation models  $F$  and  $H$ , along with the control vector  $\mathbf{u}$ , corresponding to the projectile. Assume that the noise covariances are given by

$$\begin{aligned}Q &= 0.1 \cdot I_4 \\ R &= 5000 \cdot I_2.\end{aligned}$$

Instantiate a `KalmanFilter` object with these values.

We now wish to simulate a sequence of states and observations from the dynamical system. In addition to the system parameters, we need an initial state  $\mathbf{x}_0$  to get started. Computing the subsequent states and observations is simply a matter of following equations 17.1 and 17.2.

**Problem 3.** Add a method to your `KalmanFilter` class to generate a state and observation sequence by evolving the system from a given initial state (the function `numpy.random.multivariate_normal` will be useful). To do this, implement the following:

```
def evolve(self, x0, N):
    """
```

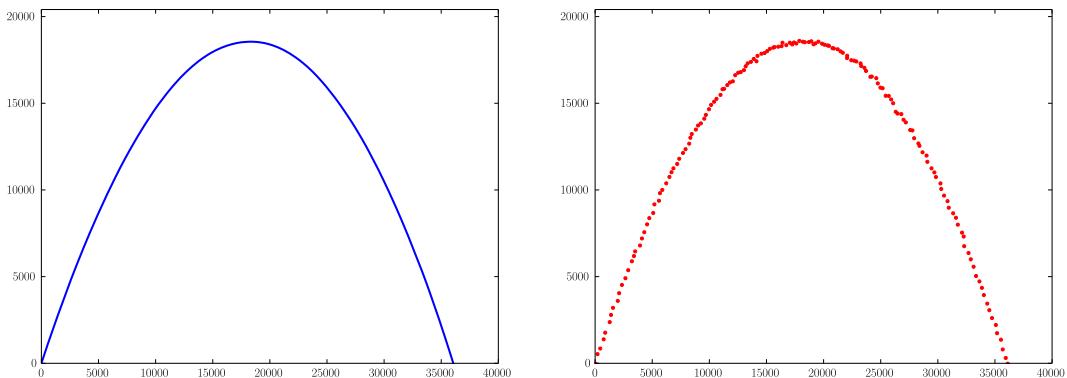


Figure 17.1: State sequence (left) and sampling of observation sequence (right).

Compute the first N states and observations generated by the Kalman system.

**Parameters**  
-----  
x0 : ndarray of shape (n,)  
    The initial state.  
N : integer  
    The number of time steps to evolve.

**Returns**  
-----  
states : ndarray of shape (n,N)  
    States 0 through N-1, given by each column.  
obs : ndarray of shape (m,N)  
    Observations 0 through N-1, given by each column.  
\*\*\*  
pass

Simulate the true and observed trajectory of a projectile with initial state

$$\mathbf{x}_0 = \begin{pmatrix} 0 \\ 0 \\ 300 \\ 600 \end{pmatrix}.$$

Approximately 1250 time steps should be sufficient for the projectile to hit the ground (i.e. for the  $y$  coordinate to return to 0). Your results should qualitatively match those given in Figure 17.1.

## State Estimation with the Kalman Filter

The Kalman filter is a recursive estimator that smooths out the noise in real time, estimating each current state based on the past state estimate and the current measurement. This process is done by repeatedly invoking two steps: Predict and Update. The predict step is used to estimate the current state based on the previous state. The update step then combines this prediction with the current observation, yielding a more robust estimate of the current state.

To describe these steps in detail, we need additional notation. Let

- $\hat{\mathbf{x}}_{n|m}$  be the state estimate at time  $n$  given only measurements up through time  $m$ ; and
- $P_{n|m}$  be an error covariance matrix, measuring the estimated accuracy of the state at time  $n$  given only measurements up through time  $m$ .

The elements  $\hat{\mathbf{x}}_{k|k}$  and  $P_{k|k}$  represent the state of the filter at time  $k$ , giving the state estimate and the accuracy of the estimate.

We evolve the filter recursively, as follows:

Predict	$\hat{\mathbf{x}}_{k k-1} = F\hat{\mathbf{x}}_{k-1 k-1} + \mathbf{u}$
	$P_{k k-1} = FP_{k-1 k-1}F^T + Q$
Update	$\tilde{\mathbf{y}}_k = \mathbf{z}_k - H\hat{\mathbf{x}}_{k k-1}$
	$S_k = HP_{k k-1}H^T + R$
	$K_k = P_{k k-1}H^T S_k^{-1}$
	$\hat{\mathbf{x}}_{k k} = \hat{\mathbf{x}}_{k k-1} + K_k\tilde{\mathbf{y}}_k$
	$P_{k k} = (I - K_kH)P_{k k-1}$

The more observations we have, the greater the accuracy of these estimates becomes (i.e the norm of the accuracy matrix converges to 0).

**Problem 4.** Add code to your `KalmanFilter` class to estimate a state sequence corresponding to a given observation sequence and initial state estimate. Implement the following class method:

```
def estimate(self,x,P,z):
    """
    Compute the state estimates using the Kalman filter.
    If x and P correspond to time step k, then z is a sequence of
    observations starting at time step k+1.

    Parameters
    -----
    x : ndarray of shape (n,)
        The initial state estimate.
    P : ndarray of shape (n,n)
        The initial error covariance matrix.
    z : ndarray of shape(m,N)
        Sequence of N observations (each column is an observation).
    """


```

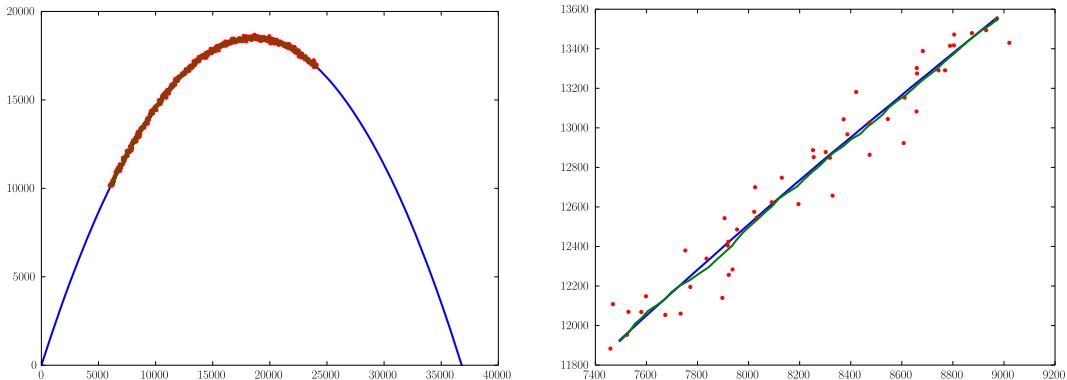


Figure 17.2: State estimates together with observations and true state sequence (detailed view on the right).

```

>Returns
-----
out : ndarray of shape (n,N)
      Sequence of state estimates (each column is an estimate).
"""
pass

```

Returning to the projectile example, we now assume that our radar sensor has taken observations from time steps 200 through 800 (take the corresponding slice of the observations produced in Problem 3). Using these observations, we seek to estimate the corresponding true states of the projectile. We must first come up with a state estimate  $\hat{\mathbf{x}}_{200}$  for time step 200, and then feed this into the Kalman filter to obtain estimates  $\hat{\mathbf{x}}_{201}, \dots, \hat{\mathbf{x}}_{800}$ .

**Problem 5.** Calculate an initial state estimate  $\hat{\mathbf{x}}_{200}$  as follows: For the horizontal and vertical positions, simply use the observed position at time 200. For the velocity, compute the average velocity between the observations  $\mathbf{z}_k$  and  $\mathbf{z}_{k+1}$  for  $k = 200, \dots, 208$ , then average these 9 values and take this as the initial velocity estimate. (Hint: the NumPy function `diff` is useful here.)

Using the initial state estimate,  $P_{200} = 10^6 \cdot Q$ , and your Kalman filter, compute the next 600 state estimates, i.e. compute  $\hat{\mathbf{x}}_{201}, \dots, \hat{\mathbf{x}}_{800}$ . Plot these state estimates as a smooth green curve together with the radar observations (as red dots) and the entire true state sequence (as a blue curve). Zoom in to see how well it follows the true path. Your plots should be similar to Figure 17.2.

In the absence of observations, we can still estimate some information about the state of the system at some future time. We can do this by recognizing that the expected state noise  $\mathbb{E}[\boldsymbol{\varepsilon}_k] = 0$  at any time  $k$ . Thus, given a current state estimate  $\hat{\mathbf{x}}_{n|m}$  using only measurements up through time  $m$ , the expected state at time  $n + 1$  is

$$\hat{\mathbf{x}}_{n+1|m} = F\hat{\mathbf{x}}_{n|m} + \mathbf{u}$$

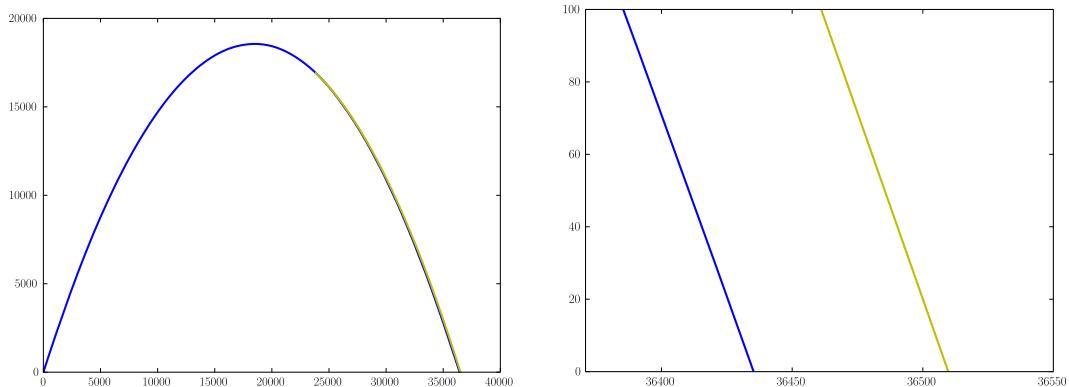


Figure 17.3: Predicted vs. actual point of impact (detailed view on right).

**Problem 6.** Add a function to your class that predicts the next  $k$  states given a current state estimate but in the absence of observations. Do so by implementing the following function:

```
def predict(self,x,k):
    """
    Predict the next k states in the absence of observations.

    Parameters
    -----
    x : ndarray of shape (n,)
        The current state estimate.
    k : integer
        The number of states to predict.

    Returns
    -----
    out : ndarray of shape (n,k)
        The next k predicted states.
    """
    pass
```

We can use this prediction routine to estimate where the projectile will hit the surface.

**Problem 7.** Using the final state estimate  $\hat{x}_{800}$  that you obtained in Problem 5, predict the future states of the projectile until it hits the ground. Predicting approximately the next 450 states should be sufficient.

Plot the actual state sequence together with the predicted state sequence (as a yellow curve), and observe how near the prediction is to the actual point of impact. Your results should be similar to those shown in Figure 17.3.

In the absence of observations, we can also reverse the system and iterate backward in time to infer information about states of the system prior to measured observations. The system is reversed by

$$\mathbf{x}_k = F^{-1}(\mathbf{x}_{k+1} - \mathbf{u} - \boldsymbol{\varepsilon}_{k+1}).$$

Considering again that  $\mathbb{E}[\boldsymbol{\varepsilon}_k] = 0$  at any time  $k$ , we can ignore this term, simplifying the recursive estimation backward in time.

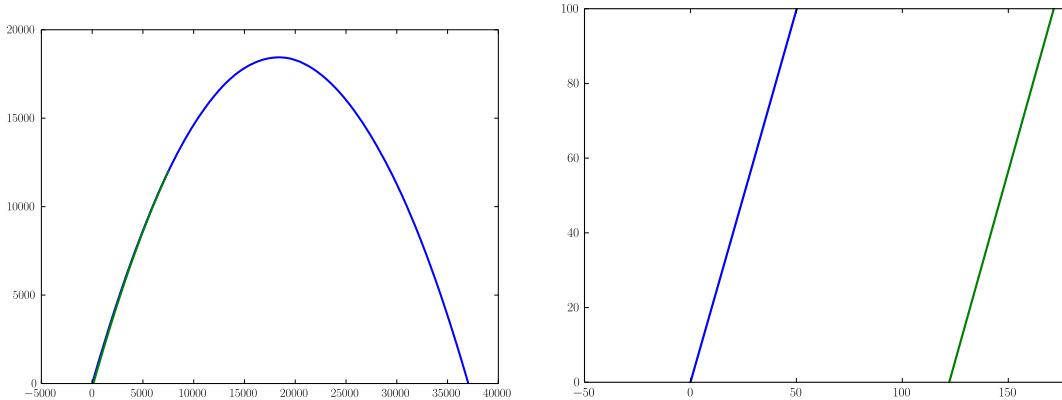


Figure 17.4: Predicted vs. actual point of origin (detailed view on right).

**Problem 8.** Add a function to your class that rewinds the system from a given state estimate, returning predictions for the previous states. Do so by implementing the following function:

```
def rewind(self, x, k):
    """
    Predict the k states preceding the current state estimate x.

    Parameters
    -----
    x : ndarray of shape (n,)
        The current state estimate.
    k : integer
        The number of preceding states to predict.

    Returns
    -----
    out : ndarray of shape (n,k)
        The k preceding predicted states.
    """
    pass
```

Returning to the projectile example, we can now predict the point of origin.

**Problem 9.** Using your state estimate  $\hat{\mathbf{x}}_{250}$ , predict the point of origin of the projectile along with all states leading up to time step 250. (The point of origin is the first point along the trajectory where the  $y$  coordinate is 0.) Plot these predicted states (in cyan) together with the original state sequence. Zoom in to see how accurate your prediction is. Your plots should be similar to Figure 17.4.

Repeat the prediction starting with  $\hat{\mathbf{x}}_{600}$ . Compare to the previous results. Which is better? Why?



# 18

## ARMA Models

**Lab Objective:** *Fit and forecast ARMA models.*

An ARMA( $p, q$ ) model is a covariance-stationary discrete stochastic process  $\{z_t\}$  that satisfies

$$z_t - \mu = \left( \sum_{i=1}^p \phi_i (z_{t-i} - \mu) \right) + a_t + \left( \sum_{j=1}^q \theta_j a_{t-j} \right) \quad (18.1)$$

where  $\mu = E[z_t]$  and  $a_t$  are identically-distributed Gaussian variables with variance  $\sigma_a^2$ . We note that the assumption that  $\{z_t\}$  is covariance-stationary is equivalent to the condition that the roots of the polynomial in  $B$

$$\phi(B) = 1 - \sum_{i=1}^p \phi_i B^i \quad (18.2)$$

lie outside of the unit circle.

The first sum on the right hand side of 18.1 is interpreted as an “autoregression” since it is a linear combination of previously observed values of  $z_t$ . The second sum is interpreted as a “moving average” of the current and previous error terms; though formally similar to an average, note that the  $\theta_j$  need not be positive nor sum to one. We say that an ARMA( $p, q$ ) model is an “autoregressive moving-average model of order  $p, q$ ”.

### Likelihood via Kalman Filter

In a general ARMA( $p, q$ ) model, the likelihood is a function of the unobserved error terms  $a_t$  and is not trivial to compute. Simple approximations can be made, but these may be inaccurate under certain circumstances. Explicit derivations of the likelihood are possible, but tedious. However, when the ARMA model is placed in state-space, the Kalman filter affords a straightforward, recursive way to compute the likelihood.

We demonstrate a state-space representation of an ARMA( $p, q$ ) model. If  $r = \max(p, q + 1)$ , we write

$$F = \begin{bmatrix} \phi_1 & \phi_2 & \cdots & \phi_{r-1} & \phi_r \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \cdots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 \end{bmatrix} \quad (18.3)$$

$$H = [1 \ \theta_1 \ \theta_2 \ \cdots \ \theta_{r-1}] \quad (18.4)$$

$$Q = \begin{bmatrix} \sigma_a^2 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \cdots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix} \quad (18.5)$$

$$w_t \sim \text{MVN}(0, Q), \quad (18.6)$$

where  $\phi_i = 0$  for  $i > p$ , and  $\theta_j = 0$  for  $j > q$ . Then the linear stochastic dynamical system

$$x_{t+1} = Fx_t + w_t \quad (18.7)$$

$$z_t = Hx_t + \mu \quad (18.8)$$

describes the same process as the original ARMA model. Note that the equation for  $z_t$  involves a deterministic component, namely  $\mu$ . The Kalman filter theory developed in the previous lab, however, assumed no deterministic component for the observations  $z_t$ , so you should subtract off the mean  $\mu$  from the time series observations  $z_t$  when using them in the predict and update steps.

Let  $\Theta = \{\phi_i, \theta_j, \mu, \sigma_a^2\}$  be the set of parameters for an ARMA( $p, q$ ) model. Suppose we have a set of observations  $z_1, z_2, \dots, z_n$ , denoted collectively by  $\{z_t\}$ . Using the chain rule, we can factorize the likelihood of the model under these data as

$$p(\{z_t\}|\Theta) = \prod_{t=1}^n p(z_t|z_{t-1}, \dots, z_1, \Theta) \quad (18.9)$$

Since we have assumed that the error terms are Gaussian, each conditional distribution in 18.9 is also Gaussian, and is completely characterized by its mean and variance. But these two quantities are easily found via the Kalman filter, namely

$$\text{mean} \quad H\hat{x}_{t|t-1} + \mu \quad (18.10)$$

$$\text{variance} \quad HP_{t|t-1}H^T \quad (18.11)$$

where  $\hat{x}_{t|t-1}$  and  $P_{t|t-1}$  are found during the Predict step. The likelihood becomes

$$p(\{z_t\}|\Theta) = \prod_{t=1}^n N(z_t; H\hat{x}_{t|t-1} + \mu, HP_{t|t-1}H^T) \quad (18.12)$$

We begin the recursion by letting

$$\hat{x}_{1|0} = \mathbb{E}(x_1) = 0 \quad (18.13)$$

$$\text{vec}(P_{1|0}) = \mathbb{E}[(x_1 - \mathbb{E}x_1)(x_1 - \mathbb{E}x_1)^T] = [I_{r^2} - (F \otimes F)]^{-1} \cdot \text{vec}(Q) \quad (18.14)$$

where `vec` flattens a matrix and  $\otimes$  is the Kronecker product (`numpy.kron`).

**Problem 1.** Write a function that computes the log-likelihood of an ARMA( $p, q$ ) model, given a time series  $z_t$ .

```
def arma_likelihood(time_series, phis=array([]), thetas=array([]), mu=0.,
                    sigma=1.):
    """
    Return the log-likelihood of the ARMA model parameters, given the time
    series.

    Parameters
    -----
    time_series : ndarray of shape (n,1)
        The time series in question, z_t
    phis : ndarray of shape (p,)
        The phi parameters
    thetas : ndarray of shape (q,)
        The theta parameters
    mu : float
        The parameter mu
    sigma : float
        The standard deviation of the a_t random variables

    Returns
    -----
    log_likelihood : float
        The log-likelihood of the model
    """
    pass
```

When done correctly, your function should match the following output:

```
>>> arma_likelihood(time_series_a, phis=array([0.9]), mu=17., sigma=0.4)
-77.6035
```

## Identification and Fitting

When modeling a data set with an ARMA( $p, q$ ) model, the order of the model must be determined, as well as the other parameters. The process of choosing  $p$  and  $q$  is called *model identification*. Different methods have been used; for example, Box and Jenkins propose a methodology that involves examining the estimated autocorrelation and partial-autocorrelation functions of the data. We will choose  $p$  and  $q$  that minimize the Akaike information criterion with a correction (AICc), given by

$$2k \left( 1 + \frac{k+1}{n-k} \right) - 2\ell(\Theta) \quad (18.15)$$

where  $n$  is the sample size,  $k = p + q + 2$  is the number of parameters in the model, and  $\ell(\Theta)$  is the maximum likelihood for the model class.

To compute the maximum likelihood for a model class, we need to optimize 18.12 over the space of parameters  $\Theta$ . We can do so by using the function from Problem 1 along with some optimization routine, such as `scipy.optimize.fmin`.

**Problem 2.** Write a function that accepts a time series  $\{z_t\}$  and returns the parameters of the model that minimize the AICc, given the constraint that  $p \leq 3$ ,  $q \leq 3$ .

```
def arma_fit(time_series):
    """
    Return the ARMA model that minimizes AICc for the given time series,
    subject to p,q <= 3.

    Parameters
    -----
    time_series : ndarray of shape (n,1)
        The time series in question, z_t

    Returns
    -----
    phis : ndarray of shape (p,)
        The phi parameters
    thetas : ndarray of shape (q,)
        The theta parameters
    mu : float
        The parameter mu
    sigma : float
        The standard deviation of the a_t random variables
    """
    pass
```

Here's a hint for performing the optimization at each step, using `scipy.optimize.fmin`.

```
>>> # assume p, q, and time_series are defined
>>> def f(x): # x contains the phis, thetas, mu, and sigma
>>>     return -1*arma_likelihood(time_series, phis=x[:p], thetas=x[p:p+q-1],
>>>                                mu=x[-2], sigma=x[-1])
>>> # create initial point
>>> x0 = np.zeros(p+q+2)
>>> x0[-2] = time_series.mean()
>>> x0[-1] = time_series.std()
>>> sol = op.fmin(f,x0,maxiter=10000, maxfun=10000)
```

The variable `sol` is a flat array of length  $p + q + 2$ , whose first  $p$  entries give the optimal values for the  $\phi$  polynomial, the next  $q$  entries give the optimal values for the  $\theta$  polynomial, and the last two entries give the optimal values for  $\mu$  and  $\sigma_a$ , respectively. Notice that we defined a wrapper function `f` to feed into the `scipy.optimize.fmin` routine. This wrapper function returns the *negative* of the log likelihood, since the optimization routine we are calling finds the minimum of a function, and we are interested in the *maximum* of the log likelihood.

Your code should produce the following output, where the input data is found in `time_series_a.txt` (it may take a minute or so to run):

```
>>> arma_fit(time_series_a)
(array([ 0.9087]), array([-0.5759]), 17.0652..., 0.3125...)
```

**Problem 3.** Use your solution from Problem 2 to fit models to the data found in `time_series_a.txt`, `time_series_b.txt`, `time_series_c.txt`. Report the fitted parameters  $p, q, \Theta$ .

## Forecasting

The Kalman filter provides a straightforward way to predict future states, by giving the mean and variance of the conditional distribution of future observations.

$$z_{t+k}|z_1, \dots, z_t \sim N(z_{t+k}; H\hat{x}_{t+k|t} + \mu, HP_{t+k|t}H^T) \quad (18.16)$$

Recall the relations

$$\hat{x}_{t+k|t} = F\hat{x}_{t+k-1|t} \quad (18.17)$$

$$P_{t+k|t} = FP_{t+k-1|t}F^T + Q \quad (18.18)$$

**Problem 4.** Forecast each data set ahead 20 intervals using the parameters discovered from Problem 3, and plot their expected values along with the original data set. Also plot the expected values plus and minus  $\sigma_{t+k}$ , and plus and minus  $2\sigma_{t+k}$  to demonstrate credible intervals.

Note that we need the values of  $\hat{x}_{n|n}$  and  $P_{n|n}$  to get started. As usual, these estimates can be found using the Predict and Update recursions. Initialize  $\hat{x}_{1|0}$  and  $P_{1|0}$  as before, run the recursions until you obtain  $\hat{x}_{n|n}$  and  $P_{n|n}$ , and then calculate the future estimates  $\hat{x}_{t+k|t}$  and  $P_{t+k|t}$ . Use these to calculate the expected value and standard deviation for forecasted values (given by  $H\hat{x}_{t+k|t} + \mu$  and  $\sqrt{HP_{t+k|t}H^T}$ , respectively).

```
def arma_forecast(time_series, phis=array([]), thetas=array([]), mu=0.,
                   sigma=1., future_periods=20):
    """
    Return forecasts for a time series modeled with the given ARMA model.
    """


```

Parameters

-----

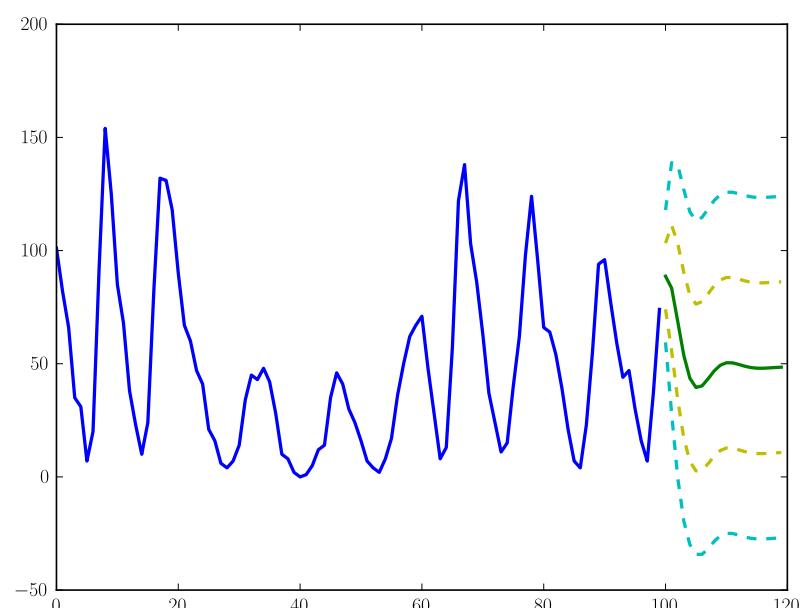
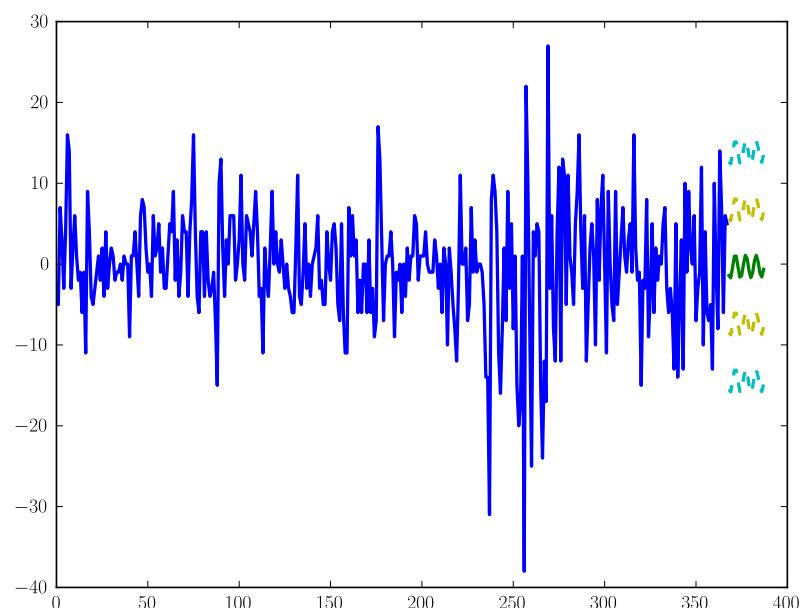
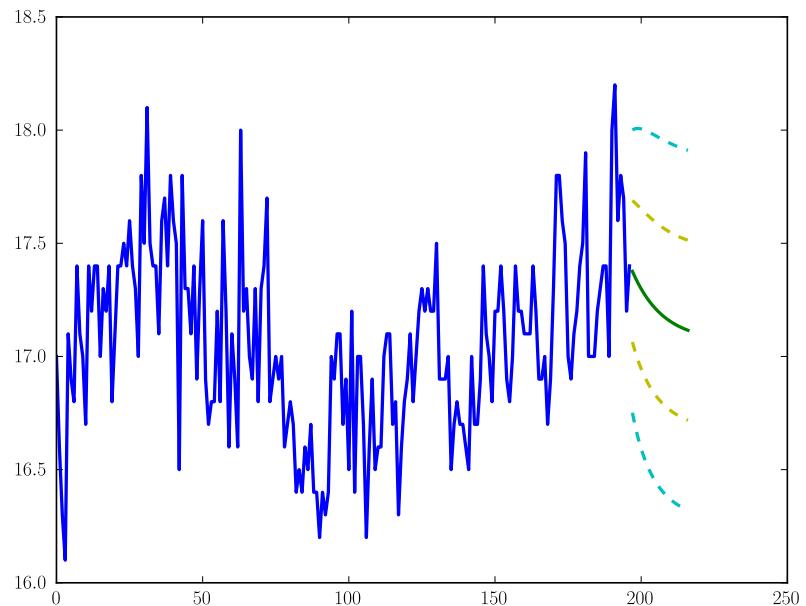
```
time_series : ndarray of shape (n,1)
    The time series in question, z_t
phis : ndarray of shape (p,)
    The phi parameters
thetas : ndarray of shape (q,)
    The theta parameters
mu : float
    The parameter mu
sigma : float
    The standard deviation of the a_t random variables
future_periods : int
    The number of future periods to return

Returns
-----
e_vals : ndarray of shape (future_periods,)
    The expected values of z for times n+1, ..., n+future_periods
sigs : ndarray of shape (future_periods,)
    The standard deviations of z for times n+1, ..., n+future_periods
"""
pass
```

You should get the following result:

```
>>> arma_forecast(time_series_a, phis, thetas, mu, sigma, 4)
(array([ 17.3762,  17.3478,  17.322 ,  17.2986]),
 array([ 0.3125,  0.3294,  0.3427,  0.3533]))
```

Your results (when using twenty future periods) should match those in Figure 18.1.





# 19

## Discrete Hidden Markov Models

**Lab Objective:** *Understand how to use discrete Hidden Markov Models.*

Given a discrete state-space Hidden Markov Model (HMM) with parameters  $\lambda$  and an observation sequence  $O$ , we would like to answer three questions:

1. What is  $\mathbb{P}(O|\lambda)$ ? In other words, what is the likelihood that our model generated the observation sequence?
2. What is the most likely state sequence to have generated  $O$ , given  $\lambda$ ?
3. How can we choose the parameters  $\lambda$  that maximize  $\mathbb{P}(O|\lambda)$ ?

The answers to these questions are centered around the *forward-backward* algorithm for HMMs. For the second question, the approach taken in this lab will be to find the state sequence maximizing the expected number of correct states. The third question is an example of *unsupervised learning*, since we are attempting to learn (or fit) model parameters using data (the observation sequence  $O$ ) that is devoid of human-provided labels (the corresponding state sequence); the algorithm does not rely on human supervision or input.

We assume throughout this lab that the HMM has a discrete state space of cardinality  $N$  and a discrete observation space of cardinality  $M$ . In this context  $\lambda = (A, B, \pi)$ , where  $A$  is a  $N \times N$  column-stochastic matrix (the state transition model),  $B$  is a  $M \times N$  column-stochastic matrix (the state observation model), and  $\pi$  is a stochastic vector of length  $N$  (the initial state distribution). Further,  $O$  is a vector of length  $T$  with values in the set  $\{1, 2, \dots, M\}$ .

