Computational Frameworks for Social Science

Matt Dickenson*

Current as of: March 21, 2012

1 Introduction

Table 1: The Software World According to Josh Cutler

Software Engineers	Computer Scientists	Hackers/Coders
IBM	Google or PhD	Start-ups
Wear Ties	Beards	Build Apps

1.1 Source Control

We will use `Git` as our source control. `Git` was a sea change in CS. Git can save versions A and $A+1$ simultaneously, so that you can work with multiple versoins of code and cooperate with other people. To use:

```
git init
git add text.txt
commit -m ‘‘first commit’’
```

Once we commit, this flushes the change set and creates "first commit" in the git log.

```
git add text.text
commit -m ‘‘after edits’’
```

*Notes for Josh Cutler's PS 398 course, Duke University, Spring 2012.

To fully preserve for posterity:

```
git push .
```

To create a new version of yourself:

```
git pull .
```

The key idea here is that you can push to/pull from arbitrary places. There is no hierarchy for what is the True version.

1.1.1 A Word on Branching

Branching is how code gets built. Branching is a way to expand a codebase while keeping an earlier version stable. When someone talks about the "hot new α version", they are talking about a branch. For instance, we may push a package to CRAN (v1.0), then start work on an update (v2.0). In the meantime, we find out a bug in v1.0 that we need to fix. With branching, this isn't a problem.

1.1.2 Naming Versions

Consider a version $X.Y.Z$. We commit $X.Y.Z + 1$ when we have fixed a bug. We commit $X.Y + 1.Z$ when we have added a feature. We commit $X + 1.Y.Z$ when we want to charge people more money.

1.1.3 Commit History

A basic rule of thumb is: if you can't describe what you did in a single line, it's too big a jump.

Sometimes in professional software you will see "commit wars" where a couple of people just push the same junk back and forth until someone fires them. For example, one guy may indent his Python code with two spaces, while another uses four. Agree on this early. Communicate.

1.2 Testing

Automated testing, that is. We're going to use code that we write to test our other code. Specifically, we're going to use "unit tests" (as opposed to "integration testing"). Testing

is contentious–people will fight about, leave companies over, or start companies because of this.

A motivating example is BDD vs. TDD–behavioral driven design *versus* test driven design. BDD people are masturbatory and annoying. TDD people write tests first and then code. They write a series of tests that suitable code should pass, and then design code. Others write code and then test it, but they think of the same test cases they did when they were coding, so they may overlook problems. Your choice will likely depend on how long you are going to have to maintain the code. Technically, TDD code has to fail a test before there is a bug.

But we are talking at a non-philosophical level about unit tests. Reasons to use TDD:

- if someone else inherits your code, it helps them to have a test suite to detect hairy portions of code
- it helps readers to see what the code is actually supposed to do
- you can refactor (rewrite) your code with more confidence, since you can quickly check all known relevant tests (without TDD, you will undoubtedly introduce bugs; with TDD you will be at least as good as you were before)
- it helps you be lazy–most people don't have enough mental RAM to remember everything a program does
- you will write better code if you know you have to test it
- if you truly understand what code is supposed to do, you should be able to assert what output it should give you

Python is dynamic (duck) code. The important thing to remember is that there are more bugs that can be introduced than, for example, C++. To see what we're talking about, let's actually write some code.

The two things that we check for are *correctness* and *robustness*. Correctness asks, "does the code do what we want it to when we give it the right input?" Robustness asks, "how does the code handle unexpected inputs?" In Python's `unittest` library, a test can have three outcomes: "." (pass), "F" (fail), or "E" (error). The key point: have a failure plan. This is important, for example, when writing a web scraper. HTML is supposed to look a certain way, but in reality it never does.

The process is iterative:

1. see a bug
2. write a test
3. fix the code so that it satisfies the test
4. repeat

Again, the argument for using code is that your mind will be more open (and thus write more appropriate tests) before you develop the tunnel vision that comes with writing code in a certain way.

2 Object-Oriented Programming

2.1 Mini-Homework

Restructure homeworks on github so that the root looks like `/HW1` and use `/HW2` for the next one.

