Topic 0: Getting Started

Python Version

```
import sys
print(sys.version)
```

Markdown

Besides cells for code, you can edit or create cells intended to contain formatted text. The "language" used for such cells is called Markdown, which you can read a little more about here. It's like a simplified HTML designed to be readable both as plain text but also rendered as formatted text.
Unfortunately, there are many flavors of Markdown and it is not always clearly documented what features are available in any particular environment. You may need to do some experimenting to figure out how to get text look the way you want, so just do your best when required.

Embedding an image in Markdown: ![Alt text](http://cse6040.gatech.edu/datasets/mystery-image.jpg)

Getting input data

Throughout the course, we'll use a variety of methods to get data for use in the notebook environment. One technique is to use magic commands or shell commands. These are code-like constructs that are specific to Jupyter but outside the base language (e.g., Python). They typically appear on lines of code prefixed by ! or %.

Here is an example that downloads a file containing a secret message:

```
# Download:
!curl -O https://cse6040.gatech.edu/datasets/message_in_a_bottle.txt.zip

# Confirm (from shell):
!echo && echo "=== Files in the current directory (from a shell command) ===" && echo && ls -al

# Confirm (from Python):
```

```
import os
print("\n=== Files in the current directory (from Python) ===\n{}".format(os.listdir('.')))
```

Download a web into a file

```
import requests
import os
import hashlib

if os.path.exists('.voc'):
    data_url = 'https://cse6040.gatech.edu/datasets/yelp-example/yelp.htm'
else:
    data_url = 'https://github.com/cse6040/labs-fa17/raw/master/datasets/yelp.htm'

if not os.path.exists('yelp.htm'):
    print("Downloading: {} ...".format(data_url))
    r = requests.get(data_url)
    with open('yelp.htm', 'w', encoding=r.encoding) as f:
        f.write(r.text)

with open('yelp.htm', 'r') as f:
    yelp_html = f.read().encode(encoding='utf-8')
```

Inputs from Zip

```
from zipfile import ZipFile
with ZipFile(filename, 'r') as input_zip:
    with input_zip.open(filename[:-4], 'r') as input_file:
        message = input_file.readline().decode('utf-8')
```

IPython

```
from IPython.display import display
```

Timing a function

Before moving on, be sure you understand what the above benchmark is doing. For more on the Jupyter "magic" command, `%timeit`, see:
http://ipython.readthedocs.io/en/stable/interactive/magics.html?highlight=magic#magic-magic

```
timing = %timeit -q -o re.match (pattern, s_n)
t_avg = sum (timing.all_runs) / len (timing.all_runs) / timing.loops / n * 1e9
```

Topic 1: Python Essentials

Basics in Python

[Python Bootcamp]

Print to output

```
print("Hello, world!")
```

Assert condition

Variable value

```
assert x_float == 1
```

Variable type

```
assert type(x_float) is float
```

Instance of class

```
isinstance(x, Number)
```

All elements are TRUE

```
all()
```

Random generators

Pick a random element from iterator

from random import choice
return choice('abcdefghijklmnopqrstuvwxyz')

Random number from 0 to 1

random()

Samples from a list

sample(range(1000), 10)

Random integer from a range

from random import randint
randint(0, 20) (includes 20)
from random import randrange
randrange(1, 10) (doesnt include 10)

Arithmetic operators

Floor division

//

Modulus

%

Collections

Lists

Filtering

[element for element in L if element != x]

Copy

L.copy()

Count ocurrences of element

[1,2,3,0].count(0)

Adding elements

Extend takes an iterable (such as a list, tuple or string), and adds each element of the iterable to the list one at a time, while append adds its argument to the end of the list as a single item. The key thing to note is that extend is a more efficient version of calling append multiple times.

list.append(‘a’)
list.extend([‘a’,’b’])

Creating tuples out of lists putting together all elements of same index

zip(*grades[1:]) * unpacks list in this case grades[1:] is a list of lists
Returns zip object, can be converted to list

Sorting

reversed(list) reverses a list
sorted(list, reverse = True, key) sorts a list and creates a new list
list.sort(key, reverse = True)
Note: Simplest difference between sort() and sorted() is: sort() doesn't return any value while, sorted() returns an iterable list.

Range

range(10) creates an iterator of 10 integers starting from 0. Or range(2,10) goes from 2 to 9.

Search for element

list.index('element')

Max

max()

Dictionary

Iterate over keys

for key in grades_by_assignment

Iterate over k,v pairs

For k,v in grade_lists.items()

Dictionary comprehension

{k:v for k, v in dict.items()}

Sort dictionary

It is not possible to sort a dict, only to get a representation of a dict that is sorted. Dicts are inherently orderless, but other types, such as lists and tuples, are not. So you need a sorted representation, which will be a list—probably a list of tuples.

To return only keys
sorted(dictionary, key=dictionary.get, reverse=True)
this sort iterates over the dictionary keys, using the number of word occurrences as a sort key .

To return a list of tuples with key, value
Import operator
sorted_x = sorted(x.items(), key=operator.itemgetter(1)) sorts by value
sorted_x = sorted(x.items(), key=operator.itemgetter(0)) sorts by key

Another way:
sorted(d.items(), key=lambda x: x[1])

Default dictionary

Python's built-in dictionaries, you always to have to check whether a key exists before updating it. For example, consider this code fragment:

```
D = {'existing-key': 5} # Dictionary with one key-value pair

D['existing-key'] += 1 # == 6
D['new-key'] += 1  # Error: 'new-key' does not exist!
```
The second attempt causes an error because 'new-key' is not yet a member of the dictionary.
So, a more correct approach would be to do the following:
```
D = {'existing-key': 5} # Dictionary with one key-value pair

if 'existing-key' not in D:
    D['existing-key'] = 0
D['existing-key'] += 1

if 'new-key' not in D:
    D['new-key'] = 0
D['new-key'] += 1
```

This pattern is so common that there is a special form of dictionary, called a default dictionary, which is available from the collections module: collections.defaultdict.
When you create a default dictionary, you need to provide a "factory" function that the dictionary can use to create an initial value when the key does not exist. For instance, in the preceding example, when the key was not present the code creates a new key with the initial value of an integer zero (0). Indeed, this default value is the one you get when you call int() with no arguments

```
from collections import defaultdict

D2 = defaultdict (int)
```

Usually, a Python dictionary throws a KeyError if you try to get an item with a key that is not currently in the dictionary. The defaultdict in contrast will simply create any items that you try to access (provided of course they do not exist yet). To create such a "default" item, it calls the function object that you pass in the constructor (more precisely, it's an arbitrary "callable" object, which includes function and type objects). For the first example, default items are created using int(), which will return the integer object 0. For the second example, default items are created using list(), which returns a new empty list object.

The constructor can be lambda: defaultdict(base_type) for dictionaryes

Counter

```
from collections import Counter
print(Counter(train.lang)) #returns a dictionary with counts per key
```

Time

from datetime import datetime

datetime_object = datetime.strptime('Jun 1 2005 1:33PM', '%b %d %Y %I:%M%p')

Debugging

from IPython.core.debugger import set_trace and then put set_trace() where the break needs to happen

n(ext) line and run this one
c(ontinue) running until next breakpoint
q(uit) the debugger

Topic 2: Pair Association mining

Problem

Association mining rule task:

Find all a → b such that Pr[b|a]>S where S is an analyst defined threshold

Pr[b|a] = Pr[b,a]/Pr(a)

Let's define 2 metrics:

Support = Na,b/M estimates Pr[b,a]
Confidence = Na,b/Na estimates Pr[b|a]

Algorithm

Given R = {r0,..,rm-1} all our receipts and A = {a0,..,an-1} the items

We'll need to tabulate T[a,b],C[a] = 0 for every a,b belonging to A
We'll add 1 to T[a,b] for each receipt if a and b are in the same receipt and 0 if they are not.
Same for C[a] we'll add 1 if a is in a receipt and 0 if not

```
For every r in R do:
    For every {a in r, b in r} do:
        T[a,b] = T[a,b] + 1
        T[b,a] = T[b,a] + 1

    For every a in r do:
        C[a] = C[a] + 1

For every (a in A, b in A) do:
    If T[a,b]/C[a] > S then:
        Output a → b
```

Observe that the bulk of the work in this procedure is just updating these tables, TT and CC. So your biggest implementation decision is how to store those. A good choice is to use a dictionary

Efficiency

If all possible pairs occur then the table will need n^2 entries

It's better to design algorithms that scale like O(n) or O(n log n)
More generally if we have an input of size n and an output of size k, we would want an algorithm whose scaling like O(n+k) or O((n+k)log(n+k)).

In computer science this type of algorithm that scales linearly O(n+k) is called work optimal.

Combinations

from itertools import combinations
combinations(set,size)

Topic 3: Math Prerequisite review

Probability

Counting

P(A) = #Outcomes A / #Total outcomes

Combinations

(n k) = n! / k!(n-k)!

Bayes Rule

P(A|B) = P(B|A) P(A) / P(B)

Calculus

Derivatives

Product rule

d/dx (a(x) * b(x)) = a(x) * db/dx + da/dx * b(x)

Chain Rule

d/dx g(h(x)) = d(g(y))/dy * dy/dx

Integration

Integration by parts

Linear Algebra

Inner Product

xTx

Outer Product

xxT

Vector i norm

||x|| = sum(|x|^i)^1/i

Infinity norm simplifies to the max element of the vector

Frobenius norm

Topic 4: Representing Numbers

Representing numbers

In general, the value of a string of d+1 digits in base b is:

[Sd Sd-1 … S1 S0]b = Sd*b^d + Sd-1*b^d-1 + .. + S1 * b + S0 = Σi=0tod Si*b^i

Computer hardware uses binary digits to represent numbers.

Bit = short hand for Binary Digit

Representing fractional values

Same as integers, but with decreasing negative i's for numbers after the decimal point starting with -1.

Representation of some numbers may be finite in base 10 but infinite in base 2, for example 0.1

As there is no infinite storage, computers approximate, then we will need to learn to live with approximation.

If we need exact values there will be a way to achieve it but it will come at a computational cost.

int(s,base) in Python gives value of a string in a given base.

Points that float

Floating point numbers are stored in computers and manipulated via the concept of floating point storage.

Floating point comes from the idea of having a fixed storage which means having to keep only a certain amount of digits. Then we want to place the fractional point arbitrarily according to the number.

To achieve this in a systematic way, we use a floating point encoding which is a normalized scientific notation consisting of a base, a sign, a fractional significand or mantissa, and a signed integer exponent. Conceptually, think of it as a tuple of the form, $(\pm,[s]b,x)$, where b is the digit base (e.g., decimal, binary); $\pm$ is the sign bit; s is the significand encoded as a base b string; and x is the exponent.

Rounding Errors

The phenomenon of losing digits because of finite precision is what is known as rounding or round-off error.

Now, rounding errors are a fact of life whenever you're trying to do a numerical Computation.

It doesn't mean you can't still write a program that gives a good answer but it does mean you should try to anticipate or at least learn to diagnose potential

The IEEE-754 Binary Encoding

Almost all modern computers store floating point values in the same way.

It's a format known as the IEEE 754 standard.

(+-, 1.S-1,S-2,..,S-d, +- xp-1,xp-2,..,x0)2

No need to store the first 1 before the fractional part, it is always one in binary.

The exponent part is slightly asymmetrical to allow for Infinite or NaN values.

Single Precision

In IEEE single precision (goes by the name of float), there is a total of 32 bits to store numbers: 1 bit for the sign, 24 bits for the significand (it's actually 23 bits because we don't need to stor the first 1) and 8 bits for the exponent [-126,127]

If you are familiar with languages like C, C++, or Java, then IEEE single-precision is the float primitive in those languages.

Machine Epsilon

Let v = s (exact number) and vhat = shat the closest float, then the relative error is:

abs(vhat - v) / abs(v) = abs(shat - s) / abs(s) <= Ɛ / abs(s) <= Ɛ

This is called the machine epsilon and equals 2^-23 or in base ten 1.19^-7

Double Precision

Python, R, and MATLAB, by default, store their floating-point values in a standard tuple representation known as IEEE double-precision format. It's a 64-bit binary encoding having the following components:

The most significant bit indicates the sign of the value.
The significand is a 53-bit string with an implicit leading one. That is, if the bit string representation of s is $s_0.s_1s_2\cdots s_d$, then $s_0=1$ always and is never stored explicitly. That also means $d=52$ and the size is actually 52 bit.
The exponent is an 11-bit string and is treated as a signed integer in the range $[-1022,1023]$.

Machine epsilon

Thus, the smallest positive value in this format $2^{-1022} \approx 2.23 \times 10^{-308}$, and the smallest positive value greater than 1 is $1 + \epsilon$, where $\epsilon = 2^{-52} \approx 2.22 \times 10^{-16}$ is known as machine epsilon (in this case, for double-precision).

Python allows to get the machine epsilon via package numpy

```
import numpy as np
EPS_S = np.finfo (np.float32).eps
EPS_D = np.finfo (float).eps
```

Other languages

If you are familiar with languages like C, C++, or Java, then IEEE double-precision format is the same as the double primitive type. The other common format is single-precision, which is float in those same languages.

v.hex()

v.hex() gives hexadecimal representation of a Python float value v = 16.0625 ==> v.hex() == '0x1.0100000000000p+4' where:

If v is negative, the first character of the string is '-'.
The next two characters are always '0x'.
Following that, the next characters up to but excluding the character 'p' is a fractional string of hexadecimal (base-16) digits. In other words, this substring corresponds to the significand encoded in base-16.

The 'p' character separates the significand from the exponent. The exponent follows, as a signed integer ('+' or '-' prefix). Its implied base is two (2)---not base-16, even though the significand is.

A framework for error analysis

Analysing floating point programs:

Suppose f(x) is a function to compute and we write an algorithm alg(x) to compute f(x).

Forward stability analysis

alg(x) will approximate f(x). The calculation of the bound |alg(x)−f(x)| <= ? is called forward stability analysis.

If the forward error is small we can claim the algorithm is forward stalbe.

Forward error can be hard to compute.

