Chapter 1
Introduction

Abstract THIS IS A DRAFT DOCUMENT FOR A WORK IN PROGRESS. COPYRIGHT BY JAMES SHARPNACK, 2020. PLEASE DO NOT DISTRIBUTE.

1.1 The two tales of data

To early statisticians, data primarily meant a sample from a population, such as the result of a survey or experiment. Sir Francis Galton (1822-1911) conducted surveys to study human genetics and inheritance of ability. The idea is that this data is a random draw from some larger population, and by collecting such data, we can glean information about unknown quantities and properties of the population as a whole. Such surveys collected data with the express purpose of calculating statistics in order to estimate desired population level quantities. Building on the work of Galton, statisticians such as R.A. Fisher (1890-1962) developed tools to quantify the uncertainty of statistics extracted from sampled data. Modern Statistics is an expanding field devoted to understanding uncertainty inherent in statistical calculations and algorithms. Now statistics are extracted from every type of dataset imaginable, from text data to videos.

To the computer scientist, data are anything stored on the computer that is used as input or intermediate results for a program. Alan Turing (1912 - 1954) conceved of a computer, which we now call a Turing machine, that works by reading and writing data on a tape of arbitrary length. This served as an early model of a computer, and central to its usefulness as a theoretical construct is the idea that data can influence the state of the processing unit, and in turn the processor will write and alter the data. Modern computer architecture is built up from this simple model, and understanding modern architectures is critical to being a successful data scientist.

One might be concerned that these two conceptions of data are in conflict. Where is randomness in the Computer Science (CS) notion of data? Why does the use of data as samples from a population seem to be more restrictive than data as anything stored in memory? If one thinks about it, the idea that data are anything that is stored

in memory is not in conflict with the use of data in Statistics. Data are anything stored in memory, and statistics are anything that is extracted from data. Instead of thinking of a statistic as just a number, we often imbue it with a probability distribution, which enables us to quantify uncertainty. Looking forward, we should think of data as both an object in memory that has to be transferred, processed, and learned from, in addition to being a random quantity. Data carries this randomness like baggage, that we must unpack to produce uncertainty and error, but luckily we can quantify this uncertainty with statistical reasoning.

Successful data scientists must be well versed in the main ideas from computer science and statistics, as well as understand the context surrounding the data. In this book, we highlight some key data science principles, such as optimization, data structures, query languages, statistical machine learning, and the grammar of graphics. In order to practice data science however, you need tools that utilize these principles at your fingertips. To this end, this book is filled with open source tools, in Python, to demonstrate and utilize these principles.

1.2 Workflow and Installations

There is a preferred way to do Data Science with Python, painstakingly developed in response to many repeated failures, which we will adopt in this book. Specifically, our sequential programming language of choice will be Python, but in addition we will use descriptive languages, such as SQL, markdown, HTML, and XML. For Python we need a package manager, and we recommend Anaconda, but you can also use pip or system-wide package managers such as APT for Ubuntu. You need a text editor such as emacs or vim and some interactive development environments, namely Jupyter, IPython, and perhaps PyCharm. You should also use a versioning system such as Git for collaboration, and will want to make your projects reproducible with tools like Make. You can think of this as your Data Science tech stack, which can be expanded upon by including other languages such as R and Julia, or other tools such as Apache Spark or D3.js.

1.2.1 Why Python?

Python was developed in 1991 by Guido van Rossum because the popular procedural languages of the time (perl, C, Fortran, etc.) were deficient for many common tasks. C (Dennis Ritchie, 1973) is great for writing fast optimized code, but it suffers from a lack of flexibility because of the need to declare variables, and the thin layer of abstraction. Perl (Larry Wall, 1987) is great for writing scripts that interact with unix, process strings, and are able to do this quickly with minimal effort. One common complaint of Perl is that it was very easy to write nearly unreadable code in Perl (aka line garbage), and it is filled with heuristics. Python is like Perl in that there

is sufficient abstraction to write code without thinking too much about memory allocation and type compatibility, but unlike Perl, it emphasizes readibility. Python and Perl are both interpreted languages, meaning that they come with interpreters that will run code line by line as you execute them, while C is compiled, meaning that you write code and then a program, called a compiler, converts it into machine code (code that the CPU can directly read).