2.2 What is OOP?

Table 2: Three Styles of Programming

Procedural	Functional	Object-Oriented (OOP)
Probably what you think of	You don't care about this	Philosophy of representation
C, R		Python
Sufficient for small projects		
Very little hierarchy or structure	list comprehensions, etc.	tightly couples data w/methods

Consider an object `Book`. In `R`, this would have a bunch of characteristics like "author" and "pages". In OOP, every book that exists is an instance of the class "books." In fact, *everything is an object* in Python: numbers, strings, classes. Think of classes as a way to give your object some structure.

2.3 Why OOP?

To stay DRY: Don't Repeat Yourself. Ultimately, you will be writing less code using OOP– even though it won't seem that way at first. A lot of it comes down to being able to say, "This is a `thing`. It should have `this` info and `these` methods." Any code can be written procedurally *or* as OOP, but the choice is which is simpler.

2.4 Inheritance

Python is a method-passing language, which means it does something called "dynamic dispatch." Say we defined:

```
class Animal(object):
  def __init__(self,name):
```

```
    self.name = name

class Cat(Animal):
  def talk(self):
    return ‘‘Meow!’’
```

So when if we set `a` equal to `Animal(‘‘Fido’’)` and `b` equal to `Cat(‘‘Sally’’)`, then when we call `b.talk()`, it will search within class Cat first, and then class Animal if it doesn't find anything.

This is known as *polymorphism*, but it won't mean much to you until you practice.

As the example `sports.py` will show, there is a trade-off between generality and specificity in how much you want a subclass to inherit from a superior class. Practicing this will force you to think about:

- What should exist at the object level?
- How should those things interact with each other?

Pro-tip: Move as far away from `global` variables as possible.

3 Pragmatic Programming

3.1 Data types

There are three data types in Python, shown in Table 3.

Table 3: Data Types

Name	Code	Details
tuple	`()`, `a=(1,2)`	immutable, i.e. you can't change it
list	`[]`, `a=[], a[0] = ''foo'', a[1]=''bar''`	
dictionary	{}, `a[''foo'']=''bar''`	use of keys

3.2 Exceptions

Any time Python "explodes" it is because an exception has been raised. `Exception` is a class, and all types of exceptions inherit from this class. Any exception based on the class `Exception` will have the attributes `msg` (what prints when the exception is raised) and `stack trace` (which identifies where the error occurred). Note that "raise" is a technical term in Python.

```
class CustomException(Exception): # inherits from Exception
  def __init__(self, value):
    self.value = value

  def __str__(self):
    return self.value

def i_call_a_function_with_errors():
  try:
    print "Calling a function...."
    #function_with_generic_error()
    #function_with_custom_error()
    #function_with_unknown_error(1)
    print "Tada!"
  except CustomException as inst: # 'as' gives us access to the exception
```

```
    print "Custom Error Caught! Error({0})".format(inst.value)
  except: # any exception is caught, even ones you don't know about
    print "Default Error Caught!"
  else: # if nothing broke, then run this block
    print "No error raised."
    traceback.print_exc() # this prints the traceback
  finally: # this block is always run
    print "Goodbye!"

def function_with_generic_error():
  raise Exception, "Foo!" # this method doesn't know what to do with
the exception

def function_with_custom_error():
  raise CustomException, "Foo Bar!" # this will be handled in the
function above}

def function_with_unknown_error(foo):
  foo.bar()
```

Caught exceptions are ones that keep the user from doing something the programmer didn't want them to do. Uncaught exceptions cause problems.

In a Python test suite, you set up a `try` block to run some code where you think a problem might happen. As soon as an exception is thrown in your `try` block, it doesn't try anything else.

Rule of thumb: If you know what to do with an error, handle it. If you don't, let it percolate up.

Exceptions are something that you've been dealing with up to now but (maybe) didn't even know about.

3.3 Algorithms

Having an instict for better and worse algorithms will let you know whether solving your problem will take hours or years.[1] An algorithm can be defined as "a series of steps that

[1] "No one in stats gets a Ph.D. without coding and the only way people get Ph.D.'s without coding in our discipline is because there are some dumb people who have made their way up, but those guys are going

achieve a desired outcome."