Backward stability analysis

In particular, alg(x) is a backward stable algorithm to compute f(x) if, for all x , there exists a "small" Δx such that
alg(x) = f(x+Δx)

For data analysis we could think that backward analysis is more natural. x represents the input data, which is often noisy. So if your program produces any answer that is within that noise, which is delta x, then you might feel perfectly fine using it.

Link between 2 analysis

Suppose alg(x) = f(x+deltax) and f(x) is differentiable, then

abs(alg(x) - f(x)) = abs(f(x+deltax) - f(x))

By Taylor's theorem: f(x+deltax) = f(x) + deltax df/dx

Which means abs(alg(x) - f(x)) = deltax df/dx which means if we show backward error is small then forward error will also be small.

Machine error

Coming back to analysing a floating point problem, let's say we have an operation a+b, then the operation will approximate the real value with fl(a+b).

fl(a+b) ≡ (a+b)(1+δ),

Well, first note that δ will in general depend on the operands and b, as well as what operation we're doing.

But you'll have relative errors for other operations, like multiplication and division. Unfortunately, we usually have no idea exactly what the value of delta is, since we don't know the true answer.

But from our understanding of floating point arithmetic, we do know it's worst case value. And that value is epsilon, as in machine precision. Then $|\delta| \leq \epsilon$

Computing a sum

Exact arithmetic analysis

Let $x \equiv (x_0, \ldots, x_{n-1})$ be a collection of input data values. Suppose we wish to compute their sum.

The exact mathematical result is
$f(x) \equiv \sum_{i=0,n-1} x_i$.

Given x, let's also denote its exact sum by the synonym $s_{n-1} \equiv f(x)$.

An algorithmic way for this computation would be:

```
Def alg_sum(x):
        S = 0.0
        For x_i in x:
                S +=x_i
        Return s
```

Considering exact arithmetics, then, it can be proven by induction that our program gives the exact result.

First, assume that the for loop enumerates each element p[i] in order from i=0 to n-1, where n=len(p). That is, assume p_i is p[i].

Let $p_k \equiv p[k]$ be the k-th element of p[:]. Let $s_i \equiv \sum_{k=0,i} p_k$; in other words, s_i is the exact mathematical sum of p[:i+1]. Thus, s_{n-1} is the exact sum of p[:].

Let $\hat{s}_{-1}$ denote the initial value of the variable s, which is 0. For any $i \geq 0$, let $\hat{s}_i$ denote the computed value of the variable s immediately after the execution of line 4, where i=`i`. When i=i=0 , $\hat{s}_0 = \hat{s}_{-1} + p_0 = p_0$, which is the exact sum of p[:1]. Thus, $\hat{s}_0 = s_0$.

Now suppose that $\hat{s}_{i-1} = s_{i-1}$. When i=i , we want to show that $\hat{s}_i = s_i$. After line 4 executes, $\hat{s}_i = \hat{s}_{i-1} + p_i = s_{i-1} + p_i = s_i$. Thus, the computed value $\hat{s}_i$ is the exact sum s_i.

If i=n, then, at line 5, the value s = ŝn−1 = sn−1, and thus the program must in line 5 return the exact sum.

Floating point analysis

After the first iteration of our program, now we have:

ŝ0 = fl[ŝ-1 + x0]

According to our model of floating point error then:

ŝ0 = (ŝ-1 + x0) (1+δ0) with δ0 rounding error associated with ŝ0, but we know ŝ-1 is 0 and therefore there is no rounding error, which means δ0 is 0 and

ŝ0 = x0

Let's also consider si as the exact partial sums, then:

ŝ1 = s1 + s1δ1

ŝ2 = s2 + s1δ1 + s2δ2 + s1δ1δ2

We can get to:

ŝk = sk + Σ i=0tok siδi + O(δlδm)

Which means that

ŝk ≈ sk + Σ i=0tok siδi

Backward stability analysis

Using a backward error analysis, you can show that

ŝ n−1 ≈ ∑i=0,n−1 xi(1+Δi) = f(x+Δ),

where Δ ≡ (Δ0,Δ1,…,Δn−1). In other words, the computed sum is the exact solution to a slightly different problem, x+Δ .

To complete the analysis, you can at last show that

$|\Delta i| \leq (n-i)\epsilon$,

where ϵ is machine precision. Thus, as long as $n\epsilon \ll 1$, then the algorithm is backward stable and you should expect the computed result to be close to the true result.

Interpreted differently, as long as you are summing $n \ll 1\epsilon$ values, then you needn't worry about the accuracy of the computed result compared to the true result:

In the case of this summation, we can quantify not just the backward error (i.e., Δi) but also the forward error. In that case, it turns out that

$|s\hat{}n-1-sn-1| \lessapprox n \epsilon |x|1$

More accurate summation

Suppose you wish to compute the sum, $s=x0+x1+x2+x3$. Let's say you use the "standard algorithm," which accumulates the terms one-by-one from left-to-right, as done by alg_sum() above.

For the standard algorithm, let the i-th addition incur a roundoff error, δi. Then our usual error analysis would reveal that the absolute error in the computed sum, $\hat{s}$, is approximately:

$\hat{s} - s \approx x0(\delta 0+\delta 1+\delta 2+\delta 3) + x1(\delta 1+\delta 2+\delta 3) + x2(\delta 2+\delta 3) + x3\delta 3.$

And since $|\delta i| \leq \epsilon$, you would bound the absolute value of the error by,

$|\hat{s} - s| \lesssim (4|x0|+3|x1|+2|x2|+1|x3|)\epsilon.$

Notice that $|x0|$ is multiplied by 4, $|x1|$ by 3, and so on.

In general, if there are n values to sum, the $|xi|$ term will be multiplied by $n-i$. Therefore sorting the collection x from smallest to largest will cause the algorithm to incur in less error.

Topic 5: Pattern matching with strings

Concatenation

''.join(['hello','world'])

Split

s.split()

Lowercase

s.lower()

Replace a set of characters with another character

"abcabc".translate({ord('a'): 'd', ord('c'): 'x'})
translate(str.maketrans("ATCG","TAGC"))
replace('toreplace','replacement')

Find a substring

string.find(substring,starting point) returns the index where the substring is found
Returns -1 if not found

Text validators

text.isdigit()
text.isspace()
text.islower()
text.isupper()
text.isalpha()

Replace in string with variable

'^a{%d}$' % n
"Matching input '{}' against pattern '{}'...".format (s_n, pattern)

Count letters

string.count(letter)

Regular expressions

There is a beautiful theory underlying regular expressions, and efficient regular expression processing is regarded as one of the classic problems of computer science. In the last part of this lab, you will explore a bit of that theory, albeit by experiment.

Metacharacters

[] is used to specify character classes. [abc] = [a-c] means match any of a, b or c
^ gets the complement set. [^5] matches any character except number 5. Also when put at the beginning of a sequence means the pattern must be matched at the beginning of the line.
\ can be used to escape other metacharacters in Python. Some sequences that start with a backlash represent certain characters.
\d and \D represent decimal digits and non-decimal digits respectively
\s and \S represent spaces and non-spaces
\w and \W match any alpha-numeric character and any non alpha-numeric character respectively
$ matches patterns at the end of the line
. matches anything except a new line character, .DOTALL can be added as a parameter so . matches also new line
\n matches new line

Counting

+ matches 1 or more of the preceding expression
* means the previous character can be matched 0 or more times
? used to match 0 or 1 of the prvious character
{m,n} matches from m to n ocurrences of the previous character

Grouping

Grouping allows us to dissect string and divide them into different subgroups,
which will allow us to match different components. We can denote groups using parentheses.
(ab)* means we can match ab 0 or multiple times

In Python

A regular expression is specially formatted pattern, written as a string. Matching patterns with regular expressions has 3 steps:
You come up with a pattern to find.

You compile it into a pattern object.
You apply the pattern object to a string, to find matches, i.e., instances of the pattern within the string.

Import re
Pattern_object = re.compile(pattern)
Matches = pattern_object.search(input)

matches.group() gives what has been matched
matches.groups() gives all the matched groups
matches.start() gives starting of the pattern in the input
matches.end() gives end of the pattern in the input
matches.span() gives starting and end of the pattern

Module-level searching. For infrequently used patterns, you can also skip creating the pattern object and just call the module-level search function, re.search().

matches_2 = re.search ('jump', input)

Other Search Methods
re.match(pattern,string) - Determine if the RE matches at the beginning of the string. Match has also groups
search() - Scan through a string, looking for any location where this RE matches.
findall() - Find all substrings where the RE matches, and returns them as a list.
finditer() - Find all substrings where the RE matches, and returns them as an iterator.

Patterns can be defined in multiple lines with argument VERBOSE:

```
re_names2 = re.compile ('''^              # Beginning of string
                    ([a-zA-Z]+)    # First name
                    \s             # At least one space
                    ([a-zA-Z]+\s)? # Optional middle name
                    ([a-zA-Z]+)    # Last name
                    $              # End of string
                  ''',
                  re.VERBOSE)
```

Groups from a match can be accessed through groups():

re_names2.match ('Rich Vuduc').groups () where re_names2 is a compiled pattern object.

Groups can be tagged and accessed via groups(‘tag’) like:
(?P<first>[a-zA-Z]+)

Non-capturing groups can be defined by (?:pattern)

Topic 6: Mining the web

Requests module

import requests

response = requests.get('http://www.gatech.edu/')
webpage = response.text # or response.content for raw bytes

https://www.yelp.com/search?find_desc=ramen&find_loc=Atlanta%2C+GA&ns=1: This URL encodes what is known as an *HTTP "get"* method (or request). It basically means a URL with two parts: a *command* followed by one or more *arguments*.

In this case, the command is everything up to and including the word search; the arguments are the rest, where individual arguments are separated by the & or #.

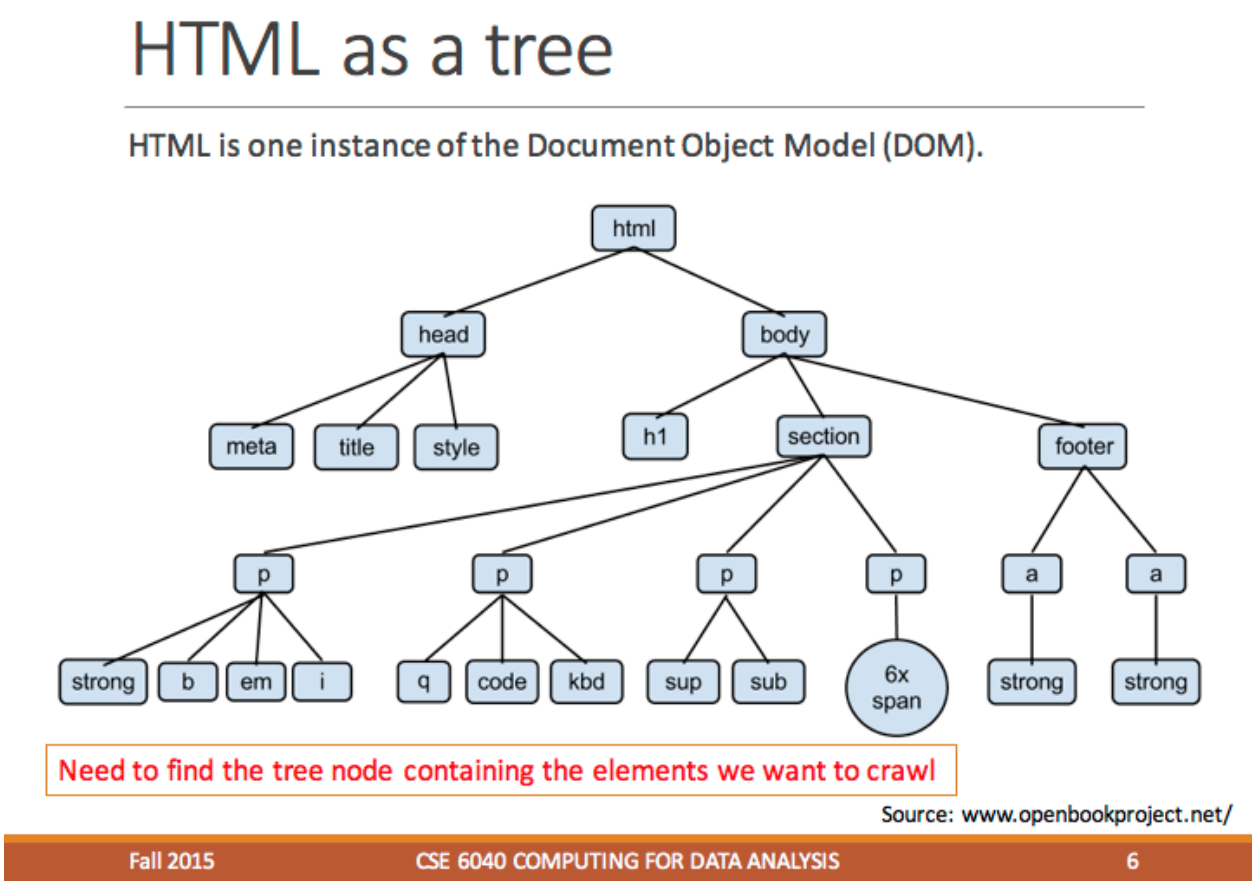
"HTTP" stands for "HyperText Transport Protocol," which is a standardized set of communication protocols that allow *web clients*, like your web browser or your Python program, to communicate with *web servers*.

