**Chapter 2: The** UNIX **Shell and Command Line; First Two Data Science Exercise**

**Section 2.1 Overview**

Chapter 1 should be regarded as setting the framework for things to come, particularly Figure 1 since most of all the exercises are designed along that work flow. These exercises are meant to give the student/reader practice in way to interrogate real world data sets, all of which have noise at some level, and all of which will require a fair amount of data munging to get them shape for further processing. Initially, you should try the first few exercises using any tools and software that you like to navigate through the exercise narrative. Indeed, it is important to self-discover tools that are efficient and applicable to the problem at hand from those that are inefficient and cause you to waste time dealing with trying to solve a problem with the wrong tool. In general, it is assumed that the reader will discover that writing and executing code and instructions on a UNIX/Linux machine is the basic platform for doing data science. There is a huge amount of ON line support and discussion forums on this topic so at any given time of you encounter difficulty (and you will) with any part of any exercise perform a Google search on something like:

*How the hell do I do this task using this tool with this code?*

However, the more specific you are the better. For example, related to exercise x.x you might ask the question:

*How do I convert a column text file to an image in python?*

From that you will discover the existence of many tools, one of which is a built-in in scipy (scientific python). This is an example of efficiency as you can accomplish this basic task in two lines of code:

import scipy.misc

scipy.misc.toimage(image\_array, cmin=0,0, cmax=some#),save(‘outfile.jpg’)

Here image\_array refers to the filename that you have given for the input text file. The lesson here is before you do some task using only the tools that you know search for alternative ways to do the same task, try different tools, and learn the most efficient tool that you can use in a way that you best understand. In that way, the next time your boss tells you to covert some text to an image, you can do it in less than a minute. Tools are constantly evolving and as a data scientist there is never any reason to live in the legacy world.

In this chapter, we start out with covering the basics of Linux/Unix operating systems (OS) and the shell interface to the OS. We introduce many of the basic shell built-in commands that can be used for a lot of file manipulation instead of having to write a program to accomplish these same tasks. While this overview of shell operations may seem primitive to advanced users, experience shows that there are plenty of young people today that have virtually no knowledge that the shell environment even exists and moreover, can be quite useful. While not required it is nevertheless beneficial that a data scientist has a working knowledge of the OS in which they are working. The material presented here is a quick overview that should allow the reader to practice and to gain the requisite working knowledge to be more efficient.

Similarly, suppose that you don’t have access to a Linux based system, what do you do? Note that if you’re a student at a university your campus it is very likely to have HPC resources that you can get an account on. To you, these HPC resources will simply appear as a UNIX shell. The basics of a UNIX shell or given in the next section. If you want to run Linux on your local machine there are many products out there (Google for them) but in particular we recommend these:

VirtualBox: This free software has been available for the last 5 years and will give you a fully operating Linux based system on your PC hardware. Like all products it has some flaws but this one we recommend as the most robust. There is a large support community for this product which can assist you with any difficulties.

Cygwin: This has been around for a very long time. It does not provide a full OS but does offer a simple shell interface for command line operations. To learn basic shell commands like *grep, awk, sed*, as discussed in the next section this is most easily accomplished by installing Cygwin.

Mac: Starting with Mac OS 10 (Mac OSX) a command line became part of the system although many Mac users do not know this. A straightforward tutorial on how to use this feature of Mac OSX can be found here: http://blog.teamtreehouse.com/introduction-to-the-mac-os-x-command-line - while this tutorial is dated, it is still highly accurate. Go to the tutorial; read about it and in 10 minutes you will be exercising shell commands on your Mac. In this world, it is far faster to read the instructions rather than thinking you can figure it out on your won. So, read the instructions.

This chapter also presents the first two data science exercises: a) about Atlantic Basin Hurricane behavior since 1900 and b) about the Arctic Sea Extent going back to 1859. Both exercises are good examples of a raw data set that needs to be formatted for analysis and that lends itself to all kinds of data exploration and we offer a structured set of steps for doing this exploration. Much like the newly developed Kaggle competitions in data science (https://www.kaggle.com/competitions) we will offer a website where readers of this work that are working through these data exercises can upload and discuss various parts of their exploration. Indeed, we include some example outputs of the Atlantic Hurricane Basin exercise and sea ice extent exercises that have been done by previous generations of students that you should use to compare against your results. When attempting the first exercise prepare to become frustrated with the time it takes to tackle the various steps but realize that you are on a learning curve. More importantly, realize that any particular data strategy that you develop can carry over to other data exercises to come. Code should be general re-usable and never specific. Initially you should try this first data exercise with any tools that you currently know how to use (including Excel) to discovery for yourself which tools lend themselves efficiently to dealing with which steps in the exercises. You are strongly encouraged to generate relevant plots outside of normal tools like Excel, Matlab, Mathematica, etc, and learn to use a new suite of tools such as those in the Google Chart Library, charts.js, Highcharts.com, etc.

Chapter 1 was concerned about the big picture ideas and trends of data science and the careers of data scientists. Equally important, this chapter is concerned with basics of data manipulation via UNIX shell commands and simple programming. Indeed, at the start we assume that the reader has little or no programming experience but that reader can still be successful at doing many of the exercises. This also means that some treatments of the various tools of programming and data science will be at a very low level – tolerate that – remember the value of this work is you will be given lots of interesting exercises to work through using data that you have never dealt with before. There is an important philosophy here that may distinguish this treatment of data science as different and perhaps more relevant than other treatments: 1) you can definitely become a data scientist without much initial programming experience – indeed this foundation for the construction of the figure x.x Venn diagram which emphasizes domain knowledge (acquired through an undergraduate degree) as on the of prime ingredients in data science.

Domain knowledge means that you can ask intelligent questions about the data – programming is merely a tool that you apply to datasets to deal with your questions. You can learn how to use and apply tools; it’s much more difficult to learn how to scientifically think without having some background of domain knowledge. The exercises in this box serve as example guides to data exploration that also require thinking. Sure, you can be an expert programmer and be facile at clever code and clever scripting, but does that mean you understand data, understand science, or able to ask questions to guide your coding? That profile also needs practice in real world data problems. So if you’re an undergraduate reader without much programming experience, do not be immediately discouraged – these days you can learning programming faster than you think but you need to be committed to learning. My advice is that you decide to take a summer and learn programming, perhaps taking a programming class at a local community college and not a university computer science department. During that summer you need to practice, fail and rebound – rinse and repeat and by the end of the summer you will be quite surprised on how competent you have become.

**Section 2.2**

**The UNIX Command Shell**

Any user interaction with any "super computer" will require using a command line interface known as a shell. This will be the means that you login to that machine and run various commands, submit jobs, etc.  The following guide and examples are based on using VirtualBox as the UNIX interface. For this example, the name of the VirtualBox is shells. A primary advantage of VirtualBox is you will have access to X-windows, an ancient technology that delivers a windowing interface from the host computer to your local platform. This is particularly useful if you want to generate graphs from some utility on the host machine and view the results on your local machine (called the client).

After you have a UNIX shell up and running on your machine you can start trying some basic commands. Note that there are many different kinds of shells and each shell has its own set of built-in commands with their own syntax, although many commands span over many shells. Again there is much on line support for any particular shell. In the examples below everything that appears in ***italics*** are the commands your will type into the UNIX shell.

For the purposes of this tutorial you do not need to know or use any text editors for files (this will be covered at the very end of this tutorial) as we will be using some command line operations to put text or characters into a file. Also, in this tutorial we will use the C-shell (csh); the default shell for VirtualBox is the Bourne Again Shell (bash). To change shells simply type *csh* in your Unix terminal. There is no one preferred shell for all and there are many different kinds of shells available; the most commonly used are the bash shell (bash), the Bourne Shell (sh), the C-shell (csh) and the T-Cshell (tcsh). In general, each of these shells can be switched to the other in any Unix command line terminal by simply typing in the short hand version of the shell name (e.g. tchs, sh ..)

**Section 2.2.1 Some UNIX syntax and hotkey basics**

* Execute commands by typing the name of the command and hitting return (or enter)
* Unix is case sensitive; Ls, LS, lS and ls would all be interpreted as different or invalid commands. In this case, ls is the correct command.
* Spaces matter in the UNIX command line; For example, *ls –l* is valid but *ls – l* is not valid and will return an error.
* man *commandname* brings up help pages for that command. . For example, type this command in your shell: *man ls* to return the help pages for that command. UNIX help pages are notoriously opaque and are best used by looking at the provided examples. But don’t worry, lots of forums exists to decipher Unix help pages into more understandable language
* If you have made a typo, often the easiest thing to do is hit CTRL-U to cancel the whole line.
* If your feeling totally out to lunch then hit CTRL-C which terminates the current running command and returns you back to the shell prompt.
* Arguments (flags) are things that are input into the command; for instance *ps –afe* feeds three flags to the ps command and returns some information about processes running on your system.
* Be aware that your backspace and/or delete key may not work as expected in some shells and the arrow keys might also not work as expected,

**Section 2.2.2 The UNIX Directory structure**

The UNIX directory structure is completely hierarchical but also very logical. One of the most important things about UNIX is knowing the directories in which your files are located. UNIX retains the term directories for these file areas but you are probably used to them being called folders. The UNIX *find* command (see below) is your best friend for this case; learn to use it. The top level directory is the root directory which is denoted by /. Under that directory there are systems level folders, which the normal user cannot edit, and user directories. The top level directory in the Users area is generally known as the home directory or Home. Figure x.x shows the Unix Directory Structure that may be relevant to accompany this particular treatise on data science from the perspective of user bigmoo. In this figure, the directories that you have access to (because you created them – see below) are colored in green. Also pay attention to the use of dots (..) or dot (.) in all the commands below. The use of this character is one of many shell typing shortcuts that are available.