Consider the task of sorting a list. This is not a problem you will ever have to solve because better people have already done it, but it's easy to visualize. We use algorithms to solve hard problems. In mathematics it's known as an NP problem. (This is just nerd's way of saying "hard.") Most problems in game theory are NP complete.

In political science, you might be using a data set with 18 million events. But the thing you're interested in might require $18,000,000^2$ or even $18,000,000^3$ observations. 18 million cubed is a big number.

To put this in a Python context, say we want to sort a list $[y_i, ...y_n]$. A terribly inefficient way to do this would be to randomly shuffle and then check to see if they're in order. There are $n!$ ways to shuffle the list, so the problem grows factorially.

In programming, we use "big O" notation to talk about complexity. The O means $\forall x f(x) < cf(x)$. So if we have a problem $n^2 + n$, we denote it $O(n^2)$ and call it "quadratic complexity." Complexity of the class $O(n)$ is called "linear," and so on. It tells us how the number of operations goes up as n grows. We use it to speak about *average* complexity of problems–after all, we can only speak in averages.

3.3.1 Selection sort

```
k = 0, L= [] \\
Loop through n-k \\
    find smallest number at j \\
    swap L[k] with L[j] \\
    k++
```

```
              [5,1,15,7,111]  # of things to check
Rd 1: k=0,j=1 [1,5,15,7,111]    5
Rd 2: k=1,j=1 [1,5,15,7,111]    4
Rd 3: k=2,j=1 [1,5,7,15,111]    3...
Rd 4: k=3,j=3
```

to die soon." – Scott DeMarchi.

3.3.2 A Brief Primer/Refresher in Discrete Math

$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$, which is a $O(n^2)$ (quadratic) complex problem. How do you prove it?

$$\begin{aligned}
\sum_{i=1}^{n} i &= n + (n-1) + (n-2) + ... + 3 + 2 + 1 \\
2 \times \sum_{i=1}^{n} i &= n + (n-1) + ... + 2 + 1 + 1 + 2 + ... + (n-1) + (n-2) \\
&= (n+1) + (n+1) + ... \\
&= n(n+1)
\end{aligned}$$

Go back to the example above. How many tries would the best case take? 5. The worst case? 5. And the average case? You guessed it–5.

3.3.3 Merge Sort

This method would take the list to be sort it, split it in half again and again until they were all disaggregated to the unit level. It would sort those and reassemble (merge) them into a two-ple (get it?). This reduces the list sort to an $O(nlog(n))$ problem. (In this case we're not even throwing away little numbers or constants from the n.)

Try this out. Come up with a list of numbers, maintain a spot in them. It's fairly easy to sort two things and interweave them. It will take $nlog_2(n)$ sorts.

3.3.4 What should I care about in optimizing an algorithm?

Some people will care about the time it takes. Others will care about the (memory) space it take. Most people who care about memory work on rocket ships or microwaves. Merge sort takes more space than selection sort, but uses less time. As with anything in life, it's a tradeoff.

3.3.5 Back to NP

P means that the problem is in polynomial time class–not that your algorithm is in P time (it may be exponential) but that the ideal answer is in P time. Computational game theorists sometimes look at *whether* something is solvable in polynomial time. If it isn't, we call it NP time–that is, not solvable. If you can prove that $P = NP$, you can break all cryptography in

Table 4: How long will it take to find y_i?

Method	List	Worst	Best	Average
Naive	$[y_1, ..., y_n]$	$O(n)$	$O(1)$	$O(\frac{n}{2})$
Binary	sorted	O(log n)		

the world, make a lot of money, and retire. If somebody says a problem is "NP," that means don't waste your time. (Actually they'll say "NP-hard," "NP-complete" and so on–that just means someone way more mathematically inclined than us has proven it so; don't argue.)

3.3.6 Quick Sort

There is a fourth sort, which we will not get into here, that most programs actually use. Quick sort is, on average $O(nlogn)$, but its worst case is n^2. Again, it all comes back to what you care about this.