In this next example, let's see how to build a "get request" with the requests module. It's pretty easy!

```
url_command = 'http://yelp.com/search'
url_args = {'find_desc': "ramen",
            'find_loc': "atlanta, ga"}
response = requests.get (url_command, params=url_args)
```

Beautiful Soup

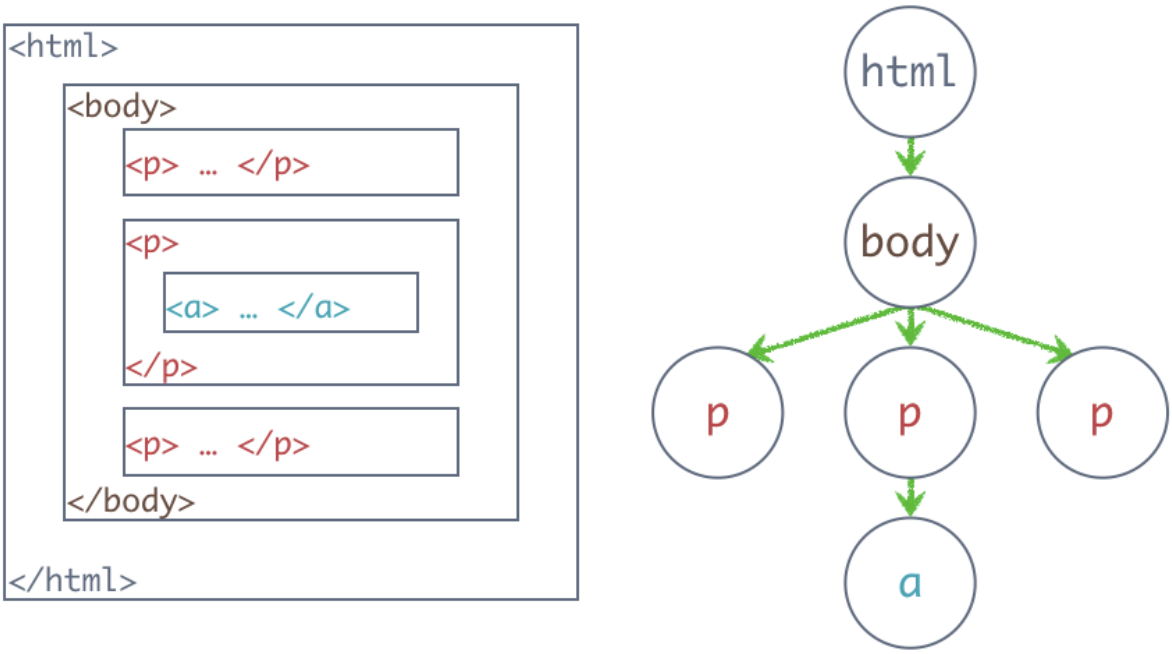
Any HTML document may be modeled as an object in computer science known as a tree:



Consider a tree is a collection of *nodes*, which are the labeled boxes in the figure, and *edges*, which are the line segments connecting nodes, with the following special structure.

- The node at the top is called the *root*. Here, the root is labeled html and abstractly represents the entire HTML document.
- Regard each edge as always "pointing" from the node at its top end to the node at its bottom end. For any edge, the node at its top end is the *parent* and the node at the bottom end is a *child*. Like real families, a parent can be a child. For example, the node labeled head is the child of html and the parent of meta, title, and style.
- The *descendant* of a node x is any node y for which there is a path from x going down to y. For example, the node labeled 6x span is a descendant of the node body. All nodes are descendants of the root.
- Any node with *no* descendants is a *leaf*.
- Any node that is neither a root nor a leaf is an *internal node*.
- There are no *cycles*. A cycle would be a loop. For instance, if you were to add an edge between the two lower rightmost nodes labeled, strong and strong, that would create a loop and the object would no longer be a tree.

The Beautiful Soup package gives you a data structure for traversing this tree.



from bs4 import BeautifulSoup

soup = BeautifulSoup(some_page, "lxml")

Soup
Soup.html
Soup.html.body
Soup.html.body.contents

Observe that the . notation allows us to reference HTML tags---that is, the stuff enclosed in angle brackets in the original HTML, e.g., <html> ... </html>, <body> ... </body>---as they are nested. But in the case of the <body> ... </body> tag, there are multiple subtags. Evidently, soup.html.body.contents contains these, as a list, which we know how to manipulate.

A tag type is: type(link) is bs4.element.Tag
Tag have names: link.name == 'a'
We can refer to tag properties tag['href'] == 'http://www.gatech.edu'
Tag can have contents tag.contents == ['Georgia Tech website']

Beautiful Soup gives us a way to search for specific tags.

soup.find_all(attrs={'class': 'indexed-biz-name'}) will return a list of tags for the specific search
Or soup.find will find the next iteration

Selenium

```
from selenium import webdriver

htmls = []

driver = webdriver.Chrome()
driver.get('http://www.racetecresults.com/Results.aspx?CId=16410&RId=278&EId=6')
htmls.append(driver.page_source)
driver.find_element_by_xpath("""//*[@id="ctl00_Content_Main_grdTopPager"]/tbody/tr/td[2]/a""").
click()
htmls.append(driver.page_source)
driver.find_element_by_xpath("""//*[@id="ctl00_Content_Main_grdTopPager"]/tbody/tr/td[3]/a""").
click()
htmls.append(driver.page_source)
driver.find_element_by_xpath("""//*[@id="ctl00_Content_Main_grdTopPager"]/tbody/tr/td[4]/a""").
click()
htmls.append(driver.page_source)
driver.quit()
```

Web API's

Luckily, many websites provide an application programming interface (API) for querying their data or otherwise accessing their services from your programs. For instance, Twitter provides a web API for gathering tweets, Flickr provides one for gathering image data, and Github for accessing information about repository histories.

These kinds of web APIs are much easier to use than, for instance, the preceding technique which scrapes raw web pages and then has to parse the resulting HTML. Moreover, there are more scalable in the sense that the web servers can transmit structured data in a less verbose form than raw HTML.

Github

```
import requests
response = requests.get ('https://api.github.com/repos/cse6040/labs-fa17/events')
response.headers['Content-Type'] = application/json; charset=utf-8
response.json() gives the response in json format.
```

JSON

The response is in JSON format, which is an open format for exchanging semi-structured data. (JSON stands for **J**ava**S**cript **O**bject **N**otation.) JSON is designed to be human-readable and machine-readable, and maps especially well in Python to nested dictionaries.

```
import json
print(type(response.json ()))
print(json.dumps(response.json()[:3], sort_keys=True, indent=2))
```

Topic 7: Tidying data

Tidy data

Definition: Tidy datasets. More specifically, Wickham defines a tidy data set as one that can be organized into a 2-D table such that

1. each column represents a *variable*;
2. each row represents an *observation*;
3. each entry of the table represents a single *value*, which may come from either categorical (discrete) or continuous spaces.

This definition appeals to a statistician's intuitive idea of data he or she wishes to analyze. It is also consistent with tasks that seek to establish a functional relationship between some response (output) variable from one or more independent variables.

A computer scientist with a machine learning outlook might refer to columns as *features* and rows as *data points*, especially when all values are numerical (ordinal or continuous).

Definition: Tibbles. Here's one more bit of terminology: if a table is tidy, we will call it a *tidy table*, or *tibble*, for short.

Pandas

In Python, the Pandas module is a convenient way to store tibbles. If you know R, you will see that the design and API of Pandas's data frames derives from R's data frames.

In a Pandas data frame, every column has a name (stored as a string) and all values within the column must have the same primitive type. This fact makes columns different from, for instance, lists.

In addition, every row has a special column, called the data frame's *index*. (Try printing irises.index.) Any particular index value serves as a name for its row; these index values are usually integers but can be more complex types, like tuples.

In [13]:

Separate from the index values (row names), you can also refer to rows by their integer offset from the top, where the first row has an offset of 0 and the last row has an offset of n-1 if the data frame has n rows.

Creating a Data Frame

Dictionary with lists

```
pd.DataFrame(
   {'lst1Tite': lst1,
    'lst2Tite': lst2,
    'lst3Tite': lst3
   })
```

Matrix

```
pd.DataFrame(np.column_stack([lst1, lst2, lst3]),
                          columns=['lst1tite', 'lst2itie', 'lst3tite'])
```

Read from csv

```
pandas.read_csv("airlines.csv", header=None, dtype=str)
```

Read from sql

Time

Convert to time

```
data['finish_time'] = pd.to_datetime(data[data['status']=='Finished']['finish_time'],
format='%H:%M:%S').dt.time
```

We can access attributes like data.finsh_time.dt.hour, weekday, dayofyear,

Compares to timestamps pd.to_datetime(‘2017-10-01’)
Has max and min attribute. Operations output Timedelta object

Counting

pd.data.column.value_counts() ouotputs count per value of column.

Explore

irises.describe()
irises.head()

df.tail()

df.shape
irises[["sepal length", "petal width"]].head()

Df.columns

data.dtypes

df.info()

Slice and dice

By column name

airport_codes[['Code']] if we pass [‘column’] the result is a data frame, if we pass ‘column’ its a pandas series.

irises[["sepal length", "petal width"]]

By index

loc works on labels in the index.
iloc works on the positions in the index (so it only takes integers).
ix usually tries to behave like loc but falls back to behaving like iloc if the label is not in the index.

Irises.iloc[5:10]

By element value

```
irises[irises["sepal length"] > 5.0]

df.query('col1 <= 1 & 1 <= col1')

.query("(Description_x == 'Atlanta\\'s Hartsfield-Jackson International Airport')
```

Regex

```
str.contains(regex)
```

Sort

```
irises.sort_values(by="sepal length", ascending=False).head(1)
irises.sort_values(by="sepal length", ascending=False).iloc[5:10]
irises.sort_values(by="sepal length", ascending=False).loc[5:10]
irises['x'] = 3.14

airline_route_lengths.sort("length", ascending=False)
```

Alter

Rename columns

```
irises.rename(columns={'species': 'type'}, inplace = True)

Airports.columns = ["id", "name", "city", "country", "code"]
```

Delete a column

```
del irises['x']

df.drop(‘column’, axis = 1)
```

Drop duplicates

```
df.drop_duplicates()
```

Drop NA’s

```
data1["Type 2"].dropna(inplace = True)
```

Fill NA’s

```
data["Type 2"].fillna('empty',inplace = True)
```

Moves index to column

df.reset_index(drop = True) #drop parameter drops the new created column

Drop index

.drop('index', axis=1)

Concatenate

pd.concat([data1,data2],axis =0,ignore_index =True) vertical concatenation

pd.concat([data1,data2],axis =1,ignore_index =True) horizontal concatenation

Change type

data['Type 1'].astype('category')

Input

pd.read_csv(local_filename)

Summaries

irises["sepal length"].max()
irises['species'].unique()

Group

df.groupby(['ORIGIN_AIRPORT_ID', 'DEST_AIRPORT_ID'], as_index=False) #as.index = True transforms group columns into indexes.

Count

groups.somecolumnname.count() #after grouping we can count specifying a column

series.value_counts()

df.count() counts not na records per column

Count distinct

nunique()

Any function

route_length_df.groupby("id").aggregate(numpy.mean)

Window functions

Rank

```
Drugs[Drugs['Med'] == 'Med A'].sort_values('Admin Date').groupby(['ID', 'Med']).head(1)
```

Other functions

Check for nulls

```
data['Type 2'].notnull().all()
```

Split string

```
pdseries.str.split('/')
```

Iterate

Over rows

```
for index, row in df.iterrows():
  ....:     print row['c1'], row['c2']
```

Compare 2 tables

```
def canonicalize_tibble(X):
    var_names = sorted(X.columns)
    Y = X[var_names].copy()
    Y.sort_values(by=var_names, inplace=True)
    Y.reset_index(drop=True, inplace=True)
    return Y

def tibbles_are_equivalent (A, B):
    A_canonical = canonicalize_tibble(A)
    B_canonical = canonicalize_tibble(B)
    cmp = A_canonical.eq(B_canonical)
    return cmp.all().all()
```

Merging data frames

```
C = A.merge (B, on=['country', 'year']) #if different column names, can use left_on, right_on.
```

In this call, the on= parameter specifies the list of column names to use to align or "match" the two tables, A and B. By default, merge() will only include rows from A and B where all keys match between the two tables.

Joins. This default behavior of keeping only rows that match both input frames is an example of what relational database systems call an *inner-join* operation. But there are several other types of joins.

- *Inner-join (A, B)* (default): Keep only rows of A and B where the on-keys match in both.
- *Outer-join (A, B)*: Keep all rows of both frames, but merge rows when the on-keys match. For non-matches, fill in missing values with not-a-number (NaN) values.
- *Left-join (A, B)*: Keep all rows of A. Only merge rows of B whose on-keys match A.
- *Right-join (A, B)*: Keep all rows of B. Only merge rows of A whose on-keys match B.

You can use merge's how=... parameter, which takes the (string) values, 'inner', 'outer', 'left', and 'right'.

Merge by index

```
pd.merge(df1, df2, left_index=True, right_index=True)

df1.join(df2)

pd.concat([df1, df2], axis=1)
```

Left join where right is null

```
first_med_b.loc[list(set(first_med_b.index) - set(first_med_a.index))]
```

Apply

Another useful primitive is apply(), which can apply a function to a data frame or to a series (column or row of the data frame).

Column

```
G['year'] = G['year'].apply(lambda x: "'{:02d}".format(x % 100))
```

Row

routes.apply(calc_dist, axis=1)

Tidying transformations: Melting and casting

Given a data set and a target set of variables, there are at least two common issues that require tidying.

Melting

First, values often appear as columns. Table 4a is an example. To tidy up, you want to turn columns into rows:

country	year	cases
Afghanistan	1999	745
Afghanistan	2000	2666
Brazil	1999	37737
Brazil	2000	80488
China	1999	212258
China	2000	213766

country	1999	2000
Afghanistan	745	2666
Brazil	37737	80488
China	212258	213766

table4

Because this operation takes columns into rows, making a "fat" table more tall and skinny, it is sometimes called *melting*.

o melt the table, you need to do the following.