### ACHTUNG!

The mathematical exposition in the lab assumes the standard 1-based indexing of vectors and matrices. Be sure to carefully translate the various formulae into 0-based indexing when implementing these methods for Python coding. This means that, in Python, your array containing the observation sequence  $O$  will actually have values in the set  $\{0, 1, \dots, M - 1\}$  so that they may be used to index the matrix  $B$  correctly.

Throughout this lab, we will be using the following toy HMM to verify your code.

```
>>> # toy HMM example to be used to check answers
>>> A = np.array([[.7, .4], [.3, .6]])
>>> B = np.array([[.1,.7],[.4, .2],[.5, .1]])
>>> pi = np.array([.6, .4])
>>> obs = np.array([0, 1, 0, 2])
```

**Problem 1.** To start off your implementation of the HMM, define a class object which you should call "hmm". Then add the initialization method, in which you should set the *self* aspects A, B, and pi to be None objects. You will be adding methods throughout the remainder of the lab.

## The Forward Pass

Our first task is to efficiently compute  $\log \mathbb{P}(O|\lambda)$ . We can do this using the *forward pass* of the forward-backward algorithm. We must take care to compute all values in a numerically stable way; we do this by properly scaling values as necessary.

We compute a scaled forward probability matrix  $\hat{\alpha}$  of dimension  $T \times N$  as follows: Let  $\hat{\alpha}_{i,:}, B_{i,:}$  denote the  $i$ -th rows of  $\hat{\alpha}$  and  $B$ , respectively, let  $\odot$  denote the Hadamard (or entry-wise) product of arrays, and let  $\langle \cdot, \cdot \rangle$  denote the standard dot product. (Note that here, using 0-based indexing and the toy HMM example,  $B_{O_3,:}$  would refer to  $[.5, .1]$ .) Then

- $c_1 = \langle \pi, B_{O_1,:} \rangle^{-1}$
- $\hat{\alpha}_{1,:} = c_1(\pi \odot B_{O_1,:})$
- For  $t = 2, \dots, T$ :
 
$$c_t = \langle A\hat{\alpha}_{t-1,:}, B_{O_t,:} \rangle^{-1}$$

$$\hat{\alpha}_{t,:} = c_t((A\hat{\alpha}_{t-1,:}) \odot B_{O_t,:})$$

The matrix  $\hat{\alpha}$  will be of use when fitting parameters, but we can compute the desired log probability using the scaling factors  $c_t$  as follows:

$$\log \mathbb{P}(O|\lambda) = - \sum_{t=1}^T \log c_t.$$

**Problem 2.** Implement the forward pass by adding the following method to your class:

```
def _forward(self, obs):
    """
    Compute the scaled forward probability matrix and scaling factors.

    Parameters
    -----
    obs : ndarray of shape (T,)
        The observation sequence
```

```

Returns
-----
alpha : ndarray of shape (T,N)
    The scaled forward probability matrix
c : ndarray of shape (T,)
    The scaling factors c = [c_1,c_2,...,c_T]
"""
pass

```

To verify that your code works, you should get the following output using the toy HMM:

```

>>> h = hmm()
>>> h.A = A
>>> h.B = B
>>> h.pi = pi
>>> alpha, c = h._forward(obs)
>>> print -(np.log(c)).sum() # the log prob of observation
-4.6429135909

```

## The Backward Pass

The backward pass of the forward-backward algorithm produces values that can be used to calculate the most likely state sequence corresponding to an observation sequence.

We compute a scaled backward probability matrix  $\hat{\beta}$  of dimension  $T \times N$  as follows:

- $\hat{\beta}_{T,i} = c_T$  for  $i = 1, \dots, N$
- $\hat{\beta}_{t,:} = c_t A^T (B_{O_{t+1},:} \odot \hat{\beta}_{t+1,:})$  for  $t = T - 1, \dots, 1$

(Above,  $A^T$  is the *transpose* of  $A$ , not the  $T$ -th power of  $A$ .)

It turns out that

$$\mathbb{P}(\mathbf{x}_t = i | O, \lambda) = \frac{\hat{\alpha}_{t,i} \hat{\beta}_{t,i}}{\sum_{j=1}^N \hat{\alpha}_{t,j} \hat{\beta}_{t,j}}$$

and so we can easily compute the most likely state at time  $t$  by

$$\mathbf{x}_t^* = \operatorname{argmax}_i \hat{\alpha}_{t,i} \hat{\beta}_{t,i}.$$

This is the solution to the second question posed at the beginning of the lab.

**Problem 3.** Implement the backward pass by adding the following method to your class:

```

def _backward(self, obs, c):
    """
    Compute the scaled backward probability matrix.

    Parameters

```

```
-----
obs : ndarray of shape (T,)
    The observation sequence
c : ndarray of shape (T,)
    The scaling factors from the forward pass

Returns
-----
beta : ndarray of shape (T,N)
    The scaled backward probability matrix
"""
pass
```

Using the same toy example as before, your code should produce the following output:

```
>>> beta = h._backward(obs, c)
>>> print beta
[[ 3.1361635  2.89939354]
 [ 2.86699344  4.39229044]
 [ 3.898812   2.66760821]
 [ 3.56816483  3.56816483]]
```

## Computing the $\delta$ and $\gamma$ Probabilities

Having implemented both parts of the forward-backward algorithm, we are closing in on the solution to question three, namely that of fitting parameters  $\lambda$  that maximize  $\mathbb{P}(O|\lambda)$ . At this stage, we combine the information accumulated in the forward-backward algorithm to produce a three-dimensional array  $\hat{\delta}$  of shape  $(T-1) \times N \times N$  whose entries are related to  $\mathbb{P}(\mathbf{x}_t = i, \mathbf{x}_{t+1} = j | O, \lambda)$ , as well as a  $T \times N$  matrix  $\hat{\gamma}$  whose entries are related to  $\mathbb{P}(\mathbf{x}_t = i | O, \lambda)$ . The relevant formulae are

$$\hat{\delta}_{t,i,j} = \frac{\hat{\alpha}_{t,i} A_{j,i} B_{O_{t+1},j} \hat{\beta}_{t+1,j}}{\sum_{k,l} \hat{\alpha}_{t,k} A_{l,k} B_{O_{t+1},l} \hat{\beta}_{t+1,l}}$$

for  $t = 1, \dots, T-1$  and  $i, j = 1, \dots, N$ ,

$$\hat{\gamma}_{t,i} = \sum_{j=1}^N \hat{\delta}_{t,i,j}$$

for  $t = 1, \dots, T-1$  and  $i = 1, \dots, N$ , and finally

$$\hat{\gamma}_{T,:} = \frac{\hat{\alpha}_{T,:} \odot \hat{\beta}_{T,:}}{\langle \hat{\alpha}_{T,:}, \hat{\beta}_{T,:} \rangle}.$$

**Problem 4.** Add the following method to your class to compute the  $\delta$  and  $\gamma$  probabilities.

```
def _delta(self, obs, alpha, beta):
```

```

"""
Compute the delta probabilities.

Parameters
-----
obs : ndarray of shape (T,)
    The observation sequence
alpha : ndarray of shape (T,N)
    The scaled forward probability matrix from the forward pass
beta : ndarray of shape (T,N)
    The scaled backward probability matrix from the backward pass

Returns
-----
delta : ndarray of shape (T-1,N,N)
    The delta probability array
gamma : ndarray of shape (T,N)
    The gamma probability array
"""

pass

```

While writing a triply-nested loop may be the simplest way to convert the formula into code, it is possible to use array broadcasting to eliminate two of the loops, which will speed up your code.

Check your code by making sure it produces the following output, using the same toy example as before.

```

>>> delta, gamma = h._delta(obs, alpha, beta)
>>> print delta
[[[ 0.14166321  0.0465066 ]
 [ 0.37776855  0.43406164]]

 [[ 0.17015868  0.34927307]
 [ 0.05871895  0.4218493 ]]

 [[ 0.21080834  0.01806929]
 [ 0.59317106  0.17795132]]]

>>> print gamma
[[ 0.18816981  0.81183019]
 [ 0.51943175  0.48056825]
 [ 0.22887763  0.77112237]
 [ 0.8039794   0.1960206 ]]

```

## Choosing Better Parameters

After running the forward-backward algorithm and computing the  $\delta$  probabilities, we are now in a position to choose new parameters  $\lambda' = (A', B', \pi')$  that increase the probability of observing our data, i.e.

$$\mathbb{P}(O | \lambda') \geq \mathbb{P}(O | \lambda).$$

The update formulas are given by

$$A'_{i,j} = \frac{\sum_{t=1}^{T-1} \hat{\delta}_{t,j,i}}{\sum_{t=1}^{T-1} \hat{\gamma}_{t,j}}$$

$$B'_{i,j} = \frac{\sum_{t=1}^T \hat{\gamma}_{t,j} \mathbf{1}_{\{O_t=i\}}}{\sum_{t=1}^T \hat{\gamma}_{t,j}}$$

$$\pi' = \hat{\gamma}_{1,:}$$

where  $\mathbf{1}_{\{O_t=i\}}$  is one if  $O_t = i$  and zero otherwise.

**Problem 5.** Implement the parameter update step by adding the following method to your class:

```
def _estimate(self, obs, delta, gamma):
    """
    Estimate better parameter values.

    Parameters
    -----
    obs : ndarray of shape (T,)
        The observation sequence
    delta : ndarray of shape (T-1,N,N)
        The delta probability array
    gamma : ndarray of shape (T,N)
        The gamma probability array
    """
    # update self.A, self.B, self.pi in place
    pass
```

Verify that your code produces the following output on the toy HMM from before:

```
h._estimate(obs, delta)
>>> print h.A
[[ 0.55807991  0.49898142]
 [ 0.44192009  0.50101858]]
>>> print h.B
[[ 0.23961928  0.70056364]
 [ 0.29844534  0.21268397]
 [ 0.46193538  0.08675238]]
>>> print h.pi
[ 0.18816981  0.81183019]
```

## Fitting the Model

We are now ready to put everything together into a learning algorithm. Given a sequence of observations, a maximum number of iterations  $K$ , and a convergence tolerance threshold  $\epsilon$ , we fit a HMM model using the following procedure:

- Randomly initialize parameters  $\lambda = (A, B, \pi)$
- Compute  $\log \mathbb{P}(O | \lambda)$
- For  $i = 1, 2, \dots, K$ :
  - Run forward pass
  - Run backward pass
  - Compute  $\delta$  probabilities
  - Update model parameters
  - Compute  $\log \mathbb{P}(O | \lambda)$  according to new parameters
  - If change in log probabilities is less than  $\epsilon$ , break
  - Else, continue

The most convenient way to randomly initialize stochastic matrices is to draw from the Dirichlet distribution, which produces vectors with nonnegative entries that sum to 1. The following Python code initializes  $M$ ,  $A$ ,  $B$ , and  $\pi$  using this technique:

```
>>> # assume N is defined
>>> # define M to be the number of distinct observed states
>>> M = len(set(obs))
>>> A = np.random.dirichlet(np.ones(N), size=N).T
>>> B = np.random.dirichlet(np.ones(M), size=N).T
>>> pi = np.random.dirichlet(np.ones(N))
```

The learning algorithm is essentially an optimization over the parameter space (i.e. the space of tuples of stochastic arrays having the proper dimensions) with respect to the objective function  $\mathbb{P}(O | \lambda)$ . The algorithm is guaranteed to increase the objective function at each iteration, so it is sure to converge. However, the objective function is riddled with local maxima, and so the outcome depends heavily on the randomly selected starting values for  $A$ ,  $B$ , and  $\pi$ . Figure 23.2 illustrates the issues involved. The log probability stays approximately constant for the first 100 iterations. This indicates that the algorithm is not exploring the parameter space enough, and the parameters found at the 100-th iteration are virtually the same as those found at the first or second iteration. After the first 100 iterations, however, the algorithm is finally able to explore more of the parameter space and hence make better progress toward increasing the objective function. The moral of the story is that you may need to train the HMM a few times, using different starting values, and then keep the model that has the highest log likelihood.

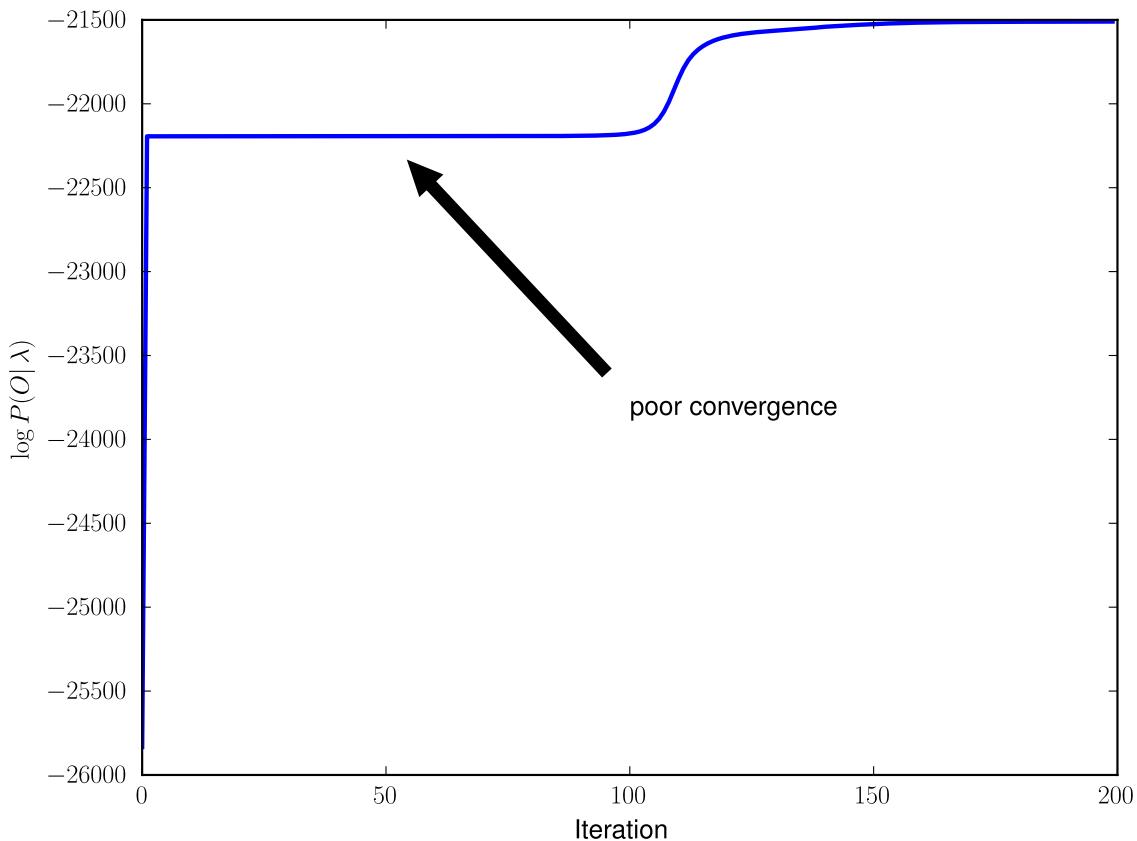


Figure 19.1: The log probabilities for a HMM trained on the Declaration of Independence data with 200 iterations. It takes over 100 iterations for the algorithm to work itself out of a poor local maximum.

**Problem 6.** Implement the learning algorithm by adding the following method to your class:

```
def fit(self, obs, A, B, pi, max_iter=100, tol=1e-3):
    """
    Fit the model parameters to a given observation sequence.

    Parameters
    -----
    obs : ndarray of shape (T,)
        Observation sequence on which to train the model.
    A : stochastic ndarray of shape (N,N)
        Initialization of state transition matrix
    B : stochastic ndarray of shape (M,N)
        Initialization of state observation matrix
    pi : stochastic ndarray of shape (N,)
        Initialization of initial state distribution
```

```

max_iter : integer
    The maximum number of iterations to take
tol : float
    The convergence threshold for change in log-probability
"""
# initialize self.A, self.B, self.pi
# run the iteration
pass

```

We now turn to the data found in the file `declaration.txt`. This file contains the text of the Declaration of Independence. We will use the sequence of characters (after stripping out punctuation and converting everything to lower-case) as our observation sequence. In order to convert the raw text into a useable data structure, we need to read in the file, process the string as necessary, and then map the characters to integer values. We provide helper code below to accomplish this task for various files in various languages:

```

>>> import numpy as np
>>> import string
>>> import codecs

>>> def vec_translate(a, my_dict):
    # translate numpy array from symbols to state numbers or vice versa
    >>> return np.vectorize(my_dict.__getitem__)(a)

>>> def prep_data(filename):
    # Get the data as a single string
    >>> with codecs.open(filename, encoding='utf-8') as f:
        data=f.read().lower() #and convert to all lower case

    >>> # remove punctuation and newlines
    >>> remove_punct_map = {ord(char): None for char in string.punctuation+"\n\r\f"}
    >>> data = data.translate(remove_punct_map)

    >>> # make a list of the symbols in the data
    >>> symbols = sorted(list(set(data)))

    >>> # convert the data to a NumPy array of symbols
    >>> a = np.array(list(data))

    >>> #make a conversion dictionary from symbols to state numbers
    >>> symbols_to_obsstates = {x:i for i,x in enumerate(symbols)}

    >>> #convert the symbols in a to state numbers

```

```
>>>     obs_sequence = vec_translate(a,symbols_to_obsstates)

>>>     return symbols, obs_sequence
```

Now apply this helper code to `declaration.txt`.

```
>>> symbols, obs = prep_data('declaration.txt')
```

**Problem 7.** You are now ready to train a HMM using the Declaration of Independence data. Use  $N = 2$  states and  $M = \text{len}(\text{set}(obs)) = 27$  observation values (26 lower case characters and 1 whitespace character), and run for 200 iterations with the default value for `tol`. Generally speaking, if you converge to a log probability greater than  $-21550$ , then you have reached an acceptable set of parameters for this dataset.

Once the learning algorithm converges, analyze the state observation matrix  $B$ . Note which rows correspond to the largest and smallest probability values in each column of  $B$ , and check the corresponding characters. The code below displays typical results for a well-converged HMM. Note that the `u` before the `"` indicates that the string should be unicode, which will be required for languages other than English.

```
>>> for i in xrange(len(h.B)):
>>>     print u"%0}, {1:0.4f}, {2:0.4f}"%format(symbols[i], h.B[i,0], h.B[←
    [i,1])
, 0.0051, 0.3324
a, 0.0000, 0.1247
c, 0.0460, 0.0000
b, 0.0237, 0.0000
e, 0.0000, 0.2245
d, 0.0630, 0.0000
g, 0.0325, 0.0000
f, 0.0450, 0.0000
i, 0.0000, 0.1174
h, 0.0806, 0.0070
k, 0.0031, 0.0005
j, 0.0040, 0.0000
m, 0.0360, 0.0000
l, 0.0569, 0.0001
o, 0.0009, 0.1331
n, 0.1207, 0.0000
q, 0.0015, 0.0000
p, 0.0345, 0.0000
s, 0.1195, 0.0000
r, 0.1062, 0.0000
u, 0.0000, 0.0546
t, 0.1600, 0.0000
w, 0.0242, 0.0000
v, 0.0185, 0.0000
```

```
y, 0.0147, 0.0058
x, 0.0022, 0.0000
z, 0.0010, 0.0000
```

What do you notice about the second column of  $B$ ? It seems that the HMM has detected a vowel state and a consonant state, without any prior input from an English speaker. Interestingly, the whitespace character is grouped together with the vowels. A HMM can also detect the vowel/consonant distinction in other languages.

**Problem 8.** Repeat the previous calculation with 3 hidden states and again with 4 hidden states. Interpret/explain your results.

Now we turn to the Russian file `WarAndPeace.txt`, which is a small subset of the book *War and Peace* by Tolstoy.

```
>>> symbols, obs = prep_data('WarAndPeace.txt')
```

**Problem 9.** Repeat the calculations for 2, and 3 hidden states for `WarAndPeace.txt`. Interpret/explain your results. Which Cyrillic characters appear to be vowels?



# 20 Gaussian Mixture Models

**Lab Objective:** *Understand the formulation of Gaussian Mixture Models (GMMs) and how to estimate GMM parameters.*

You've already seen GMMs as the observation distribution in certain continuous density HMMs. Here, we will discuss them further and learn how to estimate their parameters, given data.

The main idea behind a mixture model is contained in the name, i.e. it is a *mixture* of different models. What do we mean by a mixture? A mixture model is composed of  $K$  *components*, each component being responsible for a portion of the data. The responsibilities of these components are represented by mixture *weights*  $w_i$ , for  $i = 1, \dots, k$ . As you may have guessed, these weights are nonnegative and sum to 1. Thus component  $j$  is responsible for  $100 \cdot w_j$  percent of the data generated by the model.

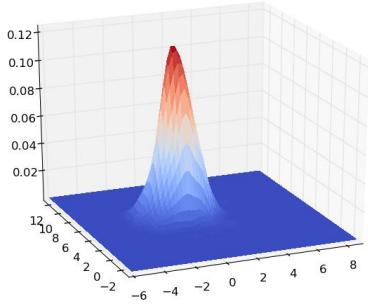
Each component is itself a probability distribution. In a GMM, each component is specifically a Gaussian (multivariate normal) distribution. Thus we additionally have parameters  $\mu_i, \Sigma_i$  for  $i = 1, \dots, K$ , i.e. a mean and covariance for each component in the GMM. It is important here to keep in mind that a GMM does not arise from adding weighted multivariate normal random variables, but rather from weighting the responsibility of each multivariate normal random variable. In the first case, we would simply have a different multivariate normal distribution, whereas in the second case we have a mixture. Refer to Figure ?? for a visualization of this.

Thus, a fully defined GMM has parameters  $\lambda = (w, \mu, \Sigma)$ . The density of a GMM is given by  $\mathbb{P}(x|\lambda) = \sum_{i=1}^K w_i \mathcal{N}(x; \mu_i, \Sigma_i)$  where

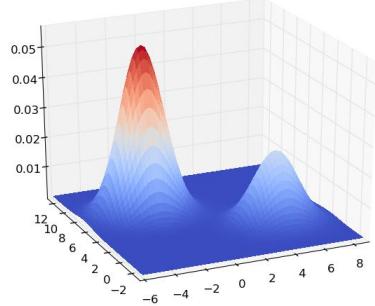
$$\mathcal{N}(x; \mu_i, \Sigma_i) = \frac{1}{(2\pi)^{\frac{K}{2}} |\Sigma_i|^{\frac{1}{2}}} e^{-\frac{1}{2}(x-\mu_i)^T \Sigma_i^{-1} (x-\mu_i)}$$

**Problem 1.** Write a function to evaluate the density of a normal distribution at a point  $x$ , given parameters  $\mu$  and  $\Sigma$ . Include the option to return the log of this probability, but be sure to do it intelligently! Also write a function that computes the density of a GMM at a point  $x$ , given the parameters  $\lambda$ , along with the log option.

Throughout this lab, we will build a GMM class with various methods. We will outline this now.



(a) Sum of weighted multivariate normal random variables.



(b) Weighted mixture of multivariate normal random variables.

**Problem 2.** Write the skeleton of a GMM class. In the `__init__` method, it should accept the non-null parameter `n_components`, as well as parameters for the weights, means, and covariance matrices which define the GMM. Include a function to generate data from a fully defined GMM (you may use your code from the CDHMM lab for this), as well as the density function you recently defined.

The main focus of this lab will be to estimate the parameters of a GMM, given observed multivariate data  $Y = y_1, y_2, \dots, y_T$ . This can be done via Gibbs sampling, as well as with EM (Expectation Maximization). We choose the latter approach for this lab. To do this, we must compute the probability of an observation being from each component of a GMM with parameters  $\lambda^{(n)} = (w_i^{(n)}, \mu_i^{(n)}, \Sigma_i^{(n)})$ . This is simply

$$\mathbb{P}(x_t = i | y_t, \lambda) \propto w_i^{(n)} \mathcal{N}(y_t; \mu_i^{(n)}, \Sigma_i^{(n)})$$

Just as with HMMs, we refer to these probabilities as  $\gamma_t(i)$ , and this is the *E*-step in the algorithm. This might seem straightforward, except this direct computation will likely lead to numerical issues. Instead, we work in the log space, which means we have to be a bit more careful.

It is feasible (and occurs quite often) that each term  $w_i^{(n)} \mathcal{N}(y_t; \mu_i^{(n)}, \Sigma_i^{(n)})$  is 0, because of underflow in the computation of the multivariate normal density. Letting  $l_i^{(n)} = \ln w_i^{(n)} + \ln \mathcal{N}(y_t; \mu_i^{(n)}, \Sigma_i^{(n)})$ , we can compute these probabilities more carefully, as follows:

$$\begin{aligned} \mathbb{P}(x_t = i | y_t, \lambda) &= \frac{e^{l_i}}{\sum_{j=1}^K e^{l_j}} \\ &= \frac{e^{l_i} e^{-\max_k l_k}}{\sum_{j=1}^K e^{l_j} e^{-\max_k l_k}} \\ &= \frac{e^{l_i - \max_k l_k}}{\sum_{j=1}^K e^{l_j - \max_k l_k}} \end{aligned}$$

which will effectively avoid underflow problems.

**Problem 3.** Add a method to your class to compute  $\gamma_t(i)$  for  $t = 1, \dots, T$  and  $i = 1, \dots, K$ . Don't forget to do this intelligently to avoid underflow!

Given our matrix  $\gamma$ , we can reestimate our weights, means, and covariance matrices as follows:

$$\begin{aligned} w_i^{(n+1)} &= \sum_{t=1}^T \gamma_t(i) \\ \mu_i^{(n+1)} &= \frac{\sum_{t=1}^T \gamma_t(i) y_t}{\sum_{t=1}^T \gamma_t(i)} \\ \Sigma_i^{(n+1)} &= \frac{\sum_{t=1}^T \gamma_t(i) (y_t - \mu_i^{(n+1)}) (y_t - \mu_i^{(n+1)})^T}{\sum_{t=1}^T \gamma_t(i)} \end{aligned}$$

for  $i = 1, \dots, K$ . These updates are the  $M$ -step in the algorithm.

**Problem 4.** Add methods to your class to update  $w, \mu$  and  $\Sigma$  as described above.

With the above work, we are almost ready to complete our class. To train, we will randomly initialize our parameters  $\lambda$ , and then iteratively update them as above.

**Problem 5.** Add a method to initialize  $\lambda$ . Do this intelligently, i.e. your means should not be far from your actual data used for training, and your covariances should neither be too big nor too small. Your weights should roughly be equal, and still sum to 1. Also add a method to train your model, as described previously, iterating until convergence within some tolerance.

We will use our work to train the “Mickey Mouse” GMM, which has parameters

$$\begin{aligned} w &= [ 0.7 \quad 0.15 \quad 0.15 ] \\ \mu_1 &= [ 0.0 \quad 0.0 ] \\ \mu_2 &= [ -1.5 \quad 2.0 ] \\ \mu_3 &= [ 1.5 \quad 2.0 ] \\ \Sigma_1 &= I_3 \\ \Sigma_2 &= 0.25 \cdot I_3 \\ \Sigma_3 &= 0.25 \cdot I_3 \end{aligned}$$

To look at this GMM, we will evaluate the density at each point on a grid, as follows:

```
>>> import matplotlib.pyplot as plt
>>> x = np.arange(-3, 3, 0.1)
>>> y = np.arange(-2, 3, 0.1)
>>> X, Y = np.meshgrid(x, y)
>>> N, M = X.shape
>>> immat = np.array([[model.dgmm(np.array([X[i,j],Y[i,j]])) for j in xrange(M)] for i in xrange(N)])
```

```
>>> plt.imshow(immat, origin='lower')
>>> plt.show()
```

See Figure 20.2 for this plot.

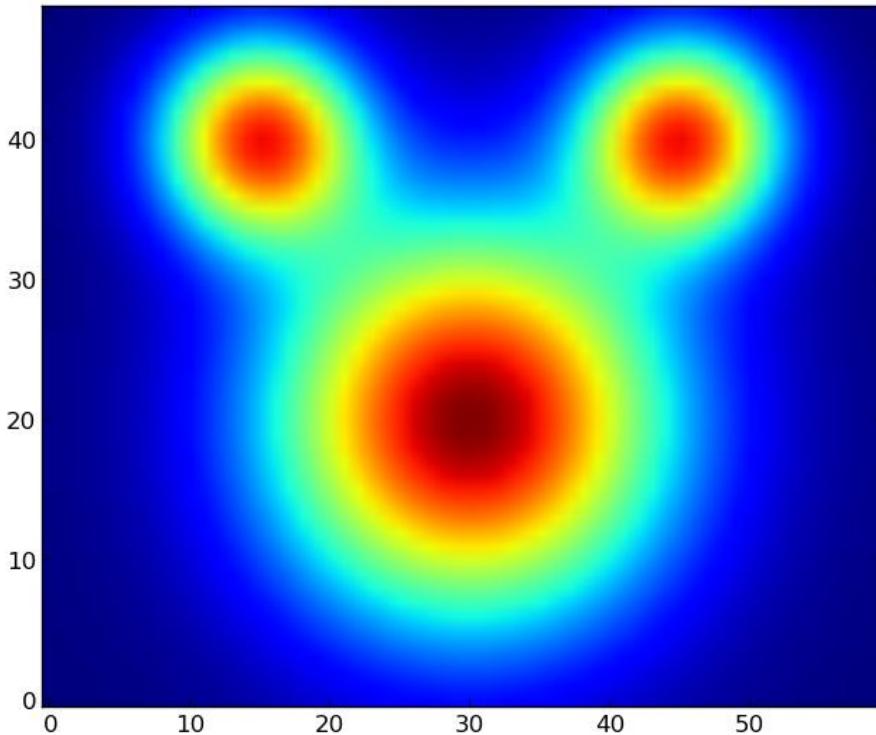


Figure 20.2: Density of true “Mickey Mouse” GMM.

**Problem 6.** Generate 750 samples from the above mixture model. Using just the drawn samples, retrain your model. Evaluate and plot your density on the grid used above. How similar is your density to the original?

How close is our trained model to the original one? We can use the symmetric Kullback-Liebler divergence to measure the distance between two probability distributions with densities  $p(x)$  and  $p'(x)$ :

$$SKL(p, p') = \left| \frac{1}{2} \int p(x) \ln \frac{p(x)}{p'(x)} dx + \frac{1}{2} \int p'(x) \ln \frac{p'(x)}{p(x)} dx \right|$$

We cannot analytically compute this, so we use a Monte Carlo approximation, which uses the fact that

$$\frac{1}{N} \sum_{i=1}^N f(x_i) \rightarrow \int f(x)p(x)dx$$

as  $N \rightarrow \infty$ , assuming that each  $x_i \sim p$ . Then we have the following approximation of the symmetric KL divergence:

$$SKL(p, p') \approx \frac{1}{2N} \left| \sum_{i=1}^N \ln \frac{p(x_i)}{p'(x_i)} + \sum_{i=1}^N \ln \frac{p'(x'_i)}{p(x'_i)} \right|$$

where  $x_i \sim p$  and  $x'_i \sim p'$ , for large  $N$ .

**Problem 7.** Write a function to compute the approximate the SKL of two GMMs. Compute the SKL between a randomly initialized GMM and the known GMM. Compute the SKL between the trained GMM and the known GMM. Is our trained model a good fit?