Don't stress too much about actually computing complexity classes–they've already been computed for almost anything interesting enough that you'd want to work on it. Just know how to choose between them when presented with options.

Remember that there is a trade-off between how much time you spend programming the algorithm and how much time it will save you. In general, think about how many times you will plausibly be running the algorithm. Most working programmers look for "satisficing" solutions–fast enough and no faster.

4 Data Structures

4.1 A Note on Naming

In Python, use UpperCamelCasing to name classes and lowerCamelCasing to name variables or methods. Variables that are member variables of a class are preceded by two underscores: "__oneTwo". These are conventions that are meaningful to Pythonistas, and should be meaningful to you as a reader of code and a contributor to projects.

4.2 Review of Sorting Algorithms

Take home points:

- Use `time.clock()` to benchmark runtimes.

Table 5: Data Structures and Their Usefulness

	Arrays	Lists	Queues/Stacks	Dictionary	Trees
Add	O(1) or O(n)	O(1), always	O(1)	O(1)	$O(\log n)^2$
Delete	O(n)	O(1)	O(1)	O(1)	O(log n)
Find	O(n)	O(n)	O(n), but we don't care	O(n)	O(log n)

- Know what your data looks like. (Are there many unique values?)
- Bubble sort is bad.

4.3 Intro to Data Structures

Why would you use one data structure over another? Because of what you want to do with it. There are several classes of data structures, with many variations on each class–look at Wikipedia. Let's talk about how lists and arrays work, and when you should use them. We typically care about how quickly we can add, delete, or find something. (See Table 5.)

4.4 Arrays

Before getting into the use of specific data structures, it is important to think about how they are represented on your computer. Your memory, sometimes referred to as RAM, is known as the machine's "heap" in computer science. When we tell the computer that an object has a certain schema, it allocates a certain amount of space based on how much space an object of that schema may take.

An array looks like `a = [1,2,''a'']`. When you do this, your computer gives the object `a` exactly 3 spaces (an imprecise term for the moment) of memory. We can index these things quickly because we are keeping track. However, the tradeoff is that adding something to list list is difficult. If we run `a.append(''b'')`, the computer creates a whole new object with four spaces and deletes the old one. The actual process of renewing the list over and over can be very time consuming.

4.5 Lists (or "Vectors")

Lists are arrays that do not allow direct indexing. Rather than allocating a block of memory, a list is stored as items that each point to the next item in the list. The only way to get

Table 6: A Stack

"b"
"a"
2
1

the second thing out of memory is to get the first thing out of memory and ask it where the second thing is hiding. If we want the fourth (or the n^{th} element, that can become a lot of asking. To find something, we have to use an algorithm, typically starting at the middle and working our way out.

The benefit of a list is that you can add things to it indefinitely. Furthermore, they can grow indefinitely as well because Python does not allocate a fixed space for the list when it was created.

4.6 Queues

(To build a queue, you need a list or an array. We're working our way up.)

Queues only support enqueue or dequeue (adding something to the end, or removing from the front, of a line). You could use either an array or a list to store these. Which is better? A list, because we aren't going to be doing searching, just adding and deleting.

Queues are managed by the rule of FIFO (First-In-First-Out).

This is not a natively supported data type in Python, so if you want one you'll have to build it yourself, or `import from collections deque`. Python also does not differentiate between arrays and lists, but in `R` it does make a difference.

4.7 Stack

As the name would suggest, this can be best visualized as a stack of things. Stacks support LIFO (Last-In-First-Out). Again, you only add or remove things. Things can only get pushed onto or popped off of the stack.

As with lists and arrays, Python conflates the difference between stacks and queues. If you're working with data of known size, this isn't a huge problem.