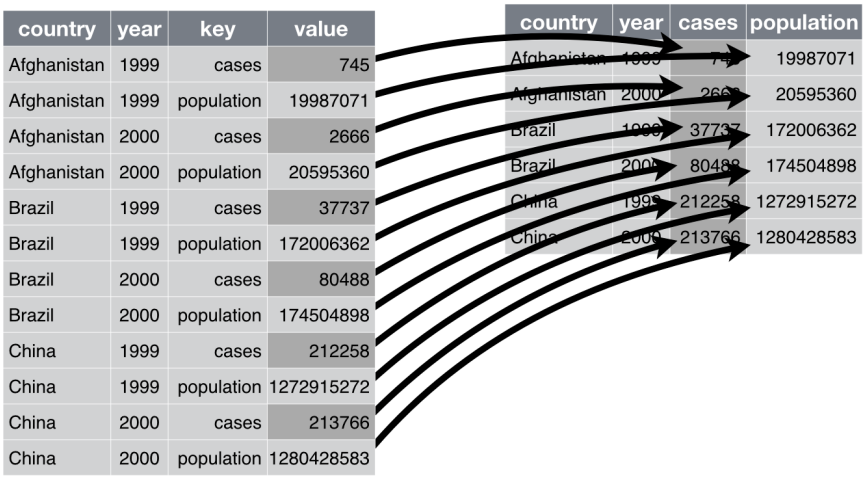
1. Extract the *column values* into a new variable. In this case, columns "1999" and "2000" of table4 need to become the values of the variable, "year".
2. Convert the values associated with the column values into a new variable as well. In this case, the values formerly in columns "1999" and "2000" become the values of the "cases" variable.

In the context of a melt, let's also refer to "year" as the new *key* variable and "cases" as the new *value* variable.

pd.melt(frame=data_new,id_vars = 'Name', value_vars= [1999,'2000'])

Casting

The second most common issue is that an observation might be split across multiple rows. Table 2 is an example. To tidy up, you want to merge rows:



country	year	key	value
Afghanistan	1999	cases	745
Afghanistan	1999	population	19987071
Afghanistan	2000	cases	2666
Afghanistan	2000	population	20595360
Brazil	1999	cases	37737
Brazil	1999	population	172006362
Brazil	2000	cases	80488
Brazil	2000	population	174504898
China	1999	cases	212258
China	1999	population	1272915272
China	2000	cases	213766
China	2000	population	1280428583

country	year	cases	population
Afghanistan	1999	745	19987071
Afghanistan	2000	2666	20595360
Brazil	1999	37737	172006362
Brazil	2000	80488	174504898
China	1999	212258	1272915272
China	2000	213766	1280428583

table2

Because this operation is the moral opposite of melting, and "rebuilds" observations from parts, it is sometimes called *casting*.

melted.pivot(index = 'Name', columns = 'variable',values='value')

Melting and casting are Wickham's terms from his original paper on tidying data. In his more recent writing, on which this tutorial is based, he refers to the same operation as *gathering*. Again, this term comes from Wickham's original paper, whereas his more recent summaries use the term *spreading*.

The signature of a cast is similar to that of melt. However, you only need to know the key, which is column of the input table containing new variable names, and the value, which is the column containing corresponding values.

Topic 8: Visualizing data and results

from IPython.display import display, Markdown
Display allows to pretty print objects.

Pyplot

matplotlib is a relatively low-level plotting library in the Python stack, so it generally takes more commands to make nice-looking plots than it does with other libraries. On the other hand, you can make almost any kind of plot with matplotlib. It's very flexible, but that flexibility comes at the cost of verbosity.

```
import matplotlib.pyplot as plt
%matplotlib inline
```

plt.show() shows the graph.

```
def spy(A, figsize=(6, 6), markersize=0.5):
    """Visualizes a sparse matrix."""
    fig = plt.figure(figsize=figsize)
    plt.spy(A, markersize=markersize)
    plt.show()
```

Customize

Title

title('title')

Labels

xlabel('')

Legend

plt.legend(loc='upper right')
fig.legend() shows the legend

Way of working

Initiate figure and add subplots

```
Fig = figure(figsize = (10,5))
Ax1 = fig.add_subplot(121)
ax1.plot(x,y)
```

Initiate subplots

fig, axes = plt.subplots(nrows=2,ncols=1), then pass axes = axes[index] to following plots
ax = plt.subplot()
ax.bar([1], career_switch.values,tick_label = career_switch.index.tolist())
ax.set_ylim([0,100])

Show the plot

plt.show()

sns.countplot(y=response['GenderSelect'],ax=ax[1])
sqft = fig.add_subplot(121), then we can refer to this plot
ax[0].axvline(salary['Salary'].median(),linestyle='dashed') adds vertical line

Histogram

plt.hist(route_lengths, bins=20)
data.Speed.plot(kind = 'hist',bins = 50,figsize = (15,15))

Bar chart

plt.bar(range(airline_route_lengths.shape[0]), airline_route_lengths["length"])

Scatter

plt.scatter(airlines["id"].astype(int), name_lengths)
data.plot(kind='scatter', x='Attack', y='Defense',alpha = 0.5,color = 'red')

Line

series.plot(kind = 'line', color = 'g',label = 'Attack',linewidth=1,alpha = 0.5,grid = True,linestyle = ':')

Boxplot

data.boxplot(column='Attack',by = 'Legendary')

3d plot

From matplotlib.pyplot import plot_wireframe, contour, quiver
ax.plot_wireframe(X,Y,Z)
ax.contour(X,Y,Z)
ax.quiver(X,Y,dX,dY, scale = .., headwidth = ..)

Bokeh

The design and use of Bokeh is based on Leland Wilkinson's Grammar of Graphics (GoG).

We call it like:

import bokeh

Bokeh is designed to output HTML, which you can then embed in any website. To embed Bokeh output into a Jupyter notebook, we need to do the following:

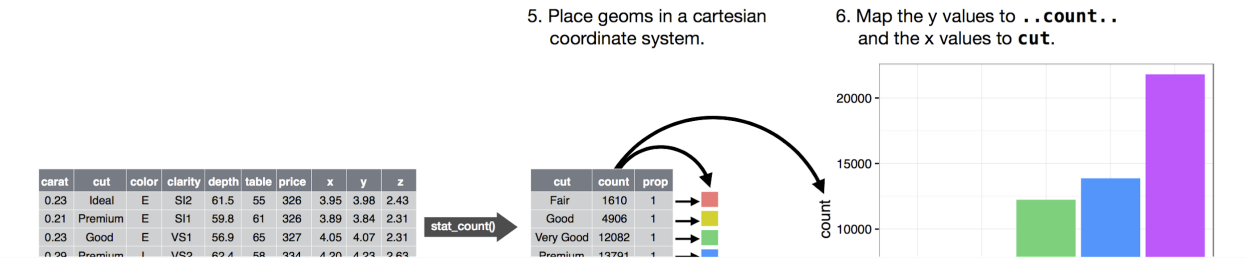
from bokeh.io import output_notebook
from bokeh.io import show
output_notebook ()

Philosophy: Grammar of Graphics

The Grammar of Graphics is an idea of Leland Wilkinson. Its basic idea is that the way most people think about visualizing data is ad hoc and unsystematic, whereas there exists in fact a "formal language" for describing visual displays.
The reason why this idea is important and powerful in the context of our course is that it makes visualization more systematic, thereby making it easier to create those visualizations through code.
The high-level concept is simple:

1. Start with a (tidy) data set.
2. Transform it into a new (tidy) data set.
3. Map variables to geometric objects (e.g., bars, points, lines) or other aesthetic "flourishes" (e.g., color).
4. Rescale or transform the visual coordinate system.

5. Render and enjoy!



(High-Level) Charts

The easiest way to use Bokeh is to call its interface for "canned" charts which can be imported:

from bkcharts import Histogram, Scatter, BoxPlot

We also import show for displaying the charts:

from bokeh.io import show and
From bokeh.io import output_notebook
output_notebook()

Histogram

The Histogram(df, c) function can take a Pandas data frame df as input and refer to the aggregation variable, c, by column name.
The plot is interactive and comes with a bunch of tools. You can customize these tools as well

Scatterplot

p = Scatter(flMid-level charts: the Plotting interface
show(p)

Boxplot

```
p = BoxPlot(flora, values='sepal length', label='petal width')
show (p)
```

Mid-level charts

Beyond the canned methods above, Bokeh provides a "mid-level" interface that more directly exposes the grammar of graphics methodology for constructing visual displays.
The basic procedure is:

- Create a blank canvas by calling bokeh.plotting.figure
- Add glyphs, which are geometric shapes.ora, x='petal width', y='sepal length')

```
from bokeh.plotting import figure

# Create a canvas with a specific set of tools for the user:
TOOLS = 'resize,pan,box_zoom,wheel_zoom,lasso_select,save,reset,help'
p = figure(width=500, height=500, tools=TOOLS)
print(p)

# Add one or more glyphs
p.triangle(x=flora['petal width'], y=flora['petal length'])
```

Other way:

Using data from Pandas.

```
from bokeh.models import ColumnDataSource
```

Here is another way to do the same thing, but using a Pandas data frame as input.

```
data=ColumnDataSource(flora)
p=figure()
p.triangle(source=data, x='petal width', y='petal length')
show(p)
```

Mapping colors

```
# Determine the unique species
unique_species = flora['species'].unique()
print(unique_species)

# Map each species with a unique color
from bokeh.palettes import brewer
color_map = dict(zip(unique_species, brewer['Dark2'][len(unique_species)]))
print(color_map)
```

Now we can more programmatically generate the same plot as above, but use a unique color for each species.

```
p = figure()
for s in unique_species:
    p.triangle(source=data_sources[s], x='petal width', y='petal length', color=color_map[s])
show(p)
```

Seaborn

```
import seaborn as sns

# The following Jupyter "magic" command forces plots to appear inline
# within the notebook.
%matplotlib inline
```

When dealing with a set of data, often the first thing we want to do is get a sense for how the variables are distributed. Here, we will look at some of the tools in seborn for examining univariate and bivariate distributions.

Plotting univariate distributions

distplot() function will draw a histogram and fit a kernel density estimate

```
import numpy as np
x = np.random.normal(size=100)
sns.distplot(x)
```

Plotting bivariate distributions

The easiest way to visualize a bivariate distribution in seaborn is to use the jointplot() function, which creates a multi-panel figure that shows both the bivariate (or joint) relationship between two variables along with the univariate (or marginal) distribution of each on separate axes.

```
mean, cov = [0, 1], [(1, .5), (.5, 1)]
data = np.random.multivariate_normal(mean, cov, 200)
df = pd.DataFrame(data, columns=["x", "y"])
```

Basic scatter plots.

The most familiar way to visualize a bivariate distribution is a scatterplot, where each observation is shown with point at the x and y values. You can draw a scatterplot with the matplotlib plt.scatter function, and it is also the default kind of plot shown by the jointplot() function:

```
sns.jointplot(x="x", y="y", data=df)
sns.lmplot(x="x", y="y", data=df, hue=”color_variable”, palette = {0:’red’, 1: ‘blue’})
```

Hexbin plots.

The bivariate analogue of a histogram is known as a “hexbin” plot, because it shows the counts of observations that fall within hexagonal bins. This plot works best with relatively large datasets. It’s availible through the matplotlib plt.hexbin function and as a style in jointplot()

```
sns.jointplot(x="x", y="y", data=df, kind="hex")
```

Kernel density estimation.

It is also posible to use the kernel density estimation procedure described above to visualize a bivariate distribution. In seaborn, this kind of plot is shown with a contour plot and is available as a style in jointplot()

```
sns.jointplot(x="x", y="y", data=df, kind="kde")
```

Visualizing pairwise relationships in a dataset

To plot multiple pairwise bivariate distributions in a dataset, you can use the pairplot() function. This creates a matrix of axes and shows the relationship for each pair of columns in a DataFrame. by default, it also draws the univariate distribution of each variable on the diagonal Axes:

```
sns.pairplot(flora)
# We can add colors to different species
sns.pairplot(flora, hue="species")
```

Visualizing linear relationships

We can use the function regplot to show the linear relationship between total_bill and tip. It also shows the 95% confidence interval.

```
sns.regplot(x="total_bill", y="tip", data=tips)
```

Visualizing higher order relationships

```
sns.regplot(x="x", y="y", data=data, order=2)
```

Strip plots.

This is similar to scatter plot but used when one variable is categorical.

```
sns.stripplot(x="day", y="total_bill", data=tips)
```

Boxplots

```
sns.boxplot(x="day", y="total_bill", hue="time", data=tips)
```

Barplot

```
sns.barplot(x="sex", y="survived", hue="class", data=titanic)
```

Violinplot

```
sns.violinplot(x,y,data, inner = ‘quart’)
```

Heatmap

```
f,ax = plt.subplots(figsize=(18, 18))
sns.heatmap(data.corr(), annot=True, linewidths=.5, fmt= '.1f',ax=ax)
```

Topic 9: Relational data

SQLite

The de facto language for managing relational databases is the Structured Query Language, or SQL ("sequel"). Many commerical and open-source relational data management systems (RDBMS) support SQL. The one we will consider in this class is the simplest, called sqlite3. It stores the database in a simple file and can be run in a "standalone" mode from the command-line. However, we will, naturally, invoke it from Python. But all of the basic techniques apply to any commercial SQL backend.

In Python, you connect to an sqlite3 database by creating a connection object.

```
Import sqlite3 as db
Conn = db.connect(‘example.db’)
```

The sqlite engine maintains a database as a file; in this example, the name of that file is example.db.

Important usage note! If the named file does not yet exist, this code creates it. However, if the database has been created before, this same code will open it. This fact can be important when you are debugging. For example, if your code depends on the database not existing initially, then you may need to remove the file first.

You issue commands to the database through an object called a cursor.

```
# Create a 'cursor' for executing commands
c = conn.cursor()
```

A cursor tracks the current state of the database, and you will mostly be using the cursor to issue commands that modify or query the database.

Tables and Basic Queries

The central object of a relational database is a table. It's identical to what you called a "tibble" in the tidy data lab: observations as rows, variables as columns. In the relational database world, we sometimes refer to as items or records and columns as attributes. We'll use all of these terms interchangeably in this course.

Create

You can create the table using the command, create table. Note: If you try to create a table that already exists, it will fail. If you are trying to carry out these exercises from scratch, you may need to remove any existing example.db file or destroy any existing table; you can do the latter with the SQL command, drop table if exists Students.

```
c.execute("create table Students (gtid integer, name text)")
```

Insert

To populate the table with items, you can use the command, insert into.