When you use the *ls* command (see below), any names with a / at the end means that name belongs to a directory and not a file and represents a subdirectory in that higher level directory (folder). To become familiar with this overall directory structure, perform the following sequence of operations where we assume that you start in your top-level directory (e.g Home)

* *mkdir bigmoo* this creates a new directory named bigmoo. The use of the ls command would then show this as bigmoo/ .
* *cd bigmoo* cd is change directory and executing that command goes to the directory named bigmoo.
* *pwd* stands for print working directory and lists what directory you are currently in. This is a useful command when you don’t know where the hell you are.
* *cd ..* (two dots) moves you back up one directory cd . (one dot) does nothing but keep you where you are)
* *cd* puts you back in your home directory no matter where you are in the system
* *cd* ~username puts you in that username’s top directory which will have read access only. So if there are other users on your system you can spy on them to see what they have been up to. More practically there might be files that they have created that you want to copy to your own directory and edit them. For example *cp ~nuts/secrets/first\_date .* (dot) would copy a filename in user nuts directory structure called first\_date (located in the subdirectory called secrets) to your account. Note the appearance of the . (dot) at the end of command; the use of the dot completes the copy statement by telling the system where to copy the file to, in this case the directory from which the copy command was issued. If you wanted to copy that file to another directory (say temp in the above directory tree figure) and give it a new name the command would be this:

*cp ~nuts/secrets/first\_date /home/bigmoo/data\_science/downloaded\_data/temp/newname*

* *cd* *~/papers* would take you to the subdirectory papers in your home directory.
* *rm –rf papers* would forceably remove all the files in the directory called papers; you should always use *rm -rif papers* to protect yourself from stupidity.
* *rmdir junk* in principle removes the directory called junk but this is generally a useless command since it only works on directories with no files in them. So do this:
  + *mkdir junk*
  + *cd junk*
  + *touch one*
  + *touch two*
  + *ls* (which shows you have two files in the directory junk name one two)
  + *cd ..*
  + *rmdir junk* and this command will not work returning the error – Directory not empty; hence use the *rm –rif* approach above to remove directories with files in them

**Section 2.2.3 Basic list of UNIX commands:**

Try all of these in your shell login. Commands are issued when you see the shell prompt which usually will end with a $ sign:

* *ls* lists your files
  + *ls –l* lists details of files
  + *ls –a* lists hidden files as well
  + *ls –tl* lists files in date order with details
* *more filename* shows the first part of a text file on your screen; space bar pages down; q quits. The alternative to the more command is *less* which seems to be preferred these days over more.
* *mv file1 file2* moves the file called file1 to a new file called file2; this is essentially a renaming command as file1 is destroyed in this process. Similarly *mv dir1 dir2* would be a way to rename an entire directory (folder).
* *cp file1 file2* copies file 1 to file 2 (now you have two copies of the same file); *cp dir1 dir2* is the equivalent operation for directories.
* *rm file1* removes the file called file1 from the system. Since there is no easy undelete command in UNIX you will want to use the flag i as in *rm –i* to interactively delete a file by asking you if you really want to delete it. This protects against stupidity and the rm command can be aliased to automatically do this (see section below on aliases – for the UNIX experts out there, yes I once did the supremely stupid UNIX thing by typing, as root, rm \*); *man rm* is a really good idea to do!
* *wc file1* returns how many lines, words and characters there are in a file. So for instance if you want to know how many lines of text there are in some big text file of data executing the wc command is a whole lot more efficiently than loading that text file in to Excel and counting the number of lines.
* *touch filename* modifies the timestamp of a file and can also create a new filename of 0 length; *touch bigmoo* will create a filename called bigmoo in your directory.

**Section 2.2.4 File management and pipes:**

* | is the unix pipe command - can be used as many times on the command line as you want; try *ls -lt | more*
* is the shell wildcard; *ls \*.txt* would show all files ending in .txt
* *cat file* lists the contents of a text file to the screen continuously – this is silly to use on big files so use less or more or a pipe (|); *cat file | more* will route the cat command through more command and give you a page listing the file:
  + *cat file2 file2 file3 > outfile* will add these three files together to create a new file called outfile
  + *cat \*.txt > outfile* adds all .txt files together
  + *cat edit.txt >> project.txt* will append the filed called edit.txt to the end of the file called project.txt
* *grep* or *fgrep* will find regular expressions (usually text) in a file name; *cat file1 file2 file3 | fgrep moo* will find which file contains the expression moo.

 find is a highly useful feature that allows you to find specific files;

*find . -name \\*.f90 -print* | *more* will find all files in all subdirectories starting with the directory you're in that have extension of f90 (Fortran code default extension) and print them on your screen. If you have a lot of these then pipe it through more. Always best to run this in your home (e.g. top) directory. Note the \\* is necessary so the find command properly recognizes the \* as a wildcard character.

 file provides information about a particular file (whether is binary, text, image, etc e.g. *file filename*)

**Section 2.2.5 Shell Shortcuts:**

It might be more efficient to go through life minimizing keystrokes. For example you really don't want to name a file this\_is\_what\_I\_did\_today.txt but suppose that you did. How do you avoid typing out the whole filestring each time you want to do a file operation; for instance removing (rm) the file. Well let’s try this by way of example:

* *set filec* – allows for autofile completion by tab (not if the set filec commend did not work, then you’re not using a C-shell; type *csh* to switch shells.
* *set history = 16* – this is the C-shell history command and recalls the last 16 command lines that you have entered
  + *mkdir keystrokes*
  + *cd keystrokes*
  + *touch this\_is\_what\_i\_did\_today.txt*
* now type *history* to see the last 3 commands listed in numerical order
* suppose you want to repeat a command then type !1 to repeat the first command (now of course you have made another subdirectory)
* to repeat the previous command type !! so type *ls* to get a directory listing and type !! to repeat that same command – not that depending on terminal emulation the up arrow will also repeat the previous command
* now since you have set file completion (e.g. the set filec command) now type *cat this* followed by a tab
* suppose you want to fine the type of this file using wildcards and just a few letters: *ls \*tod\**
* *wc this* tab completion will then run wc on that unnecessarily long filename
* Now suppose you have a series of file names named dataout1, dataout2, dataout3 … dataout101. Some of these files might have the keyword monkeygoats in them so you might run this command on dataout17: *cat dataout17 | fgrep monkeygoats* . No you really don’t want to type this 101 more times so we can use the ^ character for command line substitution. Let’s create 3 files but not with touch but using the echo command which will actually now place content in our files:
  + *echo 23skidoo > dataout1*
  + *echo skidoo23 > dataout17*
  + *echo monkeygoats >dataout51*
  + *cat dataout17 | fgrep monkeygoats* (nothing found of course)
  + *^17^51* (this has now changed the occurrence of 17 to 51 in runs the cat command again with that change).

So, if you had 50 files with the names day1, day2, day3, …day 50 and wanted to search those files for a specific expression then you can see that the use of this command line substitution shortcut would make that operation go quickly. Now of course you could always write a program to do the same thing but developing expertise in shell built-in file management commands and shortcuts really does save time in the long run and is particularly useful set of tools to employ during the data munging phase of the data science process.

The use of wildcards and file completion suggests that you ought to think about how you name filenames in such a way that allow for rapid interrogation and manipulation using command line procedures and short cuts. Files and directories should be named in a way that makes its own accounting system where the filename reminds you of what is going on.

**Section 2.2.6 Aliases and setup files**:

This is another convenient thing to set up as a short cut. I will use the C shell example which makes use of a file called .cshrc --- any "." file is a hidden file that can only be seen with the ls -a command. The .cshrc file contains a list of aliases that are shorthand to run various commands*.* For example a very good alias is this alias rm "rm -i" this allows you to now, by default, feed the -i flag to rm. The .cshrc file is run at login and sets up all the commands. For example, suppose I have the habit of typing more as mroe for some strange reason. Now I can either try to correct my erroneous ways or take advantage of them by entering this in my .cshrc file. Follow this example from within your top directory – note the .cshrc file is always located in the top directory as the system searches your top directory for executable files to execute at login:

* *echo “alias mroe more” > .cshrc* - the use of the quotes is necessary to populate all three words into the filename .cshrc
* *ls* – You will not see the file you just created because it has a . in front of the filename so it is hidden; type *ls –a* to see the directory listing that now shows this hidden file
* Now you could log out and log back in to execute this newly created .cshrc file which contains the one command but instead type *source .cshrc* to “source” that file for the system (a useful alias is *alias so source* – saves typing). In this way any edits and updates you make to the .cshrc file will become available system wide upon sourcing it.
* Now type *alias* to generate a list of aliases and you should see the alias that you just added. Work through this example:
  + *ls | mroe* should now work like it’s supposed to even with your inbreed erroneous typo – so you can correct your own bad habits.
  + *echo “remind me to do DATA SCIENCE today” > remind*
  + *alias tt “cat remind”*
  + *tt* – so if you put that alias in your .cshrc file, then every time you logged and typed *tt* you would be reminded that indeed today you need to do data science. (advanced user here: if you are root on your system then you can make a file called motd in the directory /etc which will broadcast the same message of the day to all users upon login)

In general, the use of the alias command is an excellent way to reduce the number of keystrokes that you commit during the course of your working day. A summary of much that was written above is contained in this low production value YouTube video:

https://www.youtube.com/watch?time\_continue=3&v=8gvHFUG-D3s

**Section 2.2.7 File Operations and file permissions:**

Files have three kinds of permissions, read (r), write (w) and executable (x) and three kinds of owners: a) you the user/owner, b) a group (this is generally not applicable) and c) the world. Any file which is world writable means that anyone can delete it - however, you have to go out of your way to make a world writeable file. The chmod command is how you modify permissions and you should man chmod for that help file. You can use chmod with just numeric input (see man page) but I find it more intuitive to use the oug convention (other, user, group). For example,

* *echo “permissions are confusing” > confused*
* *ls –l confuse* – this lists the details of the file name confused and the first field contains 10 characters in it; for the file confused this should read –rw-r-r—r-- ;
  + the first character will likely be a – or a d indicating whether the file is an ordinary file or a directory (some Linux OS’s will give more detail than just – or d)
  + The next three fields are permissions for the file owner which means you can read to it or you can write to it (meaning you can edit it)
  + The next three fields are group permission which are read only (again group permissions are not very relevant to the average user)
  + The next three fields are the world permissions which again are read only
  + Now type this *chmod og+w confused* followed by *ls* – you have now done a silly thing because your permissions are now –rw-rw-rw- meaning that anyone on the system can edit your file.
  + Nowhere yet do you see an x but instead of an x there is a – which means that, by default, this file is not executable. Now suppose this text file is actually a script to run commands (covered later) you will need to change that to an executable file. Suppose we want others on the system to also be able to execute this file but not edit it; *chmog og-w confused* (to turn off the silly thing you did) and then *chmod oug+x confused* now makes the file world executable with a permission string like this –rwxr-xr-x

**Section 2.2.7 Shell Scripting**

There are myriad of things that one can do with a shell script which is a shell specific executable for running a bunch of commands. People that are good at shell scripting a) are much more efficient mangers of time on any given project and b) generally able to have very flexible job skills. Scripting is a very much underappreciated tool that can be used in data munging and data organization. There are at least a billion shell scripting example web sites but this one is quite straightforward: http://parallel.vub.ac.be/documentation/linux/unixdoc\_download/Scripts.html. Note again that learning data science does require a willingness to learn a variety of things and even if you think you already know how to do something you can always learn how to do that something better. In general shell scripting is a good an efficent way for doing lots of different kinds of file operations through the use of foreach or while loops.