# 21

# Speech Recognition using CDHMMs

**Lab Objective:** *Understand how speech recognition via CDHMMs works, and implement a simplified speech recognition system.*

## 21.0.1 Continuous Density Hidden Markov Models

Some of the most powerful applications of HMMs (speech and voice recognition) result from allowing the observation space to be continuous instead of discrete. These are called Continuous Density Hidden Markov Models (CDHMMs), and they have two standard formulations: Gaussian HMMs and Gaussian Mixture Model HMMs (GMMHMMs). In fact, the former is a special case of the latter, so we will just discuss GMMHMMs in this lab.

In order to understand GMMHMMs, we need to be familiar with a particular continuous, multivariate distribution called a *mixture of Gaussians*. A mixture of Gaussians is a distribution composed of several Gaussian (or Normal) distributions with corresponding weights. Such a distribution is parameterized by the number of mixture components  $M$ , the dimension  $N$  of the normal distributions involved, a collection of component weights  $\{c_1, \dots, c_M\}$  that are nonnegative and sum to 1, and a collection of mean and covariance parameters  $\{(\mu_1, \Sigma_1), \dots, (\mu_M, \Sigma_M)\}$  for each Gaussian component. To sample from a mixture of Gaussians, one first chooses the mixture component  $i$  according to the probability weights  $\{c_1, \dots, c_M\}$ , and then one samples from the normal distribution  $\mathcal{N}(\mu_i, \Sigma_i)$ . The probability density function for a mixture of Gaussians is given by

$$f(x) = \sum_{i=1}^M c_i N(x; \mu_i, \Sigma_i),$$

where  $N(\cdot; \mu_i, \Sigma_i)$  denotes the probability density function for the normal distribution  $\mathcal{N}(\mu_i, \Sigma_i)$ . See Figure 21.1 for the plot of such a density curve. Note that a mixture of Gaussians with just one mixture component reduces to a simple normal distribution, and so a GMMHMM with just one mixture component is simply a Gaussian HMM.

In a GMMHMM, we seek to model a hidden state sequence  $\{\mathbf{x}_1, \dots, \mathbf{x}_T\}$  and a corresponding observation sequence  $\{O_1, \dots, O_T\}$ , just as with discrete HMMs. The major difference, of course, is that each observation  $O_t$  is a real-valued vector of length  $K$  distributed according to a mixture of Gaussians with  $M$  components. The parameters for such a model include the initial state distribution  $\pi$  and the state transition matrix  $A$  (just as with discrete HMMs). Additionally, for each state  $i = 1, \dots, N$ , we have component weights  $\{c_{i,1}, \dots, c_{i,M}\}$ , component means  $\{\mu_{i,1}, \dots, \mu_{i,M}\}$ , and component covariance matrices  $\{\Sigma_{i,1}, \dots, \Sigma_{i,M}\}$ .

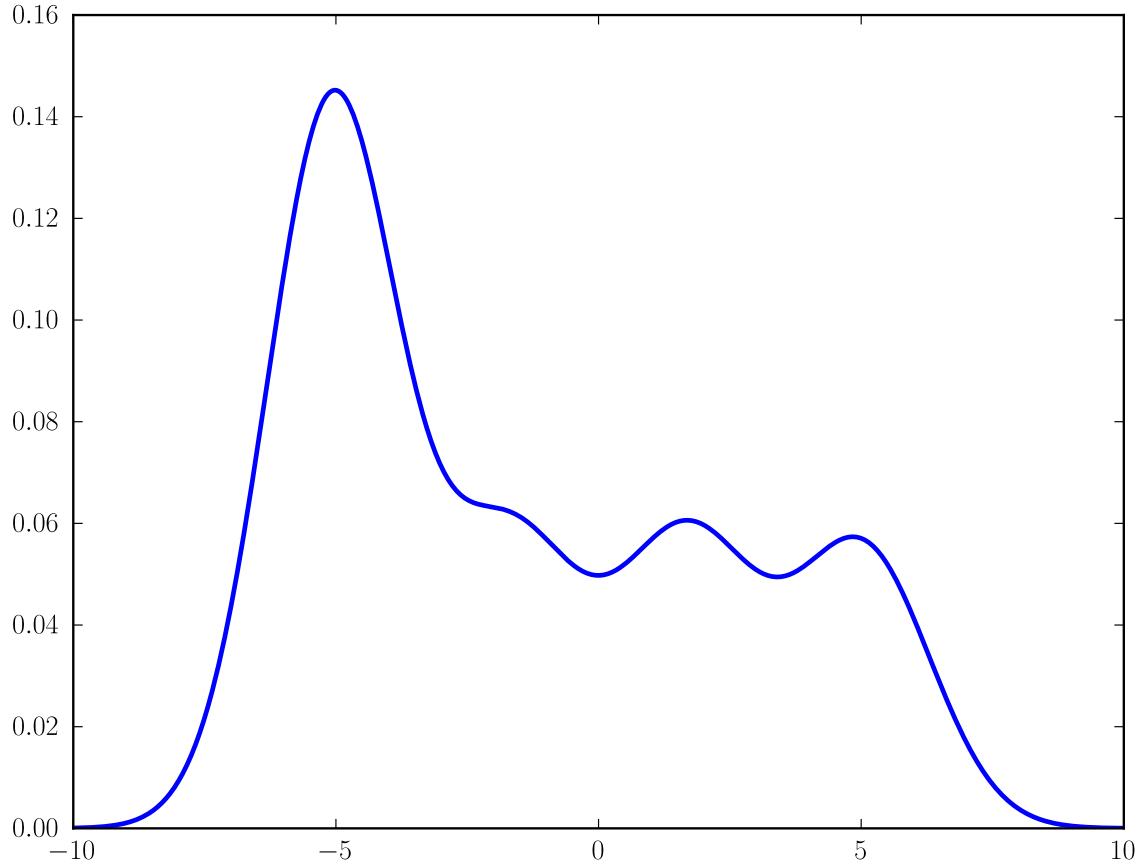


Figure 21.1: The probability density function of a mixture of Gaussians with four components.

Let's define a full GMMHMM with  $N = 2$  states,  $K = 3$ , and  $M = 3$  components.

```
>>> import numpy as np
>>> A = np.array([[.65, .35], [.15, .85]])
>>> pi = np.array([.8, .2])
>>> weights = np.array([[.7, .2, .1], [.1, .5, .4]])
>>> means1 = np.array([[0., 17., -4.], [5., -12., -8.], [-16., 22., 2.]])
>>> means2 = np.array([[-5., 3., 23.], [-12., -2., 14.], [15., -32., 0.]])
>>> means = np.array([means1, means2])
>>> covars1 = np.array([5*np.eye(3), 7*np.eye(3), np.eye(3)])
>>> covars2 = np.array([10*np.eye(3), 3*np.eye(3), 4*np.eye(3)])
>>> covars = np.array([covars1, covars2])
>>> gmmhmm = [A, weights, means, covars, pi]
```

We can draw a random sample from the GMMHMM corresponding to the second state as follows:

```
>>> sample_component = np.argmax(np.random.multinomial(1, weights[1,:]))
```

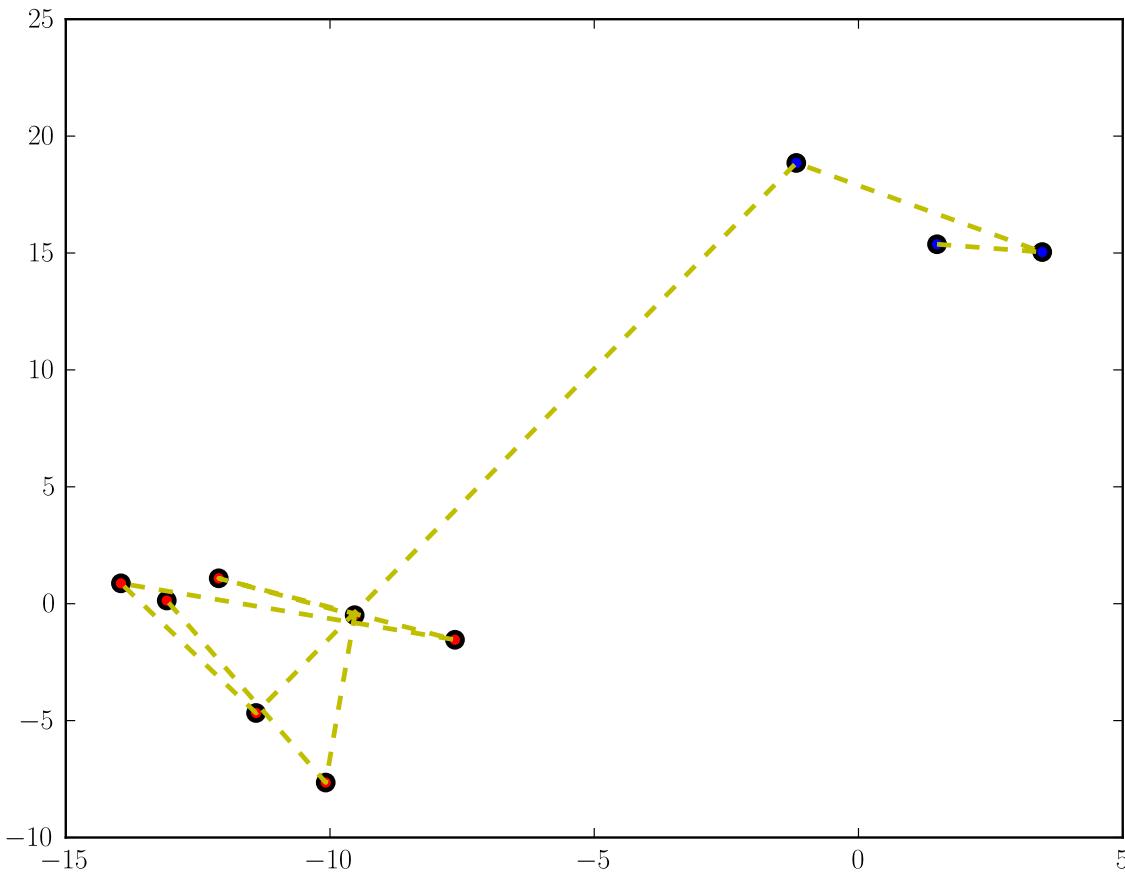


Figure 21.2: An observation sequence generated from a GMMHMM with one mixture component and two states. The observations (points in the plane) are shown as solid dots, the color indicating from which state they were generated. The connecting dotted lines indicate the sequential order of the observations.

```
>>> sample = np.random.multivariate_normal(means[1, sample_component, :], ←
    covars[1, sample_component, :, :])
```

Figure 21.2 shows an observation sequence generated from a GMMHMM with one mixture component and two states.

**Problem 1.** Write a function which accepts a GMMHMM in the format above as well as an integer  $n\_sim$ , and which simulates the GMMHMM process, generating  $n\_sim$  different observations. Do so by implementing the following function declaration.

```
def sample_gmmhmm(gmmhmm, n_sim):
    """
    Simulate sampling from a GMMHMM.
```

```

Returns
-----
states : ndarray of shape (n_sim,)
    The sequence of states
obs : ndarray of shape (n_sim, K)
    The generated observations (column vectors of length K)
"""
pass

```

The classic problems for which we normally use discrete observation HMMs can also be solved by using CDHMMs, though with continuous observations it is much more difficult to keep things numerically stable. We will not have you implement any of the three problems for CDHMMs yourself; instead, you will use a stable module we will provide for you. Note, however, that the techniques for solving these problems are still based on the forward-backward algorithm; the implementation may be trickier, but the mathematical ideas are virtually the same as those for discrete HMMs.

## Speech Recognition and Hidden Markov Models

Hidden Markov Models are the basis of modern speech recognition systems. However, a fair amount of signal processing must precede the HMM stage, and there are other components of speech recognition, such as language models, that we will not address in this lab.

The basic signal processing and HMM stages of the speech recognition system that we develop in this lab can be summarized as follows: The audio to be processed is divided into small frames of approximately 30 ms. These are short enough that we can treat the signal as being constant over these intervals. We can then take this framed signal and, through a series of transformations, represent it by mel-frequency cepstral coefficients (MFCCs), keeping only the first  $K$  (say  $K = 10$ ). Viewing these MFCCs as continuous observations in  $\mathbb{R}^K$ , we can train a GMMHMM on sequences of MFCCs for a given word, spoken multiple times. Doing this for several words, we have a collection of GMMHMMs, one for each word. Given a new speech signal, after framing and decomposing it into its MFCC array, we can score the signal against each GMMHMM, returning the word whose GMMHMM scored the highest.

Industrial-grade speech recognition systems do not train a GMMHMM for each word in a vocabulary (that would be ludicrous for a large vocabulary), but rather on *phonemes*, or distinct sounds. The English language has 44 phonemes, yielding 44 different GMMHMMs. As you could imagine, this greatly facilitates the problem of speech recognition. Each and every word can be represented by some combination of these 44 distinct sounds. By correctly classifying a signal by its phonemes, we can determine what word was spoken. Doing so is beyond the scope of this lab, so we will simply train GMMHMMs on five words/phrases: biology, mathematics, political science, psychology, and statistics.

**Problem 2.** Obtain 30 (or more) recordings for each of the words/phrases *mathematics*, *biology*, *political science*, *psychology*, and *statistics*. These audio samples should be 2 seconds in duration, recorded at a rate of 44100 samples per second, with samples stored as 16-bit signed integers in WAV format. Load the recordings into Python using `scipy.io.wavfile.read`.

If the audio files have two channels, average these channels to obtain an array of length 88200 for each sample. Extract the MFCCs from each sample using code from the file `MFCC.py`:

```
>>> import MFCC
>>> # assume sample is an array of length 88200
>>> mfccs = MFCC.extract(sample)
```

Store the MFCCs for each word in a separate list. You should have five lists, each containing 50 MFCC arrays, corresponding to each of the five words under consideration.

For a specific word, given enough distinct samples of that word (decomposed into MFCCs), we can train a GMMHMM. Recall, however, that the training procedure does not always produce a very effective model, as it can get stuck in a poor local minimum. To combat this, we will train 10 GMMHMMs for each word (using a different random initialization of the parameters each time) and keep the model with the highest log-likelihood.

For training, we will use the file we have provided called `gmmhmm.py`, as this is a stable implementation of GMMHMM algorithms. To facilitate random restarts, we need a function to provide initializations for the initial state distribution and the transition matrix.

Let `samples` be a list of arrays, where each array is the output of the MFCC extraction for a speech sample. Using a function `initialize()` that returns a random initial state distribution and (row-stochastic) transition matrix, we can train a GMMHMM with 5 states and 3 mixture components and view its log-likelihood as follows:

```
>>> import gmmhmm
>>> startprob, transmat = initialize(5)
>>> model = gmmhmm.GMMHMM(n_components=5, n_mix=3, transmat=transmat, startprob=startprob, cvtype='diag')
>>> # these values for covars_prior and var should work well for this problem
>>> model.covars_prior = 0.01
>>> model.fit(samples, init_params='mc', var=0.1)
>>> print model.logprob
```

**Problem 3.** Partition each list of MFCCs into a training set of 20 samples, and a test set of the remaining 10 samples.

Using the training sets, train a GMMHMM on each of the words from the previous problem with at least 10 random restarts, keeping the best model for each word (the one with the highest log-likelihood). This process may take several minutes. Since you will not want to run this more than once, you will want to save the best model for each word to disk using the `pickle` module so that you can use it later.

Given a trained model, we would like to compute the log-likelihood of a new sample. Letting `obs` be an array of MFCCs for a speech sample we do this as follows:

```
>>> score = model.score(obs)
```

We classify a new speech sample by scoring it against each of the 5 trained GMMHMMs, and returning the word corresponding to the GMMHMM with the highest score.

**Problem 4.** Classify the 10 test samples for each word. How does your system perform? Which words are the hardest to correctly classify? Make a dictionary containing the accuracy of the classification of your five testing sets. Specifically, the words/phrases will be the keys, and the values will be the percent accuracy.

# 22

# Gibbs Sampling and LDA

**Lab Objective:** *Understand the basic principles of implementing a Gibbs sampler. Apply this to Latent Dirichlet Allocation.*

## Gibbs Sampling

Gibbs sampling is an MCMC sampling method in which we construct a Markov chain which is used to sample from a desired joint (conditional) distribution

$$\mathbb{P}(x_1, \dots, x_n | \mathbf{y}).$$

Often it is difficult to sample from this high-dimensional joint distribution, while it may be easy to sample from the one-dimensional conditional distributions

$$\mathbb{P}(x_i | \mathbf{x}_{-i}, \mathbf{y})$$

where  $\mathbf{x}_{-i} = x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_n$ .

---

### Algorithm 22.1 Basic Gibbs Sampling Process.

---

```

1: procedure GIBBS SAMPLER
2:   Randomly initialize  $x_1, x_2, \dots, x_n$ .
3:   for  $k = 1, 2, 3, \dots$  do
4:     for  $i = 1, 2, \dots, n$  do
5:       Draw  $x \sim \mathbb{P}(x_i | \mathbf{x}_{-i}, \mathbf{y})$ 
6:       Fix  $x_i = x$ 
7:      $\mathbf{x}^{(k)} = (x_1, x_2, \dots, x_n)$ 

```

---

A Gibbs sampler proceeds according to Algorithm 22.1. Each iteration of the outer for loop is a *sweep* of the Gibbs sampler, and the value of  $\mathbf{x}^{(k)}$  after a sweep is a *sample*. This creates an irreducible, non-null recurrent, aperiodic Markov chain over the state space consisting of all possible  $\mathbf{x}$ . The unique invariant distribution for the chain is the desired joint distribution

$$\mathbb{P}(x_1, \dots, x_n | \mathbf{y}).$$

Thus, after a burn-in period, our samples  $\mathbf{x}^{(k)}$  are effectively samples from the desired distribution.

Consider the dataset of  $N$  scores from a calculus exam in the file `examscores.csv`. We believe that the spread of these exam scores can be modeled with a normal distribution of mean  $\mu$  and variance  $\sigma^2$ . Because we are unsure of the true value of  $\mu$  and  $\sigma^2$ , we take a Bayesian approach and place priors on each parameter to quantify this uncertainty:

$$\begin{aligned}\mu &\sim N(\nu, \tau^2) && \text{(a normal distribution)} \\ \sigma^2 &\sim IG(\alpha, \beta) && \text{(an inverse gamma distribution)}\end{aligned}$$

Letting  $\mathbf{y} = (y_1, \dots, y_N)$  be the set of exam scores, we would like to update our beliefs of  $\mu$  and  $\sigma^2$  by sampling from the posterior distribution

$$\mathbb{P}(\mu, \sigma^2 | \mathbf{y}, \nu, \tau^2, \alpha, \beta).$$

Sampling directly can be difficult. However, we *can* easily sample from the following conditional distributions:

$$\begin{aligned}\mathbb{P}(\mu | \sigma^2, \mathbf{y}, \nu, \tau^2, \alpha, \beta) &= \mathbb{P}(\mu | \sigma^2, \mathbf{y}, \nu, \tau^2) \\ \mathbb{P}(\sigma^2 | \mu, \mathbf{y}, \nu, \tau^2, \alpha, \beta) &= \mathbb{P}(\sigma^2 | \mu, \mathbf{y}, \alpha, \beta)\end{aligned}$$

The reason for this is that these conditional distributions are *conjugate* to the prior distributions, and hence are part of the same distributional families as the priors. In particular, we have

$$\begin{aligned}\mathbb{P}(\mu | \sigma^2, \mathbf{y}, \nu, \tau^2) &= N(\mu^*, (\sigma^*)^2) \\ \mathbb{P}(\sigma^2 | \mu, \mathbf{y}, \alpha, \beta) &= IG(\alpha^*, \beta^*),\end{aligned}$$

where

$$\begin{aligned}(\sigma^*)^2 &= \left( \frac{1}{\tau^2} + \frac{N}{\sigma^2} \right)^{-1} \\ \mu^* &= (\sigma^*)^2 \left( \frac{\nu}{\tau^2} + \frac{1}{\sigma^2} \sum_{i=1}^N y_i \right) \\ \alpha^* &= \alpha + \frac{N}{2} \\ \beta^* &= \beta + \frac{1}{2} \sum_{i=1}^N (y_i - \mu)^2\end{aligned}$$

We have thus set this up as a Gibbs sampling problem, where we have only to alternate between sampling  $\mu$  and sampling  $\sigma^2$ . We can sample from a normal distribution and an inverse gamma distribution as follows:

```
>>> from math import sqrt
>>> from scipy.stats import norm
>>> from scipy.stats import invgamma
>>> mu = 0. # the mean
>>> sigma2 = 9. # the variance
>>> normal_sample = norm.rvs(mu, scale=sqrt(sigma))
>>> alpha = 2.
>>> beta = 15.
>>> invgamma_sample = invgamma.rvs(alpha, scale=beta)
```

Note that when sampling from the normal distribution, we need to set the `scale` parameter to the standard deviation, *not* the variance.

**Problem 1.** Implement a Gibbs sampler for the exam scores problem using the following function declaration.

```
def gibbs(y, nu, tau2, alpha, beta, n_samples):
    """
    Assuming a likelihood and priors
    y_i ~ N(mu, sigma2),
    mu ~ N(nu, tau2),
    sigma2 ~ IG(alpha, beta),
    sample from the posterior distribution
    P(mu, sigma2 | y, nu, tau2, alpha, beta)
    using a gibbs sampler.

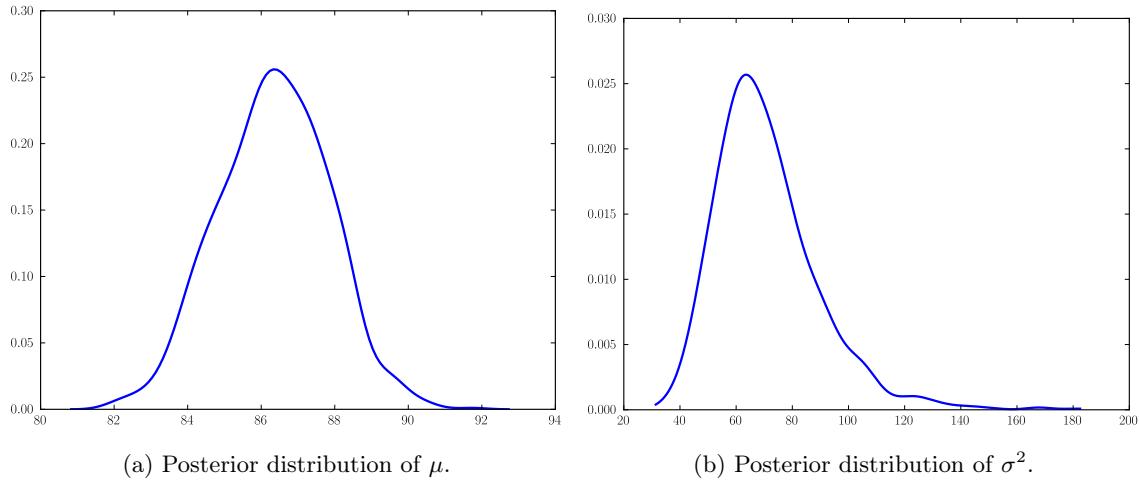
    Parameters
    -----
    y : ndarray of shape (N,)
        The data
    nu : float
        The prior mean parameter for mu
    tau2 : float > 0
        The prior variance parameter for mu
    alpha : float > 0
        The prior alpha parameter for sigma2
    beta : float > 0
        The prior beta parameter for sigma2
    n_samples : int
        The number of samples to draw

    Returns
    -----
    samples : ndarray of shape (n_samples,2)
        1st col = mu samples, 2nd col = sigma2 samples
    """
    pass
```

Test it with priors  $\nu = 80$ ,  $\tau^2 = 16$ ,  $\alpha = 3$ ,  $\beta = 50$ , collecting 1000 samples. Plot your samples of  $\mu$  and your samples of  $\sigma^2$ . How long did it take for each to converge? It should have been very quick.

We'd like to look at the posterior marginal distributions for  $\mu$  and  $\sigma^2$ . To plot these from the samples, we will use a kernel density estimator. If our samples of  $\mu$  are called `mu_samples`, then we can do this as follows:

```
>>> import numpy as np
>>> from scipy.stats import gaussian_kde
>>> import matplotlib.pyplot as plt
>>> mu_kernel = gaussian_kde(mu_samples)
>>> x_min = min(mu_samples) - 1
```

Figure 22.1: Posterior marginal probability densities for  $\mu$  and  $\sigma^2$ .

```
>>> x_max = max(mu_samples) + 1
>>> x = np.arange(x_min, x_max, step=0.1)
>>> plt.plot(x,mu_kernel(x))
>>> plt.show()
```

**Problem 2.** Plot the kernel density estimators for the posterior distributions of  $\mu$  and  $\sigma^2$ . You should get plots similar to those in Figure 22.1.

Keep in mind that the plots above are of the posterior distributions of the *parameters*, not of the scores. If we would like to compute the posterior distribution of a new exam score  $\tilde{y}$  given our data  $\mathbf{y}$  and prior parameters, we compute what is known as the *posterior predictive distribution*:

$$\mathbb{P}(\tilde{y}|\mathbf{y}, \lambda) = \int_{\Theta} \mathbb{P}(\tilde{y}|\Theta) \mathbb{P}(\Theta|\mathbf{y}, \lambda) d\Theta$$

where  $\Theta$  denotes our parameters (in our case  $\mu$  and  $\sigma^2$ ) and  $\lambda$  denotes our prior parameters (in our case  $\nu, \tau^2, \alpha$ , and  $\beta$ ).

Rather than actually computing this integral for each possible  $\tilde{y}$ , we can do this by sampling scores from our parameter samples. In other words, sample

$$\tilde{y}_{(t)} \sim N(\mu_{(t)}, \sigma_{(t)}^2)$$

for each sample pair  $\mu_{(t)}, \sigma_{(t)}^2$ . Now we have essentially drawn samples from our posterior predictive distribution, and we can use a kernel density estimator to plot this distribution from the samples.

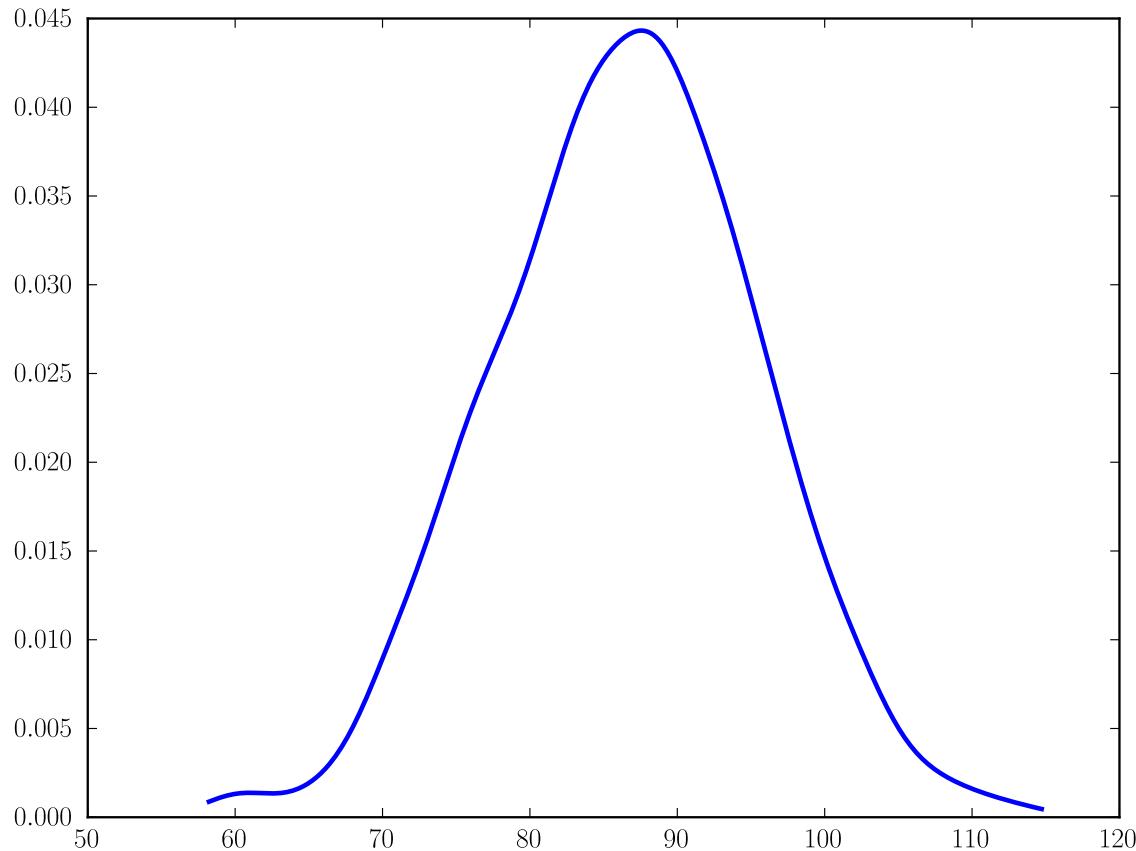


Figure 22.2: Predictive posterior distribution of exam scores.

**Problem 3.** Use your samples of  $\mu$  and  $\sigma^2$  to draw samples from the posterior predictive distribution. Plot the kernel density estimator of your sampled scores. It should resemble the plot in Figure 22.2.

## Latent Dirichlet Allocation

Gibbs sampling can be applied to an interesting problem in language processing: determining which topics are prevalent in a document. Latent Dirichlet Allocation (LDA) is a generative model for a collection of text documents. It supposes that there is some fixed vocabulary (composed of  $V$  distinct terms) and  $K$  different topics, each represented as a probability distribution  $\phi_k$  over the vocabulary, each with a Dirichlet prior  $\beta$ . What this means is that  $\phi_{k,v}$  is the probability that topic  $k$  is represented by vocabulary term  $v$ .

With the vocabulary and topics chosen, the LDA model assumes that we have a set of  $M$  documents (each “document” may be a paragraph or other section of the text, rather than a “full” document). The  $m$ -th document consists of  $N_m$  words, and a probability distribution  $\theta_m$  over the topics is drawn from a Dirichlet distribution with parameter  $\alpha$ . Thus  $\theta_{m,k}$  is the probability that document  $m$  is assigned the label  $k$ . If  $\phi_{k,v}$  and  $\theta_{m,k}$  are viewed as matrices, their rows sum to one.

We will now iterate through each document in the same manner. Assume we are working on document  $m$ , which you will recall contains  $N_m$  words. For word  $n$ , we first draw a topic assignment  $z_{m,n}$  from the categorical distribution  $\theta_m$ , and then we draw a word  $w_{m,n}$  from the categorical distribution  $\phi_{z_{m,n}}$ . Throughout this implementation, we assume  $\alpha$  and  $\beta$  are scalars. In summary, we have

1. Draw  $\phi_k \sim \text{Dir}(\beta)$  for  $1 \leq k \leq K$ .
2. For  $1 \leq m \leq M$ :
  - (a) Draw  $\theta_m \sim \text{Dir}(\alpha)$ .
  - (b) Draw  $z_{m,n} \sim \text{Cat}(\theta_m)$  for  $1 \leq n \leq N_m$ .
  - (c) Draw  $w_{m,n} \sim \text{Cat}(\phi_{z_{m,n}})$  for  $1 \leq n \leq N_m$ .