Commitment issues. The commands above modify the database. However, these are temporary modifications and aren't actually saved to the databases until you say so. (Aside: Why would you want such behavior?) The way to do that is to issue a commit operation from the connection object.

There are some subtleties related to when you actually need to commit, since the SQLite database engine does commit at certain points as discussed here. However, it's probably simpler if you remember to encode commits when you intend for them to take effect.

Insert many

Another common operation is to perform a bunch of insertions into a table from a list of tuples. In this case, you can use executemany().

Select

Given a table, the most common operation is a query, which asks for some subset or transformation of the data. The simplest kind of query is called a select.

```
c.execute("select * from Students")
results = c.fetchall()
print("Your results:", len(results), "\nThe entries of Students:\n", results)
```

Another way of iterating over the results:

For row in c.execute(selectquery).

Join Querys

The main type of query that combines information from multiple tables is the join query. Recall from our discussion of tibbles these four types:
inner-join(A, B): Keep rows of A and B only where A and B match
outer-join(A, B): Keep all rows of A and B, but merge matching rows and fill in missing values with some default (NaN in Pandas, NULL in SQL)
left-join(A, B): Keep all rows of A but only merge matches from B.
right-join(A, B): Keep all rows of B but only merge matches from A.

In SQL, you can use the where clause of a select statement to specify how to match rows from the tables being joined.

Aggregations

Another common style of query is an aggregation, which is a summary of information across multiple records, rather than the raw records themselves.

Case-insensitive grouping: COLLATE NOCASE. One way to carry out the preceding query in a case-insensitive way is to add a COLLATE NOCASE qualifier to the GROUP BY clause.

Dates and times in SQL

The CreatedDate column is actually a specially formatted date and time stamp, where you can query against by comparing to strings of the form, YYYY-MM-DD hh:mm:ss.

This next example shows how to extract just the hour from the time stamp, using SQL's strftime().

strftime(‘%H’,CreatedDate)

Timestamp only: strftime('%H:%M:%f',CreatedDate) can be compared against '00:00:00.000`156'

Module 2: The computational analysis of data

Topic 10: Intro to Numpy/Scipy for numerical computation

Numpy

Numpy is a Python module that provides fast primitives for multidimensional arrays. It's well-suited to implementing numerical linear algebra algorithms, and for those can be much faster than Python's native list and dictionary types when you only need to store and operate on numerical data.

import numpy as np

Creating an array

np.array([1,2,3,4])
np.array([1,2,3,4], [1,2,3,4]) #2 dimensional array, 2x3

Properties

Array.ndim = number of dimensions in array
Array.shape = rows,columns
len(array) = number of elements in first dimension

Range

np.arange(10) # Moral equivalent to `range`
linspace(start,end,step)

Particular values

np.zeros((3,4)) = Creates a 3x4 zeros matrix
np.ones((3,4)) = Creates a 3x4 onesmatrix
np.eye(3) = Creates a 3x3 identity matrix The eye() function returns an identity matrix and uses a floating-point type as the element type.
np.diag([1, 2, 3]) = Specify diagonal rest are zeros
np.empty((3, 4)) = Creates a 3x4 ones matrix

Random

np.random.uniform(low, high, size)
np.random.randint(low,high,size)
np.random.choice(array, size, replace)

Reshape

np.reshape(array, (nrow,ncol))

Copy columns

tile(array, reps = (from, to))

Transpose

From dataframe

df.as_matrix([cols])

From Boolean

np.where(Boolean, Yes, No)

Add columns

np.insert(matrix, newcolumn, axis)

Index and slicing

The usual 0-based slicing and indexing notation you know and love from lists is also supported for Numpy arrays. In the multidimensional case, including their natural multidimensional analogues with index ranges separated by commas.

To help save memory, when you slice a Numpy array, you are actually creating a view into that array. That means modifications through the view will modify the original array.

You can force a copy using the .copy() method: array_copy = array.copy()

And to check whether two Numpy array variables point to the same object, you can use the numpy.may_share_memory(array1, array2) function.
Two other common ways to index a Numpy array are to use a boolean mask or to use a set of integer indices.

To slice a column from a matrix and keep it as a column we need to specify A[: , i:i+1]

Matrix vector multiplication

A.dot(x)

Dense matrix storage: Column-major versus row-major layouts

For linear algebra, we will be especially interested in 2-D arrays, which we will use to store matrices. For this common case, there is a subtle performance issue related to how matrices are stored in memory.

By way of background, physical storage---whether it be memory or disk---is basically one big array. And because of how physical storage is implemented, it turns out that it is much faster to access consecutive elements in memory than, say, to jump around randomly.
A matrix is a two-dimensional object. Thus, when it is stored in memory, it must be mapped in some way to the one-dimensional physical array. There are many possible mappings, but the two most common conventions are known as the column-major and row-major layouts.

In Numpy, you can ask for either layout. The default in Numpy is row-major. Historically numerical linear algebra libraries were developed assuming column-major layout. This layout happens to be the default when you declare a 2-D array in the Fortran programming language. By contrast, in the C and C++ programming languages, the default convention for a 2-D array is row-major layout. So the Numpy default is the C/C++ convention. In your programs, you can request either order of Numpy using the order parameter. For linear algebra operations (common), we recommend using the column-major convention. In either case, here is how you would create column- and row-major matrices.

```
np.ones((n, n), order='F') #Fortran or column major
np.ones((n, n), order='C') #C or row-major
```

Colmajor layout works better when computing a scaling of the columns:

```
# Test (timing) cell: `scale_colwise_test`

# Measure time to scale a row-major input column-wise
%timeit scale_colwise(A_rowmaj)
218 ms ± 7.84 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
# Measure time to scale a column-major input column-wise
%timeit scale_colwise(A_colmaj)
63.1 ms ± 169 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)

# Convert values into Numpy arrays in column-major order
A_np = np.reshape(A_py, (n, n), order='F')
x_np = np.reshape(x_py, (n, 1), order='F')

# Here is how you do a "matvec" in Numpy:
%timeit A_np.dot(x_np)
1.44 ms ± 294 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)

Vs a function implemented on native lists:
2.12 s ± 10.2 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Comparing values

np.isclose(val1,val2,tol)

Grid of values

X, Y = meshgrid(vector1, vector2)

Vectorize a function

vectorize(function)

Linear algebra

np.linalg.lstsq()
np.linalg.cond(X) condition number
np.linalg.qr(X) gives qr decomposition of a matrix
np.linalg.norm(v) computes the norm of a vector
U, Sigma, VT = np.linalg.svd(X, full_matrices=False)

Functions

Aggregations

With axis parameter we can aggregate for a particular axis.
np.max(A) alias for np.amax(A)

np.min(A) alias for np.amin(A)
np.maximum take two arrays and compute their element-wise
np.minimum(A)
np.sum(A, axis)
np.mean(A)
np.cumsum(A, axis)

np.std(A)
np.argmin(A) gives the index of the minimum values along an axis

Element-wise/Universal

np.maximum(A,B)
np.sqrt(array)
np.frompyfunc(f, 1, 1) allows to create a custom function from a python function f.

np.cumprod(X, axis = 1)

Broadcasting

Sometimes we want to combine operations on Numpy arrays that have different shapes but are compatible. If we try to do A + 3. Technically, A and 3 have different shapes: the former is a 4×3 matrix, while the latter is a scalar (1×1). However, they are compatible because Numpy has a scheme to extend---or broadcast---the value 3 into an equivalent matrix object of the same shape, before combining them elementwise.

The broadcasting rule. One way is to learn Numpy's convention for broadcasting. Numpy starts by looking at the shapes of the objects. These are compatible if, starting from right to left, the dimensions match or one of the dimensions is 1. This convention of moving from right to left is referred to as matching the trailing dimensions.

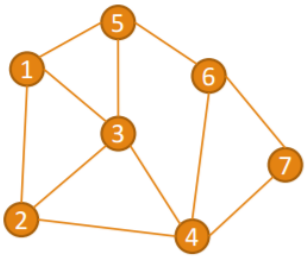
Part 3: Sparse matrix storage

Coordinate Format (COO)

In this format we store three lists, one each for rows, columns and the elements of the matrix. Look at the below picture to understand how these lists are formed.

Coordinate (COO) format

The triplets can be stored as 3 arrays: rows, cols, values.



	1	2	3	4	5	6	7
1		1	1		1		
2	1		1	1			
3	1	1		1	1		
4		1	1			1	1
5	1		1			1	
6				1	1		1
7				1		1	

`rows` = [0, 0, 0, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4, 5, 5, 5, 6, 6]

`cols` = [1, 2, 4, 0, 2, 3, 0, 1, 3, 4, 1, 2, 5, 6, 0, 2, 5, 3, 4, 6, 3, 5]

`values` = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

Note: 0-based arrays

```
x = dense_vector([1.] * num_verts)
%timeit spmv_coo(coo_rows, coo_cols, coo_vals, x)
```

Compressed Sparse Row Format

This is similar to the COO format excpet that it is much more compact and takes up less storage. Look at the picture below to understand more about this representation

Compressed sparse row (CSR) format

Suppose a sparse matrix has `nnz` nonzero entries.

`rows` = [0, 0, 0, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4, 5, 5, 5, 6, 6]

`cols` = [1, 2, 4, 0, 2, 3, 0, 1, 3, 4, 1, 2, 5, 6, 0, 2, 5, 3, 4, 6, 3, 5]

`values` = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

The COO format needs `3nnz` elements to store the matrix. Can we do better?

When the nonzeros are stored row by row, we can compress the above storage:

`rowptr` = [0, 3, 6, 10, 14, 17, 20, 22] Row pointer

`colind` = [1, 2, 4, 0, 2, 3, 0, 1, 3, 4, 1, 2, 5, 6, 0, 2, 5, 3, 4, 6, 3, 5] Column index

`values` = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1] Values

This CSR format needs `2nnz+n` elements to store the matrix.

the list-based COO and CSR formats do not really lead to sparse matrix-vector multiply implementations that are much faster than the dictionary-based methods:

Dictionary based

453 ms ± 68.1 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

COO
308 ms ± 35.9 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

CSR
369 ms ± 28.8 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

SciPy

```
import numpy as np
import scipy.sparse as sp
```

```
A_coo_sp = sp.coo_matrix((coo_vals, (coo_rows, coo_cols)),shape=m,n)
A_csr_sp = A_coo_sp.tocsr() # Alternatively: sp.csr_matrix((val, ind, ptr))

x_sp = np.ones(num_verts)

print ("\n==> COO in Scipy:")
%timeit A_coo_sp.dot (x_sp)

print ("\n==> CSR in Scipy:")
%timeit A_csr_sp.dot (x_sp)

==> COO in Scipy:
3.66 ms ± 21.2 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

==> CSR in Scipy:
3.37 ms ± 2.34 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Solve system of equations

scipy.linalg.solve()

Clustering

```
from scipy.cluster import vq
# `distortion` below is the similar to WCSS.
# It is called distortion in the Scipy documentation
# since clustering can be used in compression.
centers_vq, distortion_vq = vq.kmeans(points, k)

# vq return the clustering (assignment of group for each point)
# based on the centers obtained by the kmeans function.
# _ here means ignore the second return value
clustering_vq, _ = vq.vq(points, centers_vq)
```

Functions

Find mode

scipy.stats.mode(a, axis=0, nan_policy='propagate')

Topic 11: Ranking relational objects

Page Rank Problem

Let's model the analysis problem as follows.
Consider a "random flyer" to be a person who arrives at an airport i, and then randomly selects any direct flight that departs from i and arrives at j. We refer to the direct flight from i to j as the flight segment $i \rightarrow j$. Upon arriving at j, the flyer repeats the process of randomly selecting a new segment, $j \rightarrow k$. He or she repeats this process forever.

Let $x_i(t)$ be the probability that the flyer is at airport i at time t. Take tt to be an integer count corresponding to the number of flight segments that the flyer has taken so far, starting at $t=0$. Let p_{ij} be the probability of taking segment $i \rightarrow j$, where $p_{ij}=0$ means the segment $i \rightarrow j$ is unavailable or does not exist. If there are n airports in all, numbered from 0 to $n-1$, then the probability that the flyer will be at airport i at time $t+1$, given all the probabilities at time t, is $x_i(t+1)=\sum_{j=0,n-1} p_{ji} \cdot x_j(t)$.

Let $P \equiv [p_{ij}]$ be the matrix of transition probabilities and $x(t)=[x_i(t)]$ the column vector of prior probabilities. Then we can write the above more succinctly for all airports as the matrix-vector product,
$x(t+1)=PTx(t)$.
Since P is a probability transition matrix then there exists a steady-state distribution, $x*$, which is the limit of $x(t)$ as tt goes to infinity:
$\lim_{t \rightarrow \infty} x(t)=x* \equiv [x*_i]$.

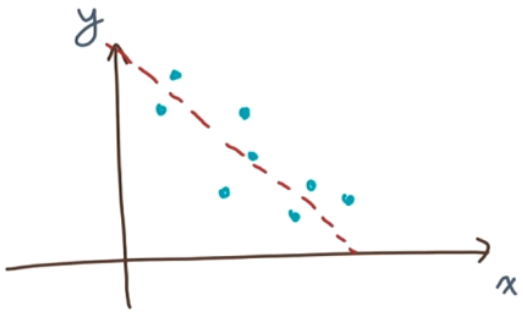
The larger x_i*, the more likely it is that the random flyer is to be at airport i in the steady state. Therefore, we can take the "importance" or "criticality" of airport i in the flight network to be its steady-state probability, x_i*.
Thus, our data pre-processing goal is to construct P and our analysis goal is to compute the steady-state probability distribution, $x*$, for a first-order Markov chain system.