A simple example to be done in a C-shell is as follows:

* Make a text file called rr using any editor (see editor section below)
* In the first line your need to put in #! /bin/csh -this tells the system that this file is an executable shell script. Now within that file edit in the following lines
* *echo “i am reading and really really like this treatment of data science”* > BigFan
* *wc bigfan*
* *echo “oh good, more royalties” > cc1*
* *cat cc1 >> bigfan*
* *more bigfan*
* *mailx –s “data science” tandfds.gmail.com < bigfan* (this will mail the file to the author of this book! – it is also possible that your particular version of UNIX/Linux doesn’t support mailx any more – but this is an example of how to send a file in email just on the command line.
* *rm BigFan* – this means that after you have run the script the evidence is gone; BigFan has been deleted.

Now you have created this script and you need to execute it:

* *Chmod oug+x rr*
* *./rr* – if you just typed rr this doesn’t tell the system where the executable file is – the use of ./ tells the system that the executable file lives in the directory you are currently in. For better organization in your top level directory you might create a subdirectory called scripts and place all your scripts there.

**Section 2.2.8 Command line file operations**

Now we are ready to start some data science via an exercise in data munging. There are some basic command line operations that can greatly aid in file organization; these include:

* sort – this allows extracting different fields (columns) in a data text file and writing those fields out to a new file
* grep – find specific instances of a term in the text dta
* head or tail – operates on the beginning or ending of a file
* cut – like sort but has more options; cut can extract various lines and columns out of data that sort can not
* awk – another, older, version of cut. Most UNIX savvy programmers are offended by an awk script and strongly prefer cut. But cut is difficult to get to work correctly and often requires time spent to verify that the actual operation was done correctly; awk is more primitive but it is much more straightforward to use.
* sed – the command line editor – this is very powerful and generally underutilized.

There is much available ON line help for how these shell built-ins can be used but again it is much better to learn by example and so this is the first of many data exercises. We now offer a messy input file for you to organize via the shell built-ins that we just discussed.

**Exercise 2.1** – manipulating a messy input file to generate a specific kind of output file

1. We start with a master input file that contains raw messy data – in this case the file is related to Hurricanes and their evolutionary status in 6 hour increments. The file starts in 1900 and goes through 2009 (we will be doing a more extended hurricane exercise later). An immediate problem with this file is that the number of fields per data line varies – in particular the file starts out with the name of the hurricane being NOT NAMED (that is 2 fields) before transition to the beginning when storms are named (e.g ABLE). In addition, the description can have two fields, i.e. TROPICAL STORM or one field, i.e. HURRICANE-1. A portion of this dilemma is shown here where the top line is seen to have 11 data fields and the bottom line has 9.

760 NOT NAMED 42.1 -61.9 45 0 1949 18190.50 EXTRATROPICAL STORM

760 NOT NAMED 43.6 -59.6 45 0 1949 18190.75 EXTRATROPICAL STORM

763 ABLE 16.5 -54.5 35 0 1950 18487.00 TROPICAL STORM

763 ABLE 17.1 -55.5 40 0 1950 18487.25 TROPICAL STORM

763 ABLE 17.7 -5.7 45 0 1950 18487.50 TROPICAL STORM

763 ABLE 18.4 -58.3 50 0 1950 18487.75 TROPICAL STORM

763 ABLE 19.1 -59.5 50 0 1950 18488.00 TROPICAL STORM

763 ABLE 20.1 -61.1 55 0 1950 18488.25 TROPICAL STORM

763 ABLE 2.1 -62.5 65 0 1950 18488.50 HURRICANE-1

763 ABLE 21.6 -63.2 70 0 1950 18488.75 HURRICANE-1

2. Retrieve the data file at <http://somewhere> and name it master1.txt

3. *wc master1.txt* to determine how many lines it contains

3. *grep –I “HURR” master1.txt > new.txt* – this creates a new file with only lines that contain HURR(icane) in them as for this data exercise we are only interested in analyzing systems during their hurricane stage of evolution.

4. *wc new.txt* to see how many lines have now been removed from the old file to make the new current data file. This is an important part of data munging – do not keep irrelevant data in your working file – reduce the master file down to the basic necessary elements to tackle the problem at hand. A common problem in raw data sets is unnecessary columns (fields) of data and you want to remove them before submitting the data file for analysis

5. *sed /s/NAMED//g new.txt > new1.txt* - this removes the word NAMED wherever it appears and is taking up one of the data fields and creates the new file new1.txt. The /g syntax at the end means global substation; if we wanted to substitute the term NAMED for MonkeyGoats then we would have put in MonkeyGoats between the two // .

6. *awk '{print $1 FS $6 FS $7}' new1.txt > new2.txt* – this explicitly tells the file new1.txt to output only columns 1, 6 and 7 to a new data file called new2.txt

Exercises 2.1 continued:

The file new2.txt contains the scientific data of interest in which column 1 is the name of the system, column 6 (now column 2 of the new data file) is the central pressure of the storm (in units of millibars – not measured in the first few years of data) and column 7 (now column 3) is the year.

7. *sort -n -k 2 new2.txt > new3.txt* – this produces the final data file which has been sorted by column 2. We have now trimmed the raw data file into scientifically useful input to some program (see full Hurricane exercise x.x). You should inspect this file with *tail -100 new3.txt | more* to view the last 100 lines.

8. Issuing *ls -l* then will show you all the new files that you have created via the above operations.

9. As a challenge you now have enough skills that you should be able to produce two separate data files, one containing all the hurricanes of the 1930s decade and one containing all the hurricanes of the 1990s decade. This might come in handy when you engage with the full hurricane exercise.

**Section 2.2.9 Some UNIX System Basics**

Often times, especially if your system seems unusually slow; you might want to interrogate the basic system state. Here we give you various tools that will help with program compilation, program running, inspecting the state of the system, killing your jobs or processes.

For system load and performance use the *top* command. (*man top* for more info)

* *ps -afe | fgrep yourusername* --- this will return all of your processes and PIDs (process ID). This is useful if you want to forcefully kill a process via *kill -9* number (where number corresponds to the process that you want to terminate).
* For disk or memory management
  + df -k (system disk capacity)
  + du -k (your own disk usage listed by directory)
* The command *ldd* to run on an executable shows the libraries that are linked to the program. This is sometimes useful if your program can't run because it is missing a library.
* For running Python on you UNIX system there are many options. Note that Python does not come with Virtual Box, you will have to download and install that on your own but this is relatively simple and usually involves a system known as Anaconda: (https://www.continuum.io/downloads). Anaconda now includes a variety of other tools and even promotes itself as a “Portal to Data Science” - it seems to be worthwhile to download and install and again, if problems arise, a support forum exists. On some systems you might have to load python as a module – load module python. But for your own system, typing python should bring it up with the prompt >>> .
* You can also run python directly from the command line using a script: python script.py. Follow this example: Edit a file called t.py and put in the following lines
  1. a=2
  2. b=4
  3. c=a+b
  4. Print c
* Now execute the command *python t.py* to discover that your computer can correctly add two numbers together.

So now you have created an executable file and placed that in some subdirectory on your system. What can you do to access that file when you are not in the same subdirectory for that executable file? We can remedy this by using the PATH environment variable to search for executable files. To being which, the command which filename\_of\_executable\_file will return the path to that file. For example,

* Type *which sed* or *which sort* and the system will return that these files are located in the directory /bin (and here / means the system root directory)
* Type *echo $PATH* to return the current PATH that is associated with your account. Since you have likely not modified this path you see a system default that probably contains /bin:/usr/bin:/usr/sbin:/usr/local/bin and a few things like that.
* From your top directory create a subdirectory called newpath (*mkdir newpath*). Now using the touch command and chmod commands create an executable file called findme. When you do an *ls* you will see findme\* returned where the \* indicates this is an executable file; return to your home directory with *cd.*
* Now type *which findme* to see that the system cannot find this file because you have not yet modified your path.
* Path modification is very shell dependent and is not dependent upon the particular Linux/Unix OS. Since we are using the C-shell here (*csh*) the following example is explicitly for that shell, but the concept holds for all shells. Importantly we do not want to write over the previous path, we want to append this new directory. Furthermore, we want to append to the path not for the specific current shell login, but for any future times you login. This means adding this new path to the .cshrc file.
  + *echo "set path = ($path /export/home/nuts/newpath)" >> ~/.cshrc*
  + *source .cshrc*
  + *which findme*

Now you have succeeded in changing your path and *echo $PATH* will show the newly appended subdirectory (newpath) as the last item in the path string. For good system and file management you do want to keep your various executable files (these are binary files) in a single folder, which by conventions is usually called /usr/local/bin . You don’t really want to be modifying your path very often as mistakes could be made that would delete your entire path.

There is a whole slow of environmental variables that can be set up within the login shell profile and there is much ON line help available. However, the typical user here doesn’t need to be very much concerned with this aspect of the UNIX OS but some documentation is available here:

http://homework.uoregon.edu/pub/tandf/env.pdf .

**Section 2.2.10 Unix File Editors**

Briefly there are a lot of editor choices, some are command line based that work in a simple terminal (like vi, vim, emacs) and others are editors that work in a windowed system (often called full screen editors like nano, pico, gedit). You should experiment with different editors (all of them have a learning curve) and simply pick one that you are the most efficient with. If your running VirtualBox on a Windows system you could, in principle, use windows editors to create a file but you will have to spend some time determining how you Windows file system maps onto the VirtualBox file system. It is probably better not to take that approach and simple learn and actual unix editor. The following example is for the editor vi (esc is shorthand for the escape key, heavily used in vi). This editor should be naturally part of any distribution and located in /bin/vi :

* *vi mycreation*
* *i* – insert text and type in the following 2 lines
  + Apples are orange
  + Monkeys are green
* esc *:wq* - this writes the file and then quits the program returning you to the shell prompt
* Now let’s edit the file again: *vi mycreation* ; if you have set history in your shell then *!vi*
* To add words to the end of the first line type *$* (skips to the end) followed by *a* – now you can type in more words like *and tasty*. Hit esc to get out of editing mode.
* Moving the arrow keys will position the cursor over the characters; Replace the letter n in orange with the letter Q by moving the cursor over that letter and typing *x* (deletes character) no insert (*i*) Q and hit esc.
* Now hit the + sign to advance to the second line; use *dw* to deleted the first word in that line (monkeys) and hit esc; to add a new line hit *$* and *a* and return. Now type in words for a new line like *whales is fat* and hit esc.
* Now hit the – sign to return to the second line and hit *dd* to delete that line – oops didn’t want to delete it, then hit esc and *u* (undelete) to restore the line
* To save this modified file esc *:wq*

Now you may end up loathing vi and thinking it is the most insane editor ever invented but the following the above example will let you create and edit basic files and is a simple way to make shell script files that usually do not contain very many lines. When you become proficient at coding, there are lots of more advanced editors that provide good screen environments for code development and debugging – a good example is the eclipse editor: https://www.eclipse.org/downloads/ .