What we end up with here for document  $m$  is  $n$  words which represent the document. Note that these words are *not* distinct from one another; indeed, we are most interested in the words that have been repeated the most.

This is typically depicted with graphical plate notation as in Figure 22.3.

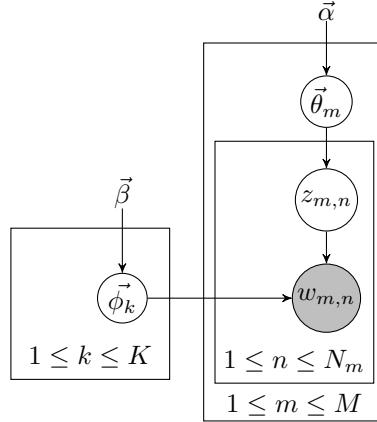


Figure 22.3: Graphical plate notation for LDA text generation.

In the plate model, only the variables  $w_{m,n}$  are shaded, signifying that these are the only observations visible to us; the rest are latent variables. Our goal is to estimate each  $\phi_k$  and each  $\theta_m$ . This will allow us to understand what each topic is, as well as understand how each document is distributed over the  $K$  topics. In other words, we want to predict the topic of each document, and also which words best represent this topic. We can estimate these well if we know  $z_{m,n}$  for each  $m, n$ , collectively referred to as  $\mathbf{z}$ . Thus, we need to sample  $\mathbf{z}$  from the posterior distribution  $\mathbb{P}(\mathbf{z}|\mathbf{w}, \alpha, \beta)$ , where  $\mathbf{w}$  is the collection words in the text corpus. Unsurprisingly, it is intractable to sample directly from the joint posterior distribution. However, letting  $\mathbf{z}_{-(m,n)} = \mathbf{z} \setminus \{z_{m,n}\}$ , the conditional posterior distributions

$$\mathbb{P}(z_{m,n} = k | \mathbf{z}_{-(m,n)}, \mathbf{w}, \alpha, \beta)$$

have nice, closed form solutions, making them easy to sample from.

These conditional distributions have the following form:

$$\mathbb{P}(z_{m,n} = k | \mathbf{z}_{-(m,n)}, \mathbf{w}, \alpha, \beta) \propto \frac{(n_{(k,m,\cdot)}^{-} + \alpha)(n_{(k,\cdot,w_{m,n})}^{-} + \beta)}{n_{(k,\cdot,\cdot)}^{-} + V\beta}$$

where

$$\begin{aligned}
 n_{(k,m,\cdot)} &= \text{the number of words in document } m \text{ assigned to topic } k \\
 n_{(k,\cdot,v)} &= \text{the number of times term } v = w_{m,n} \text{ is assigned to topic } k \\
 n_{(k,\cdot,\cdot)} &= \text{the number of times topic } k \text{ is assigned in the corpus} \\
 n_{(k,m,\cdot)}^{-(m,n)} &= n_{(k,m,\cdot)} - \mathbf{1}_{z_{m,n}=k} \\
 n_{(k,\cdot,v)}^{-(m,n)} &= n_{(k,\cdot,v)} - \mathbf{1}_{z_{m,n}=k} \\
 n_{(k,\cdot,\cdot)}^{-(m,n)} &= n_{(k,\cdot,\cdot)} - \mathbf{1}_{z_{m,n}=k}
 \end{aligned}$$

Thus, if we simply keep track of these count matrices, then we can easily create a Gibbs sampler over the topic assignments. This is actually a particular class of samplers known as *collapsed Gibbs samplers*, because we have collapsed the sampler by integrating out  $\theta$  and  $\phi$ .

We have provided for you the structure of a Python object LDACGS with several methods. The object is already defined to have attributes `n_topics`, `documents`, `vocab`, `alpha`, and `beta`, where `vocab` is a list of strings (terms), and `documents` is a list of dictionaries (a dictionary for each document). Each entry in dictionary  $m$  is of the form  $n : w$ , where  $w$  is the index in `vocab` of the  $n^{\text{th}}$  word in document  $m$ .

Throughout this lab we will guide you through writing several more methods in order to implement the Gibbs sampler. The first step is to initialize our assignments, and create the count matrices  $n_{(k,m,\cdot)}$ ,  $n_{(k,\cdot,v)}$  and vector  $n_{(k,\cdot,\cdot)}$ .

**Problem 4.** Complete the method `initialize`. By randomly assigning initial topics, fill in the count matrices and topic assignment dictionary. In this method, you will initialize the count matrices (among other things). Note that the notation provided in the code is slightly different than that used above. Be sure to understand how the formulae above connect with the code.

To be explicit, you will need to initialize  $nmz$ ,  $nzw$ , and  $nz$  to be zero arrays of the correct size. Then, in the second for loop, you will assign  $z$  to be a random integer in the correct range of topics. In the increment step, you need to figure out the correct indices to increment by one for each of the three arrays. Finally, assign `topics` as given.

The next method we need to write fully outlines a sweep of the Gibbs sampler.

**Problem 5.** Complete the method `_sweep`, which needs to iterate through each word of each document. It should call on the method `_conditional` to get the conditional distribution at each iteration.

Note that the first part of this method will undo what the `initialize` method did. Then we will use the conditional distribution (instead of the uniform distribution we used previously) to pick a more accurate topic assignment. Finally, the latter part repeats what we did in `initialize`, but does so using this more accurate topic assignment.

We are now prepared to write the full Gibbs sampler.

**Problem 6.** Complete the method `sample`. The argument `filename` is the name and location of a .txt file, where each line is considered a document. The corpus is built by method `buildCorpus`, and stopwords are removed (if argument `stopwords` is provided). Burn in the Gibbs sampler, computing and saving the log-likelihood with the method `_loglikelihood`. After the burn in, iterate further, accumulating your count matrices, by adding `nzw` and `nmz` to `total_nzw` and `total_nmz` respectively, where you only add every  $sample\_rate^{th}$  iteration. Also save each log-likelihood.

You should now have a working Gibbs sampler to perform LDA inference on a corpus. Let's test it out on Ronald Reagan's State of the Union addresses.

**Problem 7.** Create an `LDACGS` object with 20 topics, letting `alpha` and `beta` be the default values. Load in the stop word list provided. Run the Gibbs sampler, with a burn in of 100 iterations, accumulating 10 samples, only keeping the results of every 10<sup>th</sup> sweep. Plot the log-likelihoods. How long did it take to truly burn in?

We can estimate the values of each  $\phi_k$  and each  $\theta_m$  as follows:

$$\hat{\theta}_{m,k} = \frac{n_{(k,m,\cdot)} + \alpha}{K \cdot \alpha + \sum_{k=1}^K n_{(k,m,\cdot)}}$$

$$\hat{\phi}_{k,v} = \frac{n_{(k,\cdot,v)} + \beta}{V \cdot \beta + \sum_{v=1}^V n_{(k,\cdot,v)}}$$

We have provided methods `phi` and `theta` that do this for you. We often examine the topic-term distributions  $\phi_k$  by looking at the  $n$  terms with the highest probability, where  $n$  is small (say 10 or 20). We have provided a method `topterms` which does this for you.

**Problem 8.** Using the methods described above, examine the topics for Reagan's addresses. As best as you can, come up with labels for each topic. Note that if  $ntopics = 20$  and  $n = 10$ , we will get the top 10 words that represent each of the 20 topics. What you will want to do for each topic is decide what these ten words jointly represent represent. Save your topic labels in a list or an array.

We can use  $\hat{\theta}$  to find the paragraphs in Reagan's addresses that focus the most on each topic. The documents with the highest values of  $\hat{\theta}_k$  are those most heavily focused on topic  $k$ . For example, if you chose the topic label for topic  $p$  to be *the Cold War*, you can find the five highest values in  $\hat{\theta}_p$ , which will tell you which five paragraphs are most centered on the Cold War.

Let's take a moment to see what our Gibbs sampler has accomplished. By simply feeding in a group of documents, and with no human input, we have found the most common topics discussed, which are represented by the words most frequently used in relation to that particular topic. The only work that the user has done is to assign topic labels, saying what the words in each group have in common. As you may have noticed, however, these topics may or may not be *relevant* topics. You might have noticed that some of the most common topics were simply English particles (words such as *a*, *the*, *an*) and conjunctions (*and*, *so*, *but*). Industrial grade packages can effectively remove such topics so that they are not included in the results.

# 23

## Metropolis Algorithm

**Lab Objective:** *Understand the basic principles of the Metropolis algorithm and apply these ideas to the Ising Model.*

### The Metropolis Algorithm

Sampling from a given probability distribution is an important task in many different applications found throughout the sciences. When these distributions are complicated, as is often the case when modeling real-world problems, direct sampling methods can become difficult, as they might involve computing high-dimensional integrals. The Metropolis algorithm is an effective method to sample from many distributions, requiring only that we be able to evaluate the probability density function up to a constant of proportionality. In particular, the Metropolis algorithm does not require us to compute difficult high-dimensional integrals, such as those that are found in the denominator of Bayesian posterior distributions.

The Metropolis algorithm is an MCMC sampling method which generates a sequence of random variables, similar to Gibbs sampling. These random variables form a Markov Chain whose invariant distribution is equal to the distribution from which we wish to sample. Suppose that  $h : \mathbb{R}^n \rightarrow \mathbb{R}$  is the probability density function of distribution, and suppose that  $f(\theta) = c \cdot h(\theta)$  for some nonzero constant  $c$  (in practice, we assume that  $f$  is an easy function to evaluate, while  $h$  is difficult). Let  $Q : \mathbb{R}^n \times \mathbb{R}^n \rightarrow \mathbb{R}$  be a symmetric *proposal function* (so that  $Q(\cdot, y)$  is a probability density function for all  $y \in \mathbb{R}^n$ , and  $Q(x, y) = Q(y, x)$  for all  $x, y \in \mathbb{R}^n$ ) and let  $A : \mathbb{R}^n \times \mathbb{R}^n \rightarrow \mathbb{R}$  be an *acceptance function* defined by

$$A(x, y) = \min\left(1, \frac{f(y)}{f(x)}\right).$$

We can combine these functions in such a way so as to sample from the aforementioned Markov Chain by following Algorithm 23.1. The Metropolis algorithm can be interpreted as follows: given our current state  $y$ , we propose a new state according to the distribution  $Q(\cdot, y)$ . We then accept or reject it according to  $A$ . We continue by repeating the process. So long as  $Q$  defines an irreducible, aperiodic, and non-null recurrent Markov chain, we will have a Markov chain whose unique invariant distribution will have density  $h$ . Furthermore, given any initial state, the chain will converge to this invariant distribution. Note that for numerical reasons, it is often wise to make calculations of the acceptance functions in log space:

$$\log A(x, y) = \min(0, \log f(x) - \log f(y)).$$

**Algorithm 23.1** Metropolis Algorithm

---

```

1: procedure METROPOLIS ALGORITHM
2:   Choose initial point  $x_0$ .
3:   for  $t = 1, 2, \dots$  do
4:     Draw  $x' \sim Q(\cdot, x_{t-1})$ 
5:     Draw  $a \sim \text{unif}(0, 1)$ 
6:     if  $a \leq A(x', x_{t-1})$  then
7:        $x_t = x'$ 
8:     else
9:        $x_t = x_{t-1}$ 
10:    Return  $x_1, x_2, x_3, \dots$ 
```

---

Let's apply the Metropolis algorithm to a simple example of Bayesian analysis. Consider the problem of computing the posterior distribution over the mean  $\mu$  and variance  $\sigma^2$  of a normal distribution for which we have  $N$  data points  $y_1, \dots, y_N$ . For concreteness, we use the data in `examscores.csv` and we assume the prior distributions

$$\begin{aligned}\mu &\sim N(\mu_0 = 80, \sigma_0^2 = 16) \\ \sigma^2 &\sim IG(\alpha = 3, \beta = 50).\end{aligned}$$

In this situation, we wish to sample from the posterior distribution

$$p(\mu, \sigma^2 | y_1, \dots, y_N) = \frac{p(\mu)p(\sigma^2) \prod_{i=1}^N N(y_i | \mu, \sigma^2)}{\int_{-\infty}^{\infty} \int_0^{\infty} p(\mu)p(\sigma^2) \prod_{i=1}^N N(y_i | \mu, \sigma^2) d\sigma^2 d\mu}$$

. However, we can conveniently calculate only the numerator of this expression. Since the denominator is simply a constant with respect to  $\mu$  and  $\sigma^2$ , the numerator can serve as the function  $f$  in the Metropolis algorithm, and the denominator can serve as the constant  $c$ . We choose our proposal function to be based on a bivariate Normal distribution:

$$Q(x, y) = N(x | y, sI),$$

where  $I$  is the  $2 \times 2$  identity matrix and  $s$  is some positive scalar. Let's create these functions in Python:

```

import numpy as np
from math import sqrt, exp, log
import scipy.stats as st
from matplotlib import pyplot as plt
from scipy.stats import gaussian_kde

# load in the data
scores = np.loadtxt('examscores')

# initialize the hyperparameters
alpha = 3
beta = 50
mu0 = 80
sig20 = 16
```

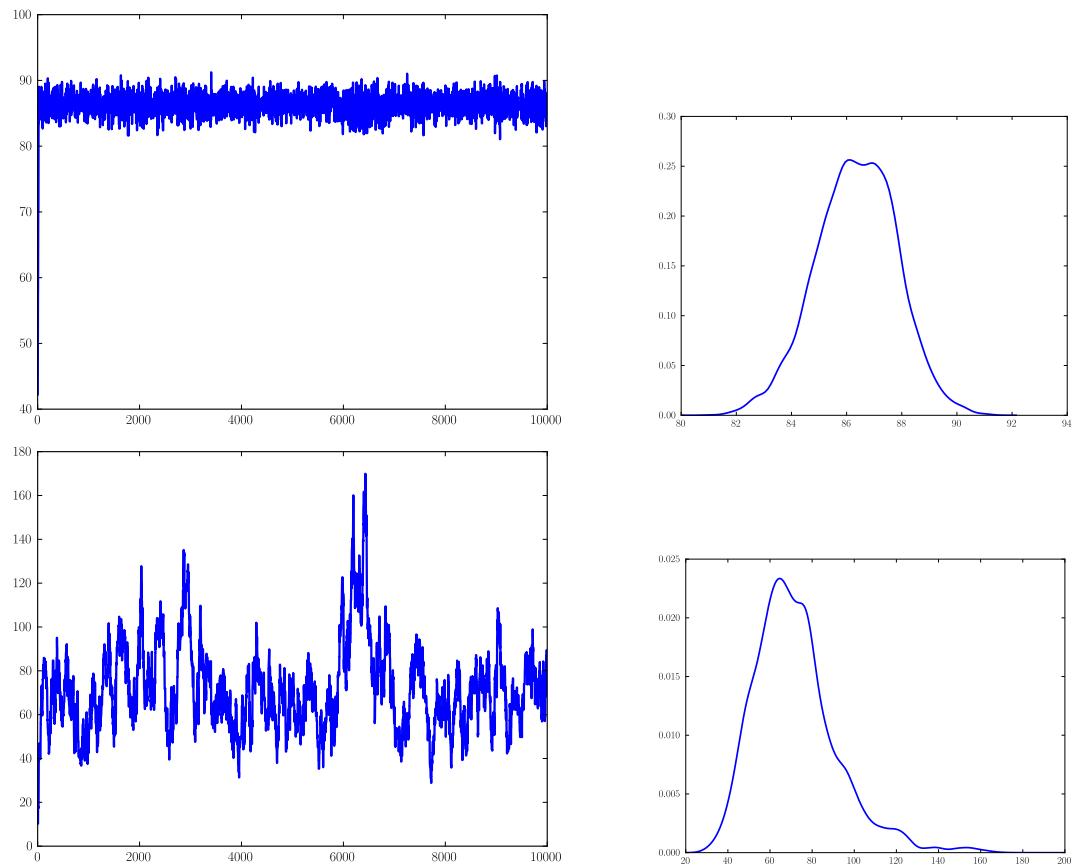


Figure 23.1: Metropolis samples and KDEs for the marginal posterior distribution of  $\mu$  (top row) and  $\sigma^2$  (bottom row).

```
# initialize the prior distributions
muprior = st.norm(loc=mu0, scale=sqrt(sig20))
sig2prior = st.invgamma(alpha,scale=beta)

# define the proposal function
def proposal(y, s):
    return st.multivariate_normal.rvs(mean=y, cov=s*np.eye(len(y)))

# define the log of the proportional density
def propLogDensity(x):
    return muprior.logpdf(x[0])+sig2prior.logpdf(x[1])+st.norm.logpdf(scores,←
        loc=x[0],scale=sqrt(x[1])).sum()
```

We are now ready to code up the Metropolis algorithm using these functions. We will keep track of the samples generated by the algorithm, along with the proportional log densities of the samples and the proportion of proposed samples that were accepted. Study the implementation below to make sure you understand the process:

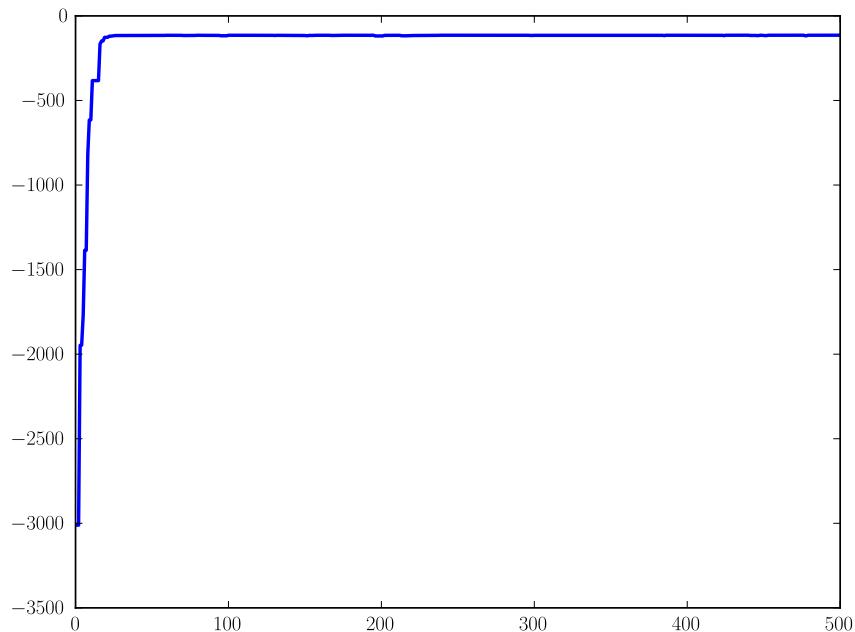


Figure 23.2: Log densities of the first 500 Metropolis samples.

```
def metropolis(x0, s, n_samples):
    """
    Use the Metropolis algorithm to sample from posterior.

    Parameters
    -----
    x0 : ndarray of shape (2,)
        The first entry is mu, the second entry is sigma2
    s : float > 0
        The standard deviation parameter for the proposal function
    n_samples : int
        The number of samples to generate

    Returns
    -----
    draws : ndarray of shape (n_samples, 2)
        The MCMC samples
    logprobs : ndarray of shape (n_samples)
        The log density of the samples
    accept_rate : float
        The proportion of proposed samples that were accepted
    """
    accept_counter = 0
    draws = np.empty((n_samples, 2))
    logprob = np.empty(n_samples)
```

```

x = x0.copy()
for i in xrange(n_samples):
    xprime = proposal(x,s)
    u = np.random.rand(1)[0]
    if log(u) <= propLogDensity(xprime) - propLogDensity(x):
        accept_counter += 1
        x = xprime
    draws[i] = x
    logprob[i] = propLogDensity(x)
return draws, logprob, accept_counter/float(n_samples)

```

Now let's sample from the posterior. We will choose an initial guess of  $\mu = 40$  and  $\sigma^2 = 10$ , and we will set  $s = 20$ . We draw 10000 samples as follows:

```

>>> draws, lprobs, rate = metropolis(np.array([40, 10], dtype=float), 20., ←
    10000)
>>> print "Acceptance Rate:", r
Acceptance Rate: 0.3531

```

We can evaluate the quality of our results by plotting the log probabilities, the  $\mu$  samples, the  $\sigma^2$  samples, and kernel density estimators for the marginal posterior distributions of  $\mu$  and  $\sigma^2$ . The code below will accomplish this task:

```

>>> # plot the first 500 log probs
>>> plt.plot(lprobs[:500])
>>> plt.show()
>>> # plot the mu samples
>>> plt.plot(draws[:,0])
>>> plt.show()
>>> # plot the sigma2 samples
>>> plt.plot(draws[:,1])
>>> plt.show()
>>> # build and plot KDE for posterior mu
>>> mu_kernel = gaussian_kde(draws[50:,0])
>>> x_min = min(draws[50:,0]) - 1
>>> x_max = max(draws[50:,0]) + 1
>>> x = np.arange(x_min, x_max, step=0.1)
>>> plt.plot(x,mu_kernel(x))
>>> plt.show()
>>> # build and plot KDE for posterior sigma2
>>> sig_kernel = gaussian_kde(draws[50:,1])
>>> x_min = 20
>>> x_max = 200
>>> x = np.arange(x_min, x_max, step=0.1)
>>> plt.plot(x,sig_kernel(x))
>>> plt.show()

```

Your results should be close to those given in Figures 23.1 and 23.2.

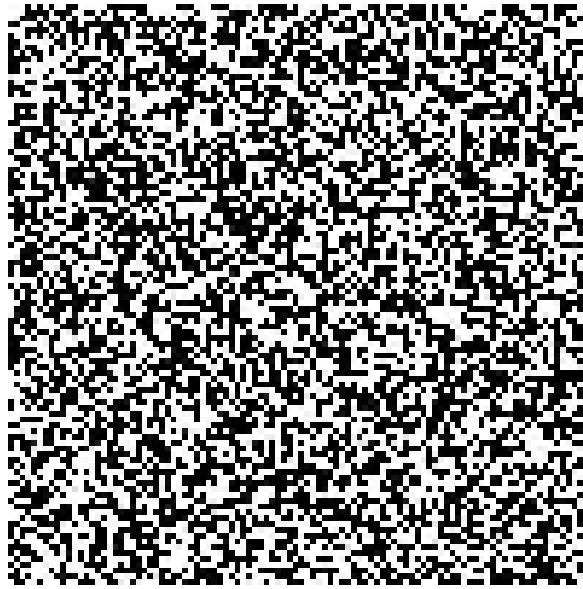


Figure 23.3: Spin configuration from random initialization.

## The Ising Model

In statistical mechanics, the Ising model describes how atoms interact in ferromagnetic material. Assume we have some lattice  $\Lambda$  of sites. We say  $i \sim j$  if  $i$  and  $j$  are adjacent sites. Each site  $i$  in our lattice is assigned an associated *spin*  $\sigma_i \in \{\pm 1\}$ . A *state* in our Ising model is a particular spin configuration  $\sigma = (\sigma_k)_{k \in \Lambda}$ . If  $L = |\Lambda|$ , then there are  $2^L$  possible states in our model. If  $L$  is large, the state space becomes huge, which is why MCMC sampling methods (in particular the Metropolis algorithm) are so useful in calculating model estimations.

With any spin configuration  $\sigma$ , there is an associated energy

$$H(\sigma) = -J \sum_{i \sim j} \sigma_i \sigma_j$$

where  $J > 0$  for ferromagnetic materials, and  $J < 0$  for antiferromagnetic materials. Throughout this lab, we will assume  $J = 1$ , leaving the energy equation to be  $H(\sigma) = -\sum_{i \sim j} \sigma_i \sigma_j$  where the interaction from each pair is added only once.

We will consider a lattice that is a  $100 \times 100$  square grid. The adjacent sites for a given site are those directly above, below, to the left, and to the right of the site, so to speak. For sites on the edge of the grid, we assume it wraps around. In other words, a site at the farthest left side of the grid is adjacent to the corresponding site on the farthest right side. Thus, a single spin configuration can be represented as a  $100 \times 100$  array, with entries of  $\pm 1$ .

**Problem 1.** Write a function that initializes a spin configuration for an  $n \times n$  lattice. It should return an  $n \times n$  array, each entry of which is either 1 or  $-1$ , chosen randomly. Test this for the grid described above, and plot the spin configuration using `matplotlib.pyplot.imshow`. It should look fairly random, as in Figure 23.3.

**Problem 2.** Write a function that computes the energy of a wrap-around  $n \times n$  lattice with a given spin configuration, as described above. Make sure that you do not double count site pair interactions!

Different spin configurations occur with different probabilities, depending on the energy of the spin configuration and  $\beta > 0$ , a quantity inversely proportional to the temperature. More specifically, for a given  $\beta$ , we have

$$\mathbb{P}_\beta(\sigma) = \frac{e^{-\beta H(\sigma)}}{Z_\beta}$$

where  $Z_\beta = \sum_\sigma e^{-\beta H(\sigma)}$ . Because there are  $2^{100 \cdot 100} = 2^{10000}$  possible spin configurations for our particular lattice, computing this sum is infeasible. However, the numerator is quite simple, provided we can efficiently compute the energy  $H(\sigma)$  of a spin configuration. Thus the ratio of the probability densities of two spin configurations is simple:

$$\begin{aligned} \frac{\mathbb{P}_\beta(\sigma^*)}{\mathbb{P}_\beta(\sigma)} &= \frac{e^{-\beta H(\sigma^*)}}{e^{-\beta H(\sigma)}} \\ &= e^{\beta(H(\sigma) - H(\sigma^*))} \end{aligned}$$

The simplicity of this ratio should lead us to think that a Metropolis algorithm might be an appropriate way by which to sample from the spin configuration probability distribution, in which case our acceptance probability would be

$$A(\sigma^*, \sigma) = \begin{cases} 1 & \text{if } H(\sigma^*) < H(\sigma) \\ e^{\beta(H(\sigma) - H(\sigma^*))} & \text{otherwise.} \end{cases}$$

By choosing our transition matrix  $Q$  cleverly, we can also make it easy to compute the energy for any proposed spin configuration. We restrict our possible proposals to only those spin configurations in which we have flipped the spin at exactly one lattice site, i.e. we choose a lattice site  $i$  and flip its spin. Thus, there are only  $L$  possible proposal spin configurations  $\sigma^*$  given  $\sigma$ , each being proposed with probability  $\frac{1}{L}$ , and such that  $\sigma_j^* = \sigma_j$  for all  $j \neq i$ , and  $\sigma_i^* = -\sigma_i$ . Note that we would never actually write out this matrix (it would be  $2^{10000} \times 2^{10000}$ !!!). Computing the proposed site's energy is simple: if the spin flip site is  $i$ , then we have  $H(\sigma^*) = H(\sigma) + 2 \sum_{j:j \sim i} \sigma_i \sigma_j$ .

**Problem 3.** Write a function that proposes a new spin configuration given the current spin configuration on an  $n \times n$  lattice, as described above. This function simply needs to return a pair of indices  $(i, j)$ , chosen with probability  $\frac{1}{n^2}$ .

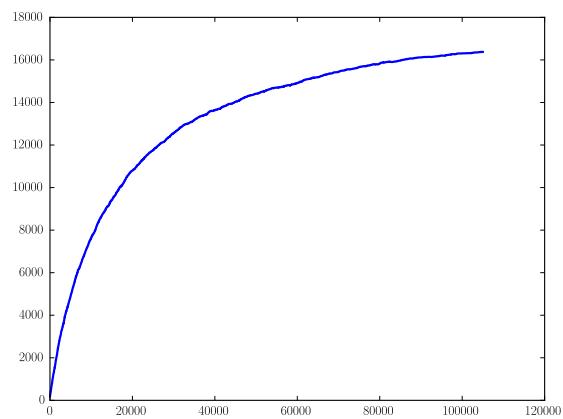
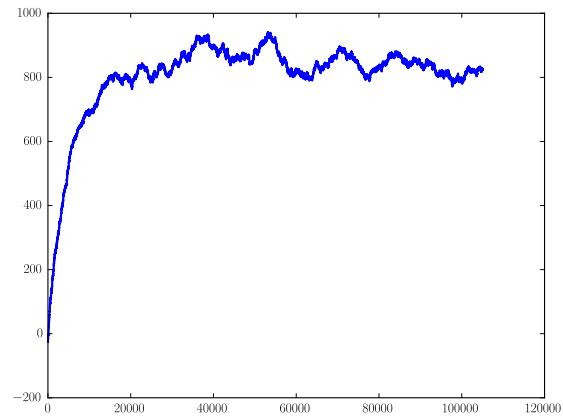
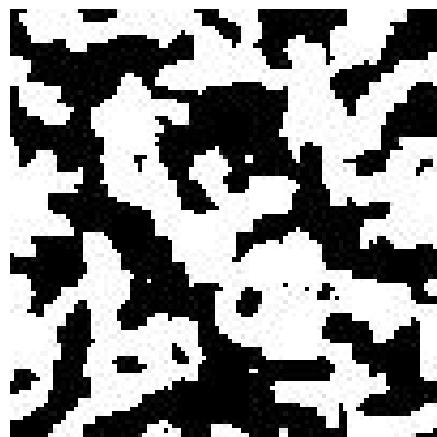
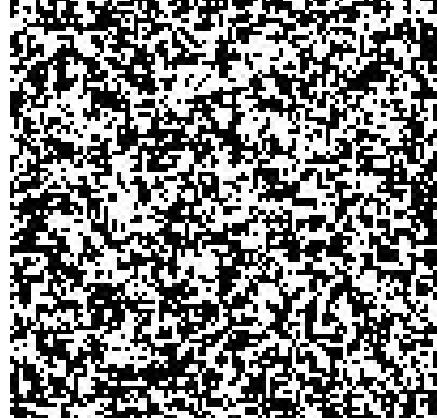
**Problem 4.** Write a function that computes the energy of a proposed spin configuration, given the current spin configuration, its energy, and the proposed spin flip site indices.

**Problem 5.** Write a function that accepts or rejects a proposed spin configuration, given the current configuration. It should accept the current energy, the proposed energy, and  $\beta$ , and should return a boolean.

To track the convergence of the Markov chain, we would like to look at the probabilities of each sample at each time. However, this would require us to compute the denominator  $Z_\beta$ , which—as we explained previously—is generally the reason we have to use a Metropolis algorithm to begin with. We can get away with examining only  $-\beta H(\sigma)$ . We should see this value increase as the algorithm proceeds, and it should converge once we are sampling from the correct distribution. Note that we don't expect these values to converge to a specific value, but rather to a restricted range of values.

**Problem 6.** Write a function that initializes a spin configuration for an  $n \times n$  lattice as done previously, and then performs the Metropolis algorithm, choosing new spin configurations and accepting or rejecting them. It should burn in first, and then iterate  $n\_samples$  times, keeping every 100<sup>th</sup> sample (this is to prevent memory failure) and all of the above values for  $-\beta H(\sigma)$  (keep the values even for the burn-in period). It should also accept  $\beta$  as an argument, allowing us to effectively adjust the temperature for the model.