Topic 12: Linear Regression

Lesson 0: Overview

Overview of linear regression

Observe: $\{(x_i, y_i) \mid 0 \le i < m\}$

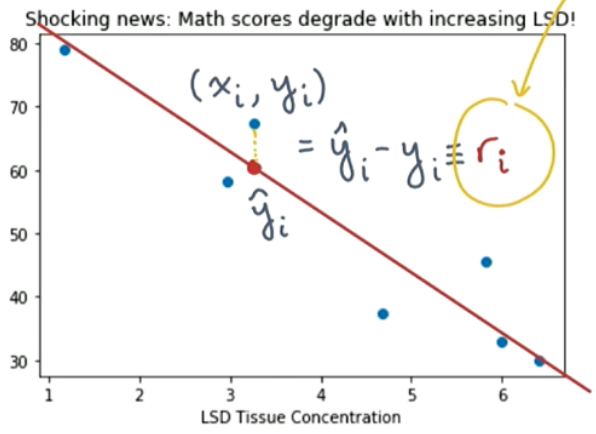


Believe that

$$y = \alpha \cdot x + \beta$$

Overview of linear regression

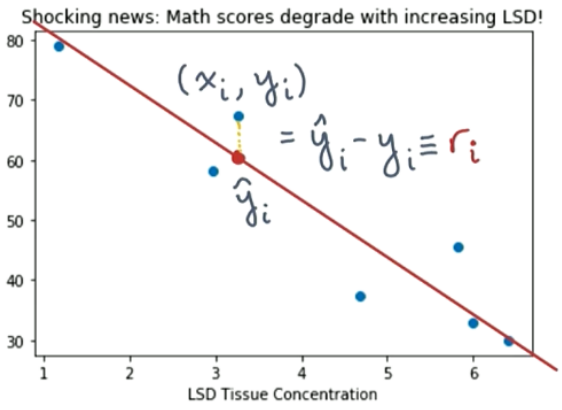
residual



Believe that

$$\hat{y}_i = \alpha \cdot x_i + \beta$$

Overview of linear regression



Computational task:

Choose α & β s.t.

$$\sum_{i=0}^{m-1} (r_i)^2$$

Sum-of-squared residuals is minimized.

Lesson 1: Minimizing the residual

Minimize f(alpha, beta) = Sum i:0,n (alpha * xi + beta - yi)^2

We need to compute partial derivatives over alpha and beta:

d/dalpha f(alpha, beta) = Sum i:0,m-1 2 (alpha * xi + beta - yi) xi
= 2 alpha Sum i:0,m-1 xi^2 + 2 beta Sum i:0,m-1 xi + Sum i:0,m-1 2 xi yi
= 2 alpha (xTx) + 2 beta (uTx) - 2 (xTy)

d/dbeta f(alpha, beta) = 2 alpha (uTx) + 2 m beta - 2 (uTy)

At the extreme point we obtain a system of equations with 2 unknowns where:

d/dalpha f(alpha, beta) = 0 and d/dbeta f(alpha, beta) = 0
And we can obtain solving the system:

Beta* = 1/m uT (y - alpha* x)

Alpha* = [xTy - 1/m (uTx) (uTy)] / xTx - 1/m (uTx) ^ 2

Minimizing the Residual

$$f(\alpha,\beta)=\sum_{i=0}^{m-1}\left(\alpha x_i+\beta-y_i\right)^2$$

$$\frac{\partial}{\partial\alpha}f(\alpha,\beta)=\sum_{i=0}^{m-1}2\left(\alpha x_i+\beta-y_i\right)x_i$$

Minimizing the Residual

$$f(\alpha,\beta)=\sum_{i=0}^{m-1}\left(\alpha x_i+\beta-y_i\right)^2$$

$$\frac{\partial}{\partial\alpha}f(\alpha,\beta)=2\alpha(x^Tx)+2\beta(u^Tx)-2(x^Ty)$$

$$x\equiv\begin{bmatrix}x_0\\x_1\\\vdots\\x_{m-1}\end{bmatrix}\quad y\equiv\begin{bmatrix}y_0\\y_1\\\vdots\\y_{m-1}\end{bmatrix}\quad u\equiv\begin{bmatrix}1\\1\\\vdots\\1\end{bmatrix}$$

Minimizing the Residual

$$f(\alpha,\beta)=\sum_{i=0}^{m-1}\left(\alpha x_i+\beta-y_i\right)^2$$

$$\frac{\partial}{\partial\alpha}f(\alpha,\beta)=2\alpha(x^Tx)+2\beta(u^Tx)-2(x^Ty)$$

$$\frac{\partial}{\partial\beta}f(\alpha,\beta)=2\alpha(u^Tx)+2m\beta-2(u^Ty)$$

Minimizing the Residual

$$f(\alpha,\beta)=\sum_{i=0}^{m-1}\left(\alpha x_i+\beta-y_i\right)^2$$

$$0 = 2\alpha_*(x^Tx) + 2\beta_*(u^Tx) - 2(x^Ty)$$

$$0 = 2\alpha_*(u^Tx) + 2m\beta_* - 2(u^Ty)$$

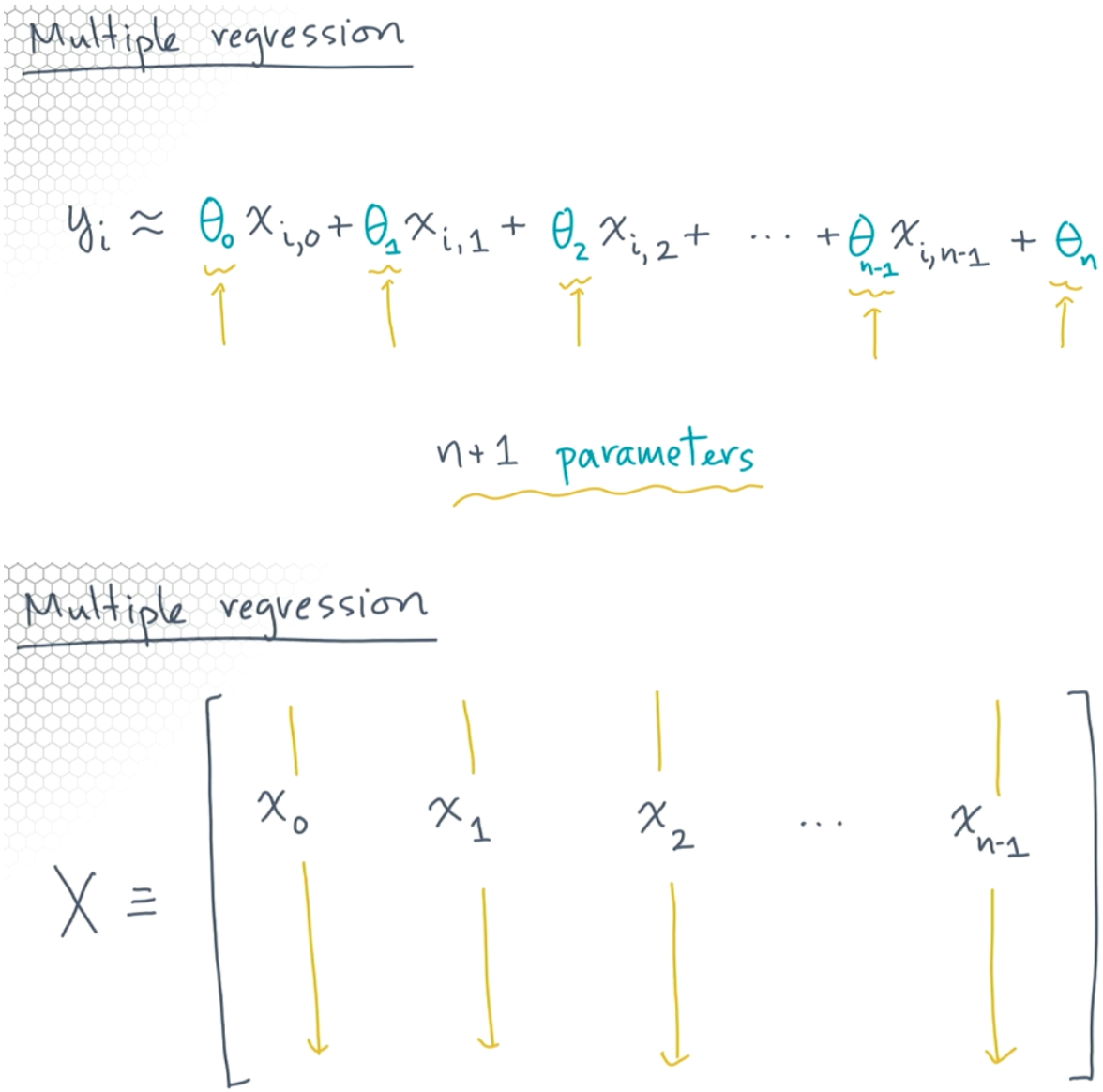
Minimizing the Residual

$$f(\alpha,\beta)=\sum_{i=0}^{m-1}\left(\alpha x_i+\beta-y_i\right)^2$$

$$\beta_* = \frac{1}{m}u^T(y-\alpha_* x)$$

$$\alpha_* = \frac{x^Ty - \frac{1}{m}(u^Tx)(u^Ty)}{x^Tx - \frac{1}{m}(u^Tx)^2}$$

Lesson 2: Multiple linear regression and matrices



Multiple regression

$$X \equiv \begin{bmatrix} x_0 & x_1 & x_2 & \cdots & x_{n-1} \\ & & & & \end{bmatrix}$$

$$x_j \equiv \begin{bmatrix} x_{0,j} \\ x_{1,j} \\ \vdots \\ x_{m-1,j} \end{bmatrix} \rightarrow y \equiv \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{m-1} \end{bmatrix}$$

Multiple regression

$$y \approx \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_{n-1} x_{n-1} + \theta_n u$$

$$u \equiv \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}$$

Multiple regression

$$X \equiv \begin{bmatrix} x_0 & x_1 & x_2 & \cdots & x_{n-1} & 1 \end{bmatrix}$$

Redefine X to include n^{th} column of ones

Multiple regression

$$y \approx X\theta$$

$$\theta \equiv \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_n \end{bmatrix}$$

Multiple regression

$$r \equiv X\theta - y \qquad r = \begin{bmatrix} r_0 \\ r_1 \\ \vdots \\ r_{m-1} \end{bmatrix}$$

residual (vector)

Multiple regression

$$r \equiv X\theta - y$$

Recall: $\|r\|_2^2 = r^T r = \sum_{i=0}^{m-1} (r_i)^2$

Computational task: Choose $\theta^* = \underset{\theta}{\text{argmin}} \|r\|_2^2$

We estimate yihat = theta0 xi,0 + .. + thetan-1 xi,n-1 + thetan where xi,j is the ith observation of the jth predictor. And theta j the jth parameter corresponding to the jth predictor. And there one last thetan for the intercept.

We can think of this as:

Matrix X = [x0 + .. + xn-1 + u] where xi is the column vector xi = [x0,i, .. , xm-1,i] that corresponds to the observations for the ith predictor and u a column vector of ones.

And vector theta containing [theta0, .. , thetan].