**Section 2.4**

**A general approach to data problems**

Now it is time to practice on a real data set. For the present purposes feel free to use any tool that you want. For instance, you could try to do this exercise in Excel and quickly find it to be cumbersome. Why? – because many parts of this problem require customization and this is the reason that one writes code, to customize data to a particular task to remove the constraints from the non-customizable black box approach. Before delving into the data, there are a few general rules (similar to the data rules of Section x.x) to cover to serve as a strategy guide:

* Plot the data early and often to get a feel for the results you’re generating and the likely level of noise in the data.
* Use powerful command line procedures for certain data sorts or global edits – like you already did in Exercise 2.1 .
* Build command line pipelines that can produce a graphical output whenever you re-run your code on the data so that new code results are instantly shown to you. For instance, it is very likely that the package matplotlib (https://matplotlib.org) is part of your python distribution so you could, in principle, use a python script and pipe that output to matplotlib to generate simple plots.
* Search for existing tools to adapt – especially for any visualization needs.
* Practice, practice, practice; mistakes, mistakes, mistakes – this is the best cycle to learn data science by application to real data. This is likely not the way that you would be taught in any data science class which is really why you have to work with data to learn data science. This requires time and the reader here has to make that commitment to advance.

The second general aspect has to do with scientific programming which in its basic form always follows the same work flow structure:

* Turn raw, unstructured data into some formatted file structure that turns the data into an array of N rows by Y columns. Be aware of the size of the raw data as quickly as possible.
* Sort the data in ways that are convenient to the problem at hand. Often a good strategy is to never work with the entire data set but only work with a section of that data (stored as a separate filename) for testing various aspects of your code. For instance, if you want to know if your code correctly identifies the central pressure values in the data set, there is no reason to run this test over 100,000 entries when it can be done over 10 entries as long as the line to line formatting is consistent.
* Loop through the data applying conditional statements (like an if statement) to various parts of the data to identify various characteristics. This often called conditional flow control.
* Run mathematical operations on those pieces of the data that satisfy the previous conditions. Often these operations can be done by referring to a library subroutine that you just call from your program. For instance, in Excel (not recommended) if you simple want to find the average of a column of data, you use the built in average function in excel rather than manually computing that average. Python has a large array of various efficient tools in this regard and some of this will be practiced in Exercise 2,2 .
* Produce good visual and organized output for initial analysis
* Now use the science part of data science to think about the new questions that have been raised by your first analysis run and go back to the looping step and change the conditions.

The above procedure is the way that any scientific data set is basically analyzed. The tools that you use to perform these steps are entirely up to you. But your data science is guided by creativity, reflective thinking, and the scientific process much more than it is guided by how you code, what you code, or the coding tools that you use. All of the ensuing exercises will have multiple parts to allow you practice in data organization, management, manipulation, processing, analysis, and representation. All assignments have built-in failure modes and obstacles to overcome because this is the best way to learn this stuff. The expectation is that any reader attempting these exercises will encounter a failure mode – if you easily give up – then data science is not a good career choice and it is better to find that out sooner rather than later.

Finally, there is never any one right way to do the exercise; multiple approaches using multiple tools is what the exercises are designed to encourage. In addition, time management is extremely important and you should keep a log of the time it takes for you to do various aspects of this first exercise. This will help guide you on how to best spend your time and what tools to best use for various parts. Based on previous versions of this assignment given to undergraduate and graduate students a complete journey will likely take about 20 hours if you already have come programming competency. However, it will take longer if you need to develop some competency during the course of trying to complete the exercise. Hence a commitment to learning data science by doing data science is necessary to proceed through these exercises. Indeed, a portfolio of your results (see examples in Section x.x) can be viewed by prospective employers which might facilitate your entry into this career. Another way to think about doing these exercises, and one that would give you another set of skills, is engaging with your friends and forming your own data science team to tackle the various parts of the problem. In this way, you can practice program management, collaborative code sharing (likely using **github** which will be discussed later), and overall time management in synthesizing the various parts of a large and complex data set into coherent output. In fact, we strongly encourage readers to do these exercises via a data team approach. With all that now said, here we go!

**Section 2.4 Your First Data Journey**

**Exploring the Atlantic Basin Hurricane database**

Exercise 2.2 Exploring the Atlantic Basin Hurricane Database from 1900-2017 – the source for the data base can be found here: http://weather.unisys.com/hurricane/atlantic/

Follow all of the instructions below, explicitly:

Step 1: Download the data file (at http:homework.uoregon.edu/tandf/master1.txt.tgz). This file is a gzipped file so you will have to find an application to unzip it - gzip (or gunzip) compresses or uncompresses a file; gzip produces files with the ending '.gz' appended to the original filename

This data is a master data file for Atlantic Basin Hurricanes. Note that the data fields in this file contain both numbers and alpha characters. You will have to deal with this.

As always, run *wc* on the uncompressed file to determine the number of lines in the data file. In this file there are nine columns of information:

* Column 1 is Storm ID
* Column 2 is name of storm
* Column 3,4 is latitude and longitude
* Column 5/6 is wind speed and central pressure
* Column 7 is year
* Column 8 is some running date field in units of hours. This serves as a timestamp for the entire data set and generally each new line for a storm is in increments of 6 hours.
* Column 9 is a descriptor and includes the Hurricane category (1,2,3,4 or 5)

You are to construct a set of program that calculates what all the next steps ask for. To use the science part of data science you will also be asked to make comments on your results - this is what a data scientist does!

Exercise 2.2 continued.

**Step 2:** Referring to some of the tools used in Exercise 2.1 – break up the master file into individual files that contain Hurricane only events as a function of decade (I.e. the 1910s, 20s, etc).

**Step 3**: Determine the average central pressure and standard deviation for all the listed hurricane events for each decade and produce a plot of that average plus standard deviation as a function of decade. A good tool to use to produce that plot can be found at: https://www.highcharts.com/demo/box-plot.

If Hurricane central pressure is a proxy for overall storm strengthen and you work for a company that insists hurricanes are getting stronger with time so we need to raise insurance rates for people on the East Coast, what would data driven decision making imply?

**Step 4**: Calculate the frequency of individual Hurricanes (starting from 1900) (this excludes systems that remain as Tropical Storms but these should already have been eliminated from the decade data files) - in units of number of hurricanes per decade. Produce a plot and from that predict the number of hurricanes in the year 2030. Provide a scientific explanation of the uncertainty associated with this prediction. Once again, your insurance company insists that global climate change results in increasing numbers of Hurricanes. Is this borne out by the data? If not, are you going to get fired because your analysis failed to confirm this expected truth? This is the world that the data scientist must live in by defending the important notion that knowledge comes out of data; knowledge does not come from belief.

**Step 5:** You have been hired by the state of Florida to produce a hurricane impact outlook starting in 2020. From the data, develop an impact parameter which is basically the wind speed of the hurricane when it makes Florida Landfall multiplied by the number of hours the system exists as a hurricane when it’s in Florida. (so now you have to determine if the coordinates of a storm place it in the boundaries of Florida – while this can be done explicitly it is likely more efficient to simply represent Florida by to rectangles because exactness is really never needed when making an approximate prediction.

For your output result make a plot showing this impact factor as a function of time starting in 1950 (note there will be some years where this factor is zero) and extrapolate that plot out to the year 2030. Comment on whether the data shows that the impact factor for the state of Florida is likely to be increasing in the future.

**Step 6**: On a decade basis, produce a Pie Chart showing the respective contributions of Cat 1,2,3,4 or 5 systems.

**Step 7**: For hurricanes that reach Category 3 and beyond, determine the total number of hours a storm spends at that level. Plot the average number of hours per decade as a function of decade? What trends can you notice?

**Step 8:** Produce a histogram for minimum central pressures for the entire set of Hurricane storm IDs**.**  Note that doing this step is a coding challenge in that you have to identify a minimum value for every set of unique storm ID lines and it’s easy to make mistakes here – so you need to verify that your code is actually working.

Exercise 2.2 continued:

**Step 9:** Let's define a standard of 1000 MB and a "hurricane threat" level of 970 mb. Calculate the "spin up time" (the time it takes to go from 1000 MB to 970 MB) for each storm and produce a histogram of spin up times. Compare the average spin up time for the period of 1990-2010 to 1950 - 1970 and comment on any trend you might see.

**Step 10:** From the histogram of central pressures, determine the central pressure that is reached only 5% of the time and generate a list of named hurricanes and the year in which they occurred. Find the fastest 3 systems in terms of spin up times from 970 MB to this 5 % threshold and report on those systems.

**Step 11**: It is well established that the peak activity of Atlantic Basin Hurricanes occurs around Sept 10. Using the timestamp data convert that to month and day (there are a variety of ways to do this) and plot a histogram of the times when a particular storm has reached its maximum intensity as defined by wind speed in this case (not central pressure). This histogram should confirm the Sept 10 expected result.

**Step 12:** Define a latitude box that represents a possible intensification zone SouthEast of the Caribbean. Define this box as:

* Latitude 15-22 degrees
* Longtitude 50-65 W degrees (make sure you get the longitude correct)

a) Count the number of storms per decade that go through this box and make a table of that shows these events. Comment on any possible decadal to decadal differences and speculate (e.g. scientific thinking) on what might be the cause.

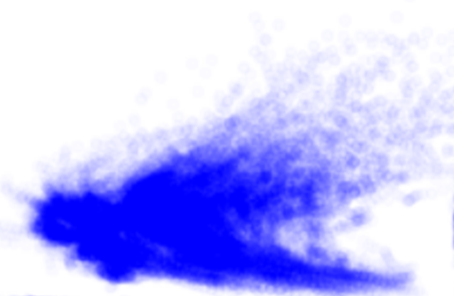
b) Using a bubble chart where the radius of the bubble is Cat 1,2,3,4 or 5 at their maximum category intensity - plot all of the storms that go through this box where the Y-axis is latitude and the X-axis is year starting in 1950.

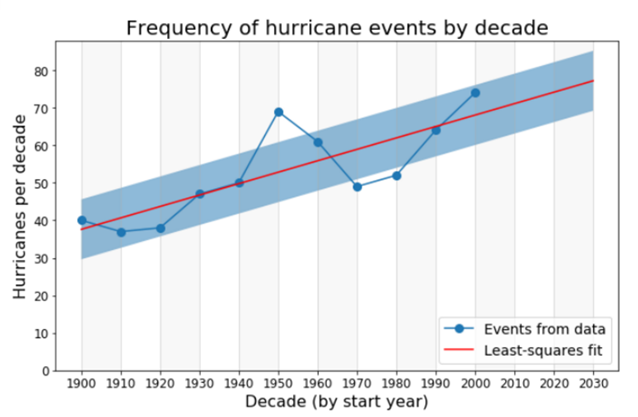
c) Apply Poisson statistics to this box to see if any decade is unusual compared to the average rate. Poisson statistics is not very hard and is discussed, with examples, in the statistics section of this work.