**Problem 7.** Test your Metropolis sampler on a  $100 \times 100$  grid, with 200000 iterations, with  $n\_samples$  large enough so that you will keep 50 samples, testing with  $\beta = 1$  and then with  $\beta = 0.2$ . Plot the proportional log probabilities, and also plot a late sample from each test using `matplotlib.pyplot.imshow`. How does the ferromagnetic material behave differently with differing temperatures? Recall that  $\beta$  is an inverse function of temperature. You should see more structure with lower temperature, as illustrated in Figures 23.4b and 23.4d.

(a) Proportional log probs when  $\beta = 1$ .(c) Proportional log probs when  $\beta = 0.2$ .(b) Spin configuration sample when  $\beta = 1$ .(d) Spin configuration sample when  $\beta = 0.2$ .



# 24

# Principal Component Analysis and Latent Semantic Indexing

**Lab Objective:** *Understand the basics of principal component analysis and latent semantic indexing.*

## Principal Component Analysis

Understanding the variance in complex data is one of the first tasks encountered in exploratory data analysis. For an example, consider the scatter plot displaying the sepal and petal lengths of 100 different irises shown in Figure 24.1. There are three distinct types of iris flowers present: *setosa*, *versicolor*, and *virginica*. Considering this data, we might ask how to best distinguish the different types of irises based on their given sepal and petal lengths. We can answer this question by finding the characteristic that causes the greatest variance in the data. (Greater variance implies a greater ability to distinguish between data points. If the variance is very small, the data are clustered tightly together, and it is difficult to distinguish well.)

Upon examination, we see that the petal length ranges between 3 and 7 cm, while the sepal length only ranges between 5 and 8 cm. We might be tempted to say that the most distinguishing aspect of irises is their petal length, but this is only considering the features of the data individually, and not collectively. The two features of the data are clearly correlated, and a more careful consideration would lead us to conclude that the most distinguishing aspect of irises is their overall size. Some irises are much larger than others, while the sepal and petal lengths stay roughly in proportion.

Principal Component Analysis (PCA) is a multivariate statistical tool used to orthogonally change the basis of a set of observations from the basis of original features (which may be correlated) into a basis of uncorrelated (in fact, orthonormal) variables called the *principal components*. It is a direct application of the singular value decomposition (SVD) from linear algebra. More specifically, the first principal component will account for the greatest variance in the set of observations, the second principal component will be orthogonal to the first, accounting for the second greatest variance in the set of observations, etc. The first several principal components capture most of the variance in the observation set, and hence provide a great deal of information about the data. By projecting the observations onto the space spanned by the principal components, we can reduce the dimensionality of the data in a manner that preserves most of the variance.

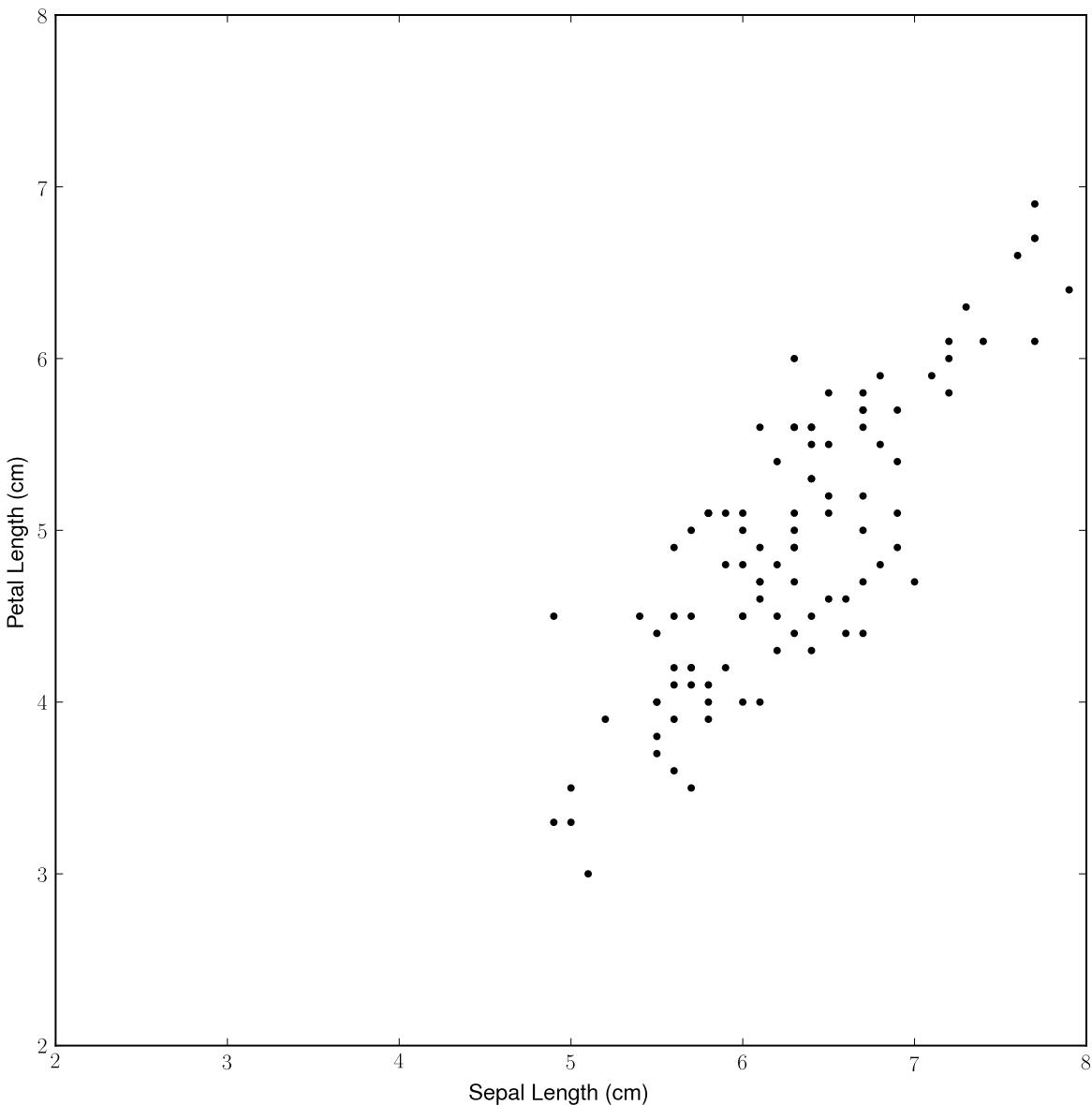


Figure 24.1: Sepal Length vs. Petal Length for 100 iris flowers. Note the strong correlation of these variables.

In our iris example, the two principal components are shown in Figure 24.2. The first principal component, corresponding intuitively to iris size, accounts for 96% of the variance in the data. The second, which accounts for only 4% of the variance, corresponds to the relative sepal and petal length of irises of the same size.

### Computing the Principal Components

We now explore how to use the SVD to compute the principal components of a dataset. Throughout this lab we will use the `sklearn` iris data set, which can be obtained as follows:

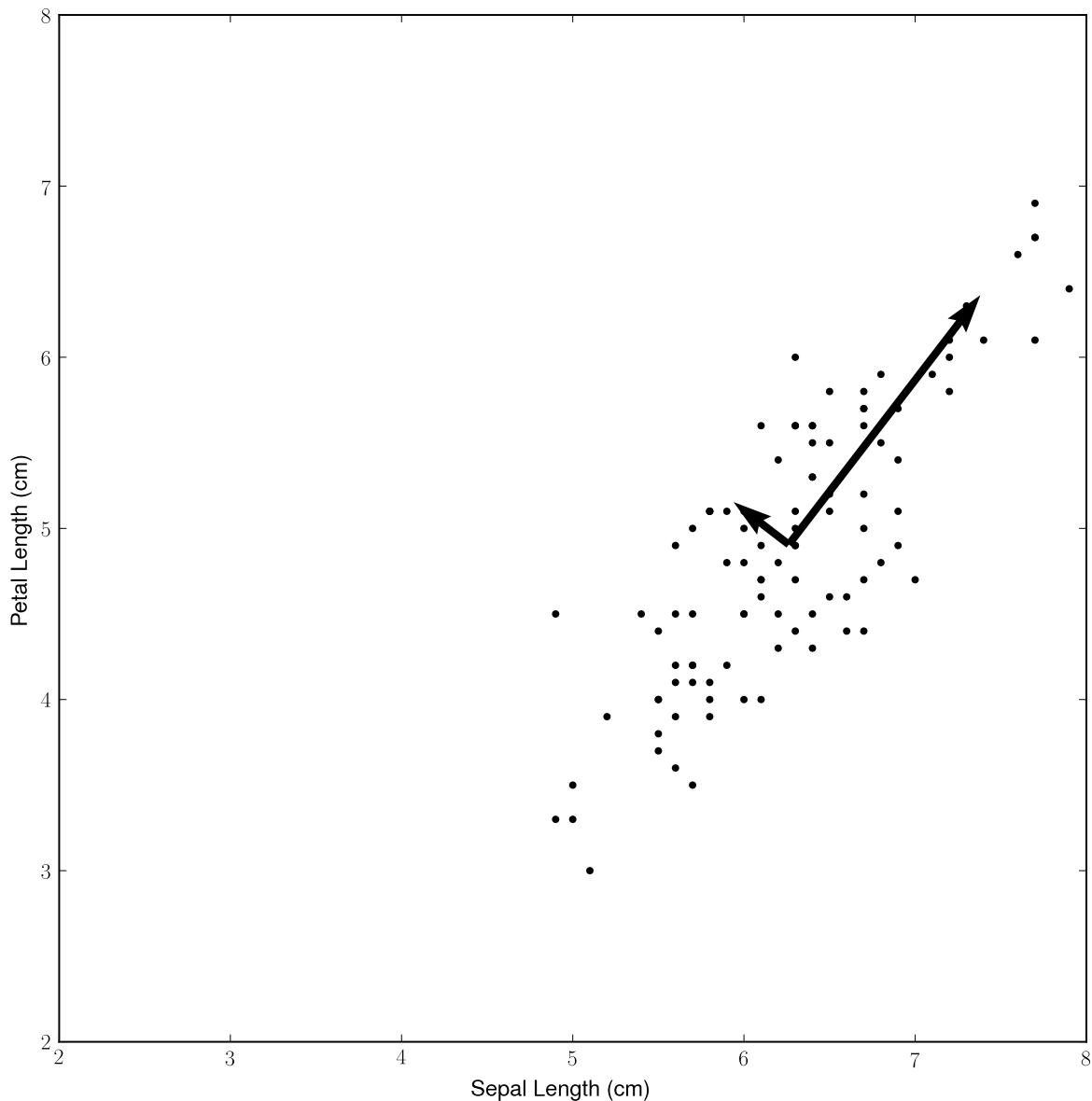


Figure 24.2: The vectors indicate the two principal components, which are weighted by their contribution to the variance.

```
>>> import numpy as np
>>> from scipy import linalg as la
>>> import sklearn.datasets as datasets
>>> iris = datasets.load_iris()
>>> X = iris.data
```

We represent the collection of observations as an  $n \times m$  matrix  $X$ , where each row of  $X$  is an observation, and each column is a specific feature. Let  $k = \min(m, n)$ . We will use this later. In the iris example,  $X$  contains 150 observations, each consisting of 4 features (so  $k = 4$ ), as shown below:

```
>>> X.shape
(150L, 4L)
>>> iris.feature_names
['sepal length (cm)',
 'sepal width (cm)',
 'petal length (cm)',
 'petal width (cm)']
```

The first step in PCA is to pre-process the data. In particular, we first translate the columns of  $X$  to have mean 0. The data may then be optionally scaled to remove discrepancies arising from different units of measure (i.e. centimeters vs meters), and we call the new matrix containing the centered and scaled data  $Y$ . In this lab, we will not have any scaling issues, so we won't address this issue any further. Thus we can pre-process our iris data simply as follows:

```
>>> Y = X - X.mean(axis=0)
```

We next compute the truncated SVD of our centered and scaled data,

$$Y = U\Sigma V^T$$

where  $U$  is  $n \times k$ ,  $\Sigma$  is a  $k \times k$  diagonal matrix containing the singular values of  $Y$  in decreasing order along the diagonal, and  $V$  is  $m \times k$ . The columns of  $V$  are the principal components (which form an orthonormal basis for the space spanned by the observations), and the corresponding singular values provide us information about how much variance is captured in each principal component. More specifically, let  $\sigma_i$  be the  $i$ -th non-zero singular value. Then the value

$$\frac{\sigma_i^2}{\sum_{j=1}^k \sigma_j^2}$$

is the percentage of the variance captured by the  $i$ -th principal component. We compute the truncated SVD of the iris data and show the variance percentages for each component below:

```
>>> U,S,VT = la.svd(Y, full_matrices=False)
>>> S**2/(S**2).sum() # variance percentages
array([ 0.92461621,  0.05301557,  0.01718514,  0.00518309])
```

In general, we are only interested with the first several principal components. But just how many principal components should we keep? There are a number of ways to decide this. One is to only keep the first two principal components, as these enable us to project the data into 2-dimensional space, which is easy to visualize. Another way is to only keep the set of principal components accounting for a certain percentage (say 80%) of the variance. A third method is to examine the *scree plot* of the variance percentages for each principal component, as in Figure 24.3. Upon examination of the iris scree plot, we see that there is a distinct change after the first principal component. This method is referred to as finding the "elbow" of the scree plot, and we keep all the principal components on the left of the elbow. In the case of the iris data, that is simply the first principal component, which accounts for 92% of the variance.

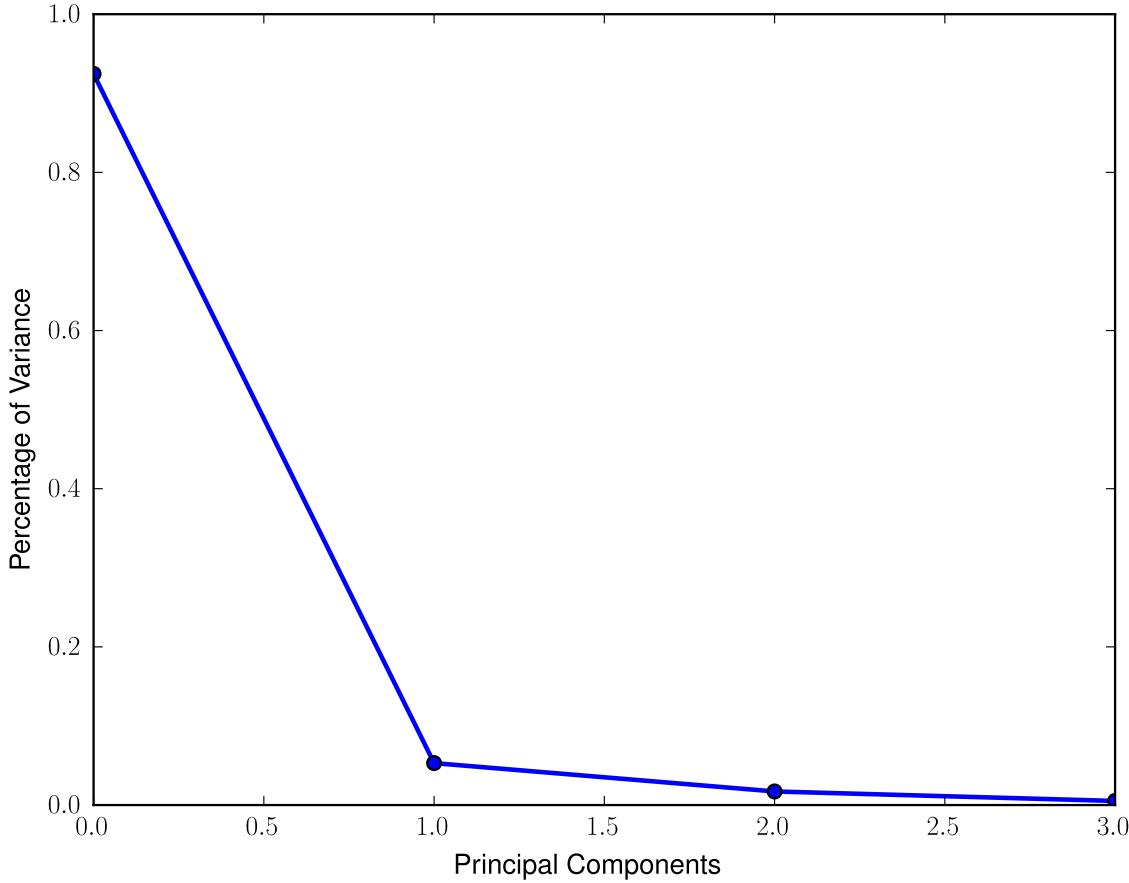


Figure 24.3: Scree plot of the percentage of variance for PCA on the iris dataset.

Once we have decided how many principal components to keep (say the first  $l$ ), we can project the observations from the original feature space onto the principal component space by computing

$$\hat{Y} = U_{:,l} \Sigma_{:l,:}$$

where  $\Sigma_{:l,:}$  is the first  $l$  rows and columns of  $\Sigma$  and  $U_{:,l}$  is the first  $l$  columns of  $U$ . Using the SVD formula, note that

$$\hat{Y} = Y V_{:,l},$$

where  $V_{:,l}$  is the first  $l$  columns of  $V$ . In this way, we see that the  $i$ -th row of  $\hat{Y}$  is simply the projection of the  $i$ -th observation onto the orthonormal set of the first  $l$  principal components. Under this projection, the data is represented in fewer dimensions, and in such a way that accentuates the variance (which can help with finding patterns within the data).

In Figure 24.4 we display the transformed iris data set, plotting the first principal component against the second. This reduction helps us to see the distinctions between the three different species, using only two dimensions instead of the full four dimensions of the feature space.

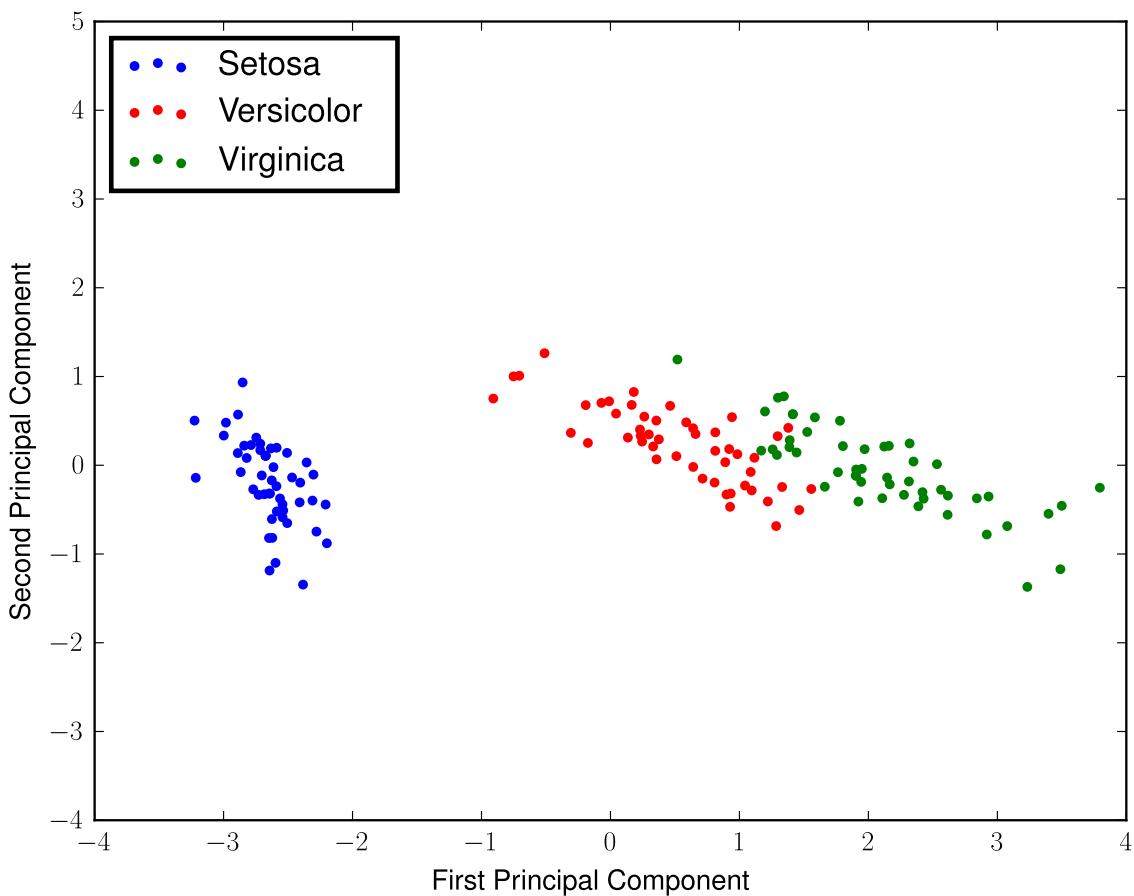


Figure 24.4: Plot of the transformed iris data, keeping only the first two principal components.

**Problem 1.** Recreate the plot shown in Figure 24.4 by performing PCA on the iris dataset, keeping the first two principal components.

*Note:* If `Yhat` is your  $150 \times 2$  array of transformed observations, you can access the rows corresponding to the setosa flowers as follows:

```
>>> Yhat[iris.target==0]
```

To get the rows corresponding to versicolor and virginica specimens, simply replace the 0 with 1 and 2, respectively.

## Latent Semantic Indexing

*Latent Semantic Indexing* (LSI) is an application of PCA which applies the ideas we have discussed to the realm of natural language processing. In particular, LSI employs the SVD to reduce the dimensionality of a large corpus of text documents in order to enable us to evaluate the similarity between two documents. Many information-retrieval systems used in government and in industry are based on LSI.

To motivate the problem, suppose we have a large collection of documents dealing with various statistical and mathematical topics. How can we find an article about PCA? We might consider simply choosing the article which contains the acronym *PCA* the greatest number of times, but this is a crude method. A better way is to use a form of PCA on the collection of documents.

In order to do so, we need to represent the documents as numerical vectors. A standard way of doing this is to define an ordered set of words occurring in the collection of documents (called the *vocabulary*), and then to represent each document as a vector of word counts from the vocabulary. More formally, let our vocabulary be  $V = \{w_1, w_2, \dots, w_m\}$ . Then a document is a vector  $x = (x_1, x_2, \dots, x_m) \in \mathbb{R}^m$  such that  $x_i$  is the number of occurrences of word  $w_i$  in the document. In this setup, we represent the entire collection of  $m$  documents as an  $n \times m$  matrix  $X$ , where  $m$  is the number of vocabulary words and  $n$  is the number of documents in our collection, each row being a document vector. As expected, we let  $X_{i,j}$  be the number of times term  $j$  occurs in document  $i$ . Note that  $X$  is often a sparse matrix, as any one document likely doesn't contain most of the vocabulary words. This mode of representation is called the *bag of words* model for documents.

We calculate the SVD of  $X$  without centering or scaling the data so that we may retain the sparsity. We now have  $X = U\Sigma V^T$ . Once we have selected the number of principal components to keep, say  $l$ , we can represent the corpus of documents by the matrix

$$\widehat{X} = U_{:,l}\Sigma_{:,l}V_{:,l} = X V_{:,l}.$$

Note that  $\widehat{X}$  will no longer be a sparse matrix, but it has dimensions  $n \times l$ , which is much smaller than  $n \times m$  when  $l \ll m$ .

Now that we have our documents represented in terms of the first  $l$  principal components, we can find the similarity between two documents. Our measure for similarity is just the cosine of the angle between the vectors; a small angle (and hence large cosine) indicates greater similarity, while a large angle (hence small cosine) indicates greater dissimilarity. Recall that we can use the inner product to find the cosine of the angle between two vectors. Under this metric, the similarity between document  $i$  and document  $j$  (represented by the  $i$ -th and  $j$ -th row of  $\widehat{X}$ , notated  $\widehat{X}_i$  and  $\widehat{X}_j$ , respectively) is just

$$\frac{\langle \widehat{X}_i, \widehat{X}_j \rangle}{\|\widehat{X}_i\| \|\widehat{X}_j\|}.$$

To find the document most similar to document  $i$ , we simply compute

$$\operatorname{argmax}_{j \neq i} \frac{\langle \widehat{X}_i, \widehat{X}_j \rangle}{\|\widehat{X}_i\| \|\widehat{X}_j\|}.$$

We now discuss some practical issues involved in creating the bag of words representation  $X$  from the raw text. Our dataset will consist of the US State of the Union addresses from 1945 through 2013, each contained in a separate text file in the folder **Addresses**. We would like to avoid loading in all of the text into memory at once, and so we will *stream* the documents one at a time.

The first thing we need to establish is the vocabulary set, i.e. the set of unique words that occur throughout the collection of documents. A Python set object automatically preserves the uniqueness of the elements, so we will create a set, and then iteratively read through the documents, adding the unique words of each document to the set. As we read in each document, we will remove punctuation and numerical characters and convert everything to lower case. The following code will accomplish this task:

```
>>> # get list of filepaths to each text file in the folder
>>> import string
>>> from os import.listdir
>>> path_to_addresses = "./Addresses/"
>>> paths = [path_to_addresses + p for p in os.listdir(path_to_addresses) if p[-4:]==" .txt"]

>>> # helper function to get list of words in a string
>>> def extractWords(text):
>>>     trans = string.maketrans("", "")
>>>     return text.strip().translate(trans, string.punctuation+string.digits).lower().split()

>>> # initialize vocab set, then read each file and add to the vocab set
>>> vocab = set()
>>> for p in paths:
>>>     with open(p, 'r') as f:
>>>         for line in f:
>>>             vocab.update(extractWords(line))
```

We now have a set containing all of the unique words in the corpus. However, many of the most common words do not provide important information. We call these *stop words*. Examples in English include *the*, *a*, *an*, *and*, *I*, *we*, *you*, *it*, *there*, etc; a list of common English stop words is given in `stopwords.txt`. We remove the stop words from our vocabulary set as follows, and then fix an ordering to the vocabulary by creating a dictionary whose key-value pairs are of the form (word, index):

```
>>> # load stopwords
>>> with open("stopwords.txt", 'r') as f:
>>>     stopwords = set([w.strip().lower() for w in f.readlines()])

>>> # remove stopwords from vocabulary, create ordering
>>> vocab = {w:i for i, w in enumerate(vocab.difference(stopwords))}
```

We are now ready to create the word count vectors for each document, and we store these in a sparse matrix  $X$ . It is convenient to use the `Counter` object from the `collections` module, as this object automatically counts the occurrences of each distinct element in a list.

```
>>> from scipy import sparse
>>> from collections import Counter
>>> counts = [] # holds the entries of X
>>> doc_index = [] # holds the row index of X
```

```

>>> word_index = [] # holds the column index of X

>>> # iterate through the documents
>>> for doc, p in enumerate(paths):
>>>     with open(p, 'r') as f:
>>>         # create the word counter
>>>         ctr = Counter()
>>>         for line in f:
>>>             ctr.update(extractWords(line))
>>>         # iterate through the word counter, store counts
>>>         for word, count in ctr.iteritems():
>>>             try: # only look at words in vocab
>>>                 word_index.append(vocab[word])
>>>                 counts.append(count)
>>>                 doc_index.append(doc)
>>>             except KeyError: # if word isn't in vocab, skip it
>>>                 pass

>>> # create sparse matrix holding these word counts
>>> X = sparse.csr_matrix((counts, [doc_index,word_index]), shape=(len(paths),←
    len(vocab)), dtype=np.float)

```

**Problem 2.** Using the techniques of LSI discussed above—applied to the word count matrix  $X$ , and keeping the first 7 principal components—find the most similar and least similar speeches to both Bill Clinton’s 1993 speech and to Richard Nixon’s 1974 speech. Are the results plausible?

*Hint:* Since  $X$  is a sparse matrix, you will need to use the SVD method found in `scipy.sparse.linalg`. This method operates slightly differently than the SVD method found in `scipy.linalg`, so make sure to read the documentation.

The simple bag of words representation is a bit crude, as it fails to consider how some words may be more important than others in determining the similarity of documents. Words appearing in few documents tend to provide more information than words occurring in every document. For example, while the word *war* might not be considered a stop word, it is likely to appear in quite a few addresses, whereas *Afghanistan* will not. Thus two speeches sharing the word *war* ought to be considered more related than two speeches sharing the word *war*. So while  $X_{i,j}$  is a good measure of the importance of term  $j$  in document  $i$ , we also need to consider some kind of global weight for each term  $j$ , indicating how important the term is over the entire collection. There are a number of different weights we could choose; we choose to employ the following approach:

Let  $t_j$  be the total number of times term  $j$  appears in the entire collection of documents. Define

$$p_{i,j} = \frac{X_{i,j}}{t_j}.$$

We then let

$$g_j = 1 + \sum_{i=1}^m \frac{p_{i,j} \log(p_{i,j} + 1)}{\log m},$$

where  $m$  is the number of documents in the collection. We call  $g_j$  the *global weight* of term  $j$ . We replace each term frequency in the matrix  $X$  by weighting it globally. Specifically, we define a matrix  $A$  with entries

$$A_{i,j} = g_j \log(X_{i,j} + 1).$$

We can now perform LSI on the matrix  $A$ , whose entries are both locally and globally weighted.

To calculate the matrix  $A$  in a streaming manner, we must alter our code above somewhat:

```
>>> from itertools import izip
>>> from math import log
>>> t = np.zeros(len(vocab))
>>> counts = []
>>> doc_index = []
>>> word_index = []

>>> # get doc-term counts and global term counts
>>> for doc, path in enumerate(paths):
>>>     with open(path, 'r') as f:
>>>         # create the word counter
>>>         ctr = Counter()
>>>         for line in f:
>>>             words = extractWords(line)
>>>             ctr.update(words)
>>>         # iterate through the word counter, store counts
>>>         for word, count in ctr.iteritems():
>>>             try: # only look at words in vocab
>>>                 word_ind = vocab[word]
>>>                 word_index.append(word_ind)
>>>                 counts.append(count)
>>>                 doc_index.append(doc)
>>>                 t[word_ind] += count
>>>             except KeyError:
>>>                 pass

>>> # get global weights
>>> g = np.ones(len(vocab))
>>> logM = log(len(paths))
>>> for count, word in izip(counts, word_index):
>>>     p = count/float(t[word])
>>>     g[word] += p*log(p+1)/logM

>>> # get globally weighted counts
>>> gwcounts = []
>>> for count, word in izip(counts, word_index):
>>>     gwcounts.append(g[word]*log(count+1))

>>> # create sparse matrix holding these globally weighted word counts
>>> A = sparse.csr_matrix((gwcounts, [doc_index, word_index]), shape=(len(paths) <-
, len(vocab)), dtype=np.float)
```

**Problem 3.** Repeat Problem 2 using the matrix  $A$ . Do your answers seem more reasonable than before?



# 25 Naive Bayes

**Lab Objective:** *Implement Naive Bayes Classification Models.*

## Introduction

Naive Bayes classification methods are a good introduction to machine learning techniques. They are relatively straightforward to understand and to implement, while they are also very effective for certain applications. However, they are somewhat limited by their strong dependence on assumptions of independence.