What's going on under the hood here? When you call a function, it gets put in your stack. Your computer has a huge chunk of memory for the stack. In the pseudocode below, the user

calls `a`, which in turn calls `b`, which in turn calls `c` and returns 1. How does something get popped off? A function is popped off when it returns. A recursive function just puts itself on the stack over and over again.[3] Fortunately, Python tries to help you (in the terminal at least) and tries to keep you from overflowing the stack with recursive functions.

```
def a():
    b()

def b():
    c()

def c():
    a = 1

>>> a()
>>> 1
```

When a problem happens, the stack trace is returned, which will tell you all of the functions that failed to catch it. (Tattle-tale.)

Because a computer core only does one thing at a time, a stack is an efficient way to handle it. An eight-core computer has eight stacks.

4.8 Dictionary/Hashtable

Dictionaries provide a quick lookup for arbitrary things. It is called a hashtable because it uses a hash function: it assigns a unique identifier (typically a number) to the object that the arbitrary key should point to.

```
a = {}
a["foo"] = "bar"
```

When we do the above code, we tell the computer "a + 'foo' = memory location." The pointer that gets us from input to output is the hash function. The main point here is that you shouldn't be doing a lot of searching through a dictionary, because it is bad for that. What dictionaries are good for is adding and removing a lot of things very quickly, and to

[3] "When you try to push something onto the stack when the stack is full, you have overflown the stack."

access them (but not sort them). This is one of the most commonly used data structures for a reason: you don't know how many data points you will have, but you want to get to them quickly. Understanding lists and dictionaries are the most important points for today.

4.9 Trees

Trees come in many shapes, but it is easiest to visualize a binary tree (think game theory; each node has at most two branches). The top is the root, anything that doesn't have another node after it is a leaf, and the distance between the root and the farthest leaf is the height of the tree. Depending on the properties you enforce for your tree, you can get interesting patterns. If you put numbers into your tree in a certain way, you can access them very quickly: everything to the right is bigger than its parent node, everything to the left is smaller than its parent node.

The nodes of the tree are stored with pointers to the other items. Each node points to its right and left child (which may be empty if the node is a leaf).

You can think of a list as a degenerate case of a tree. These are pretty easy to create, so many people make their own. Useful terms to keep in mind are "max heap" and "min heap," in which the greatest number (or the smallest number) is on top. There is also an importance heap, which is beyond the scope of the current discussion.

4.10 Graphs

Graphs are a computer scientist's favorite thing. They consist of nodes and edges. Nodes can contain or not contain things, depending on what you want the graph to look like. Edges can be weighted or unweighted, depending on your problem. Weighted edges are interesting (an NP hard problem) in the travelling salesman problem. In this problem, the edges are weighted by the mileage between the cities. Edges can be directed or non-directed, which governs how the graph can be traversed. A graph is "cyclic" if you can get back to your starting point for any smaller portion of the graph.

To make things more confusing, people like to abbreviate. A DAG is a Directed Acyclic Graph. Trees can be thought of as degenerate cases of graphs, and are by definition DAGs.

No one uses graphs in Python, but you will for your homework. You will handle them as objects. Another way is with an adjacency matrix (which can be weighted or binary).

5 Input/Output

After a code review of last week's homework, we are done talking about algorithms and complexity for their own sake. Just know that as your problem gets big, you will feel the pain of picking a bad algorithm.

The slowest part of a computer is a hard drive, so most algorithms are optimized not only by time, but by minimizing the work you're doing on the hard disk. Given that everything else in your computer happens at the speed of light, going around a hard drive at 5000 or 7200 rpm is a snail's pace. This is why "defragging" would speed up a machine, and why the hard drive is the first thing to go bad. Wider availability of solid-state drives (SSD) will change the things we have to say, but for now they are not widespread enough that we can ignore hard disk read/write time.

Reading from a disk is really no different than reading data from the internet, or calling `print`. The machine is calling something called `stdout` and putting it on your screen. (Recall that in Zed Shaw's Python book you used `stdin` by calling `raw_input()`.)

5.1 filestuff.py

There is a graceful way to open files. Use it.

(Note that the readfile.txt called by filestuff.py is just the text from our department home page.)

When you see "`for l in f:`" in Python, Python will understand that to mean using the native linebreaks in the file.

`f.seek(0)` is how we move the pointer back to the 0^{th} byte of the file. A byte is one character in space representation.