For these reasons, Python's popularity has grown, so that now it is the second most popular language after Java. One advantage of this is that there exist tons of great packages for Python, such as numpy, scipy, matplotlib, etc. That is why we often prefer Python for data science. C and Fortran are going to be typically faster, but it is often wiser to go out and find a package that already implements what you want to do, so that you don't have to code everything from scratch (and it will probably be a much faster implementation). While most of this book will discuss using different packages in Python, you need to learn the basic Python syntax, because inevitably you will need to program basic things by hand. Another fundamental programming language for the data scientist is R, which has more extensive statistical packages than Python. R is not as universally used as Python though, and for some things, such as working with unstructured data and text, it is a bit cumbersome. Ideally, you will be familiar with both languages and will use whichever is more well suited for the task at hand.

1.2.2 Unix Shell

Throughout this book, it will be helpful to have access to a unix shell such as bash. Often this is called the terminal, and the basic idea is that you can enter in commands such as `$ whois google.com`, and it will run a program, instead of having to click on it through your windows manager and graphical user interfaces (gui). Throughout this book, the `$` symbol represents the beginning of a bash line, and should not actually be typed. The command line gives you greater control and more specificity when working with your computer. The command `whois`, for example, lets you find out that google.com was registered on 15 Sept 1997 in California by Google LLC. In Mac OS X or Linux, you can simply open up the terminal. In Windows however you will either need to install a bash shell or use the Ubuntu on Windows subsystem. For windows, I recommend just installing Anaconda (see below) and using the Anaconda prompt if you need a shell.

Different systems will have different commands available. For example, whois was not installed by default on my system and I had to install it first. On the Linux distribution Ubuntu, I can use the apt package manager as in `$ apt install whois`, but in Mac OS X, you should use the brew package manager. As in all other things, google.com is your friend, if you run into errors or don't remember specific commands, just google it and descend down whatever rabbit-hole seems promising.

For our purposes, it will be handy to have a unix shell for a few reasons. One is that when using some aspects of jupyter, such as the nbconvert command for

making slides in presentation mode, it may be helpful to declare variables in the command line. We will go into the details of specific commands in bash IN A LATER CHAPTER, but for now, you should know that a typical bash command takes the form `$ command -o {-}{-}option argument1 argument2` where 'command' is a lower case command, '-o' is a single letter option or flag, '--option' is a long form option, and 'argument1', 'argument2' are arguments. An example is the command `$ ls -la` which lists the files in your current directory, and the 'l' and 'a' flags modify the format of the displayed output. In general, your first line of attack on complicated data will be the command line, where you can display the size, location, sample the first few lines, and do some basic string processing.

Makefiles are a way to organize the commands that need to be run to reproduce an analysis. Make is typically used for building source code (for example, C++ code that needs to be compiled before running), but it turns out to be an excellent tool for reproducing data processing steps. In your code directory you can add a file `makefile` which obeys the following syntax:

```
target: prerequisites
    bash commands
```

and you can run the target with `$ make target`. This will run the prerequisites if it is another target or check for a file if it is not, and then run the bash commands below if these checks pass. For example, the following makefile is typical:

```
all: munge vis

munge:
  data_munge.py

model: intermediate.data
  run_model.py

clean: intermediate.data
  rm intermediate.data
```

In this example, `make munge` will run `data_munge.py`, `make model` will check for the file `intermediate.data` and then run `run_model.py`. Simply `make` will run both the munge and model targets, and `make clean` will remove `intermediate.data` if it exists.

1.2.3 Python Installation

When we talk about Python, we are talking about the Python language, but the interpreter is what actually executes the code. CPython is the most common interpreter, and it is called CPython because it is implemented in C. Throughout this book, we will be using Python 3, which differs significantly enough from Python 2 that some

of the code presented here will not work with the Python 2 interpreter. There are likely many ways to install the Python 3 CPython interpreter on your system. If you want to minimize the amount of time that you spend on installation issues, then you can use the Anaconda distribution of Python (`https://anaconda.org`).

Anaconda is a Python distribution that comes with its own Package installer, conda. To install you should go to the `anaconda.org` download page and choose your operating system. It also comes with the scipy stack, jupyter, and ipython. You can also use it to install R through the r-essentials package, and this will allow you to run an R kernel in jupyter. Julia is another language that you may be interested in, and you may find that you can write much faster scripts if you are coding something from scratch. You can also install julia with conda.