Recall that “the classification problem” tries to assign the correct label to a given set of features (called a *feature vector*). For example, suppose we wish to give the correct labels (names) to two different pieces of fruit. The given features of the first fruit are that it is red and round, and the features of the second are that the fruit is long and yellow. If we assign to these fruits the names “apple” and “banana” respectively, then these are our labels.

It is common in classification problems to start with a set of correctly-labeled feature vectors called a *training set*. We use the training set to train our algorithm to make predictions. It is also common to have another smaller set of correctly-labeled feature vectors called a *test set*. To verify the effectiveness of our algorithm, we predict the labels of the test set, and then compare the predicted labels to the true labels.

Recall that Bayes rule for random variables gives that

$$P(Y|X) = \frac{P(X|Y)P(Y)}{\int P(X|Y)P(Y)dy}$$

where  $P(Y)$  is our prior distribution and  $P(X|Y)$  is our likelihood function.

Suppose that we have a set of features that we wish to label. Let  $x = (x_1, \dots, x_n)$  be this feature vector and let  $C = \{c_1, \dots, c_k\}$  be our set of possible labels for  $x$ . We may apply Bayes rule to this problem as follows:

$$P(c_i|x) = P(c_i|x_1, \dots, x_n) = \frac{P(x_1, \dots, x_n|c_i)P(c_i)}{P(x_1, \dots, x_n)}$$

If we make no further assumptions, this problem is intractable. To effectively estimate even the simplest case where each feature is a boolean value would require us to estimate around  $k2^n$  parameters. The problem gets exponentially worse if we were to consider non-boolean features.

However, if we can make the assumption that the features are conditionally-independent of one another, the problem can be simplified dramatically. If we make this assumption and apply Bayes rule, we have that

$$P(c_i|x_1, \dots, x_n) = \frac{P(x_1|c_i)P(x_2|c_i)\dots P(x_n|c_i)P(c_i)}{P(x_1, \dots, x_n)}.$$

In this case, we only need to estimate  $kn$  parameters. The Naive Bayes classification algorithm chooses the label with the highest probability. Since this is independent of the denominator in Bayes rule, we can simplify the problem further. Given an unlabeled feature vector  $x = (x_1, \dots, x_n)$ , we assign the label

$$c = \operatorname{argmax}_{i \in \{1, \dots, k\}} P(c_i) \prod_{j=1}^n P(x_j|c_i).$$

To assign a label to a set of features, we calculate each  $P(x_j|c_i)$  and choose a prior  $P(c_i)$ . We then choose the argmax as above. The calculation of conditional probabilities and our choice of a prior will depend on the type of problem that we are solving.

## Gaussian Classifiers

Gaussian classifiers are commonly used when dealing with continuous data. We assume that each feature is normally distributed and conditionally-independent of all other features. We may then calculate  $\mu_{j,i}$  and  $\sigma_{j,i}^2$  (the mean and variance) corresponding to each feature of each class. Then  $P(x_j|c_i)$  can be calculated using the gaussian pdf

$$P(x_j|c_i) = \frac{1}{\sqrt{2\pi\sigma_{j,i}^2}} \exp -\frac{(x_j - \mu_{j,i})^2}{2\sigma_{j,i}^2}.$$

For example, suppose we have a training set with labels indicating the sex of a person (1 for female, 2 for male), and features consisting of hair length and height (features 1 and 2, respectively). Further suppose that the mean hair length of the women in the training set is 15 centimeters with standard deviation 1.5 cm, and the mean height is 1.25 meters with standard deviation 6 cm. Similarly, suppose that the mean hair length of the men in the training set is 5 centimeters with standard deviation 2.5 cm, and the mean height is 1.75 meters with standard deviation 7 cm. Under this setup, we have

$$\begin{array}{ll} \mu_{1,1} = 15, & \sigma_{1,1} = 1.5, \\ \mu_{2,1} = 1.25, & \sigma_{2,1} = .06, \\ \mu_{1,2} = 5, & \sigma_{1,2} = 2.5, \\ \mu_{2,2} = 1.75, & \sigma_{2,2} = .07. \end{array}$$

If we wish to classify a person with hair length 17 centimeters who is 1.4 meters tall, we calculate the probability of each label using the parameters given above and a uniform prior ( $P(F) = P(M) = \frac{1}{2}$ ) as follows:

$$\begin{aligned} P(F | 17, 1.4) &= P(F) \left( \frac{1}{\sqrt{2\pi\sigma_{1,1}^2}} \exp -\frac{(17 - \mu_{1,1})^2}{2\sigma_{1,1}^2} \right) \left( \frac{1}{\sqrt{2\pi\sigma_{2,1}^2}} \exp -\frac{(1.4 - \mu_{2,1})^2}{2\sigma_{2,1}^2} \right) \\ &= .016 \\ P(M | 17, 1.4) &= P(M) \left( \frac{1}{\sqrt{2\pi\sigma_{1,2}^2}} \exp -\frac{(17 - \mu_{1,2})^2}{2\sigma_{1,2}^2} \right) \left( \frac{1}{\sqrt{2\pi\sigma_{2,2}^2}} \exp -\frac{(1.4 - \mu_{2,2})^2}{2\sigma_{2,2}^2} \right) \\ &= 1.7 \times 10^{-11} \end{aligned}$$

The Female label has a greater probability given the feature vector, and so we classify the person as Female.

A nice way to visualize how a classifier works is to plot the decision boundaries for two-dimensional subspaces of the feature vector space. For example, the decision boundaries for a Gaussian Naive Bayes classifier trained on a dataset consisting of the sepal widths and sepal lengths of three different types of flowers are shown in Figure 25.1.

## Working in Log Space

In the example presented above, notice that the value of  $P(M | 17, 1.4)$  is very small. This is often the case in classification problems; certain classes may be very unlikely, and so calculating these probabilities may lead to numerical underflow. This is especially pronounced in the Naive Bayes model, which involves taking the product of several numbers between 0 and 1. A useful technique to avoid underflow is to perform all of the computations in logarithmic space, where the products all become sums. When we do so, the Naive Bayes label assignment is

$$c = \underset{i \in \{1, \dots, k\}}{\operatorname{argmax}} \log P(c_i) + \sum_{j=1}^n \log P(x_j | c_i).$$

Since the logarithm is a monotone increasing function, the argmax is the same whether in log space or in the original formulation.

**Problem 1.** Download the `seeds_dataset.txt` file. This file contains 7 features describing 3 species of wheat.

1. Area
2. Perimeter
3. Compactness
4. Length
5. Width
6. Asymmetry Coefficient

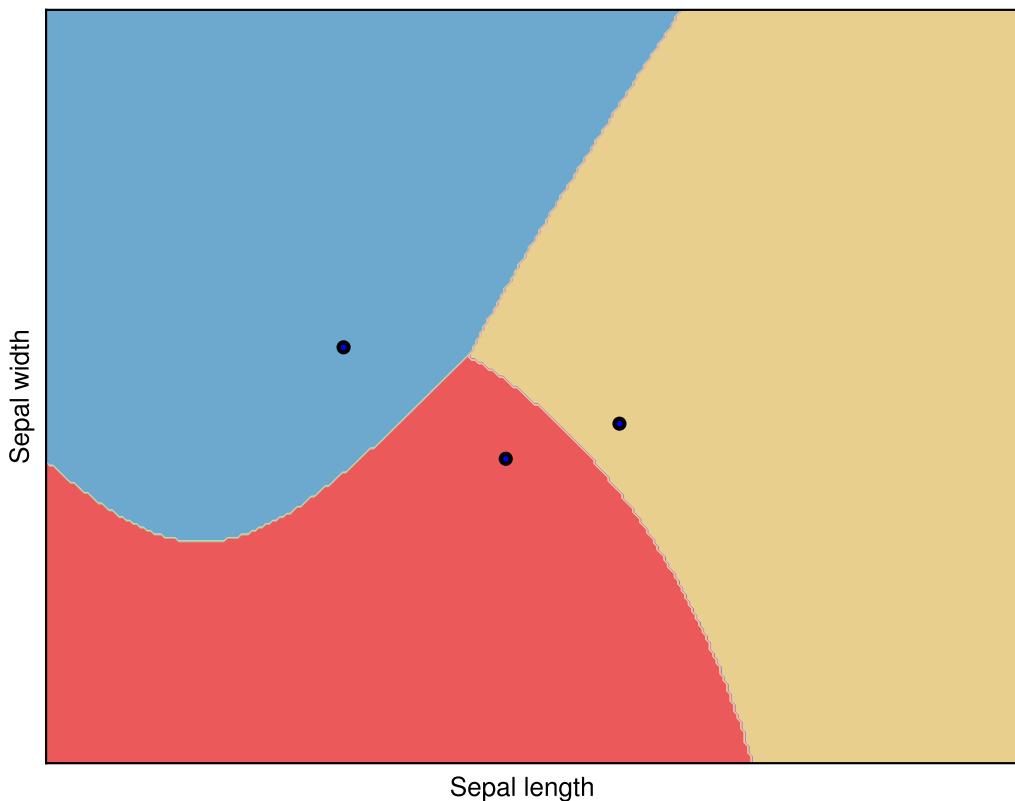


Figure 25.1: Decision boundaries for a Gaussian Naive Bayes classifier on the iris flower dataset, together with the means for each flower. Each point in the plane represents a 2-dimensional feature vector, and the color associated with each point indicates which class label was assigned to that feature vector.

7. Groove length

The species of wheat are

1. Kama
2. Rosa
3. Canadian

The measurements of the kernels are real valued, making this a good example on which to try our Gaussian classifier. Make a function that classifies a subset of the data in this file and returns the accuracy of your calculations by proceeding in the following manner:

1. Randomly select a test set consisting of 40 vectors. Make the remaining vectors into your training set.
2. Calculate the mean and variance for each feature of each label using the training set.

3. Using a uniform prior, predict the labels of your test set.
4. Compare your predictions to the correct labels of the test set. In particular, report the accuracy of the prediction, which is the number of correctly predicted test samples divided by the total number of test samples.

We may also use SciPy's `sklearn` library to implement a Gaussian classifier. After importing the library, we can create a new classifier and train it with just a few lines of code. To create a Gaussian classifier, use the following code.

```
from sklearn.naive_bayes import GaussianNB
nb_classifier = GaussianNB()
```

Given a training set, we can also quickly train the classifier to a certain problem. This requires the training set and labels as two arguments.

```
nb_classifier.fit(training_set, labels)
```

Once the classifier has been trained, we can predict the labels for a test set.

```
pred_labels = nb_classifier.predict(test_set)
```

The `predict` method returns an array of labels for the test set.

**Problem 2.** Repeat the previous problem using `sklearn`'s Naive Bayes classifier. Check that your implementation from the previous problem predicts the same labels as the `sklearn` implementation does.

## Document Classification and Spam Filters

Naive Bayes classifiers are often used in document classification, a major example being spam detection. When it comes to document classification, a common choice for the feature vector is simply a count for the number of times each word in the specified vocabulary occurs in the document. For example, suppose we are trying to classify a document, and the vocabulary of relevant words is the ordered set

(bank, tree, wealth, money, river, water).

Suppose that the document is the sentence

"The woman deposited her money in the bank, and then made her way down to the bank of the river, contemplating her wealth."

Then the feature vector for this document is

(2, 0, 1, 1, 1, 0).

Notice in particular that the  $i$ -th entry of the feature vector indicates the number of occurrences of the  $i$ -th vocabulary word in the document. Such a feature vector is often called a *word-count vector*. Notice that the count vector ignores words in the document that are not part of the vocabulary, and it also disregards the order of the words. This simple representation of text documents is known as the *bag-of-words model* or the *vector space model*. The hypothesis that drives spam filters is that spam messages will use words with different frequencies. For example, a spam message will often have a sales pitch, so the word “buy” and “cheap” will appear often. On the other hand, legitimate messages will probably use different language. Thus, it is reasonable to use word-count vectors as our feature vectors when attempting to distinguish between spam and legitimate email.

We now introduce formalisms to derive the Naive Bayes model for document classification. Let  $V = (v_1, v_2, \dots, v_n)$  be an ordered list of words, called the vocabulary, and let  $x = (x_1, x_2, \dots, x_n)$  be a word-count vector. Let  $\{c_1, c_2, \dots, c_k\}$  be the set of classification labels. In the case of continuous data and Gaussian Classifiers, each class label was associated with corresponding mean and variance parameters, and these determined the likelihood of the feature vector given the class label. In the case of document classification, each class label  $c_i$  has a corresponding probability vector  $(p_{i,1}, p_{i,2}, \dots, p_{i,n})$  whose entries are nonnegative and sum to 1. This probability vector defines a categorical probability distribution over the vocabulary  $V$ , where  $p_{i,j}$  represents the probability of seeing word  $v_j$  given class label  $c_i$ . With this notation in place, our Naive Bayes model takes the form

$$c = \operatorname{argmax}_{i \in \{1, \dots, k\}} P(c_i) \prod_{j=1}^n p_{i,j}^{x_j}.$$

Given a training set of labeled documents, we can calculate the prior probabilities  $P(c_i)$  and word probabilities  $p_{i,j}$  as follows. Each prior probability  $P(c_i)$  is simply the proportion of training documents that have the label  $c_i$ . Next, let  $\text{count}(c_i, v_j)$  denote the number of occurrences of word  $v_j$  among all training documents that have label  $c_i$ . Then we have

$$p_{i,j} = \frac{\text{count}(c_i, v_j) + 1}{\sum_{j=1}^n (\text{count}(c_i, v_j) + 1)}.$$

(Note that adding 1 to the number of occurrences of each word is known as *add-one smoothing*, and is a common technique to prevent over-fitting.)

Once we have calculated these parameters (the “fitting” stage), we are ready to classify new documents (the “prediction” stage) using the argmax equation given above. Remember to perform calculations in log space to prevent numerical underflow.

**Problem 3.** Implement a Naive Bayes model for document classification. We provide an interface below.

```
class naiveBayes(object):
    """
    This class performs Naive Bayes classification for word-count document←
    features.
    """

    def __init__(self):
        """
        Initialize a Naive Bayes classifier.
        """
        pass
```

```

def fit(self,X,Y):
    """
    Fit the parameters according to the labeled training data (X,Y).

    Parameters
    -----
    X : ndarray of shape (n_samples, n_features)
        Each row is the word-count vector for one of the documents
    Y : ndarray of shape (n_samples,)
        Gives the class label for each instance of training data. ←
        Assume class labels are in {0,1,...,k-1} where k is the ←
        number of classes.
    """
    # get prior class probabilities P(c_i)
    # (you may wish to store these as a length k vector as a class ←
    # attribute)

    # get (smoothed) word-class probabilities
    # (you may wish to store these in a (k, n_features) matrix as a ←
    # class attribute)

    pass

def predict(self, X):
    """
    Predict the class labels of a set of test data.

    Parameters
    -----
    X : ndarray of shape (n_samples, n_features)
        The test data

    Returns
    -----
    Y : ndarray of shape (n_samples,)
        Gives the classification of each row in X
    """
    pass

```

**Problem 4.** In this problem, you will train a Naive Bayes classifier using a corpus of emails extracted from the Enron dataset.

Load in the data from `SpamFeatures.txt`. This is a text file containing a whitespace-delimited numerical array with several thousand columns and several thousand rows, each row representing an email as a count vector. Also load in the data from `SpamLabels.txt`, which is a text file containing a 1 (for legitimate email) or 0 (for spam email) on each line, in correspondence with the rows of the count vector array. Using your document classification implementation, do the following:

1. Randomly create a test set from the data (500 documents), leaving the remaining documents as the training set.
2. Create a Naive Bayes classifier and fit it using the training set.
3. Predict the labels of the test set and compare them to the true labels (by reporting the classification accuracy).

Next, perform the same task using `sklearn`'s implementation and the same training and testing sets:

```
>>> # assume train_vectors, train_labels, and test_vectors are defined
>>> from sklearn.naive_bayes import MultinomialNB
>>> mnb = MultinomialNB()
>>> mnb.fit(train_vectors, train_labels)
>>> predicted = mnb.predict(test_vectors)
```

Again report the accuracy of the predicted labels. The result should be on par with those produced by your own implementation.

# 26

# Logistic Regression

**Lab Objective:** *Understand the basics of Logistic Regression, and apply to the Titanic problem.*

## Binary Logistic Regression

A *Logistic Regression Model* is a probability model that can be used to predict outcomes for a set of data. Usually “Logistic Regression” refers to what is more appropriately called *binary logistic regression*. This is a model which can assign data points to one of two sets, and is used in many different fields. One common medical example is predicting whether or not a patient has a particular disease. Based upon several factors, which may be both continuous (age, height, weight) and categorical (gender, race), we can quantify the probability of infection. This probability is computed by way of the *logistic function*, which, given the contributing factors, will return a probability value between 0 and 1. This probability will then be used to assign a label to our input data ('infected' or 'not infected', for example) by using some cut-off value, and will depend on the need for accuracy in the specific application. Success corresponds to a label of 1, and failure to a label of 0.

The logistic function takes in as input any real number and returns a value between 0 and 1, and is defined explicitly as

$$\phi(t) = \frac{1}{1 + e^{-s}}, \quad (26.1)$$

where  $s$  is some combination of the input variables  $x_1, \dots, x_n$ . The graph of this function can be seen in Figure ???.  $\phi(x)$  can then be interpreted as the probability of success. In some cases it is not possible to find a closed-form expression for the correct combination of the input variables. However, in many cases we can achieve reasonable accuracy by using a linear combination of  $x_i, \dots, x_n$ , i.e.

$$s = c_0 + c_1 x_1 + \dots + c_n x_n.$$

This can be written more compactly as

$$s = \mathbf{c}^T \mathbf{x},$$

where  $\mathbf{x} = (1, x_1, \dots, x_n)^T$  and  $\mathbf{c} \in \mathbb{R}^{n+1}$ .

Given a training set of labeled data points, a Logistic Regression Model will find the optimal  $\mathbf{c}$  for the data, and can then be used to predict labels for further data points.

## The Titanic Problem

The Titanic dataset is especially useful for Logistic Regression. This dataset is composed of actual data obtained concerning the passengers on the ill-fated Titanic voyage, given in the bulleted list below. Using logistic regression, we can predict whether or not a passenger survived based on this data. We do so by training a model on a portion of the dataset, the training set, and predicting labels for the remaining data, the test set.

Before beginning our classification, however, we will first need to process our data. The Titanic dataset contains much more information than is currently relevant for our purposes. You can obtain this data in Excel Spreadsheet form at [Insert link here]. We recommend using pandas to read in and process the data. The columns are as follows:

- **pclass**: An integer in {1, 2, 3} which describes the class the passenger was in.
- **survived**: The dependent variable. 1 indicates survival, and 0 death.
- **name**: A string containing the passenger's name.
- **sex**: A string, either 'male' or 'female'.
- **age**: Either an integer or a float.
- **sibsp**: An integer giving the number of siblings and/or spouse who embarked with the passenger.
- **parch**: An integer giving the number of parents and/or children who embarked with the passenger.
- **ticket**: A string containing the transaction code for the ticket(s) purchased.
- **fare**: A float giving the cost of the ticket purchased.
- **cabin**: A string giving the assigned sleeping cabin for the passenger (note that the majority of this column is blank).
- **embarked**: A string in {S, C, Q} corresponding to the location of the passenger's embarkment, Southampton (UK), Cherbourg (France), or Queenstown (Ireland), respectively.
- **boat**: An int or string for those who survived giving which life boat they rode in.
- **body**: An int giving the number of body for those who died who were found and identified.
- **home.dest**: A string giving the home of or location to which the passenger was headed.

**Problem 1.** Create a function called `initialize` which will process the Titanic data set into useable format by doing the following:

1. Choose the coulmns that you believe will be relevant in predicting the survival of the passengers, and drop the other columns. You may not use `boat` or `body`, as these are dependent on whether or not the passenger survived. Be sure to include `survived`, which will be separated later as the independent variable, as well as `sex` and `pclass`.
2. Since `sex` is really a binary variable, make it one explicitly by changing "female" and "male" to be binary values.

3. Drop the rows that contain missing values. Make sure you have a significant number of rows left. If you have too few, you may need to choose fewer columns to keep before deleting the incomplete rows.
4. Because the `pclass` column is an integer in  $\{1, 2, 3\}$ , it will be treated as a ranked variable instead of simply a categorical variable. It may be useful to rank this variable, or it may mess up our classification. Include a keyword argument `pclass_change` with default `True`. If it is set to `True`, eliminate this ranking by dividing `pclass` into two binary columns. Make one column a boolean for being 1<sup>st</sup> class and the other a boolean for being 2<sup>nd</sup> class. (This means that a value of 1 would correspond to [1, 0], 2 to [0, 1], and 3 to [0, 0].)
5. Split the remaining rows into a training and a test set using a 60/40 split. Be sure and pick random rows for each group and not rows in any particular order.

Have your function return the training set and the test set, in that order.

## Model Evaluation

Now that we have our training set, we can train a model, which can be used to obtain the probability of success for each data point. The label chosen depends heavily on the probability cut-off value mentioned previously, which we represent as  $\tau$ . In the simplest manner, we can simply pick a value of  $\tau = 0.5$ , which will then assign the label with the highest likelihood. However, it is often beneficial to choose a different value of  $\tau$ . In regards to the probability of infection for serious diseases, it might be best to give a patient medicine if they have even a 10 percent chance of infection. So how can we find the “best” cut-off value?

In order to determine this, we need to discuss how to measure the accuracy of the labels predicted. Say that we have picked a cut-off value  $\tau$ , and have assigned labels to the test set. Using the predicted labels and the actual labels, we can obtain four important values: the number of *true positives*, *false positives*, *true negatives*, and *false negatives*, which are abbreviated TP, FP, TN, and FN, respectively. These values are integers which together sum to the number of labels predicted. You can see the definition of these in Table ??? (until the figure is up, true positives are the points with predicted label 1 and true label 1, false positives have predicted label 1 true label 0, true negatives predicted label 0 true label 0, and false negatives true label 1 predicted label 0). We can now use these to report our accuracy in various metrics:

- *Prediction accuracy* is defined as  $\frac{TP+TN}{TP+FN+FP+TN}$ , and is the percentage of correctly predicted cases.
- *Sensitivity*, also known as the *true positive rate* or *TPR*, is given by the fraction of correctly predicted cases where the actual outcome is 1,  $\frac{TP}{TP+FN}$ .
- *Specificity*, the *true negative rate*, or *TNR* is the proportion of correctly predicted cases where the true outcome is 0,  $\frac{TN}{FP+TN}$ .
- *False Positive Rate*, or *FPR*, is the proportion of incorrectly predicted cases where the true outcome is 0, and is given by  $\frac{FP}{TP+TN}$ .

All of these depend strongly on the value chosen for  $\tau$ .

A *roc curve* is useful in measuring the accuracy of a model. To make a roc curve, we pick many values for  $\tau$ , and obtain the False Positive Rate and the True Positive Rate for each. Then we plot the data points  $(FPR_\tau, TPR_\tau)$  for each value of  $\tau$  chosen and connect them into a curve. A completely random label assignment would result in a nearly-linear roc curve, while a more accurate assignment would result in a more steeply-rising curve (see Figure ???). A good choice for  $\tau$  is the one that intersects the family of lines  $y = x + b$  at only one point, which intuitively is the point closest to the vertex  $(0, 1)$ . Mathematically, this is given by

$$\arg \max_{\tau} (TPR_\tau - FPR_\tau). \quad (26.2)$$

**Problem 2.** Use the function declaration below to find the best value for  $\tau$ . You should use evenly spaced values from 0 to 1, exclusive.

```
def best_tau(predicted_labels, true_labels, n_tau=100, plot=True):
    """
    Parameters
    -----
    predicted_labels : ndarray of shape (n,)
        The predicted labels for the data
    true_labels : ndarray of shape (n,)
        The actual labels for the data
    n_tau : int
        The number of values to try for tau
    plot : boolean
        Whether or not to plot the roc curve

    Returns
    -----
    best_tau : float
        The optimal value for tau for the data.
    """
    pass
```

Now that we have a good value for  $\tau$ , we can quantify the accuracy of a model. We will do so for a few different types of models for the Titanic data you initialized previously. For the first two, we will use the logistic classifier found in `sklearn.linear_model.LogisticRegression`. The first model we use will be our “Unchanged Logistic Classifier”, which will use our Titanic data with `pclass` unchanged. The second model is the “Changed Logistic Classifier”, which use the data with `pclass` changed.

When using this package to create a classifier, we need to input a keyword argument `C`. This value represents the inverse of the regularization strength. Different values will yield different results. A better value for `C` will yield a more steeply-rising roc curve. The model can find the coefficients (the vector `c`), along with the probabilities of failure and success for each label. With these in hand, we can use the function `sklearn.metrics.roc_curve` to obtain the False Positive Rates, the True Positive Rates, and the optimal value for  $\tau$ . You will need to pass in the test data, the probability of success, and the keyword argument `pos_label = 1`. The accuracy of the model with input value `C` can then be obtained using `sklearn.metrics.auc`, which will give the area under the curve. The larger the area, the more steeply-rising curve we have, and the better the model. Note that we create a single roc curve using one value for `C` and multiple values for  $\tau$ .

**Problem 3.** Use the following function declaration to return the auc score for the two Logistic Regression models described.

```
def auc_scores(unchanged_logreg, changed_logreg):
    """
    Parameters
    -----
    unchanged_logreg : float in (0,1)
        The value to use for C in the unchanged model
    changed_logreg : float in (0,1)
        The value to use for C in the changed model

    Returns
    -----
    unchanged_auc : float
        The auc for the unchanged model
    changed_auc : float
        The auc for the changed model
    """
    pass
```

We can test a Naive Bayes model against our Logistic Regression model for both the unchanged and changed models to see the comparative accuracy. Use the model `MultinomialNB`, found in `sklearn.naive_bayes`. You can use it in the same manner as `LogisticRegression`, except instead of passing in the keyword argument `C`, you will pass in a keyword argument `alpha` corresponding to a smoothing parameter.

**Problem 4.** Add input variables `unchanged_bayes` and `changed_bayes` to your function from the previous problem to obtain the auc for each of these models. Your function should return all four areas, unchanged logistic regression, changed logistic regression, unchanged Bayes, and changed Bayes, in that order.

Different values for `C` and `alpha` will yield different results. We seek to find those that will maximize the area under the curve. One way to do so is to pick a number of evenly-spaced points between 0 and 1, exclusive, and try each one in turn, keeping the value that yields the greatest accuracy.

**Problem 5.** Use the function declaration below to find the optimal values for `C` and `alpha` as described.

```
def find_best_parameters(choices):
    """
    Parameters
    -----
    choices : int
        The number of values to try for C and alpha

    Returns
    -----
    best : list of length 4
        The best values for C for the unchanged and changed logistic
        regression models, and the best values for alpha for the
        unchanged and changed Naive Bayes models, respectively.
    """
    pass
```

Now that we have found the optimal inputs for these functions, we can test them against one another.

**Problem 6.** Create a function called `results` which will graph of the roc curves for each of the methods, and will print out the names of the models with their corresponding areas, in numerically descending order.

# 27

# Classification Trees

**Lab Objective:** *Understand how to build a classification tree and use it to predict survival of Titanic passengers.*

Classification trees are a class of decision trees, and are used in a wide variety of settings where labeled training data is available, and where the desired outcome is a model which is able to accurately assign labels to unlabeled data. We assume that each sample  $d$  has  $P$  attributes, which can be real-valued or categorical, and that each sample belongs to some class  $k$ , where there are  $K$  classes. The tree is composed of many *nodes*, which represent a decision point (i.e. a question is asked about the sample which has a boolean response). If the response is `True`, then the sample is “pushed” down the tree to the left child node. If the response is `False`, then the sample is “pushed” down the tree to the right child node. A *leaf* node is a node that has no child node, i.e. it is the end of the line and there is not a question asked. Each leaf has a classification assigned to it, and an unlabeled sample is labeled with that classification upon arrival at the leaf node.

How do we train a classification tree? We start with a labeled data set  $D$  and choose the best attribute  $p$  and value  $x$  by which to *split* the data. We have now partitioned  $D$  into two sets, which we may then split as well. We continue in this manner until some stopping criterion is met (often a maximum depth of the tree). To formalize this, we need several definitions.

**Definition 27.1.** Let  $D$  be a data set with  $K$  different classes. Let  $N_k$  be the number of samples labeled class  $k$  for each  $1 \leq k \leq K$ , and let  $f_k = \frac{N_k}{N}$  where  $N$  is the total number of samples in  $D$ . We define the Gini impurity to be

$$G(D) = 1 - \sum_{k=1}^K f_k^2.$$

**Problem 1.** Write a function that accepts a list of class assignments and a list of all the  $K$  possible classes, and computes the Gini impurity.

**Definition 27.2.** We define the split  $s_D(p, x)$  of the data set  $D$  on attribute  $p$  using value  $x$ , to be a partition  $D_1, D_2$  such that

1.  $d_p \leq x$  for all  $d \in D_1$  and  $d_p > x$  for all  $d \in D_2$ , where  $d_p$  is the value of attribute  $p$  in  $d$ , assuming real values; or
2.  $d_p = x$  for all  $d \in D_1$  and  $d_p \neq x$  for all  $d \in D_2$ , where  $d_p$  is the value of attribute  $p$  in  $d$ , assuming categorical values.

**Problem 2.** Write a function that computes the split of a data set for a given variable  $p$  and given value  $x$ . It should return the partitioned data set, as well as the partitioned class labels.

**Definition 27.3.** Let  $s_D(p, x) = D_1, D_2$  be a split. We define the information gain of this split to be

$$I(s_D(p, x)) = G(D) - \sum_{i=1}^2 \frac{|D_i|}{|D|} \cdot G(D_i)$$

**Problem 3.** Write a function that computes the information gain for the split of a data set for a given variable  $p$ , value  $x$ .

We define the optimal split of a data set to be

$$s_D^* = s_D(p^*, x^*),$$

where

$$p^*, x^* = \operatorname{argmax}_{p,x} I(s_D(p, x)).$$

From this partition, we create two child nodes, assigning the left child node data set  $D_1$ , and right child node data set  $D_2$ .

**Problem 4.** Write a function that computes the optimal split of a data set. You may need to separate this into two tasks: finding the optimal split for each attribute  $p$ , and then choosing the optimal split over all the attributes.

Let's put all of this together to create the full classification tree.

**Problem 5.** Write a class called `Node` that creates and trains a classification tree. It should accept a training data set  $D$ , class labels  $y$ , current depth (which when initialized should be 1), some maximum depth which is greater than 1, and some tolerance for the Gini impurity (say 0.2). Use recursion to build the tree, i.e. after determining the optimal split, create two new nodes (`leftchild` and `rightchild`), with incremented depth. If the depth equals the maximum depth or the Gini impurity for a node is less than the tolerance, assign the majority label to the node and do not split further.

**Problem 6.** Write a method for the class `Node` that prints out the tree structure. For each node it should show which attribute  $p$  and value  $x$  provide the optimal split, and for the leaf nodes, it should show the assigned label. You may use your own creativity for how to display this.