**Section 2.4.1**

**Some results from the analysis of Hurricanes**

Congratulations, you have just completed your first intensive data exercise. For the most part you have duplicated an extensive analysis done by others (Google on HURDAT) to find this. Most importantly, you have shown that the various aspects of the data do not strongly support the notion that the frequency of Atlantic Based Hurricanes is increasing with time, contrary to wide spread media reports every time a new Hurricane hits the US (e.g. Hurricane Harvey, August 25, 2017). Now of course is the data science challenge of presenting these results in a way that leads to better data decision making as well as better informing policy makers about this natural phenomenon. In this section (and yes you can use this to compare your results against) we show some of the results that can be generated; these results refer to the time period 1900-2009 but nonetheless should be similar when the last 8 years of data have been added. A side note here: this kind of exercise provides a good example of crowd sourced science as perhaps one of you or your data team will discover results that no one has noticed before.

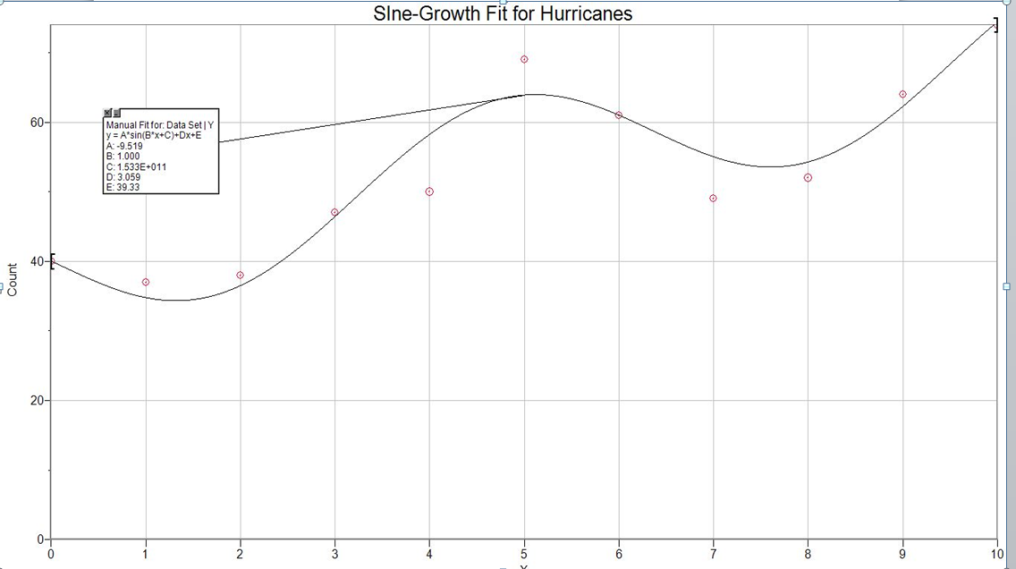
Although not asked for in the assignment, it is possible to produce an image that shows the density of hurricane tracks as a way to visualize the entire data set to use as a reference. An example image, done in matplotlib, is shown here.

A black box approach, meaning we will turn off the science part of our data science brain, with respect to hurricane frequency vs decade is shown here, were clearly the red line projects a bad future. But is this result sensible and should we believe it? Well our science mind tells us there are two possible problems associated with this representation:

a) Maybe hurricanes were more difficult to detect in the past (< 1940)

b) Maybe the data is cyclical in nature and a linear fit is inappropriate.

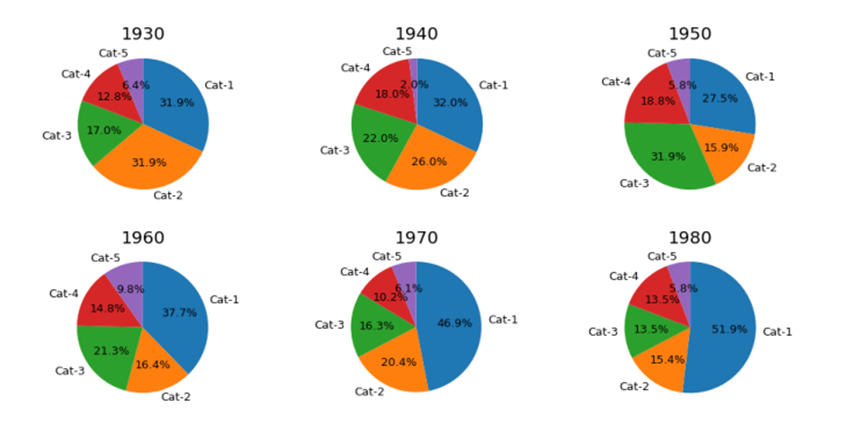
These are the kinds of questions the data scientist must ask and investigate so that the policy world does not have a Pavlovian response to the red line of doom ending up with the result of how data science can produce misleading information.

A better approach results from the application of domain knowledge and realizing that hurricane formation and evolution may be somewhat cyclical in nature with certain decades being more active than other decades. So, let’s code up a sine wave fit to this data because that kind of fit is not an option in our black box of curve fitting choices. The results of a sine wave fit are shown here -yes it still is trending upward but extrapolating the sine wave would then produce a downward trend in the decade to come. The data problem here is that the sampled time span is only equal to approximately one period, and you need several periods (well definitely more than one) to better characterize the situation. This all means that the honest data answer to the question of whether or not hurricane frequency is increasing is not well known – and sometimes or even many times, uncertainty is the principal outcome of data science as applied to real data.

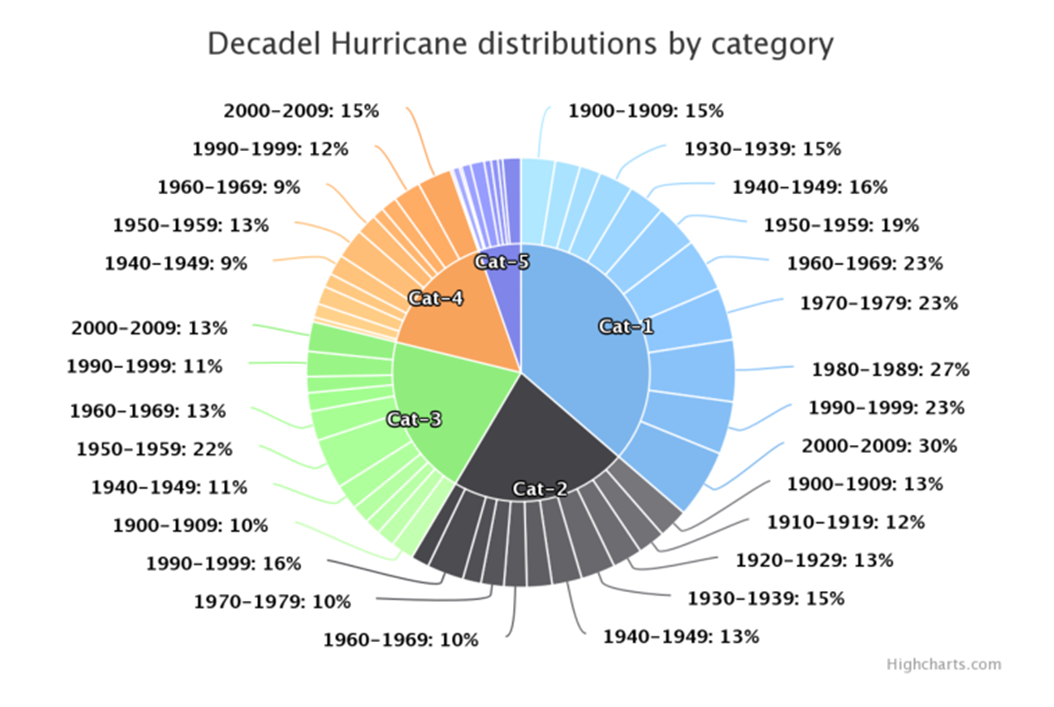
For the question about impacts, the kind of plot that you produce is important to the best kind of interpretation as shown below.

|  |  |
| --- | --- |
| The linear fit black box approach would strictly predict a slowly rising impact parameter in the future but clearly a linear fit is inappropriate and predicts a much less dire future than is actually likely to happen in Florida. | The histogram more accurately captures the strong fluctuations in this parameter where the data qualitatively suggests a spike in activity every 20 years or so followed by relative quiessence. So the most appropriate prediction would be another spike of activity sometime around 2025. |
| For determining storm events in Florida there a multitude of open source geomaps now available to be used as a plot reference. So again, in doing this project if you wondered “gee did someone already make a map of Florida that I can put points on” - well yes, Google on that to discover these resources because discovering new tools is an important part of practicing data science that can improve overall efficiency. |  |

When producing visualized output it is important to choose the kind of visualization that is easy to understand. You can spend a significant amount of time on the style over substance issue and end up producing an overly complicated graphic which takes a longer time to process. The production of pie charts in response to Step 6 of the previous exercise provides a good example, as shown below. Below is a traditional kind of pie chart for some of the decades in question. While there are six individual figures this information can be processed relatively quickly – for instance which decades have the highest percentage of Cat 1 storms – in about five seconds your eye-brain has identified the 1970s and 1980s.