If you are a Linux or Mac user the terminal is sufficient, but for Windows you probably do not have a shell that you use often. In Windows, Anaconda ships with a bash shell from which you can run all the necessary commands. Just search for the anaconda prompt and then pin it to your desktop. From the shell, you should be able to run the commands `$ python`, `$ ipython`, `$ jupyter notebook`. When you run the command `$ ipython`, then you should see a header like the following.

```
Python 3.7.4 (default, Aug 13 2019, 20:35:49)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.8.0 -- An enhanced Interactive Python.
```

Of course, 'Python 3.7.4' is you current version which may differ. You may want to start with updating conda: `$ conda update conda`. In order to see what packages you have installed in conda run `$ conda list` and then you should see a list of packages with their versions, for example,

```
numpy                     1.17.2          py37haad9e8e_0
scikit-learn              0.21.3          py37hd81dba3_0
jupyter                   1.0.0                  py37_7
```

which tells us that numpy is installed at version 1.14.0, in addition to the versions of many other installed packages such as scikit-learn and jupyter. You should see numpy, scipy, matplotlib, ipython, jupyter, pandas listed. One check that you could make is to make sure that your listed ipython version (7.8.0 in the above display), matches that listed by `$ conda list`. If you do not see them, then install them with `$ conda install packagename` where 'packagename' is replaced with your package.

Suppose that you did not find your Anaconda install when you opened the python interpreter, what is likely happening is that you have a prior installation of Python. You can see what is happening if you inspect the PATH environment variable, in linux run `$ echo $PATH`. You will see a list of absolute paths, which indicate the places in which the shell looks for a given executable. It will run the first instance that it finds, so all you have to do is put the Anaconda install path at the beginning of this list. In Linux, open your `~/.profile` file with a text editor such as emacs or vi and add a line

```
PATH="PATH_TO_PYTHON:$PATH"
```

where `PATH_TO_PYTHON` is replaced with that path to anaconda. In Mac OS X edit the file `~/.bash_profile` and add the line

```
export PATH=PATH_TO_PYTHON:$PATH
```

which does the same thing.

Then run this with `$ source ~/.profile` (or `$ source ~/.bash_profile` for Mac). In Windows, you can edit the path variable by using the Environment Variables in the Control Panel. Just search in Windows 8 for "environment variables" and it should come up. You can edit the Path variable there by clicking "Path" and "Edit".

After you have become accustomed to building and maintaining packages with Anaconda, then it will be useful to become well versed in conda environments. An environment is simply a directory which contains the packages that you have installed, but on a given machine you can have many environments. This is useful if a module or script that you are writing requires specific versions that are not on your current install, then . The use of environments also lends itself to reproducibility because you can share the environment via an `environment.yaml` file. To create an environment you will use the command `$ conda create --name thisenv` where you replace 'thisenv' with a better name for the environment. Then to activate the environment you simply use `$ conda activate thisenv`, and then the standard conda commands you use will be within this environment. You should see your prompt change to `(thisenv) $` to remind you that you are currently within the conda environment. To deactivate simply use `$ conda deactivate`.

1.2.4 Text Editors and IDEs

Files on your computer are just a bunch of bytes (strings of 0s and 1s) on your drive. A plain text editor reads these as characters via ASCII, which is a dictionary that converts bytes to characters (like how Ribosomes convert RNA base pairs to amino acids). So you can start a new file using an editor like emacs, vi, or notepad, and write something there like "Hello world." and then save it as 'hello.txt' or 'hello' or whatever. Then it writes the bytes that those characters correspond to on your drive. If you do the same thing in Microsoft Word and save it as a Word file, then Word converts it to a different set of bytes and this process is proprietary (it is legally protected by licenses like the recipe for CocaCola). If you open a Word document in emacs it will look like `320317^Q340241261^Z341^@^@^@^@^@^@^@^@` which is not very helpful.