**Problem 7.** Write a method for the class `Node` that assigns the class label for a new sample. You will probably have to make this method recursive also.

We would like to test our classifier on a real data set. Provided for you is a data set on about 1000 passengers aboard the Titanic. We would like to predict their survival or death depending on several attributes: class ( $1^{st}$ ,  $2^{nd}$ , or  $3^{rd}$ ), gender (male or female), and age.

**Problem 8.** Using the Titanic data set, train a classification tree with a maximum depth of 10 nodes and Gini impurity tolerance .1, and predict labels for the test set. What is your misclassification rate? Print out the tree structure. Is it what you expected? Was there any optimal split which surprised you?

The free parameters which we can vary are the maximum depth and the Gini impurity tolerance. Higher values for the maximum depth creates more refined, specific trees, as does a smaller Gini impurity tolerance. In this case, we are making the classifier more *complex*. As such, it will perform better on the data on which it is trained (it has learned the training data well), but perform worse on new, test data (it is not very generalizable). Keeping the Gini impurity tolerance at 0.1 and increasing the maximum depth yields the following interesting misclassification curves. What is the take away message?

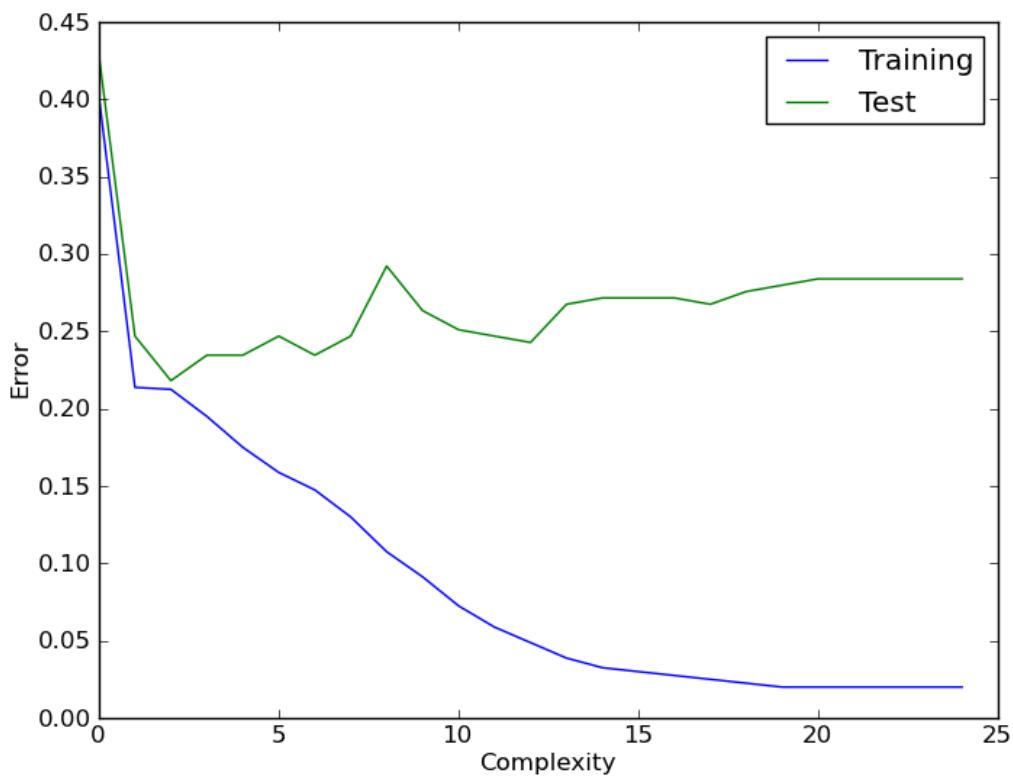


Figure 27.1: Misclassification rate on training data and test data with increasing complexity.

# 28

## Random Forests

**Lab Objective:** *Understand how to build a random forest and use it to predict survival of Titanic passengers.*

A *random forest* is just what it sounds like—a collection of trees. Each tree is trained randomly, meaning that at each node, only a small, random subset of the attributes is available by which to determine the next split. Each trained tree in the forest casts a vote for the labeling of a new sample, and the sample is labeled according to the majority vote of the trees.

Your approach to the classification tree may have been sloppy, depending on how careful you were about odd cases (say trying to split a data set on gender when each sample is male). It doesn't affect us much when all the attributes are available on which to split, unless we grow the tree too deeply. However, with the random forests these odd cases crop up more frequently, as we only have a small subset of attributes to choose from. We need to be more careful then, and keep track of which attributes are still available to split on and only consider these.

**Problem 1.** Modify your code for classification trees so that we keep track of which attributes are available to split on at each node. We can only split on an attribute if it assumes two or more distinct values present in the data set. For example, in the Titanic data set, once we have split on gender, we can never split on it again in any descendant node, since one child data set will only have males and the other will only have females.

We must next add the randomness to our trees.

**Problem 2.** Modify your code for classification trees so that each tree is trained *randomly*, i.e. when determining the optimal split, randomly select a small subset of the available variables, and use them to split on. You should be able to specify the size of the subset. If the number of available variables is smaller than the size of the random subset, then terminate the node (make it a leaf node).

We can now train the whole forest.

**Problem 3.** Make a class *Forest* which trains a collection of random trees. Use the following implementation:

```
class Forest(data, targets, Gini, max_depth, num_trees, num_vars):
    """
    Train a collection of random trees.

    Parameters
    -----
    data : ndarray of shape (n,k)
        Each row is an observation.
    targets : ndarray of shape (K,)
        The possible labels or classes.
    Gini : float
        The Gini impurity tolerance
    max_depth : int
        The maximum depth for the the trees
    num_trees : int
        The number of trees in the forest.
    num_vars : int
        The number of variables randomly selected at each node.
    """

    pass
```

Note that `num_vars` should be small, i.e.  $\text{num\_vars} \approx \sqrt{P}$  where  $P$  is the total number of attributes in the data set. The number of trees in the forest should be somewhat large, say greater than 100.

**Problem 4.** Write a method that assigns a label to a new sample, by considering the majority vote of the trees.

Let us reexamine the Titanic data set and see if we get any significant improvement.

**Problem 5.** Train a random forest on the Titanic data set, and for your inputs use `num_vars` = 2 and 100 trees. Let the Gini impurity tolerance be 0.1 and the maximum depth be 10. What is your misclassification rate? Was there any significant improvement?

# 29

## K-Nearest Neighbors and Support Vector Machines

**Lab Objective:** *Implement the k-Nearest Neighbor (KNN) and binary Support Vector Machine (SVM) classifiers.*

For numerical data, one of the most simple classification methods is the k-nearest neighbor (KNN) classifier, which labels a new sample according to the majority vote of the nearest  $k$  training samples. As  $k$  is the only parameter for the model, this choice determines the effectiveness of the classifier. Throughout this lab we will explore how different values of  $k$  affect the accuracy of our classifier.

Suppose we have numerical data  $\mathbf{x}_1, \dots, \mathbf{x}_N$  and associated labels  $y_1, \dots, y_N$ , along with a metric  $d$  on our feature space. We define the  $k$ -neighborhood of a new sample  $\mathbf{x}$  to be

$$n(\mathbf{x}, k) = \{\mathbf{x}_i : d(\mathbf{x}_i, \mathbf{x}) < d(\mathbf{x}_j, \mathbf{x}) \text{ for all but fewer than } k \text{ samples } \mathbf{x}_j\}$$

Thus the  $k$ -neighborhood of a new sample is the set of samples from our training set which are the  $k$  closest samples according to our metric.

**Problem 1.** Write a function that computes the  $k$ -neighborhood of a sample  $\mathbf{x}$  given  $k$  and a training set. Assume the use of the Euclidean metric.

We define the  $k$ -neighborhood votes of a new sample  $\mathbf{x}$  to be

$$v(\mathbf{x}, k) = \{y_i : \mathbf{x}_i \in n(\mathbf{x}, k)\}$$

The label assigned to  $\mathbf{x}$  according to the standard KNN classifier is the mode of the  $k$ -neighborhood votes.

**Problem 2.** Write a function that labels a new sample  $\mathbf{x}$  given  $k$  and a training set. Assume the use of the Euclidean metric.

**Problem 3.** Write a KNN class which accepts initial training data and training labels. It should have a method to classify new samples, given a value of  $k$ . Load the iris dataset from `sklearn.datasets`, and by separating the data into training and testing sets, implement your class. Test your classifier on the test data given different values of  $k$ . What are the misclassification rates?

Different values of  $k$  lead to different results. Essentially, our choice of  $k$  determines how far-reaching we would like the influence of a sample to be. Larger  $k$  means that a sample is influenced by points farther away from it. Smaller  $k$  means that a sample is influenced only by the few points nearest it. In either case, extreme choices of  $k$  (too small or too big) often yield poor results.

Another powerful classifier is the support vector machine (SVM). There are two main ideas in this classifier: maximum-margin hyperplanes and kernel functions. The first is simply the thought that the simplest binary classifier is a separating hyperplane, the best being the hyperplane that is “farthest” from the nearest two points of opposing classes, while perfectly partitioning the training data. There are very few interesting classification problems where this is possible in the standard feature space. However, if we can transform the feature space into a higher-dimensional space, then we might be able to find such a hyperplane. Unfortunately, working in this higher-dimensional space can be quite costly, which is where the kernel functions come into play.

The second big idea is that instead of working directly in the higher-dimensional space, we can choose our transformation in such a way that we can use *kernel* functions for any necessary computations whose domain is the product space of the original feature space with itself. There are many, sometimes exotic, kernel functions to choose from, though in practice only a few forms are used. We let  $\phi(\mathbf{x})$  be the transformation of  $\mathbf{x}$  into the higher-dimensional space, and we let this transformation be determined by some kernel function

$$k(\mathbf{x}_i, \mathbf{x}_j) = \phi(\mathbf{x}_i)^T \phi(\mathbf{x}_j).$$

We assume now that our labels are simply  $\pm 1$ , and we assume  $\phi(\mathbf{x}) \in \mathbb{R}^K$ . We consider the hyperplane defined by  $f(\phi(\mathbf{x})) = \mathbf{w}^T \phi(\mathbf{x}) + b$ , where  $\mathbf{w} \in \mathbb{R}^K$  and  $b \in \mathbb{R}$ . We wish to find  $\mathbf{w}$  and  $b$  such that  $f(\phi(\mathbf{x}_i)) > 0$  if  $y_i = 1$  and  $f(\phi(\mathbf{x}_i)) < 0$  if  $y_i = -1$ . Thus we need  $\mathbf{w}$  and  $b$  to satisfy  $y_i (\mathbf{w}^T \phi(\mathbf{x}) + b) > 0$  for all  $i = 1, \dots, N$ . Additionally, we would like the distance between the boundary  $\mathbf{w}^T \phi(\mathbf{x}) + b = 0$  and the nearest points to be maximized. We can determine this distance geometrically to be

$$\frac{2}{\|\mathbf{w}\|},$$

and is called the margin. Thus we would like to solve the following optimization problem:

$$\begin{aligned} &\text{minimize } \|\mathbf{w}\| \\ &\text{subject to } y_i (\mathbf{w}^T \phi(\mathbf{x}) + b) > 0 \quad \text{for } i = 1, \dots, N. \end{aligned}$$

Considering the Lagrangian of this optimization problem, we have the dual formulation of this optimization problem as

$$\begin{aligned} &\text{maximize } \sum_{n=1}^N a_n - \frac{1}{2} \sum_{i=1}^N \sum_{j=1}^N a_i a_j y_i y_j \phi(\mathbf{x}_i)^T \phi(\mathbf{x}_j) \\ &\text{subject to } a_i \geq 0 \quad i = 1, \dots, N \\ &\quad \sum_{i=1}^N a_i y_i = 0. \end{aligned}$$

This is simply a quadratic programming problem, where the objective function is  $\mathbf{a}^T \mathbf{1} - \frac{1}{2} \mathbf{a}^T Q \mathbf{a}$ , where

$$Q_{ij} = y_i y_j \phi(\mathbf{x}_i)^T \phi(\mathbf{x}_j)^T = y_i y_j k(\mathbf{x}_i, \mathbf{x}_j).$$

Quadratic programming problems have nice solutions, so this will be easy to solve. The classifier function  $f(\mathbf{x}) = \sum_{i=1}^N a_i y_i k(\mathbf{x}, \mathbf{x}_i)$  also has a nice closed-form solution.

Given a training set of size `n_samples` and a kernel  $k$ , with data  $X$  and target  $Y$ , we can use `cvxopt` to solve this quadratic programming problem:

```
>>> import cvxopt
>>> import numpy as np
>>> K = np.zeros((n_samples,n_samples))
>>> for i in xrange(n_samples):
>>>     for j in xrange(n_samples):
>>>         K[i,j] = k(X[i,:], X[j,:])
>>> Q = cvxopt.matrix(np.outer(Y, Y) * K)
>>> q = cvxopt.matrix(np.ones(n_samples) * -1)
>>> A = cvxopt.matrix(Y, (1, n_samples))
>>> b = cvxopt.matrix(0.0)
>>> G = cvxopt.matrix(np.diag(np.ones(n_samples) * -1))
>>> h = cvxopt.matrix(np.zeros(n_samples))
>>> solution = cvxopt.solvers.qp(Q, q, G, h, A, b)
>>> a = np.ravel(solution['x'])
```

From this value  $a$ , our kernel  $k$ , a training set  $X$ , and target  $Y$ , we have everything we need to build an SVM classifier. But what should our kernel  $k$  be? There are three common kernels used:

$$\text{Polynomial: } k(\mathbf{x}, \mathbf{y}) = (\mathbf{x}^T \mathbf{y} + a)^d$$

$$\text{Radial Basis Function: } k(\mathbf{x}, \mathbf{y}) = e^{-\gamma \|\mathbf{x} - \mathbf{y}\|^2}$$

$$\text{Sigmoid: } k(\mathbf{x}, \mathbf{y}) = \tanh(\mathbf{x}^T \mathbf{y} + r)$$

**Problem 4.** Write an SVM class. Upon initialization, it should accept a training data set and training target set. It should have a method called `setKernel` which accepts one of the three kernel types, and defines a kernel function for the object. It should also have a method to train the classifier, and another method to predict the class of a new sample. It should predict 1 if  $f(\mathbf{x}) > 0$  and -1 if  $f(\mathbf{x}) < 0$ .

A data set on breast cancer has been provided to you. This is from a breast cancer database from the University of Wisconsin Hospital, Madison, from Dr. W. H. Wolberg. The attributes are (in order): clump thickness, uniformity of cell size, uniformity of cell shape, marginal adhesion, single epithelial cell size, bare nuclei, bland chromatin, normal nucleoli, and mitoses, all on a scale from 1 to 10. The targets are either 1 or -1, signifying malignant or benign, respectively.

**Problem 5.** Load the data set. Separate it into a training set and a test set. Train SVMs on the data, trying each kernel with various parameter values. What are your misclassification rates?

# 30 Crime Mapping

**Lab Objective:** *Use Gaussian Mixture Models and Kernel Density Estimates to map homicides in Baltimore.*

Suppose we are moving to Baltimore, and we need to find a place to live. We would like to be aware of the crime (homicides in particular) in each neighborhood, as this will certainly influence our decision. We could simply look at a map of all homicides in the city over several years, as shown in Figure 30.1.

This is helpful, but we would like to find the largest geographical area of Baltimore from which to choose a home, with only a small fraction (say 5%) of the homicides. This is equivalent to finding the smallest geographical area of Baltimore containing 95% of all homicides. To do this, we'll need to estimate the probability density of homicides in Baltimore.

We will use two approaches to estimating this pdf. The first is via GMMs.

**Problem 1.** Unpickle the file `homicides`, which contains the approximate longitude (first column) and latitude (second column) of a homicide in Baltimore. Train a 3-component GMM on this data set.

We can plot this density over the Baltimore region as in the previous lab, using `matplotlib.pyplot.imshow`.

**Problem 2.** Compute the density at each point on a sufficiently fine grid of the Baltimore region. Save a plot of this pdf.

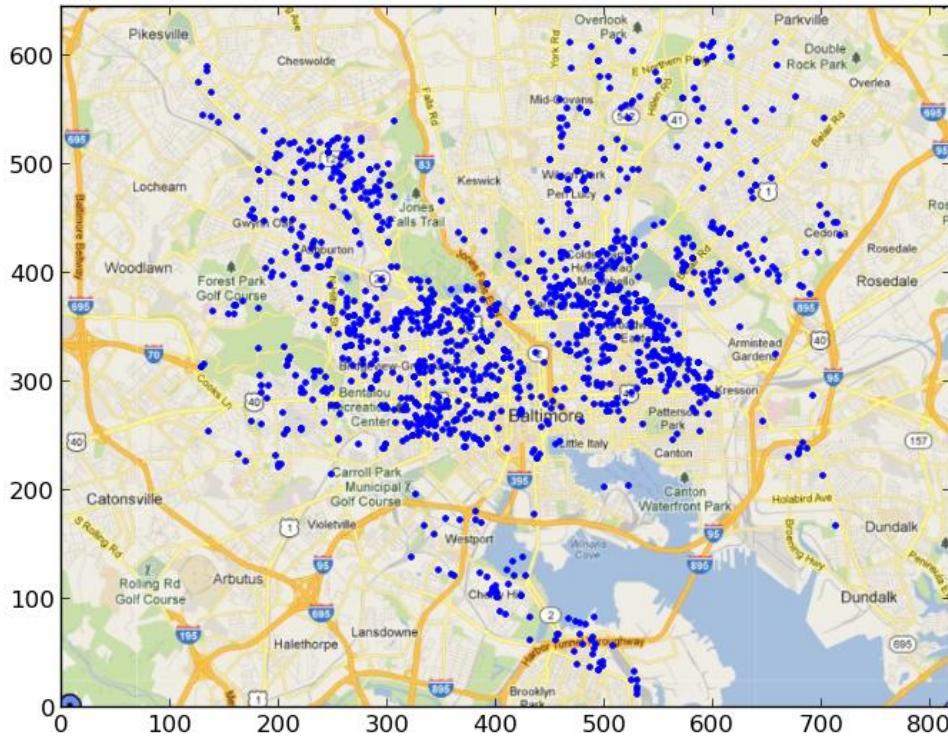


Figure 30.1: Map of homicides in Baltimore over several years.

From this model, we can get a more probabilistic viewpoint of crime in the Baltimore area, as in Figure 30.2.

Alternatively, we can use a Kernel Density Estimate (KDE) of our data. A KDE is a method of smoothing data. We initially examined a scatterplot of the homicide data. Suppose that given  $N$  data points, we replace each data point  $x_i$  with some symmetric probability distribution  $K_h$ . We call this distribution  $K_h$  a *kernel*, and it is centered at the origin and has some smoothing parameter  $h$ . If we scale each kernel by  $\frac{1}{N}$ , then their sum will be a probability distribution. Thus the KDE for the data  $x_1, \dots, x_n$  is

$$\hat{f}_h(x) = \frac{1}{n} \sum_{i=1}^n K_h(x - x_i) = \frac{1}{nh} \sum_{i=1}^n K\left(\frac{x - x_i}{h}\right)$$

This formulation allows us to choose a kernel family  $K$ , and then alter it with a smoothing parameter  $h$  to get various kernels. Of course, this formulation only makes sense for univariate data (otherwise  $h$  couldn't simply be a scalar).

Choosing  $h$  is often the most challenging aspect of dealing with KDEs. If  $h$  is chosen too small, then the pdf becomes too multimodal (a large mode at each data point). If it is too large, it is too smooth, approaching a uniform distribution. For univariate data with a Gaussian kernel, an optimal choice for the bandwidth is  $h \approx 1.06\hat{\sigma}n^{-1/5}$  where here,  $\hat{\sigma}$  is the standard deviation of our data.

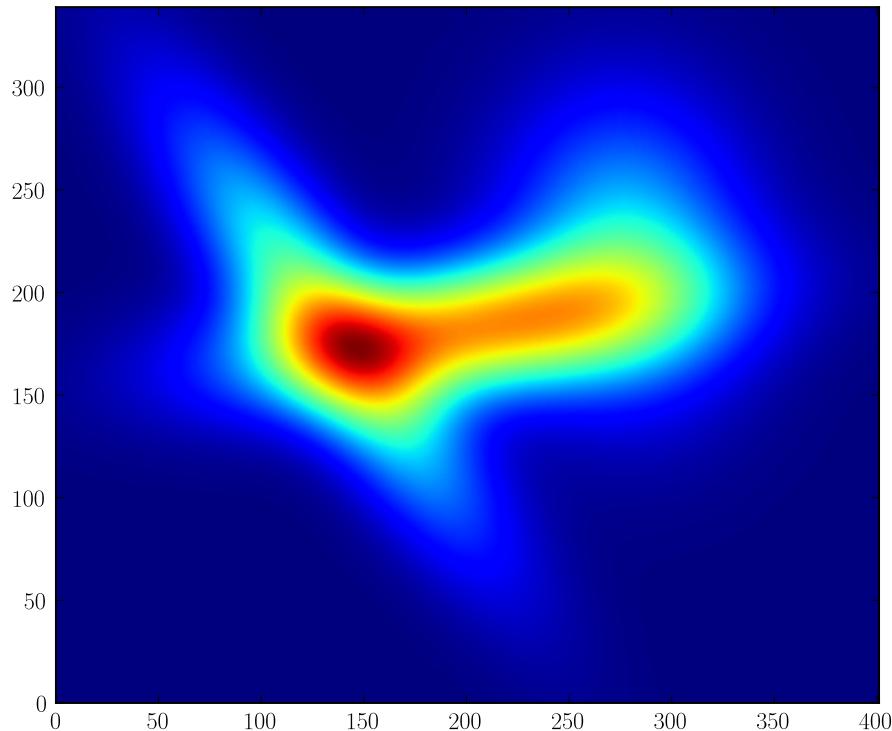


Figure 30.2: GMM density over the Baltimore region.

For multivariate data, the KDE is

$$\hat{f}_{\mathbf{H}}(x) = \frac{1}{n} \sum_{i=1}^n K_{\mathbf{H}}(x - x_i) = \frac{1}{n|\mathbf{H}|^{\frac{1}{2}}} \sum_{i=1}^n K(\mathbf{H}^{-\frac{1}{2}}(x - x_i))$$

where  $\mathbf{H}$  is a bandwidth matrix, and  $K$  is some multivariate symmetric density centered at the origin. Choosing  $\mathbf{H}$  is even more difficult in the multivariate case, but decent results can be obtained by choosing the diagonal matrix  $\mathbf{H}$  where

$$H_{ii} = \left(\frac{4}{d+2}\right)^{\frac{2}{d+4}} \cdot n^{\frac{-2}{d+4}} \cdot \hat{\sigma}_i^2$$

where  $d$  is the dimension of the data, and  $\hat{\sigma}_i$  is the standard deviation of the  $i^{th}$  dimension. We will use `scipy`'s implementation of the KDE.

The `gaussian_kde` class is initialized with data where each column is a sample, the number of rows being the dimension of the space, and the number of columns being the number of samples. We can evaluate the density of a new sample using the method `evaluate`.

```
>>> import scipy as sp
>>> from scipy import stats
>>> kernel = stats.gaussian_kde(data)
```

```
>>> print kernel.evaluate(sp.zeros(data.shape[0]))
```

**Problem 3.** Using `stats.gaussian_kde`, initialize a KDE with the homicide data (make sure it's in the right format!), and compute the KDE of each point on your grid of Baltimore. Plot the density and save your plot.

With this approach, we can get an alternative (and likely better) probabilistic view of crime in Baltimore, as shown in Figure 30.3.

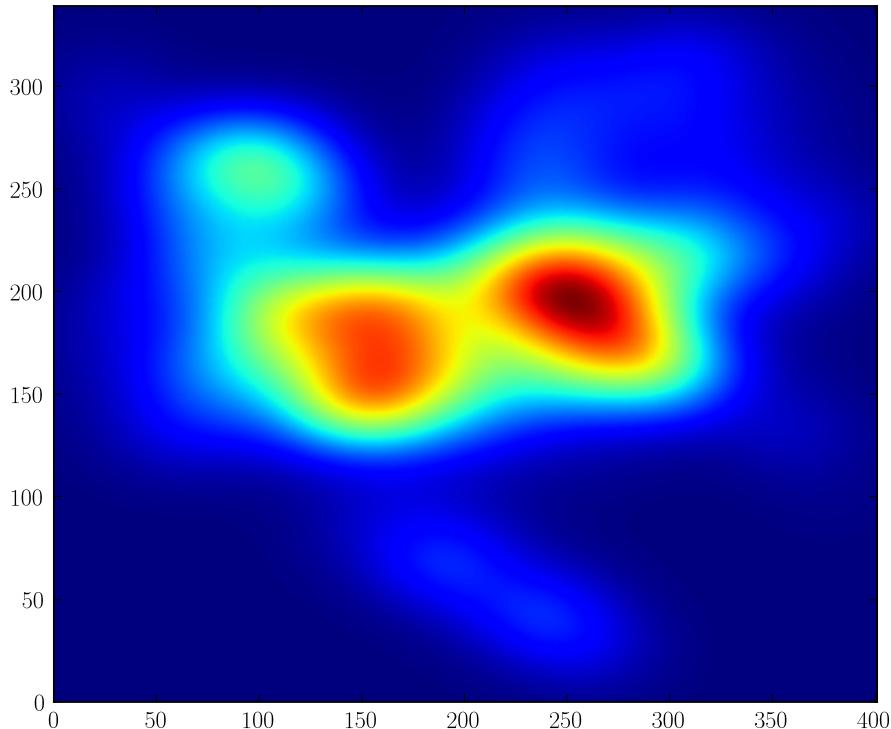


Figure 30.3: KDE over the Baltimore region.

We must still solve the initial problem, i.e. find the smallest geographical area of Baltimore containing 95% of all homicides. We do this by “vertically integrating” our densities. In other words, we choose some threshold  $t$ , find all points  $x$  on our grid such that  $\hat{f}_{\mathbf{H}}(x) \geq t$ , and compute the integral of our KDE over this region. If the length of each side for each square of our grid is  $h$ , then we can estimate this integral by

$$\sum_{x: \hat{f}_{\mathbf{H}}(x) > t} \hat{f}_{\mathbf{H}}(x) \cdot h^2$$

Ultimately, we are trying to find the region where this integral is approximately 0.95. Since  $t$  is the only thing we can vary, this becomes an optimization problem over  $t$ , where we are minimizing the objective function

$$g(t) = \left| \sum_{x:\hat{f}_{\mathbf{H}}(x)>t} \hat{f}_{\mathbf{H}}(x) \cdot h^2 - 0.95 \right|$$

**Problem 4.** Find this 95% region for both the GMM and the KDE and plot it using `imshow`, where each “safe” point on the grid is 0 and each “unsafe” region is 1.



# 31

## Image Recognition Tasks

**Lab Objective:** *Use the KNN and SVM algorithms to solve two image recognition problems.*

Two important image recognition problems are character recognition and face recognition. The first is generally framed as a post office problem. Every day, millions of pieces of mail are sent through the US Postal Service each day. This requires an automated way of routing much of the mail. The problem is to automatically determine the zip code of the addressee for a piece of mail. There are two parts to this problem: find the zip code on the letter, and then determine what it is. We will only consider the second part in this lab.

Given that we have an image of a single digit, how can we decide what it is without human intervention? This is a classification problem, with the classes being the digits 0 through 9. We will use both the KNN and SVM classifiers to predict each digit.

We will use the `digits` data set from `sklearn.datasets` for our data, using the method `sklearn.datasets.load_digits()`. Each sample is an  $8 \times 8$  image which has been flattened into a length-64 vector.

**Problem 1.** Load the digits data and separate it into a training set and a test set.

The module `sklearn.neighbors` has a nice class `KNeighborsClassifier` that implements the KNN classifier.

**Problem 2.** Find and read some of the documentation for the aforementioned class. Implement a KNN classifier on the training set. What is the misclassification rate on your test set?

While the SVM was originally designed as a binary classifier, it has been extended into a multi-class classifier as well. We won't go into the details here, but one common extension is to train  $K$  different SVMs (where  $K$  is the number of classes), each being a "one-versus-all" classifier. After some calibration, a new sample is predicted to be the class  $k$  where  $f_k(\mathbf{x})$  is greatest,  $f_k$  being the function defining the hyperplane for class  $k$  against all other classes. Again, `sklearn.svm` has a nice class `SVC` that implements this.

The module `sklearn.grid_search` provides a nice way to find the classifier that performs the best on the training set, considering a grid of parameters. We will use this to help us find an optimal SVM for the digits data set, where we use a radial basis function for the kernel.

```
>>> from sklearn.grid_search import GridSearchCV
>>> from sklearn.svm import SVC
>>> param_grid = {'C': [1e3, 5e3, 1e4, 5e4, 1e5], 'gamma': [0.0001, 0.0005, ←
    0.001, 0.005, 0.01, 0.1],}
>>> clf = GridSearchCV(SVC(kernel='rbf', class_weight='auto'), param_grid)
>>> clf = clf.fit(training_data, training_target)
```

**Problem 3.** Predict the values for the test set using the SVM we just trained. What is your misclassification rate? How does the SVM's performance compare to the KNN classifier?

We now consider our second image recognition problem: face recognition. This is much harder than simply recognizing a digit on a white envelope, as there is bound to be so much more noise and variation.

We use as our data set the “Labeled Faces in the Wild”, a database of face photographs. In fact, we will only consider a small subset of this database, including only images of Ariel Sharon, Colin Powell, Donald Rumsfeld, George W Bush, Gerhard Schroeder, Hugo Chavez, and Tony Blair. Through `sklearn.datasets` we can download and process this data set rather easily, though it might take some time.

```
>>> from sklearn.datasets import fetch_lfw_people
>>> people = fetch_lfw_people(min_faces=70, resize=0.4)
>>> data = people.data
>>> target = lfw_people.target
```

This data set consists of 1288 images of size  $50 \times 37$ , each flattened. The targets are digits from 0 to 6, corresponding with the ordered names above. This might seem counterintuitive, considering the main ideas of SVMs, but it is sometimes useful to reduce the dimensionality of our feature space before implementing an SVM, using PCA so we can retain as much information as possible while still reducing the dimensionality.

```
>>> from sklearn.decomposition import PCA
>>> pca = PCA(n_components=150, whiten=True).fit(data)
>>> data_pca = pca.transform(data)
```

**Problem 4.** Separate the data set into training data and test data. Create a PCA object fit to the training data. With this object, reduce the dimensionality of both the training data and the test data.

**Problem 5.** Train an SVM on the image set with the same parameter grid search used above. Label the test data. What is your misclassification rate?

Let's also compare and see how well the KNN classifier does on this data set.

**Problem 6.** Train a KNN classifier on the image data set. What is your misclassification rate on the test data? Does this surprise you?

This allows us to end this course with a very important take-home message: the No Free Lunch Theorem. In essence, this theorem states that there is no single machine learning classifier to rule them all—each has its strengths and weaknesses. More specifically, if there is a ML classifier that outperforms all other classifiers on a data set, then we can find a data set where a different classifier will be superior. This means that we have to be intelligent in how we choose which classifiers to try, because there isn't any “go-to” classifier that will always work.