Then yhat = Xtheta

And we have residuals r = Xtheta - y

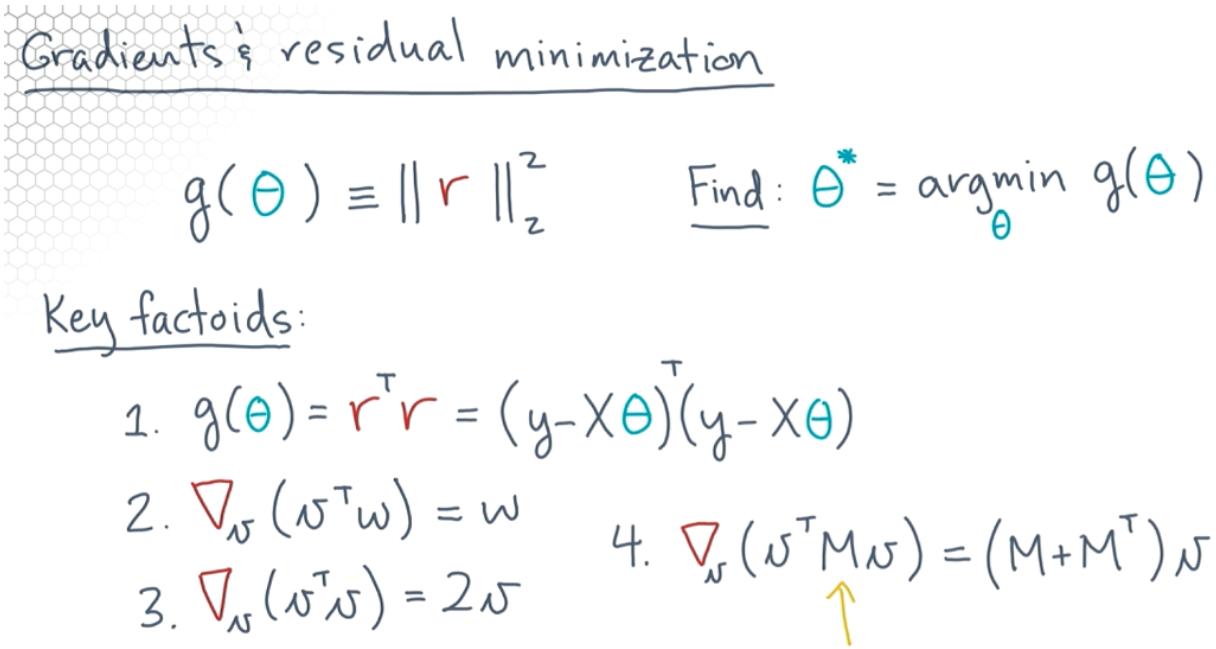
Computational task

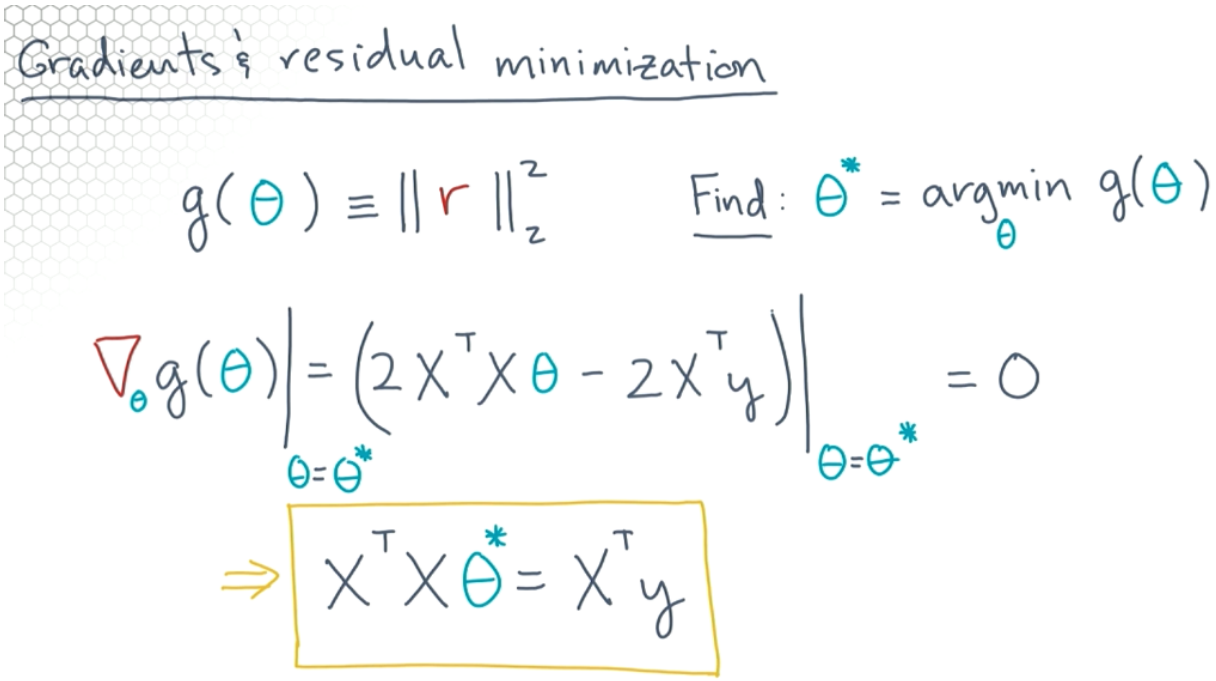
Recall that ||r||2 ^ 2 = rTr = sum i:0,m-1 (ri) ^ 2

Choose theta* = argmin theta ||r||2 ^2

Choose the value of theta, that minimizes the two norm of the residual squared.

Lesson 3: Gradients and residual minimization





Let's see how to calculate that minimum. Now, to do that, you will need the vector analog of taking the first derivative. So that mathematical tool is something called a gradient.

Gradient of g with respect to theta is:

∇theta g(theta) = [dg/dtheta0, .. , dg/dthetan-1]

The gradient points in the direction of steepest ascent, but that's not all, the gradient is also 0 at extreme points.
So that's the same thing as what happens to the one dimensional derivative.

Gradient factoids

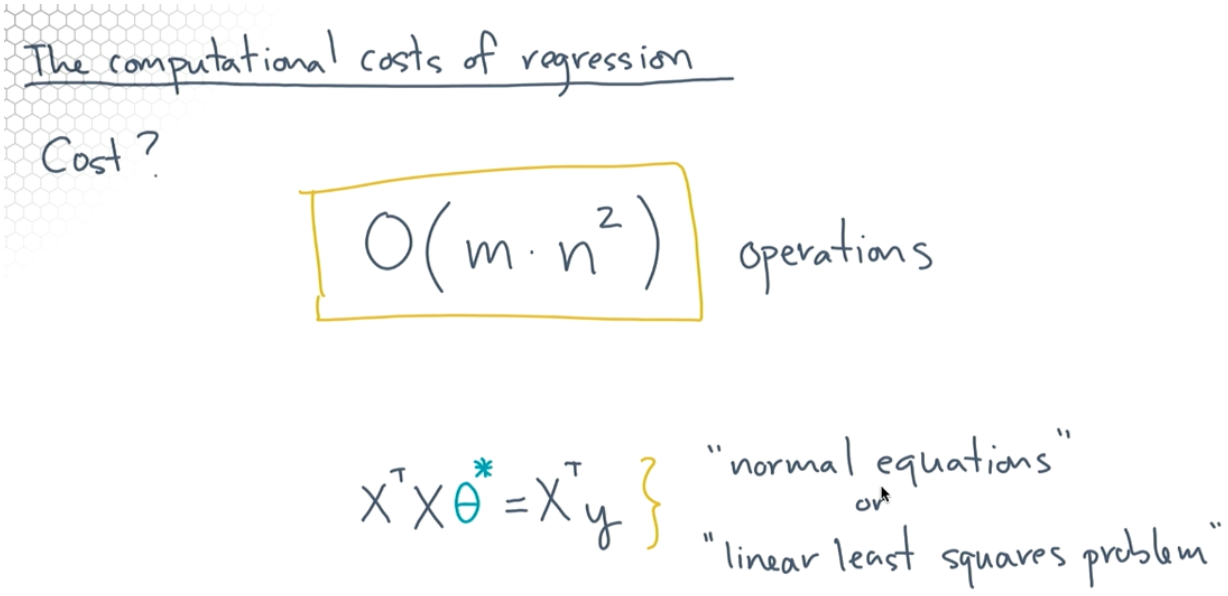
1) g(theta) = rTr
2) ∇v (vTw) = v
3) ∇v (vTv) = 2v
4) ∇v (vTMv) = (M + MT) v

We can show that:
∇theta g(theta) = 2XTXTheta - 2XTy

To find the minimum we need to solve then:

XTXTheta* = XTy which is referred as the normal equation

Lesson 4: Computational Costs



O(m*n^2) where m is the number of observations and n the number of predictors.

Lesson 5: An incremental Algorithm

An online algorithm for regression

online or incremental

$X^T X \theta^* = X^T y$

vs.

offline or batched

cost $= O(m \cdot n^2)$

One-at-a-time: $\sim n^2 + 2n^2 + 3n^2 + \dots + mn^2$

An online algorithm for regression

online or incremental

$X^T X \theta^* = X^T y$

vs.

offline or batched

cost $= O(m \cdot n^2)$

One-at-a-time: $\sim O(m^2 n^2)$

An online algorithm for regression

$$X \equiv \underbrace{\begin{pmatrix} x_0 & x_1 & \cdots & x_{n-1} & x_n \end{pmatrix}}_{\text{by columns}}$$

$$X^T X \theta^* = X^T y$$

n predictors

1 "dummy": $x_n \equiv (1)$

An online algorithm for regression

$$X \equiv \begin{pmatrix} \hat{x}_0^T \\ \hat{x}_1^T \\ \vdots \\ \hat{x}_{m-1}^T \end{pmatrix} \Bigg\} \text{by rows (m observations)}$$

$$X^T X \theta^* = X^T y$$

An online algorithm for regression

Let:

$(\hat{x}_k, y_k) \equiv k^{th}$ data point

$(k \geq 0)$

$\hat{\theta}(k) \equiv$ current estimate of θ^*

$$X^T X \theta^* = X^T y$$

Goal: Compute $\hat{\theta}(k+1)$

An online algorithm for regression

Initial idea:

$$r_k \equiv y_k - \hat{x}_k^T \hat{\theta}(k)$$

$$X^T X \theta^* = X^T y$$

Correction?

$$\hat{\theta}(k+1) \leftarrow \hat{\theta}(k) + \Delta_k$$

Suppose:

$$\Delta_k \equiv \frac{\hat{x}_k}{\|\hat{x}_k\|_2^2} r_k$$

An online algorithm for regression

Initial idea:

$$r_k \equiv y_k - \hat{x}_k^T \hat{\theta}(k)$$

$$X^T X \theta^* = X^T y$$

Correction?

$$y_k - \hat{x}_k^T \left[\hat{\theta}(k) + \Delta_k \right] = 0$$

Suppose:

$$\Delta_k \equiv \frac{\hat{x}_k}{\|\hat{x}_k\|_2^2} r_k$$

Flaw: Might break all previous predictions!

An online algorithm for regression

"Hack":

$$\Delta_k \equiv \frac{\hat{x}_k}{\|\hat{x}_k\|_2^2} r_k \cdot \phi$$

"fudge factor"

$\epsilon (0, 1)$

$$X^T X \theta^* = X^T y$$

An online algorithm for regression

"Hack":

$$\Delta_k \equiv \frac{\hat{x}_k}{\|\hat{x}_k\|_2^2} r_k \cdot \phi$$

$$X^T X \theta^* = X^T y$$

Theorem: let λ_{max} be the largest eigenvalue of $X^T X$.

For any $0 < \phi < \frac{2}{\lambda_{max}}$, $\lim_{k \to \infty} \hat{\theta}(k) = \theta^*$

$\Rightarrow$ Least mean square (LMS) algorithm

An online algorithm for regression

"Hack":

$$\Delta_k \equiv \frac{\hat{x}_k}{\|\hat{x}_k\|_2^2} r_k \cdot \phi$$

$$X^T X \theta^* = X^T y$$

Theorem: let λ_{max} be the largest eigenvalue of $X^T X$.

For any $0 < \phi < \frac{2}{\lambda_{max}}$, $\lim_{k \to \infty} \hat{\theta}(k) = \theta^*$

??

We need an incremental or online algorithm vs the batch or offline algorithm just discussed.

With current method the cost is O(m*n^2). If we apply the same method getting one observation at the time, the cost will go up to O(m^2*n^2).

Let X = [x0T, .. , xm-1T] the matrix composed by row vectors containing the observations.

Let (xkT, yk) be a just arrived data point which we want to include into the model.

theta(k) = current estimate of thetastar

Then our goal is to compute theta (k+1).

Let’s compute rk = yk - xkT*theta(k)

We want to correct theta:

theta(k+1) = theta(k) + deltak

Suppose we choose

Topic 13: Classification

Lesson 0: Classification tasks

Beyond regression, another important data analysis task is classification, in which you are given a set of labeled data points and you wish to learn a model of the labels. The canonical example of a classification algorithm is logistic regression, the topic of this notebook. Although it's called "regression" it is it is really a model for classification.

We will take the labels to be a binary vector, $yT≡(y0,y1,…,ym−1)T$

Here, you'll consider binary classification. Each data point belongs to one of $c=2$ possible classes. By convention, we will denote these class labels by "0" and "1." However, the ideas can be generalized to the multiclass case, i.e., $c>2$, with labels $\{0,1,…,c−1\}$.

You'll also want to review from earlier notebooks the concept of gradient ascent/descent (or "steepest ascent/descent"), when optimizing a scalar function of a vector variable.

Notation

To develop some intuition and a classification algorithm, let's formulate the general problem and apply it to synthetic data sets.

Let the data consist of m observations of dd continuously-valued predictors. In addition, for each data observation we observe a binary label whose value is either 0 or 1.

Just like our convention in the linear regression case, represent each observation, or data point, by an agumented vector, $\hat{x}_i^T = \equiv(x_{i,0}, x_{i,1}, ...x_{i,d-1}, 1)$

That is, the point is the d coordinates augmented by an initial dummy coordinate whose value is 1. This convention is similar to what we did in linear regression.

We can also stack these points as rows of a matrix, X, again, just as we did in regression:
$X \equiv [\hat{x}_0^T, \hat{x}_1^T, .. , \hat{x}_{m-1}^T]$

Linear discriminants and the heaviside function

Suppose you think that the boundary between the two clusters may be represented by a line. A linear boundary is also known as a linear discriminant. Any point x on this line may be described by $\theta^T x = 0$, where θ is a vector of coefficients:

$\theta \equiv (\theta_0, \theta_1,.., \theta_d)$

For example, suppose our observations have two predictors each (d=2). Let the corresponding data point be $x^T \equiv (x_0,x_1,x_2 = 1.0)$

Then, $\theta^T\hat{x} = 0 \Longrightarrow \theta_0 x_0 + \theta_1 x_1 + \theta_2 = 0 \Longrightarrow x_1 = -\theta_2/\theta_1 - \theta_0/\theta_1 x_0$

So that describes points on the line. However, given any point x in the d - dimensional space that is not on the line, $\theta^T x$ still produces a value: that value will be positive on one side of the line ($\theta^T x>0$) or negative on the other ($\theta^T x<0$).

In other words, you can use the linear discriminant function, $\theta^T x$, to generate a label for each point x : just reinterpret its sign!

If you want "0" and "1" labels, the heaviside function, H(y) , will convert a positive y to the label "1" and all other values to "0."

$H(y) \equiv \{1 \text{ if } y>0, 0 \text{ if } y \leq 0\}$

Lesson 1: The logistic function

The logistic (or sigmoid) function as an alternative discriminant

Since the labels are 0 or 1, you could look for a way to interpret labels as probabilities rather than as hard (0 or 1) labels. One such function is the logistic function, also referred to as the logit or sigmoid function.

$G(y) \equiv 1 / 1 + e \hat{} -y$

The logistic function takes any value in the range $(-\infty, +\infty)$ and produces a value in the range (0,1). Thus, given a value y, we can interpret $G(y)$ as a conditional probability that the label is 1 given y, i.e., $G(y) \equiv \Pr[\text{label is } 1|y]$

Why logistic

Consider a set of 1-D points generated by a mixture of Gaussians. That is, suppose that there are two Gaussian distributions over the 1-dimensional variable, $x \in (-\infty, +\infty)$, that have the same variance ($\sigma 2$) but different means ($\mu 0$ and $\mu 1$).

Show that the conditional probability of observing a point labeled "1" given x may be written as, $\Pr[l=1|x] \propto 1/1+e-(\theta 0x+\theta 1)$, for a suitable definition of $\theta 0$ and $\theta 1$.

Hints. Since the points come from Gaussian distributions,
$\Pr[x|l] \equiv 1/\sigma\sqrt{2\pi} \exp(-(x-\mu l)\hat{}2/2\sigma\hat{}2)$.

To rewrite $\Pr[l|x]$ in terms of $\Pr[x|l]$, recall Bayes's rule (also: Bayes's theorem):
$\Pr[l=1|x] = \Pr[x|l=1]\Pr[l=1] / \Pr[x]$

Where the denominator can be expanded as
$\Pr[x] = \Pr[x|l=0] \Pr[l=0] + \Pr[x|l=1] \Pr[l=1]$

You may assume the prior probabilities of observing a 0 or 1 are given by $\Pr[l=0] \equiv p0$ and $\Pr[l=1] \equiv p1$.