One way to write code is to just select a text editor that you like and stick with it for all of your coding needs. Common choices are emacs, vim, sublime, notepad++, atom, etc. All of these have syntax highlighting, but you may need to do some work to enable it depending on your install. The oldest of these editors are vim and emacs, and they have their own hotkeys and interfaces. For remote computing it is often nice

to already be familiar with vim (or emacs). This is because they can be run entirely in a bash shell (in the terminal). The following is a list of recommended text editors:

1. Vim: an open source text editor that is preinstalled on all operating systems other than Windows; has a terminal interface; best for remote computing; syntax highlighting for Python
2. Emacs: open source text editor with terminal interface; may have to install on server (with apt install emacs for example) for remote computing; syntax highlighting for Python
3. Atom: open source GUI for text editing; good GitHub integration
4. Sublime Text: open source GUI for text editing; written in Python with great packages

The simplest way to write and run Python scripts is to edit a file, for example, run in terminal `$ emacs text.py` and enter

```
import os
for fn in os.listdir():
  print(fn)
```

After saving and exiting (in emacs `Ctrl-X Ctrl-S Ctrl-X Ctrl-C`) then run the script with `$ python test.py`. This printed the contents of my current directory:

```
perceptron.py
scrabble.py
test.py
proc_rst.py
```

Because Python is interpreted, we are able to run the above script line by line in the Python shell. When we run `$ python` we get a prompt that we can use to run lines such as `> import os; os.listdir()` which will output the contents of the directory as well. Throughout, we will use > to indicate that we are using a Python shell (although you may see the prompt as >>>). You can use the Python shell as a sophisticated calculator, as in `> 5003.38 + 134.56 - 2500`. This works because a just-in-time (JIT) compiler is running in the background, and it compiles each line or function when it needs to. Typically, the downside to this flexibility is that the code you write in Python is slower than if we had written it in C.

Interactive development environments (IDE) are development tools with Python interpreters and other tools like tab completion (when you hit Tab it auto-completes the code snippet). IPython is an IDE that acts like a Python shell and has more extensive features than the native Python shell. Conda comes with IPython or you can install with pip as in `$ pip install ipython`. IPython has magic commands that are not part of the Python language, and they are prepended with `%` as in `%time` (we will use this magic command for profiling in the next Chapter). Often I will use ipython to test out code snippets and then use the magic command `%save 1-30 temp.py` to write lines 1 through 30 from IPython to a temporary

file then move these to a module (a Python file) that I am working on. Then using the `%run` magic command I run the module to import the functions that I have just written. In this way, I have my favorite editor, emacs, open with the module that I am working on and the temporary code file from IPython, alongside IPython for testing and debugging the code. You can find a complete tutorial of IPython in the ipython readthedocs: `https://ipython.readthedocs.io/`.

All of the editors mentioned above have a Python extension that allow you to have a Python prompt in the editor. This can expedite the process of copying code from IPython to the editor, although depending on the type of script, this may not be that important. In vim and emacs, the Python extension is called python-mode. Conda also has an IDE, called spyder, that is well suited to data science applications. PyCharm is another popular IDE for Python, and it can be found on the JetBrains website: `https://www.jetbrains.com/pycharm/`.

1.2.5 Jupyter

When you use IPython at the command line, it is running the IPython kernel in the background. The kernel is a process that runs your code and does things like completing code and running magic functions. Jupyter is an application that communicates with the IPython kernel but provides a sophisticated interface that runs in your web browser. If you are using conda you probably already have jupyter installed. If you run `$ jupyter notebook` then it should open a browser tab and you should see the notebook dashboard. This has files in you current directory which you can navigate around in. Jupyter is an application for working in IPython notebook files, which have extensions `.ipynb`. These are json files (we will see JSON return IN A LATER CHAPTER) that not only saves the code that you have run, but also the output and markdown around the code.

Jupyter is well suited for presenting results of an analysis, and is not that well suited for time intensive scientific computing. If I am working with a moderately sized dataset, and am doing exploratory data analysis, then I will often use Jupyter. More often I will develop a module for computationally intensive processing of a dataset using the text editor and IPython. I use the module that I have written to process the dataset, which typically will output summary statistics, smaller resulting datasets, and results of analyses. Then I load these into jupyter and describe and document the analysis and results with markdown and visualizations. A good rule of thumb is that there should be no cells in your notebook that take more than 30 seconds to run, and most cells should be nearly instantaneous. If you have something that is taking a long time to run then separate it into a module and run it in the command line (perhaps on a server).