# 32

## K-Means Clustering

**Lab Objective:** *Understand the basics of k-means clustering, and apply to the problem of clustering earthquake epicenters.*

### Clustering

In Lab 24, we analyzed the iris dataset using PCA; we have reproduced the first two principal components of the iris data in Figure 32.1. Upon inspection, a human can easily see that there are two very distinct groups of irises. Can we create an algorithm to identify these groups without human supervision? This task is called *clustering*, an instance of *unsupervised learning*.

The objective of clustering is to find a partition of the data such that points in the same subset will be “close” according to some metric. The metric used will likely depend on the data, but some obvious choices include Euclidean distance and angular distance. Throughout this lab we will use the metric  $d(x, y) = \|x - y\|_2$ , the Euclidean distance between  $x$  and  $y$ .

More formally, suppose we have a collection of  $\mathbb{R}^K$ -valued observations  $X = \{x_1, x_2, \dots, x_n\}$ . Let  $N \in \mathbb{N}$  and let  $\mathcal{S}$  be the set of all  $N$ -partitions of  $X$ , where an  $N$ -partition is a partition with exactly  $N$  nonempty elements. We can represent a typical partition in  $\mathcal{S}$  as  $S = \{S_1, S_2, \dots, S_N\}$ , where

$$X = \bigcup_{i=1}^N S_i$$

and

$$|S_i| > 0, \quad i = 1, 2, \dots, N.$$

We seek the  $N$ -partition  $S^*$  that minimizes the within-cluster sum of squares, i.e.

$$S^* = \arg \min_{S \in \mathcal{S}} \sum_{i=1}^N \sum_{x_j \in S_i} \|x_j - \mu_i\|_2^2,$$

where  $\mu_i$  is the mean of the elements in  $S_i$ , i.e.

$$\mu_i = \frac{1}{|S_i|} \sum_{x_j \in S_i} x_j.$$

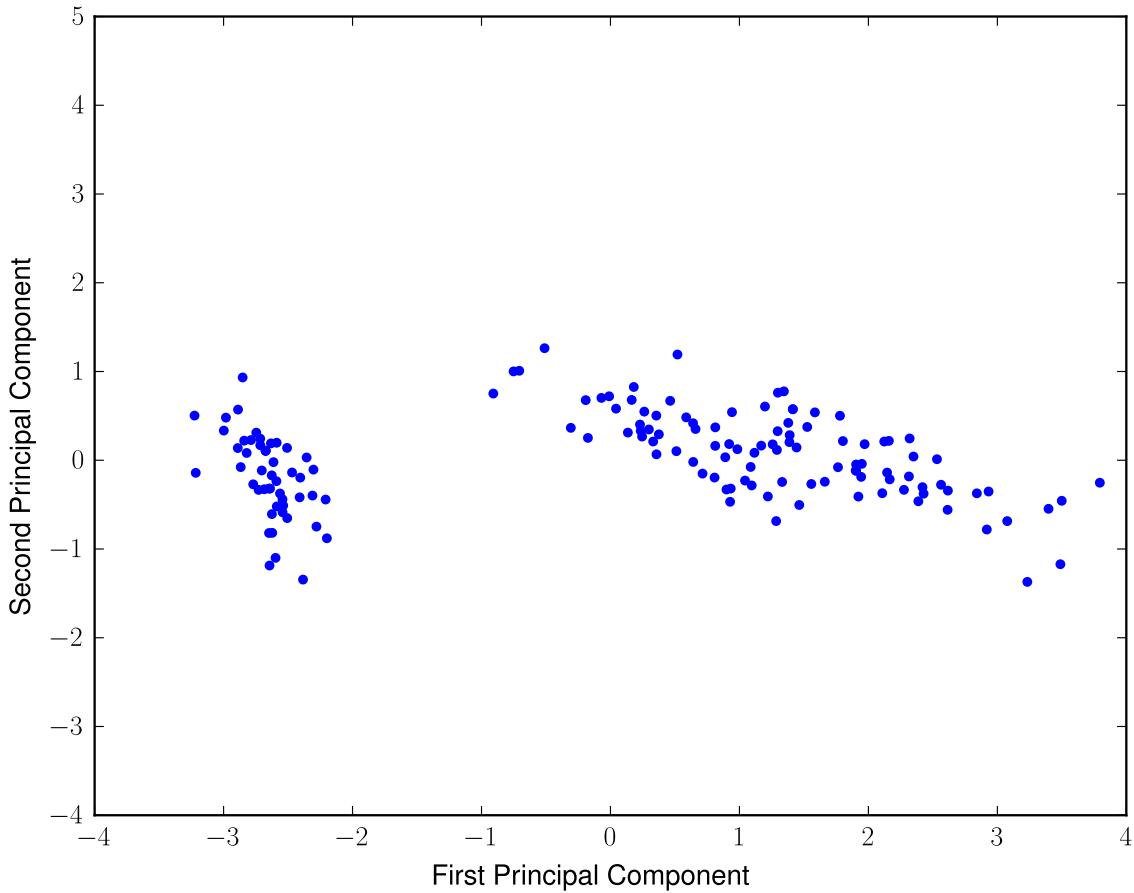


Figure 32.1: The first two principal components of the iris dataset.

## The K-Means Method

Finding the global minimizing partition  $S^*$  is generally intractable since the set of partitions can be very large indeed, but the *k-means* algorithm is a heuristic approach that can often provide reasonably accurate results.

We begin by specifying an initial cluster mean  $\mu_i^{(1)}$  for each  $i = 1, \dots, N$  (this can be done by random initialization, or according to some heuristic). For each iteration, we adopt the following procedure. Given a current set of cluster means  $\mu^{(t)}$ , we find a partition  $S^{(t)}$  of the observations such that

$$S_i^{(t)} = \{x_j : \|x_j - \mu_i^{(t)}\|_2^2 \leq \|x_j - \mu_l^{(t)}\|_2^2, l = 1, \dots, N\}.$$

We then update our cluster means by computing for each  $i = 1, \dots, N$ . We continue to iterate in this manner until the partition ceases to change.

Examine Figure 32.2, which shows two different clusterings of the iris data produced by the *k-means* algorithm. Note that the quality of the clustering can depend heavily on the initial cluster means. We can use the within-cluster sum of squares as a measure of the quality of a clustering (a lower sum of squares is better). Where possible, it is advisable to run the clustering algorithm several times, each with a different initialization of the means, and keep the best clustering. Note also that it is possible to have very slow convergence. Thus, when implementing the algorithm, it is a good idea to terminate after some specified maximum number of iterations.

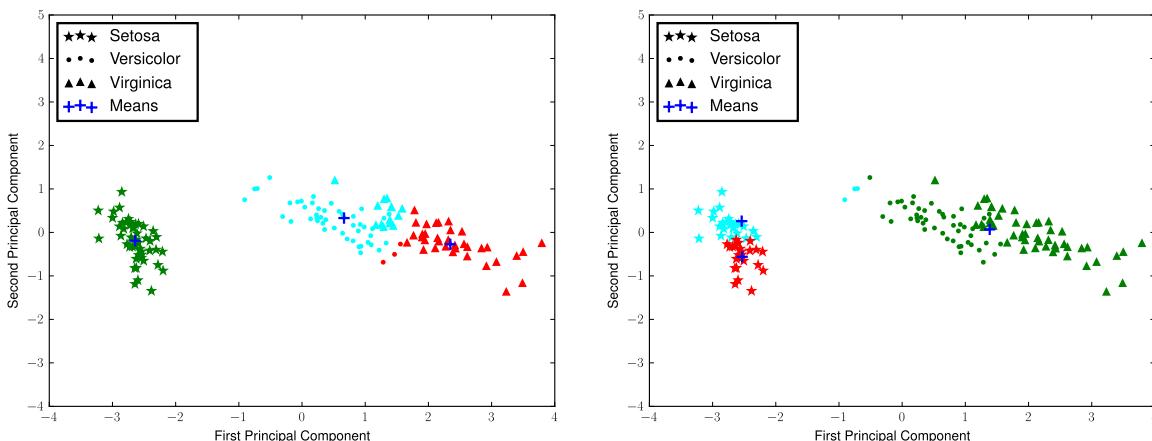


Figure 32.2: Two different K-Means clusterings for the iris dataset. Notice that the clustering on the left predicts the flower species to a high degree of accuracy, while the clustering on the right is less effective.

**Problem 1.** Implement the *k-means* algorithm using the following function declaration.

```
def kmeans(data,n_clusters,init='random',max_iter=300):
    """
    Cluster a dataset using the k-means algorithm.

    Parameters
    -----
    data : ndarray of shape (n,k)
        Each row is an observation.
    n_clusters : int
        The number of clusters.
    init : string or ndarray of shape (n_clusters,k)
        If init is the string 'random', then randomly initialize the ←
        cluster means.
        Else, the initial cluster means are given by the rows of init.
    max_iter : int
        The maximum allowable number of iterations.

    Returns
```

```
-----
means : ndarray of shape (n_cluster,k)
    The final cluster means, given as the rows.
labels : ndarray of shape (n,)
    The i-th entry is an integer in [0,n_clusters-1] indicating
    which cluster the i-th row of data belongs to relative to
    the rows of means.
measure : float
    The within-cluster sum of squares quality measure.
...
pass
```

Test your function on the first two principal components of the iris dataset. Run it 10 times, using a different random initialization of the means each time. Retain the clustering with the smallest within-cluster sum of squares. Your clustering should be similar to the first clustering in Figure 32.2.

## Detecting Active Earthquake Regions

Suppose we are interested in learning about which regions are prone to experience frequent earthquake activity. We could make a map of all earthquakes over a given period of time and examine it ourselves, but this, as an unsupervised learning problem, can be solved using our k-means clustering tool.

Our data is contained in 6 text files, each with earthquake data throughout the world covering a time period of one month, giving us data from January 2010 through June 2010. These files contain a lot of information which isn't of interest to us at the present time; all we would like to extract from them is the location of each earthquake, which appears in characters 21 through 33 of each line. Characters 21 through 26 contain the latitude of each epicenter, character 26 denoting North or South, and characters 27 through 33 contain the longitude of each epicenter, character 33 denoting East or West. We need to divide each value by 1,000 to represent these as degrees and decimals.

**Problem 2.** Load the earthquake data into a  $n \times 2$  array, where each row gives the longitude and latitude of an earthquake in degrees. Multiply South latitudes and West longitudes by  $-1$ . Create a scatter plot of the resulting data. You should be able to see the outlines of some of the continents and tectonic plates (since these are often areas of significant seismic activity). Your plot should match Figure 32.3.

We want to cluster this data into active earthquake regions. For this task, we might think that we can regard any epicenter as a point in  $\mathbb{R}^2$  with coordinates being their latitude and longitude. This, however, would be incorrect, because the earth is not flat. We must recognize that latitude and longitude are best viewed as a variation of spherical coordinates in  $\mathbb{R}^3$ , and we should interpret them as such. Since our *k-means* algorithm is based on Euclidean distance, we need to transform our data into 3-dimensional Euclidean coordinates.

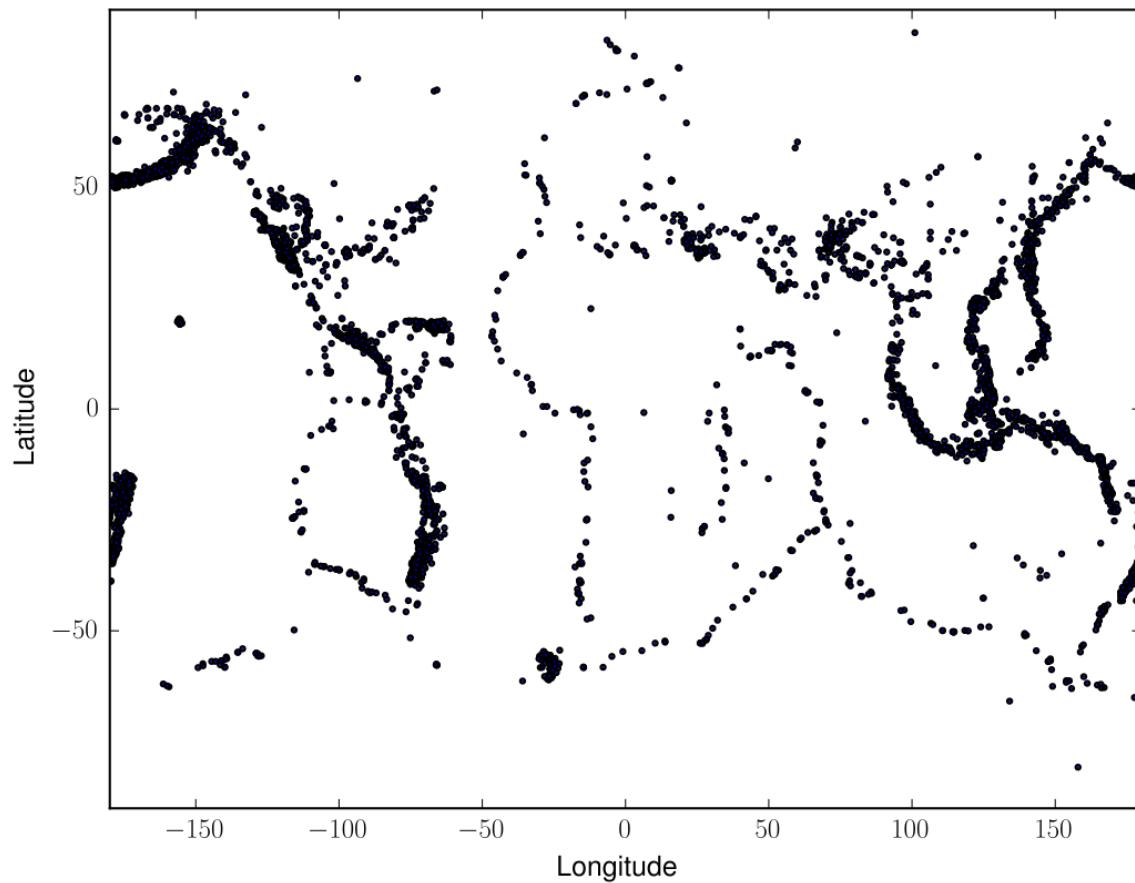


Figure 32.3: Earthquake epicenters over a 6 month period.

A simple way to accomplish this transformation is to first transform the latitude and longitude values to spherical coordinates, and then to Euclidean coordinates. Recall that a spherical coordinate in  $\mathbb{R}^3$  is a triple  $(r, \theta, \varphi)$ , where  $r$  is the distance from the origin,  $\theta$  is the radial angle in the  $xy$ -plane from the  $x$ -axis, and  $\varphi$  is the angle from the  $z$ -axis. In our earthquake data, the longitude is already the appropriate  $\theta$  value, and the  $\varphi$  value (in degrees) is simply  $90^\circ$  minus the latitude. For simplicity, we can take  $r = 1$ , since the earth is roughly a sphere. We can then transform to Euclidean coordinates using the following relationships:

$$\begin{array}{ll} r = \sqrt{x^2 + y^2 + z^2} & x = r \sin \varphi \cos \theta \\ \varphi = \arccos \frac{z}{r} & y = r \sin \varphi \sin \theta \\ \theta = \arctan \frac{y}{x} & z = r \cos \varphi \end{array}$$

**Problem 3.** Transform your earthquake data into three dimensional Euclidean coordinates. Be sure to consider if and when you need to transform your data from degrees to radians.

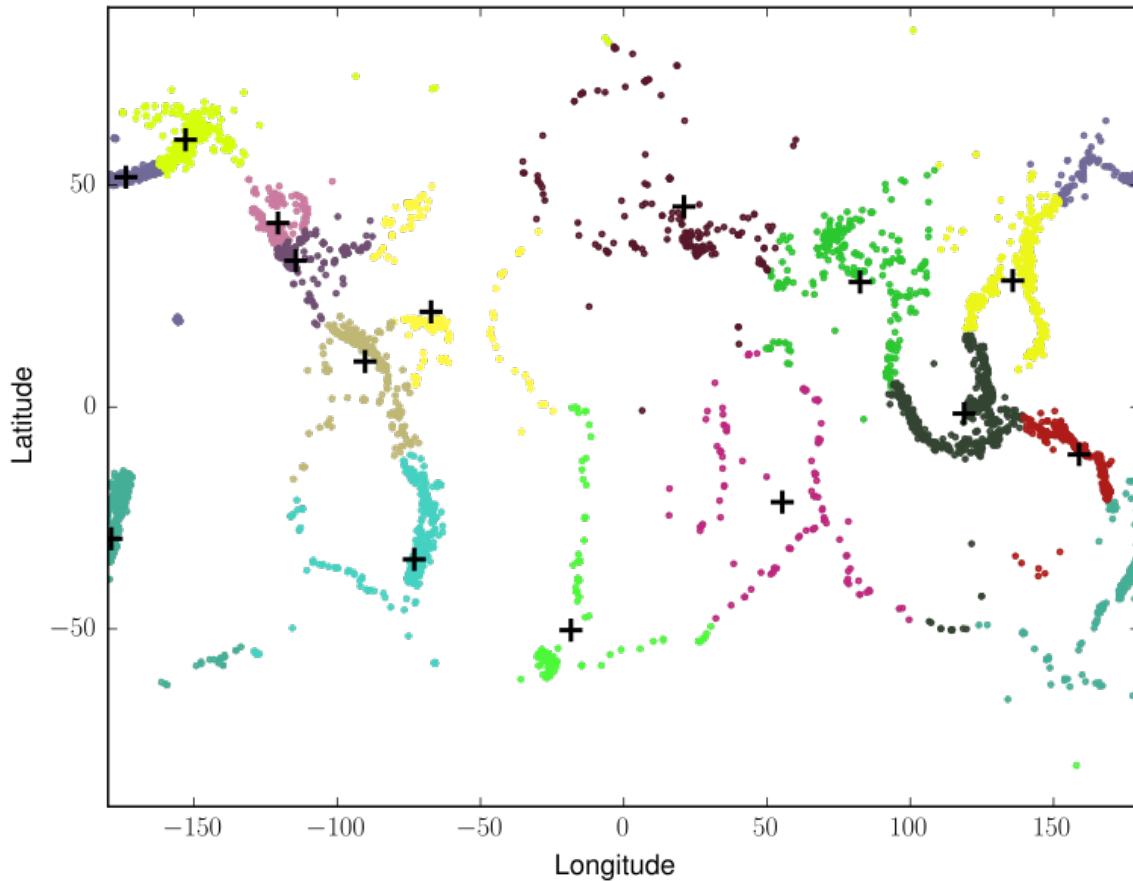


Figure 32.4: Earthquake epicenter clusters with  $N = 15$ .

We are now ready to cluster the earthquake data using the Euclidean coordinates. We need to address one further issue, however. Notice that each earthquake data point has norm 1 in Euclidean coordinates, since it lies on the surface of a sphere of radius 1. We also need to ensure that our cluster means have norm 1. Otherwise, the means can't be interpreted as locations on the surface of the earth. Furthermore, the *k-means* algorithm will struggle to find good clusters. A solution to this problem is to normalize the mean vectors at each iteration, so that they are always unit vectors. Thus, we need to add optional functionality to our `kmeans` function.

**Problem 4.** Add a keyword argument `normalize=False` to your `kmeans` function, and add code to normalize the means at each iteration, should this argument be set to `True`. Use your function to cluster the earthquake data into 15 clusters. Run this 10 times, keeping the best clustering. Transform the cluster means back to latitude and longitude coordinates (when calculating  $\theta$  using the inverse tangent, use `numpy.arctan2` or `math.atan2`, so that that correct quadrant is chosen). Create a scatter plot showing each cluster mean, along with the earthquake epicenters color-coded according to their cluster. Your plot should resemble that of Figure 32.4.

Though plotting our results in two dimensions gives us a good picture, we can see that this is not entirely accurate. There are points that appear to be closer to a different cluster center than the one to which they belong. This comes from viewing the results in only two dimensions. When viewing in three dimensions, we can see more clearly the accuracy of our results.

**Problem 5.** Add a keyword argument `3d=False` to your `kmeans` function, and add code to show the three-dimensional plot instead of the two-dimensional scatter plot should this argument be set to `True`. Maintain the same color-coding scheme as before. Use `mpl_toolkits.mplot3d.Axes3D` to make your plot.

## Spectral Clustering

We now turn to another method for solving a clustering problem, namely that of Spectral Clustering. As you can see in Figure ???, it can cluster data not just by its location on a graph, but can even separate shapes that overlap others into distinct clusters. It does so by utilizing the spectral properties of a Laplacian matrix. Different types of Laplacian matrices can be used. In order to construct a Laplacian matrix, we first need to create a graph of vertices and edges from our data points. This graph can be represented as a symmetric matrix  $W$  where  $w_{ij}$  represents the edge from  $x_i$  to  $x_j$ . In the simplest approach, we can set  $w_{ij} = 1$  if there exists an edge and  $w_{ij} = 0$  otherwise. However, we are interested in the similarity of points, so we will weight the edges by using a *similarity measure*. Points that are similar to one another are assigned a high similarity measure value, and dissimilar points a low value. One possible measure is the *Gaussian similarity function*, which defines the similarity between distinct points  $x_i$  and  $x_j$  as

$$s(x_i, x_j) = e^{-\frac{\|x_i - x_j\|^2}{2\sigma^2}}$$

for some set value  $\sigma$ .

Note that some similarity functions can yield extremely small values for dissimilar points. We have several options for dealing with this possibility. One is simply to set all values which are less than some  $\epsilon$  to be zero, entirely erasing the edge between these two points. Another option is to keep only the  $T$  largest-valued edges for each vertex. Whichever method we choose to use, we will end up with a weighted *similarity matrix*  $W$ . Using this we can find the diagonal *degree matrix*  $D$ , which gives the number of edges found at each vertex. If we have the original fully-connected graph, then  $D_{ii} = n - 1$  for each  $i$ . If we keep the  $T$  highest-valued edges,  $D_{ii} = T$  for each  $i$ .

As mentioned before, we may use different types of Laplacian matrices. Three such possibilities are:

1. The *unnormalized Laplacian*,  $L = D - W$
2. The *symmetric normalized Laplacian*,  $L_{sym} = I - D^{-1/2}WD^{-1/2}$
3. The *random walk normalized Laplacian*,  $L_{rw} = I - D^{-1}W$ .

Given a similarity measure, which type of Laplacian to use, and the desired number of clusters  $k$ , we can now proceed with the Spectral Clustering algorithm as follows:

- Compute  $W$ ,  $D$ , and the appropriate Laplacian matrix.
- Compute the first  $k$  eigenvectors  $u_1, \dots, u_k$  of the Laplacian matrix.

- Set  $U = [u_1, \dots, u_k]$ , and if using  $L_{sym}$  or  $L_{rw}$  normalize  $U$  so that each row is a unit vector in the Euclidean norm.
- Perform  $k$ -means clustering on the  $n$  rows of  $U$ .
- The  $n$  labels returned from your `kmeans` function correspond to the label assignments for  $x_1, \dots, x_n$ .

As before, we need to run through our  $k$ -means function multiple times to find the best measure when we use random initialization. Also, if you normalize the rows of  $U$ , then you will need to set the argument `normalize = True`.

**Problem 6.** Implement the Spectral Clustering Algorithm by calling your `kmeans` function, using the following function declaration:

```
def specClus(measure,Laplacian,args,arg1=None,kiters=10):
    """
    Cluster a dataset using the k-means algorithm.

    Parameters
    -----
    measure : function
        The function used to calculate the similarity measure.
    Laplacian : int in {1,2,3}
        Which Laplacian matrix to use. 1 corresponds to the unnormalized,
        2 to the symmetric normalized, 3 to the random walk normalized.
    args : tuple
        The arguments as they were passed into your k-means function,
        consisting of (data, n_clusters, init, max_iter, normalize). Note
        that you will not pass 'data' into your k-means function.
    arg1 : None, float, or int
        If Laplacian==1, it should remain as None
        If Laplacian==2, the cut-off value, epsilon.
        If Laplacian==3, the number of edges to retain, T.
    kiters : int
        How many times to call your kmeans function to get the best
        measure.

    Returns
    -----
    labels : ndarray of shape (n,)
        The i-th entry is an integer in [0,n_clusters-1] indicating
        which cluster the i-th row of data belongs to.
    """
    pass
```

We now need a way to test our code. The website <http://cs.joensuu.fi/sipu/datasets/> contains many free data sets that will be of use to us. Scroll down to the "Shape sets" heading, and download some of the datasets found there to use for trial datasets.

**Problem 7.** Create a function that will return the accuracy of your spectral clustering implementation, as follows:

```
def test_specClus(location,measure,Laplacian,args,arg1=None,kiters=10):
    """
    Cluster a dataset using the k-means algorithm.

    Parameters
    -----
    location : string
        The location of the dataset to be tested.
    measure : function
        The function used to calculate the similarity measure.
    Laplacian : int in {1,2,3}
        Which Laplacian matrix to use. 1 corresponds to the unnormalized,
        2 to the symmetric normalized, 3 to the random walk normalized.
    args : tuple
        The arguments as they were passed into your k-means function,
        consisting of (data, n_clusters, init, max_iter, normalize). Note
        that you will not pass 'data' into your k-means function.
    arg1 : None, float, or int
        If Laplacian==1, it should remain as None
        If Laplacian==2, the cut-off value, epsilon.
        If Laplacian==3, the number of edges to retain, T.
    kiters : int
        How many times to call your kmeans function to get the best
        measure.

    Returns
    -----
    accuracy : float
        The percent of labels correctly predicted by your spectral
        clustering function with the given arguments (the number
        correctly predicted divided by the total number of points.
    """

    pass
```



# 33

## Bayesian Search

**Lab Objective:** *Understand how Bayesian methods can be used for search and rescue operations.*

In the past half century, statistical methods have seen a rise in popularity for search and rescue problems. *Bayesian search theory* has been successfully used to find lost sea vessels and aircraft, including the USS *Scorpion*, the MV *Derbyshire*, the SS *Central America*, and Air France Flight (AF) 447. It was also used to search for a lost nuclear bomb after a B-52 bomber crashed in Spain in 1966.

In this lab, we will simulate search procedures for the AF447 disaster, using our knowledge of Bayes' Theorem and estimations of ocean depth in the vicinity of the last point of radio contact with the flights captains.

On June 1<sup>st</sup>, 2009, AF447 departed Rio de Janeiro, Brazil, flying to Paris, France. Air traffic controllers lost contact with the aircraft at 2:10 AM, nearly halfway between South America and Western Africa. The location of final contact was approximately 2.98°N, 30.59°W. Figure ?? shows the flight path (and part of the planned flight path) of AF447.

The search operations proved unsuccessful for nearly two years. After an unsuccessful first year of searching, the consulting company Metron was asked to develop a probabilistic search procedure to find the flight. Using Bayesian search methods, a large portion of AF447's debris field was discovered within a week of resuming the search. Within a month, the black boxes were recovered.

Bayesian search theory consists of combining our belief of where a lost vessel should be located with our belief of our success of finding it in a location, should we search there. A simple way of expressing our belief of where a lost vessel should be found is with a probability distribution, centered at the last point of contact. Our belief of successfully finding the vessel in a given location can be a function of ocean depth (or, for a land based search, a function of density of foliage, difficulty of searching a mountainous region, etc.). We simplify this by discretizing the possible locations into a square grid.

Formally, let  $A_i$  be the event that we successfully find the vessel when we search in location  $i$ , and let  $B_i$  be the event that the vessel is in location  $i$ . Then let  $p$  be a probability distribution for the location of the vessel over the grid, so

$$p_i = \mathbb{P}(B_i).$$

Let  $q$  be a set of  $N$  Bernoulli parameters, each denoting the probability of success if we search in its location, i.e.

$$q_i = \mathbb{P}(A_i|B_i).$$



Figure 33.1: Flight path of AF447.

We also assume no chance of success in finding the vessel in location  $i$  if it is *not* located there.

This allows us to compute

$$\begin{aligned}
 r_i &= \mathbb{P}(A_i) \\
 &= \sum_j \mathbb{P}(A_i \cap B_j) \\
 &= \sum_j \mathbb{P}(A_i|B_j)\mathbb{P}(B_j) \\
 &= \mathbb{P}(A_i|B_i)\mathbb{P}(B_i) \\
 &= p_i q_i
 \end{aligned}$$

We proceed by searching in location  $k$ , where  $k = \text{argmax}_i r_i$ .

**Problem 1.** Write a function that accepts two  $20 \times 20$  arrays,  $p$  and  $q$ , and returns the index pair  $i, j$  of the next search location.

If a search successful, then our search is over. If unsuccessful, we update our probabilities  $p_i$  given our recent data  $\theta$  (our unsuccessful attempt) according to Bayes Rule, which in this case, is

$$\mathbb{P}(B_i | \theta) = \frac{\mathbb{P}(\theta | B_i)\mathbb{P}(B_i)}{\mathbb{P}(\theta | B_i)\mathbb{P}(B_i) + \mathbb{P}(\theta | B_i^c)\mathbb{P}(B_i^c)}$$

This has two different solutions, depending on where we searched, yielding the following for our posterior probability:

$$\tilde{p}_i = \begin{cases} \frac{(1-q_i)p_i}{(1-q_i)p_i + (1-p_i)} = p_i \frac{1-q_i}{1-p_i q_i} & \text{if we searched in location } i \\ \frac{p_i}{(1-q_i)p_i + (1-p_i)} = \frac{p_i}{1-p_i q_i} & \text{if we searched in location } j \neq i \end{cases}$$

**Problem 2.** Write a function that accepts two  $20 \times 20$  arrays,  $p$  and  $q$ , as well as an index pair  $i, j$  of search coordinates for the most recent unsuccessful search, and returns the posterior probabilities  $\tilde{p}_{k,l}$  for each possible location  $k, l$ , where  $1 \leq k, l \leq 20$ .

Using our computed posterior probability and our success probabilities, we compute  $r$  again, and choose a search location, continuing this procedure until success.

**Problem 3.** Write a function to simulate the search procedure. It should accept two  $20 \times 20$  arrays,  $p$  and  $q$ , where  $p$  is our initial prior on the location of the vessel and  $q$  contains the probability of success for each grid location. It should also accept the actual location  $i, j$  of the vessel. During the simulation process, assume that  $q$  is correct, i.e. given that we search in location  $i, j$ , we will successfully find the vessel with probability  $q_{i,j}$ . The function should return the number of search iterations until success, as well as the location of the vessel (but don't hard code this!).

We have roughly examined the ocean depths in the vicinity of the crash, and compute probabilities of successfully finding it in a location, given that we search there (this is simply a function of depth). We have also computed an initial prior on the location, with locations nearer the final point of contact having higher probabilities than locations further away. These are represented as  $20 \times 20$  arrays, and are located in the files `depthProbs` and `prior`, respectively.

**Problem 4.** Unpickle the two files mentioned above, and test your previous function. Once it's working properly, write a function that runs the previous function `n_sim` times, and prints out the shortest search length, longest search length, and average search length. Test it with `n_sim = 500` at locations 9, 10 and 12, 5.