5.2 urlgrabbing.py

URL: *U*niversal *R*esource *L*ocator/Lookup

HTML is a contract for how you should structure the document so the web browser knows what to draw on the screen. When we talk today, we're discussing "html in its proper form." This is an ideal that does not exist on 99 percent of the web. Fortunately, other people have solved this problem for you–use libraries.

Here's an example of XML and what it allows you to to:

```
<thing>
```

```
  <nested_thing>

  <\nested_thing>
<\thing>
```

Notice that the identifiers inside the "¡¿" are arbitrary. You can find actual HTML anywhere by right-clicking and selecting "view page source."

Here's a stylized example:

```
<html>
  <head>         # not visible
    <title>
  <\head>

  <body>
    <h1>Foo</h1> # a header block (headers go from h1 to h6,
decreasing in size)
    <p>bar<\b>   # a paragraph tag
    <a href="google.com">google</a> # a hyperlink
    <img src = "pic.png"> # an image
    <div>        # a divider
    <table>      # care to guess?
    <span>
    <hr>

  </body>
</html>
```

Libraries for parsing HTML are available in almost every language. A good one in Python is BeautifulSoup.[4]

Before you go running off to spider everything, a few things to note:

- sitemap.xml = what to index
- robots.txt = what not to index

[4]In OS X, install from terminal using `easy_install BeautifulSoup` or `pip install BeautifulSoup`.

- rel="nofollow" = if you're a crawler, don't go there

Your laptop on an automated search is much faster than the way people normally use the internet. Use a delay–a second or two will suffice.

6 More Applications: Twitter API

6.1 Things we didn't discuss last week (on purpose)

HTML has a bunch of encoded versions of characters, such as ` `. Every program that interacts with the web will have a library for dealing with HTML entities. Anything that starts with "&" and ends with ";" is an HTML entity.

Another problem that you may run into is *relative* URLs. You do not actually have to specify the root of a page to access it; HTML just assumes you want the same root that you are on currently. The Python library `urlparse` can deal with this through the command `urljoin`.

6.2 Application Programming Interfaces (API)

You already know what these are, even if you do not realize it. An API can be thought of as a contract for how a message is passed. We thought about this on a smaller scale when we implemented functions. For example, in Python the code

```
def foo(bar):
    print bar
```

has a contract to accept `bar` and return the printed form.

There are different types of API's. One is SOAP and it is bad. If you have the choice, do not use it. The preferred option is REST, which stands for "Representational State Transfer."[5]

6.2.1 REST

In a REST API, all things are objects. We do not mean this in quite the same sense as OOP, but it means that we are doing things to other things. If this makes perfect sense to

[5]As an interesting piece of history, the same individual conceptualized REST and HTTP. Both are contracts for information delivery.

you, do not worry about the alternative. All of the things that the API works with (users, pages, coordinates, etc.) have unique URI's, which saves a lot of confusion. All things also have a representation; you know about the HTML representation, but you could also get the XML or the JSON representation. The important point is that we can elicit different representations of the same object from the API.

Note that the Twitter API has a rate limit of 350 requests per hour.

6.2.2 HTTP

HTTP is a way of representing object. Important types of objects include clients (e.g. browsers, urllib2, etc.) and servers (the thing you crashed if you failed to put a `time.sleep()` command in your last homework). Observe that `git://` has a different representation that `http://`, which is itself different from `https://`.

6.2.3 CRUD

You will encounter this acronym when discussing what an object can do. It stands for "Create, Read, Update, Delete." When you access this object through HTML, we are using its *read* attribute. HTTP has four actions: *G*et, *P*ost, *P*ut, *D*elete. Most everything you have done on the web up to this point is post (which corresponds to create) and get (which corresponds to read).

On Twitter, when you read a tweet, you are getting it. When you post it you are creating it. When you delete it, you are deleting it (makes sense, right?). All of these things happen to the same URI, and someone has packaged them into a nice neat library for you.