In the left-hand corner of the dashboard, you can click new and select the Python 3 interpreter. This will open a notebook, which you can rename. The notebook consists of cells that you can write code in and run via the IPython kernel. When you are working with a cell, you can either be in command mode or edit mode. In command

mode, you can run the cell, move it around within the notebook, change the cell type to markdown etc. You can get to edit mode by hitting Enter over a cell. In edit mode, you are editing the contents, and still have access to IPython tools like tab completion. You can run the cell with Ctrl-Enter or Shift-Enter (to move down a cell also), and can exit to command mode with Esc (there are many tutorials online for Jupyter where you can find other hotkeys).

One very nice thing about Jupyter is that you can add markdown cells around the code cells to document the code, interpret results, and provide background. Markdown is a descriptive markup language that allows for easy structural formatting of text. For example a cell in markdown mode with `### Header` will produce a large font header. You can easily add lists, tables, links, code snippets, quotes, embed HTML and images, and LaTeX equations. For details on markdown syntax and equations in Jupyter, see the following: `\https://https://jupyter-notebook.readthedocs.io/`.

1.2.6 Collaborating with Git

Git is an open source software (code that is free to use and develop by anyone) that provides version control. Imagine that you are working with team of people on the same file, say you all have access to the same Dropbox directory. You could all change the same file, but then when any of you syncs your changes then it will overwrite the other changes. You could set times to edit, such as Don edits from 10am-12pm, Peggy edits from 12pm-2pm, and Joan edits from 2pm-4pm. Or you could keep versions of files by changing the file name, so when I edit `lucky_strikes_v3.py` then I edit and save as `lucky_strikes_v4.py`. These all seem cumbersome, and versioning systems provide a better way. Git (and other versioning systems) provides the following features:

1. A history of changes to files serving as a backup.
2. Developers can work concurrently and then with the help of git merge their changes.
3. Tracing what changes were made by whom when.

In any directory you can run `$ git init` and then it will add a .git folder at that directory. This is the root directory for your new repository and it will keep records of the files that are being tracked. Typically, you will want to track files that are human readable code files, git is not really made for storing data and other large files that you will not be editting by hand. If you have a file, say module.py that you want to track you add it with `$ git add module.py` and now the current version is staged for commit. You can see the status of files in the directory with `$ git status`. Once you are ready to commit the changes use `$ git commit -m "some message"` with a message that describes the commit in the quotes. Then if you have a remote repository set up you can push these changes to the remote repository with `$ git push origin remote`.

Why do all of this? As we have mentioned, it is mostly for collaboration. If someone else pushed their changes to the remote repository first, then you will get an error telling you to pull these changes before you push your own. Then you will rull `$ git pull`, in which case, it will update your local repository. You with then get a notification that a conflict has occurred and where, in which case you will have to go into that file and resolve the conflict. Typically, there is some update that your colleague has done that you need to make your code consistent with. The file that is in conflict will have annotation that looks like

```
<<<<<<< HEAD
Your changes may differ from
=======
their changes.
>>>>>>> commit-number
```

And you have to resolve the conflict by merging the two changes. Once the file is to your liking then you should git add the file, commit again, and push.

Git is a distributed versioning system meaning that there is no distinction between a server and a client repository-all repositories are created equal. Github is a company that provides git repositories on their computers that you can use as your remote servers. You can set up an account on github.com and then start your own repository. It will then show you how to initialize the repository on your computer and then you can add collaborations in the Settings tab. The collaborators can clone your repository with `$ git clone https://github.com/username/reponame` and the start working with it locally.

1.3 A First DS Project

The goal of this project is to use the College Scorecard data provided by the U.S. Department of Education to visualize the trend in undergraduate population of UC Davis. This task is simpler than most Data Science projects, but it is enough to demonstrate our workflow and how to structure a repository. To see clone this project and reproduce these steps clone the repository: `https://github.com/jsharpna/first_project`. First, I created a folder called `first_project` (which will serve as our root directory henceforth), added a `/README.md` file to this directory which describes the project and the file structure in markdown. In the README file, I include instructions to get the data, which can be found at `https://collegescorecard.ed.gov/data/` (which is a 296 MB ZIP file). You should create a data folder `\data` and then unzip the data in that directory (you can also instruct git to ignore that directory by adding `\data` to a `.gitignore` file).