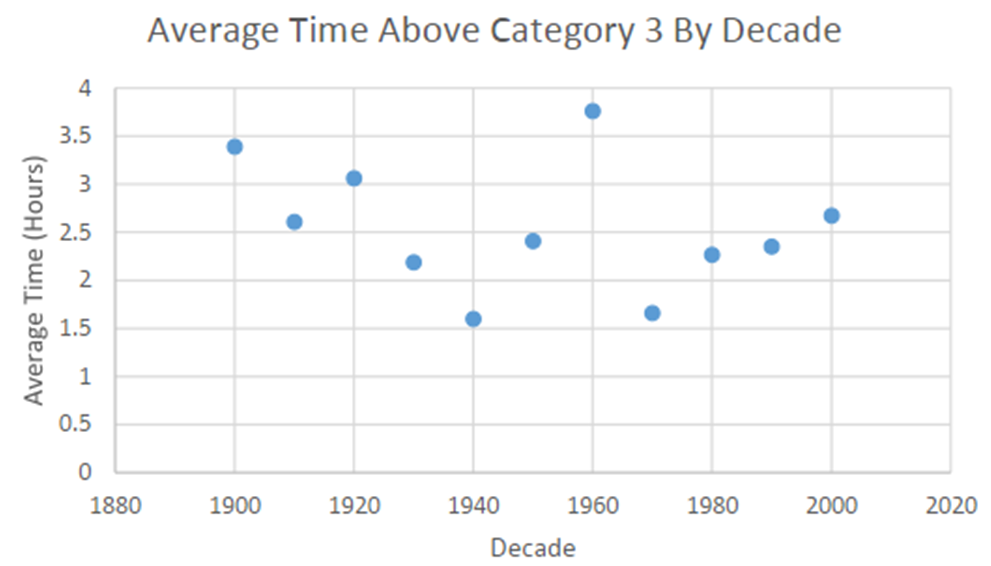


A much fancier pie chart alternative (produced at Highcharts.com) in which all the information is incorporated into a single graphic is this one, but you have to look at this for a long time to understand the information:



So when producing visualized output you must think about your audience. Yes the maker of the above figure probably does understand this format, but if you have to train your audience to process information in a new, non-intuitive format, then that is probably counterproductive. So while slick graphics might have sex appeal for the superficially minded, you the data scientist want to rapidly communicate the substance of data exploration so as to facilitate data driven policy.

Regarding step 7, the result that you should have found is actually similar in form to the event distribution in that there is considerable decade to decade fluctuation and no trend is clear.



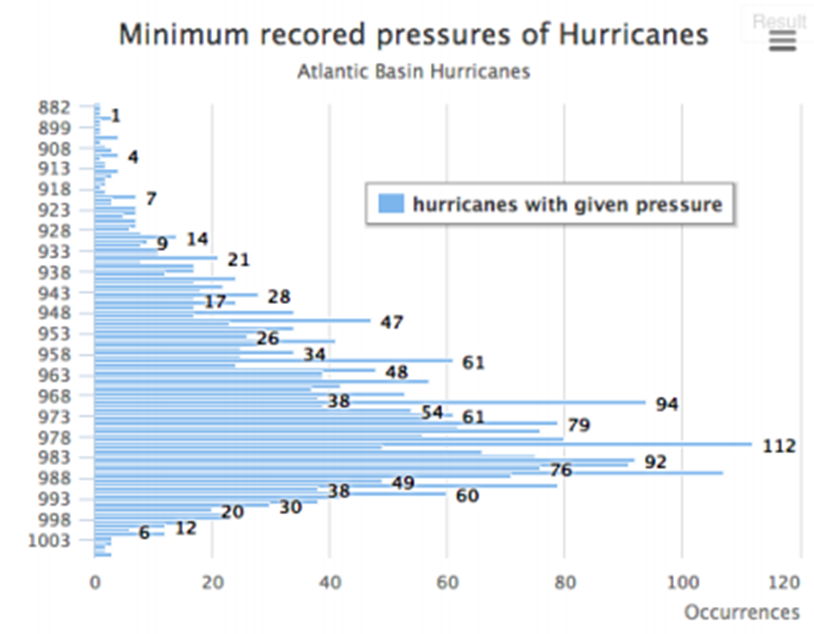
Generating histograms is another area where there are multiple approaches but, as will be emphasized later, the appearance, and hence the communication of an external audience, of a histogram is very much affected by how the data is binned. So you can subvert your audience into believing things that aren’t really there in the data but manipulating the nature of the histogram. In the table below we show four different versions of histograms for the same data.

* Panel A: Has too many bins so that the data has been over resolved and you do not visually see a continuous distribution of bin counts.
* Panel B: Has too few bins (data is over-resolved) but clearly shows the tail to very low pressures
* Panel C: Like panel B but has added the mean line (green) and the line which is 3 standard deviations () from that mean
* Panel D: Gives two representations of binning in one plot.

|  |  |
| --- | --- |
| Panel A: | Panel B: |
| Panel C | Panel D |

Note also (again covered in the statistics section) that the standard deviation () per bin is simply given by the square root of the counts in that bin. This can be used as criteria to test levels of significant difference in various bin counts. Note finally that this distribution is not normal since we have applied a threshold condition (e.g. 1000 MB) – nonetheless you can still calculate a mean and for this distribution. However, because the distribution is highly skewed the application of Gaussian statistics is not very meaningful. Indeed, if you wanted to define the 1% level for Hurricanes from this distribution it is probably best to use the data rather than to set a level which is 2.35below the mean.

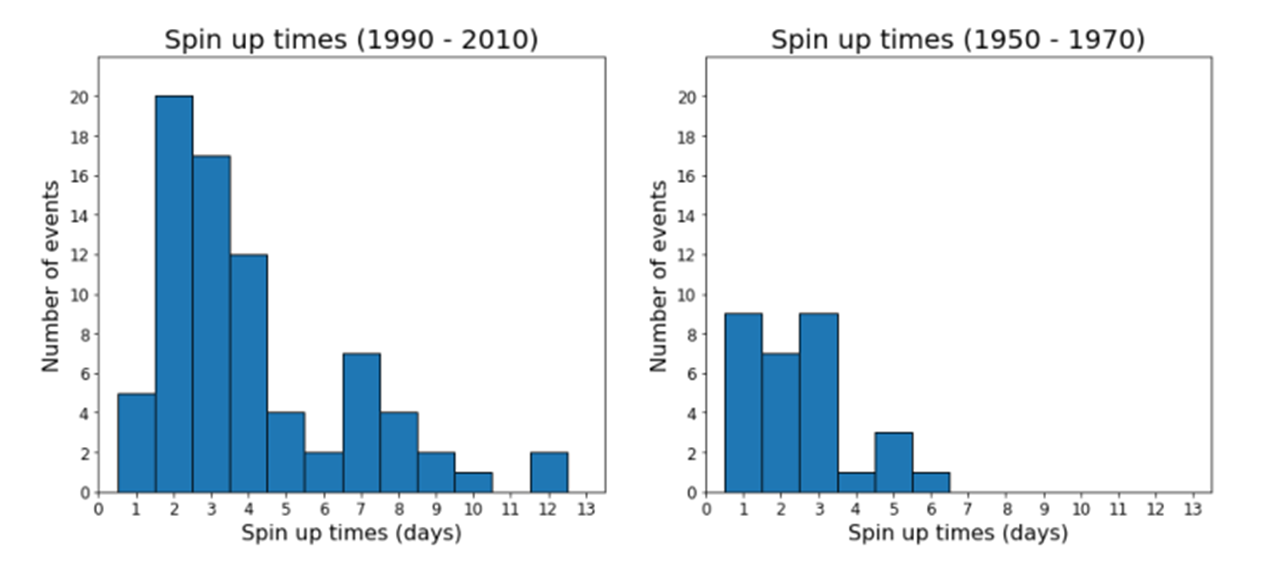
Again, let us suppose that you are working for the insurance company and their PR person comes to you and asks “How should we define a strong vs. weak hurricane for the public from this distribution”? Well, that’s a tough question – if you say, choose the median value ½ of all hurricanes are strong which is likely the message that you don’t want to send since strong hurricanes should be less common than weak hurricanes. So from this distribution alone it seems quite ambiguous what the pressure line divide would be between strong and weak. Perhaps the most informative graphic that could be instructed from this may look something like this:

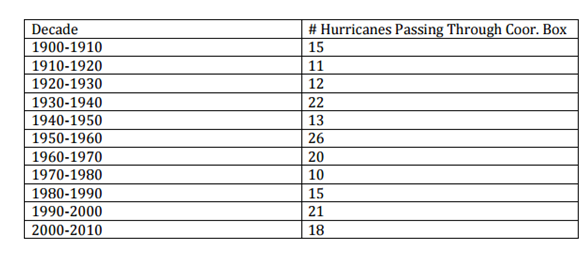


Actual students that have done this exercise struggle the most with Step 9. For instance, some systems are probably not even named Hurricanes at the 1000 MB level so you have to deal with the fact that your boss has given you slightly unclear directions. The best way to tackle Step 9 is to generate a new data file which three fields:

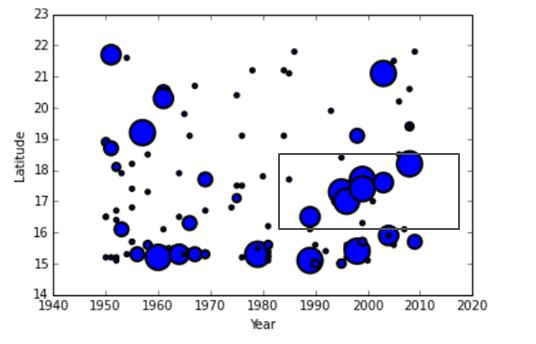
* System ID
* Timestamp
* Central Pressure

In this way, you have a single new dataset to deal exclusively with this step. One possible output production would be this which would require further statistical testing to see if there has been an increase of spin up times.



Finally, in relation to the Caribbean counting box we have this table. A quick perusal shows there as a factor of 2 dynamic range in these counts of storms and we will assess level of significance via Poisson statistics in chapter x.x. However, once again the table below shows why domain knowledge is important. Without any domain knowledge, the data analyst might put this table of numbers through some black box. But with domain knowledge, the data scientists would realize, hey, wait a minute, was our ability to detect Hurricanes in this geographic box in 1920 the same as it today? Well probably not and therefore the raw data here is biased – it contains a selection effect which is that the detection efficiency of hurricanes as a function of decade is not the same. Later on we offer possible ways that you can correct these kinds of datasets of what amounts to incompleteness.

Finally, the production of the required bubble chart does actually reveal a possible new scientific result. This is what data exploration is all about and the use of visual output is the best way of detecting possibly new results. Yes one has to follow up on the level of significance and other details on this new result, but the new result would likely never have been detected by simply examining text-based output. As highlighted by the box in the figure below, the data suggest that in recent times, stronger storms are developing in this region, that previously had seen no development. Is this because the sea surface temperatures in this region are increasing more rapidly relative to other regions? Is this because the upper level shearing winds are becoming less in this region which then allows for more storm development? Or is it something else? The point here is data science has uncovered a possible new component of Atlantic Basin Hurricane development and evolution – this is ultimate value of data science – discovering new phenomena.