A URI is made up of 5 things: domain, protocol, path, query string, hash. You should already be familiar with the domain, protocol, and path. Query strings start with "?" and are concatenated with "&". Hashes start with–gasp–"#".

There's a lot going on on the server side too. The server will depend on you giving it requests that have a representation (e.g. MIME, XML). It will give you back the info you ask for, along with a code (see Table 7).

6.3 The Twitter API

First, you have to register an application with Twitter via `http://dev.twitter.com/`. You can register before your code is done. They will give you the keys you need, which will keep

Table 7: Server Codes

Code	What it means	What probably happened
200	OK	Something good
500	Server error	
503	Temporarily down	
404	Not found	Page deleted
420	Enhance your calm	exceeded ratelimit
418	I'm a little teapot	April Fool's

people from using your app, enables the permissioning system, and keeps track of your rate limiting.

You will need to install the library `tweepy`.

You will also have been given a Consumer Key and a Consumer Secret. Tweepy will need those.

Create an API object, which will keep you from having to do all of the things we talked about above as far as getting, posting, etc.

See https://dev.twitter.com/docs/api for all of the cool things you can do with your API.

This has been a brief introduction, because you will learn it best by doing it.

7 Databases

This is a huge topic, but fortunately you can get functional very quickly and learn as you go. Most of the actual work that goes on with databases is plain vanilla, but there are some strange extreme-use cases that take some know-how and experience.

There are a number of different kinds of databases:

- Relational database (what we'll do): Mysql, sqlite, SQL Server, PostGRES (many of these are free)
- Key-value Store (the other main one): MongoDB, TokyoCabinet, Redis, Cassandra, CouchDB
- Variations on these two

We use databases instead of, e.g., CSV files because CSV's are not optimized for fast searching or any other such thing that we actually care about. Smart people have spent a

lot of time optimizing databases for the kind of searching we're doing since the 1970's.

When we talk about a database, two concepts are important: *schemas* and *rows* of data. The schemas in a database are strict. By this, we mean that the database will only return valid data (i.e. no strings when we want a number–if we set it up that way). The simplest way to think about databases, at least at first, is like a spreadsheet: a table with rows and columns. The schema defines what columns you have, how they are named, and what *type* they have. In addition to type (e.g. `int`, `str`–actually SQL uses `varchar` instead of string), you can also define the allowable length, whether or not `Null` is allowed, and whether or not uniqueness is required. You may not want to allow nulls, for example, when building a Twitter database, so that you know every row you look at has a valid username. Uniqueness prevents (or allows) duplicate information. Often joint uniqueness is helpful when dealing with large data, for example name and social security number as opposed to name alone. Note that SQL databases *require* you to have a unique (or joint unique) identifier for each row.

Why are they called 'relational databases?' Good question. Mostly because people who work on databases are striving to "normalize" the data. Say we have a database with three columns, and a value in one of them changes. The concept of normalization means that the nominal identifier in our database (say, a name) is also assigned a unique identifier (usually a number), so that if a change is made to the nominal identifier, we can still keep track of what happened. It also allows us to make changes quickly, and is optimized for both time and memory.

7.1 Schemas

Relational databases assume you're going to normalize, so they make it very quick to organize tables. Consider the example in Tables 8 and 9.

Table 8: An Example Database

Book_Title	author_id
Bible	1
Farenheit 451	2
October Country	2
Cat's Cradle	3
Catch 22	NULL

Now say we join them using an *inner join*. That would return Table 10: There are also

Table 9: A Normalizing Table

1	God
2	Bradbury
3	Vonnegut
4	Tolkien

Table 10: An Inner Join

Bible	God
F451	Bradbury
OC	Bradbury
CC	Vonnegut

left and right *outer joins* that work similarly, but will return NULL for objects that it cannot match.

7.2 ORM

ORM stands for **o**bject-**r**elational **m**odel. It is a mapping of objects to database tables. We will look at what the SQL commands do, and how we could right them on our own. If you have questions about SQL, the internet is a vast and wonderful place to find answers.