I will be collaborating with Barbara on this project via Git, so let's initialize a git repository with `$ git init` in the root directory. Then you can add the README with `$ git add README.md`, and make the first commit with `$ git commit -m "readme"`. I am using a remote repository, hosted on

github.com, so I also created a new repository there, added Barbara as a collaborator, and ran the following:

```
git remote add \
  origin https://github.com/jsharpna/first_project.git
```

Finally, I pushed my changes with `git push -u origin master`. Throughout, the development process Barbara and I commit and push changes to the repository frequently, looking over each others changes, and providing informative commit messages (after the '-m' flag in the commit).

Typically large public datasets have data dictionaries provided with other documentation for the dataset in question. The College Scorecard data documentation can be found at `https://collegescorecard.ed.gov/data/documentation/`, and there you can find that the undergraduate enrollment variable is labeled `UGDS`. After unzipping the files in the College Scorecard, I explored the dataset in the shell, for example, the head command we obtain the following:

```
$ head MERGED1996_97_PP.csv
UNITID,OPEID,OPEID6,INSTNM,CITY,STABBR,ZIP,ACCREDAGENCY,
INSTURL,NPCURL,SCH_DEG,HCM2,MAIN,NUMBRANCH,PREDDEG,HIGHD
...
```

The CSV files here begin with a header, which is the first line, and I am focusing on the UGDS variable. To find UC Davis, I will use the grep command:

```
$ grep Davis CollegeScorecard/*.csv | grep California
CollegeScorecard/MERGED1996_97_PP.csv:110644,00131300,00
1313,University of California-Davis,Davis,CA,95616-8678,
...
```

This seems to filter out the rows with only UC Davis, and so I will save this as my preprocessed data. I will include this short script in `/code/makefile`:

```
munge:
  grep Davis ../data/CollegeScorecard_Raw_Data/MERGED*.csv \
    | grep California > ../data/davis.dat
  python data_munge.py
```

I will also run this command so that I can work with davis.dat.

Now I am ready to start coding in Python, and I will start by opening an IDE such as PyCharm. We will not go into the details of Python code in this chapter, but will discuss the basic structure of the script `data_munge.py`. The IDE allows me to test code as I write it by providing a Python shell where I can run commands like the following:

```
> davis = open('../data/davis.dat','r')
> line = davis.readline()
> line.split('.csv:')
['../data/CollegeScorecard_Raw_Data/MERGED1996_97_PP',
 '110644,00131300,001313,University of California-Davis,
```

```
Davis,CA,95616-8678,NULL,NULL,NULL,NULL,NULL,1,1,3,4,1,6
...
```

I will see that this script separates out the filename from the data for each record involving UC Davis. At this moment, I clean up the code and create a function with a docstring that outputs a list of these split strings, adding it to `data_munge.py`. The function looks like the following:

```
def extract_davis(filename):
  """separate the filename from the data in davis.dat"""
```

The string below the function definition is a docstring, and typically you should use something brief for a small function and something longer that lists the arguments and output for a larger, more important function. Once I am satisfied with my work, I add the files that I modified, `data_munge.py` and `makefile`, commit and push with Git.

Barbara and I start working concurrently on `data_munge.py`, we both add functions for different tasks. Barbara pushes her work to the repository before me, and when I go to push there is an error indicating that I should pull first. I run `git pull` and I see that Barbara has added a new function, `get_index`, and also added in some code below the functions which tests does some of the preprocessing. I remove any conflicts in the parts where we both worked, then commit and push again. We continue this process until we have a script that can extract the undergraduate population and years and saves those in an intermediate data file.

Now we are ready to present the results of the project, and we will visualize and explore the previous data file. I started a Jupyter notebook with the command `$ jupyter notebook` in the `/notebooks` directory, which I renamed `analysis.ipynb`. As an introduction, I included a description of the project in markdown cells at the beginning, including the data source and the purpose of the project. I added a code cell which produced a visualization describing the increase in undergraduate population, and concluded with a markdown cell. Finally, I added the location of this in the README file, added the notebook with Git, and committed the changes.

1.4 This Book