**Section 2.4**

**The second data exercise: Arctic Sea Ice Extent**

Here the scientific problem of interest is the rate at which the Arctic is losing sea ice. This problem is one of the many planetary responses to global climate change. Analysis of this data does have significant implications for policy and the way that policy is done. That will be covered in the very last step of the exercise. This problem is also not one of just idle intellectual curiosity it has very much been identified as a likely security problem for the US: (http://www.climatechangenews.com/2017/02/02/pentagon-arctic-melt-requires-updated-us-strategy/). So, when proceeding through this exercise do so within the context that you are a data consultant hired by the Pentagon to advise them of when the Arctic Ocean will become ice free (Santa may also want to hire you as a consultant to deal with this problem). This exercise will center on another common element of working with data, namely feature extraction, curve fitting, and long-term prediction from time series data

Exercise 2.3: The Arctic Sea Ice database

The datafile below has been mainly produced from data originally stored at the University of Wisconsin and has been added to from more recent data at from data at the National Snow and Ice Center (<https://nsdic.org>). Note that since 1979 measures of Arctic Sea Ice extent have been done using satellite imaging and are much more accurate and reliable than previous measures. This will factor into some of the steps below.

**Step 1:** Get the data file (<http://homework.uoregon.edu/pub/class/sciprog/icedata.txt>)

* Column 1 = year
* Column 2 = July
* Column 3= August
* Column 4 = September

All listed units are in square kilometers.

**Step 2:** Average the 3 months and then differentiate (dy/dt) this curve in 5 years intervals and plot the resulting slope vectors - this can be done using a finite difference approach or any number of ways.

**Step 3**: Using any numerical integration (see data methods and statistics section) technique that you like, compute the total area of the curve from 1870 to 1950 and compare that to the area under the curve from 1950 to 2015.

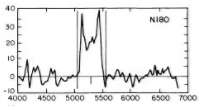
**Step 4:** Refer to this document http://homework.uoregon.edu/pub/class/355/noise.html about smoothing noisy data. Smooth the data in three different ways and produce a plot that shows all three smoothing treatments on the same plot.

* box car (moving average) of width 5 years
* Gaussian kernel of width 5 years
* Exponential smoothing with greatest weight given to the last 20 years

**Step 5**: Calculate the ratio of July to September extent and plot that ratio as a function of time. From that plot alone, can you determine if the ice is now melting faster?

**Step 6:** There are some who believe that the recent behavior in the Arctic is consistent with previous cycles of warming and cooling. Prior to 1950 there are two suggested cooling periods which allow the Summer Sea Ice Extent to remain larger than average. To extract these features from the data, requires data windowing and baseline subtraction. This is a common form of "spectral feature analysis".

For example, here is a 21-cm spectrum of a galaxy in which the (emission) feature (positive) is windowed and a baseline is fit to the data outside the window and subtracted to produce a flat baseline plus feature. The area under the feature is of interest.



You will determine if similar “emission” type features are present in the sea ice data set. The simplest way to start is to plot the data from about 1900 to 1960 and see if your eye selects any candidates.

Exercise 2.3: The Arctic Sea Ice Continued

**Step 6 continued**: There are supposedly two events in this data set of this cooling producing higher than average sea ice extents for a few years. Select those events by defining their time window and then fit a baseline to the data that is outside the event regions (refer to the “solution” diagrams at the end of this section if you’re feeling stuck) and then subtract the data from the baseline to produce a flat time series spectrum. Determine the total area of each event (you will have to decide when the event begins and ends) and compare that to the average area from the period 1870 to 1950, that you determined previously, to determine the overall amplitude of this cooling (i.e. how fast did the arctic increase its sea ice during this cooling period?)

**Step 7:**  When do we melt Santa’s home?

You are to make this determination using two sets of data.

a) Using the entire data set, determine a smooth functional form that best fits the data an extrapolate that to zero to determine when Norwegian Cruise lines should be taking bookings. DO NOT USE A POLYNOMIAL fit in this case. Nothing in nature is fit by a POLYNOMIAL. Produce a graph with your fitted line to the data, extrapolated to zero. Hint: a function known as a sigmoid is a good functional fit together, but you can also add straight line fits together if you want, that fit just won’t be very smooth

b) Now use only the satellite data, which starts in 1979 and use only the September data. Fit a linear regression to the data (you can use any package you want for that) but then also fit some kind of power or exponential law to the data. You will find the two zero predictions to be quite different and once of them is scary. Plot those two fits on the same graph.

c) Now present your results to the policy think tank or the Santa relocation project –

http://www.greenpeace.org/international/en/news/Blogs/makingwaves/santa-relocation-project/blog/43488/

on your best guess for the year in which the Arctic Ice Extent reaches zero. And yes, this is a real-life problem!

We present some graphical solutions to the problems posed in the previous steps in the section immediately below and so if you’re stuck refer to those graphical results to help nudge you on your way to completion.

**Section 2.5.1**

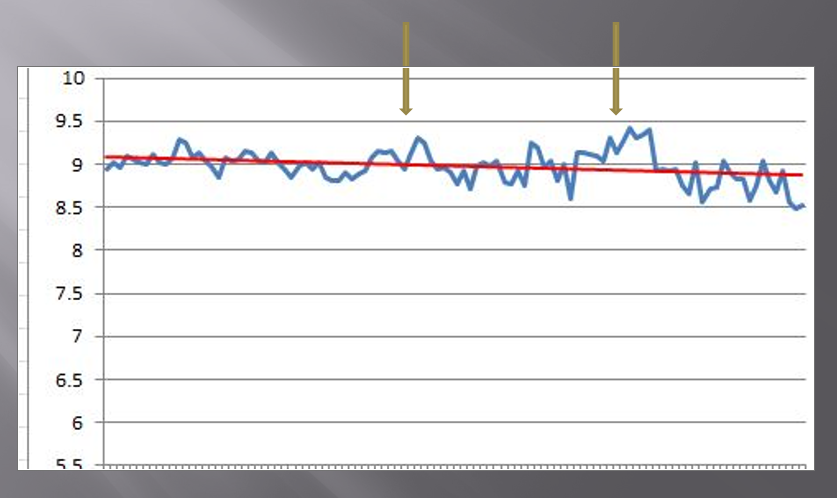
**Some results of the Arctic Sea Ice exploration exercise:**

Before discussing some of the specifics we offer some general “rules” for dealing with time series data:

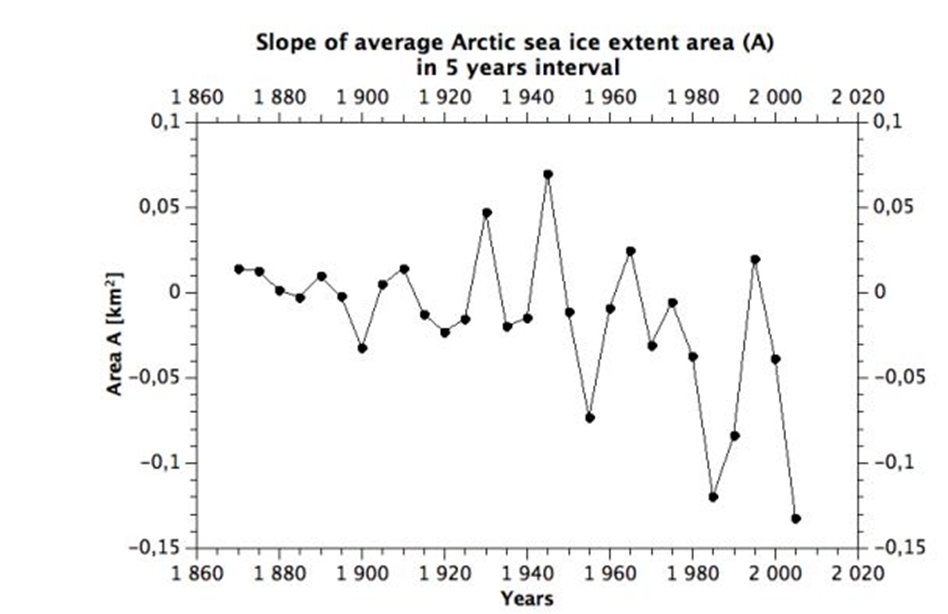
* For any time-series problem, plot the data first at some sensible scale and do simple smoothing to see if there is underlying structure vs just all random noise.
* Do a simple preliminary VISUAL analysis – fit a line to all or parts of the data, just so you get some better understanding
* EXCEL is actually convenient for this

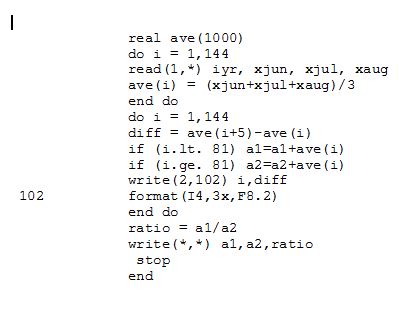
Always plot the data on some sensible scale, for instance the image in Panel A uses 0 as the minimum value for the y-axis while the image in Panel B uses the actual data minimum (5.5) so that the “noise” or volatility in the data is best seen. Both representations of the data, however, clearly show that at some point in time, the data enter into a non-linear regime.

|  |  |
| --- | --- |
| Panel A: | Panel B: |

You were told that the positive “emission” features occurred prior to 1960. A simple linear fit to that data, which does show a small baseline decrease over this time period, can visually suggest to you the existence of these features and to some extent you can also seem above in Panel B and this serves as a useful starting point on feature extraction. In the image below, the second feature in time appears to be the strongest while the first feature is kind of iffy – but the significance of these features can be tested (again see data methods and statistics).

The behavior of the dy/dt plot over 5-year intervals (step 2 above) clearly shows the increase in volatility recently compared to the distant past and this kind of plot is one of the better ways to reveal this behavior if it exists, and is useful to employ on any time-series data that has volatility.



Concerning step 3 – there is no need to do any kind of fancy numerical integration for this kind of data which has high intrinsic. Simply add the average data by year together in the respective time domains. The important lesson here is to look at the quality of the data first before deciding to invest computing resources to deal with it. When you heard the term “numerical integration” you probably thought that this was a more complicated issued that it actually is. A simple FORTAN program to do the relevant sums of the averages is shown here: in this case, the year cutoff between one time domain and the other occurs in line 81 of the data which has 144 original lines. While this code is not efficient and is very much brute force, it solves the problem in a straightforward manner and results in area ratio of about 1.4 between the former and latter time periods. So, this quick and dirty result tells you that the Arctic has lost about 40% of the Ice extent area it once had. With respect to smoothing the data (Step 4) there is always a tradeoff between smoothing width and data resolution. There is no formula to optimally determine this – you have to experiment with different procedures. For this data set the choice of smoothing does not much matter although the Gaussian Kernel approach is likely the best to use on any time series (see data methods and statistics).

|  |  |
| --- | --- |
|  |  |

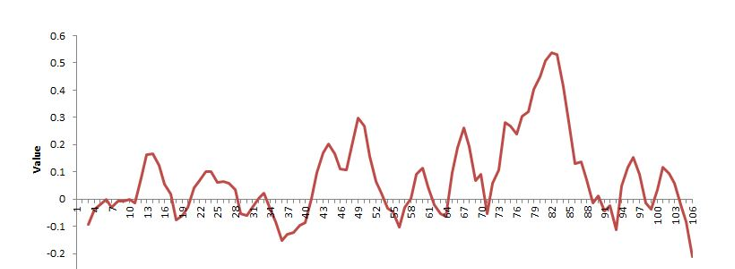
Some more details of the smoothing procedure are shown here:

|  |  |
| --- | --- |
| Here we compare exponential smoothing (blue line) with the box smoothing (smoothingwidth of 5 years). In this case the spectral slope of the exponential function is set to k=0.1 which weights the first part of the time-series more than the later part – the result is less of a decline in recent years compared to the red curve but which one should be properly used in the scientific sense? Perhaps neither one: | In this case of exponential smooth we set k=0.5 so all the data is equally weighted and the result is indistinguishable form the red line. Once again domain knowledge is needed to help make decisions on what kind of smoothing choice should be made. |
|  | In this panel, we now compare exponential smoothing at k =0.9 (recent years weighted more) to kernel smooth of width 5 years. This representation of smoothing differences clearly shows that the kernel approach provides a smoother time series and minimizes the fluctuations so that the long-term behavior is best discerned. Thus, we strongly recommend that such kernel smoothing be routinely used on this kind of data. |

In this exercise, step 6 has traditionally been the most difficult to do and has often produced erroneous results. The student seems to make the process of feature extraction by baseline fitting more difficult than it really is. So here we explicitly show the steps.