8 Intro to Classifiers

We will discuss the most mathematically tractable, easiest to understand classifier: naive Bayes classifiers. They are also the easiest to implement, and happen to work shockingly well on lots of classifiers. The output of a classifier is the probability that a thing belongs in a set. A simple introductory example is the classification of email as spam/not spam.

To build a classifier, you first need to understand the following things about your problem domain:

- what you want to classify (the *document*)
- *features* of the document
- which *features* you want to emphasize for classification (i.e. which will be good predictors)
- the *categories* you wish to sort to

8.1 Bayes' Rule: Review and Example

Bayes' Rule is about taking some conditional probability and then using it to calculate other conditional probabilities.

$$P(\text{spam}|\text{features}) = \frac{P(\text{features}|\text{spam})P(\text{spam})}{P(\text{features})}$$

We will use what is called the "bag of words" approach. In the example above, our features are word counts. The "bag of words" approach only deals with individual words rather than phrases. We don't want to evaluate all of the n-grams in a 1,000-word text document because the combinatorics are huge. To deal with this, we look at the text with the assumption that words appear independently. Note that this is a huge assumption that does not hold in reality, but it makes the math clean and works fine for most cases.[6]

$$\begin{aligned} P(\text{spam}|\text{"male enhancement"}) &= \frac{P(\text{"male enhancement"}|\text{spam})P(\text{spam})}{P(\text{"male enhancement"})} \\ &= \frac{P(\text{"male"}|\text{spam})P(\text{spam})P(\text{"enhancement"}|\text{spam})}{P(\text{"male"})P(\text{"enhancement"})} \end{aligned}$$

Here is an example from the Stanford AI course:

Table 11: Spam vs. Ham Phrases

Spam	Ham
Offer is secret	Play sports today
Click secret link	Went play sports
Secret sports link	Secret sports event
	Sports is tody
	Sports cost money

[6]Recall our earlier discussion about "most cases."

Let's use the data in Table 11 to estimate the conditional probabilities:

$$\begin{aligned} P(\text{"secret"—spam}) &= \frac{3}{9} \\ P(\text{"secret"—ham}) &= \frac{1}{15} \\ P(\text{spam—"sports"}) &= \frac{P(\text{"sports"—spam})P(\text{spam})}{P(\text{"sports"}|\text{spam})P(\text{spam})P(\text{"sports"}|\text{ham})P(\text{ham})} \\ &= \frac{\frac{1}{9} \times \frac{3}{8}}{\frac{1}{9} \times \frac{3}{8} + \frac{1}{3} \times \frac{5}{8}} \end{aligned}$$

Notice that this is probability for an individual word. Also, realize that our preferences over Type I versus Type II errors might cause us not to make 50 percent our threshold for categorization.

8.2 LaPlace Smoothing

One problem with Bayes' Rule is that in the naivest of cases it does not know how to handle something it has not seen before. A LaPlace smoother adds a small constant and re-adjusts according to the rules of probability to ensure that there are not zero probabilities that a message is spam. (This is akin to adding .001 if you are trying to log event counts, etc.)

Compare the naive and Laplace-smoothed estimators below:

$$\begin{aligned} MLp(x) &= \frac{count(x)}{N} \\ LSp(x) &= \frac{count(x) + k}{N + k|x|} \end{aligned}$$

Notice that the LS estimate converges to the ML estimator when the number of categories $|x|$ is large. k is arbitrary–choose what works. 1 is usually a good place to start.

8.3 Pseudo-Implementation

A minimum-working example of a classifier consists of three things: a hash of words, a training method, and a classifier method.

Some other terms to keep in mind when implementing are:

- word breaking: a method to recognize that Doctor, Dr., and Dotcor all represent essentially the same thing to the human mind

- word stemming: a method to recognize that "angry" and "anger" each consist of "angr-" and are thus functionally equivalent for purposes of classification (often this results in a substantial improvement in probability calculation)

- stop words: words like "in, at" and "the" can lead to spurious results and take up space in your data storage, so most folks ignore them

There are libraries for all of these made by people with a lot of experience–use those. (Note that all of our examples today are English-specific; word stemming may not work in many languages, and word breaking can be extremely hard.)

References