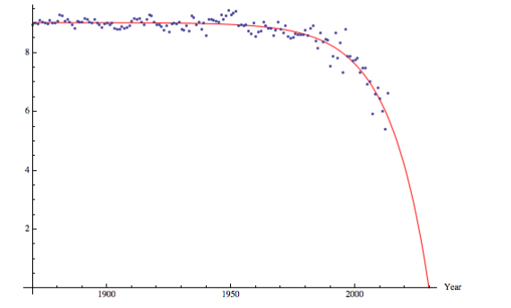
|  |  |
| --- | --- |
| Panel A | The very first thing to do is plot the data at a sensible scale and just do a linear fit. This simple linear fit does include features so is wrong but it serves as an initial guide which suggests there are a few peaks that might be present. What you also notice is that the strongest peak is about .4 units above the this linear fit and that serves as a guide for expected amplitudes. For example, the peak on the far left is much weaker and looks to be about .2 units above the baseline fit |
| Panel B | The next thing is to avoid a common mistake. You cannot have expected features within the data set that you are using to fit the baseline. If this happens you will merely subtract out the features thus defeating the entire purpose of feature extraction. So, yes, the spectrum is flattened due to baseline subtraction, but is also now garbage. |
| Panel C: the windowed spectrum | So now let’s define where we think the features are. Notice here that we are not considering the left most peak as a feature but are only windowing two features. In the second feature (about time step 80) we have chosen to include the peak at time step 65 as part of the overall feature. This is a judgement call and maybe that window should have been moved closer to time step 71 (6 years later). The portions of the spectrum denoted by the green double arrow are then used to determine the baseline |
| Panel D: the cubic spline function  In python, a cubic spline can be done in just a few lines of code  >>> import numpy  >>> x,y = numpy.loadtxt(“xy.txt”, unpack=True)  >>> p = numpy.polyfit(x,y,deg=3)  >>> print p | Now we fit a cubic spline (third order polynomial) to the baseline pixels (in the figure to the left we have just removed the data points that define our features) as that function is generally good for most baselines – higher order polynomials should not be used. The actual polynomial function is given so for every x point you then compute y and then subtract the y data point from that computed value of y to produce your flattened spectrum |
| Panel E: the baselined spectrum | After flattening the peak of the main feature has increased in amplitude above the baseline in Panel A from 0.4 units to 0.6 units now that that feature has been properly baselined. Note that we performed this series of operations on unsmoothed data just to explain the process; you should begin the Panel A part with your smoothed spectrum. This will produced figure x.x |

So now your excited because you have extracted some features and show this to your unenlightened supervisors whose first and generally only response is that there are no features there this is just all noise. Well no, it’s not just all noise as we will demonstrate below, but in dealing with low signal-to-noise (S/N) features (and almost all of climate data is low S/N) it is very difficult to effectively communicate that you have a statistically significant result. Most of the policy world reacts to the appearance of figures long before it will assimilate a table showing some statistical results. To be an effective data scientist some means you have to sell your result in creative ways, much the same that his is done in most peer reviewed scientific publications. So how can we show the strongest feature is statistically real (more of this is covered in the data methods and statistics section? Here we show the flattened version of the spectrum smoothed with a 5-year Gaussian kernel:



To determine the S/N of the feature between points 70 and 90 we want to compare the area of that feature with respect to the area of a noise feature that is 20 years wide. While one wants to do numerical integration under the feature for a more precise value of the integral here we just consider the feature as a triangle of width 20 and height 0.5 so the area = ½\*20\*0.5 = 50 units. We then compute the r.m.s. error of the data outside the feature which just means calculating the standard deviation of the 86 data points outside the feature (most are on the left-hand side). Note that this way of estimating the r.m.s. error will overestimate that error because we are including the feature between 40 and 52 and we probably shouldn’t do that. But boldly going ahead, that computed r.m.s error is about 0.1 units (or roughly ¼ of the peak to peak fluctuations in the data – this is a quick and dirty way of estimating this error from a graph). The similar noise area would be ½\*20\*.1 or 10 units. The ratio of the feature to this noise area feature is then 5. This means that the feature is 5 standard deviations above the noise. In science, 3 standard deviations (3) is often considered the threshold for detection because such a threshold has only a 1 in 1000 chance of being random noise. Here we choose to be more conservative at 5. Hence, this feature is real and suggests there was about a 10-year period where the Arctic cooling did allow sea ice to grow to larger extents.

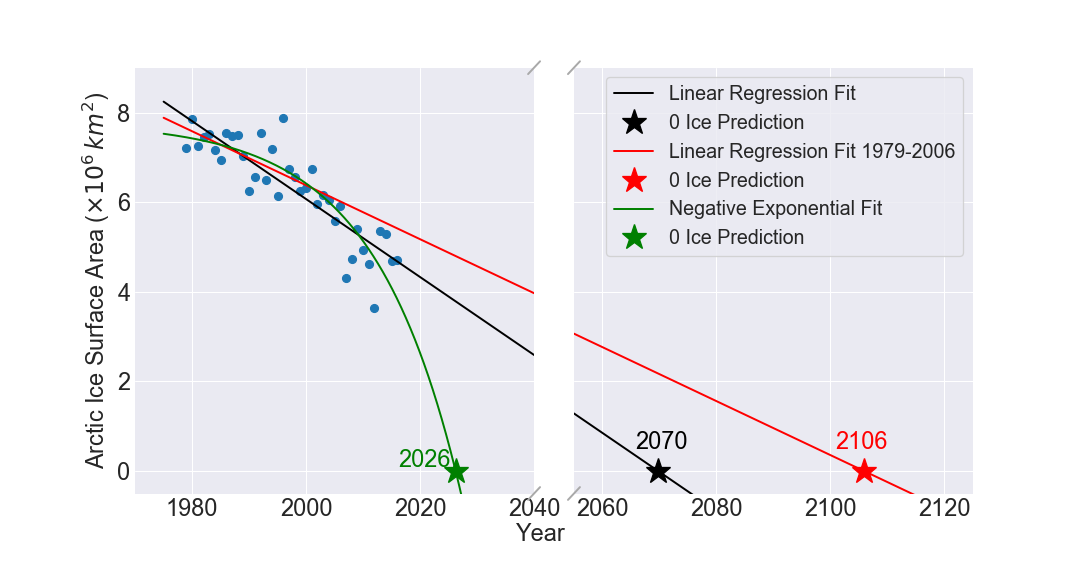
Finally, when will the extent extrapolates to zero. Below we show one model result which uses a damped exponential on that part of the data which is in the satellite era (1979) and beyond and leads to a zero-ice extent in the year 2030 – wait a minute, that can’t be right, isn’t this a problem, why aren’t we doing anything about this? This cutoff exponential form is:



So now what does the data scientist tell the policy world? Are they going to believe this result? Are they educated enough about data to know what a non-linear curve fit actually is? And all of this was produced by undergraduate students in some random class at the University of Oregon, what do they know? In short, telling a data story in a manner that leads to receptiveness of the results that the data imply, especially when they are unpopular, represents a real challenge to the data scientist but it is a challenge that they should not ignore else one is doomed to the kind of situation that Copernicus had to endure (see Chapter 1).

An updated version of this policy dilemma appears in the August 17th blog theenergycollective.com as a piece co-authored with a former data science student. We excerpt the relevant part of that piece to close out this chapter.

The rate of Arctic Sea Ice loss provides a good example of the difference between linear and non-linear fits to the data when policy extrapolates to the issue of “when will the Arctic Ocean be free of ice in September”. The data, complied from the National Snow and Ice Data Center (NSIDC) is shown in Figure 1 which plots average September sea ice extent vs time. The time record time starts in 1979, the first year of satellite measurements. In Figure x.x we show three fits to this data that are each extrapolated to zero ice extent in some future year.



The various results depicted here give rise to the following three scenarios:

1. Suppose that in 2007 a policy concern arises about an ice-free Arctic ocean. Some international committee is formed to study the issue and produce policy recommendations. Well, the red line in Figure 1 would show the requisite linear fit to the data available at that time (i.e. 1979-2006) that extrapolates to zero ice in the year 2106. The committee meeting is quite short: the problem won’t occur for 100 years so why worry about it now?
2. Okay, we now reconvene the committee 10 years later to update the situation based on 10 years additional data. Well, now the linear extrapolation from 1979-2016 (black line above) leads to the ice-free Arctic in 2070. Yes, the new data has produced a shorter timescale but, hey, that’s still 50 years away so again, why worry about it now?
3. But wait, what is that “weird” green line on this infographic? Well, the green line, which produces a prediction of 2035 (less than 20 years from now) is the best fitting non-linear relation to the data. Does this matter?

The difference between the linear and non-linear fit predictions is 30-40 years, which is significant in terms of human decision-making timescales. In the linear policy world, we would just punt on the issue since the crisis point is way into the future. However, if the non-linear approach yields the correct trend, but we remain stuck with our linear mindset then it’s quite likely that policy will be set after the time when children can visit Santa’s home on a cruise ship.

The lesson here is clear: when you are on an accelerating rate of change, the future becomes harder to predict – which adds uncertainty to the overall process. Instead of paralyzing the policy process, this increased uncertainty should focus efforts for policy to be based on more accurate trend forecasting. In this case, the data clearly show that the rate of Arctic Sea Ice loss is accelerating.