The point of this derivation is to show you that the definition of the logistic function does not just arise out of thin air. It also hints that you might expect a final algorithm for logistic regression based on using $G(y)$ as the discriminant will work well when the classes are best explained as a mixture of Gaussians.

Generalizing to d - dimensions. The preceding exercise can be generalized to d - dimensions. Let θ and x be $(d+1)$ - dimensional points. Then,
$\Pr[l=1|x] \propto 1/1+\exp(-\theta Tx)$.

Properties

$G(y) = e^y/e^y+1$ (P1)
$G(-y)=1-G(y)$ (P2)
$dG/dy=G(y)G(-y)$ (P3)
$d/dy[\ln G(y)]=G(-y)$ (P4)
$d/dy\ln[1-G(y)]=-G(y)$(P5)

Verification

Answers. In all of the derivations below, we use the fact that $G(y)>0$
(P1). Multiply the numerator and denominator by e^y

(P2). Start with the right-hand side, $1-G(y)$, apply some algebra, and then apply (P1).

$1-G(y)=e^y+1/e^y+1 - e^y / e^y+1 = 1/ e^y + 1 \cdot e^-y/e^-y = e^-y / e^-y + 1 = G(-y)$.

(P3). By direct calculation and application of (P1):

$dGdy=d/dy(1+e-y)^-1 = -(1+e^-y)^-2 \cdot (-e^-y) = 1/1+e^-y \cdot e^-y/1+e-y = G(y) \cdot G(-y)$.

(P4). By the chain rule and application of (P3):

$d/dy\ \ln G(y) = (d/dG\ \ln G)\ dG/dy = 1/G(y) \cdot G(y)G(-y)=G(-y)$.

(P5). By combining (P2), variable substitution and the chain rule, and (P4),
$d/dy\ \ln[1-G(y)] = d/dy\ \ln G(-y) = [d/dz\ \ln G(z)] \cdot dz/dy$ where $z \equiv -y = G(-z) \cdot (-1) = -G(y)$.

Lesson 2: Maximum likelihood estimation

Previously, you determined θ for our synthetic dataset by hand. Can you compute a good θ automatically? One of the standard techniques in statistics is to perform a maximum likelihood estimation (MLE) of a model's parameters, θ. Indeed, you may have seen or used MLE to derive the normal equations for linear regression in a more "statistically principled" way.

"Likelihood" as an objective function. MLE derives from the following idea. Consider the joint probability of observing all of the labels (of the actual data), given the points and the

parameters, θ: Pr[y | X, θ].

Suppose these observations are independent and identically distributed (i.i.d.). Then the joint probability can be factored as the product of individual probabilities,

Pr[y | X,θ] = Pr[y0,…,ym−1|x̂0, …, x̂m−1, θ] = Pr[y0 | x̂0, θ]···Pr[ym−1|x̂m−1,θ] =

∏i:0,m−1 Pr[yi | x̂i,θ]

The maximum likelihood principle says that you should choose θ to maximize the chances (or "likelihood") of seeing these particular observations. Thus, Pr[y | X,θ] is now an objective function to maximize.

For both mathematical and numerical reasons, we will use the logarithm of the likelihood, or log-likelihood, as the objective function instead. Let's define it as:

L(θ;y,X) ≡ log{∏i:0,m−1 Pr[yi | xi, θ]} = ∑i:0,m−1 log Pr[yi | xi,θ]

We are using the symbol log, which could be taken in any convenient base, such as the natural logarithm (lny) or the information theoretic base-two logarithm (log2y).

The MLE fitting procedure then consists of two steps:

- For the problem at hand, decide on a model of Pr[yi | x̂i,θ]
- Run any optimization procedure to find the θ that maximizes L(θ;y,X)

Lesson 3: Applying MLE to logistic regression

Let's say you have decided that the logistic function, G(x̂iTθ) = G(θTx̂i), is a good model of the probability of producing a label yi given the observation x̂iT.

Under the i.i.d. assumption, you can interpret the label yi as the result of flipping a coin, or a Bernoulli trial, where the probability of success (yi = 1) is defined as gi = gi(θ) ≡ G(x̂iTθ)

Thus, Pr[yi|x^i,θ] ≡ gi ^ yi · (1−gi) ^ 1−yi. (The probability changes according to the labels)

The log-likelihood in turn becomes,
L(θ;y,X) = ∑i:0,m−1 yi ln(gi) + (1−yi) ln (1−gi) =
∑i:0,m−1 yi ln(gi / 1−gi) +ln(1−gi) = (from logistic properties)
∑i:0,m−1 yi θTx̂ i + ln(1−gi).

You can write the log-likelihood more compactly in the language of linear algebra.

Convention 1. Let u≡(1,…,1)T be a column vector of all ones, with its length inferred from context. Let A=(a0a1⋯an−1)be any matrix, where {ai} denote its n columns. Then, the sum of the columns is:

∑i:0,n−1 a i= (a0 a1 ⋯ an−1)⋅(11⋮1)T = Au.

Convention 2. Let A=(aij) be any matrix and let f(y) be any function that we have defined by default to accept a scalar argument y and produce a scalar result. For instance, f(y)=lny or f(y)=G(y). Then, assume that B=f(A) applies f(⋅) elementwise to A , returning a matrix B whose elements bij = f(aij)
With these notational conventions, convince yourself that these are two different ways to write the log-likelihood for logistic regression.

(V1) L(θ;y,X) = yT lnG(Xθ) + (u−y)T ln [u−G(Xθ)]

(V2) L(θ;y,X) = yT Xθ + uT ln G(−Xθ)

Lesson 4: Gradient Ascent

To optimize the log-likelihood with respect to the parameters, θ , you want to "set the derivative to zero" and solve for θ. For example, recall that in the case of linear regression via least squares minimization, carrying out this process produced an analytic solution for the parameters, which was to solve the normal equations. Unfortunately, for logistic regression---or for most log-likelihoods you are likely to ever write down---you cannot usually derive an analytic solution. Therefore, you will need to resort to numerical optimization procedures.

Gradient ascent, in 1-D.

A simple numerical algorithm to maximize a function is gradient ascent (or steepest ascent). If instead you are minimizing the function, then the equivalent procedure is gradient (or steepest) descent. Here is the basic idea in 1-D.

Suppose we wish to find the maximum of a scalar function f(x) in one dimension. At the maximum, df(x)dx=0

Suppose instead that dfdx≠0 and consider the value of f at a nearby point, x+s , as given approximately by a truncated Taylor series:

f(x+s) = f(x) + s df(x)dx + O(s2)

To make progress toward maximizing f(x), you'd like to choose s so that f(x+s) > f(x). One way is to choose $s=\alpha \cdot \text{sign}(dfdx)$, where $0<\alpha \ll 1$is "small:"

$f(x+\alpha \cdot \text{sign}(dfdx)) \approx f(x) + \alpha|dfdx| + O(\alpha 2)$

If α is small enough, then you can neglect the $O(\alpha 2)$ term and f(x+s) will be larger than f(x), thus making progress toward finding a maximum.

This scheme is the basic idea: starting from some initial guess x, refine the guess by taking a small step s in the direction of the derivative, i.e., sign(dfdx).

Gradient ascent in higher dimensions.

Now suppose x is a vector rather than a scalar. Then the value of f at a nearby point f(x+s), where s is a vector, becomes

$f(x+s) = f(x) + sT\nabla xf(x) + O(// s // ^2),$

where $\nabla xf(x)$ is the gradient of f with respect to x. As in the 1-D case, you want a step s such that f(x+s)>f(x). To make as much progress as possible, let's choose s to be parallel to $\nabla xf(x)$, that is, proportional to the gradient:

$s \equiv \alpha \nabla xf(x) / // \nabla xf(x) //$.

Again, α is a fudge (or "gentle nudge?") factor. You need to choose it to be small enough that the high-order terms of the Taylor approximation become negligible, yet large enough that you can make reasonable progress.

The gradient ascent procedure applied to MLE.

Applying gradient ascent to the problem of maximizing the log-likelihood leads to the following algorithm.

1. Start with some initial guess, θ(0).
2. At each iteration t≥0 of the procedure, let θ(t) be the current guess.
3. Compute the direction of steepest ascent by evaluating the gradient, $\Delta t \equiv \nabla \theta(t)\{L(\theta(t);y,X)\}$
4. Define the step to be $st \equiv \alpha \Delta t / // \Delta t //$, where α is a suitably chosen fudge factor.
5. Take a step in the direction of the gradient, $\theta(t+1) \leftarrow \theta(t)+st$
6. Stop when the parameters don't change much or after some maximum number of steps.

This procedure should remind you of one you saw in a prior notebook (the least mean square algorithm for online regression!). As was true at that time, the tricky bit is how to choose αα. There is at least one difference between this procedure and the online regression procedure you learned earlier. Here, we are optimizing using the full dataset rather than processing data points one at a time. (That is, the step iteration variable tt used above is not used in exactly the same way as the step iteration in LMS.)

Another question is, how do we know this procedure will converge to the global maximum, rather than, say, a local maximum? For that you need a deeper analysis of a specific L(θ;y,X), to show, for instance, that it is convex in θ.

Gradient of the likelihood

$\nabla\theta\{L(\theta;y,X)\} = XT[y-G(X\cdot\theta)]$.

From (V2) $L(\theta;y,X) = yTX\theta + uT\ lnG(-X\theta)$.

Thus,

$\nabla\theta\{L(\theta;y,X)\} = \nabla\theta(yTX\theta) + \nabla\theta(uT\ lnG(-X\theta))$.

Let's consider each term in turn.

For the first term, apply the gradient identities to obtain

$\nabla\theta(yTX\theta) = \nabla\theta(\theta TXTy) = XTy$.

For the second term, recall the scalar interpretation of $uT\ lnG(-X\theta)\ \sum j:0,m-1\ lnG(-\tilde{x}jT\theta)$.

The i-th component of the gradient is $\partial/\partial\theta i\ \sum j=0m-1lnG(-x\hat{}jT\theta)=\sum j=0m-1\partial\partial\theta ilnG(-x\hat{}jT\theta)$.

Let's evaluate the summand:

$\partial\partial\theta ilnG(-x\hat{}jT\theta)=[ddzlnG(z)]\cdot[\partial z\partial\theta i]_$Let $z\equiv-x\hat{}jT\theta=G(-z)\cdot\partial\partial\theta i(-x\hat{}jT\theta)=-G(x\hat{}jT\theta)\cdot xji$.

Thus, the ii-th component of the gradient becomes

$[\nabla\theta(uTlnG(-X\theta))]i=-\sum j=0m-1G(x\hat{}jT\theta)\cdot xji$.

In other words, the full gradient vector is

$\nabla\theta(uTlnG(-X\theta))=-XTG(X\theta)$.

Putting the two components together,
$\nabla\theta\{(\theta;y,X)\}=XTy-XTG(X\theta)=XT[y-G(X\theta)]$.

See the notebook for newton and gradient proof its in notebook

Topic 14: Clustering via k-means

We previously studied the classification problem using the logistic regression algorithm. Since we had labels for each data point, we may regard the problem as one of supervised learning. However, in many applications, the data have no labels but we wish to discover possible labels (or other hidden patterns or structures). This problem is one of unsupervised learning. How can we approach such problems?

Clustering is one class of unsupervised learning methods. In this lab, we'll consider the following form of the clustering task. Suppose you are given

a set of observations, $X\equiv\{\hat{x}\ i|0\leq i<n\}$, and
a target number of clusters, k .

Your goal is to partition the points into k subsets, $C0,\ldots,Ck-1\subseteq X$, which are

disjoint, i.e., $i\neq j\implies Ci\cap Cj=\varnothing$
but also complete, i.e., $C0\cup C1\cup\cdots\cup Ck-1=X$

Intuitively, each cluster should reflect some "sensible" grouping. Thus, we need to specify what constitutes such a grouping.

The k - means clustering criterion

Here is one way to measure the quality of a set of clusters. For each cluster C , consider its center μ and measure the distance $//\ x-\mu\ //$ of each observation $x\in C$ to the center. Add these up for all points in the cluster; call this sum is the within-cluster sum-of-squares (WCSS). Then, set as our goal to choose clusters that minimize the total WCSS over all clusters.

More formally, given a clustering $C=\{C0,C1,\ldots,Ck-1\}$, let

$WCSS(C)\equiv\sum i:0,k-1\ \sum x\in Ci\ //\ x-\mu i\ //\ \hat{}2$,

where μi is the center of Ci. This center may be computed simply as the mean of all points in Ci, i.e.,

$\mu i \equiv 1|Ci| \sum x \in Cix$.

Then, our objective is to find the "best" clustering, $C*$, which is the one that has a minimum WCSS.

$C* = \text{argmin}C\ WCSS(C)$.

The standard k - means algorithm (Lloyd's algorithm)

Finding the global optimum is NP-hard, which is computer science mumbo jumbo for "we don't know whether there is an algorithm to calculate the exact answer in fewer steps than exponential in the size of the input." Nevertheless, there is an iterative method, Lloyd's algorithm, that can quickly converge to a local (as opposed to global) minimum. The procedure alternates between two operations: assignment and update.

Step 1: Assignment. Given a fixed set of k centers, assign each point to the nearest center:

$Ci = \{\hat{x}: // \hat{x} - \mu i // \ \leq \ // \hat{x} - \mu j //\ ,1 \leq j \leq k\}$

Step 2: Update. Recompute the k centers ("centroids") by averaging all the data points belonging to each cluster, i.e., taking their mean:

$\mu i = 1/|Ci| \sum \hat{x} \in Ci\ \hat{x}$

It will be convenient to use our usual "data matrix" convention, that is, each row of a data matrix X is one of m observations and each column (coordinate) is one of d predictors. However, we will not need a dummy column of ones since we are not fitting a function.

$X \equiv (\hat{x}0T \vdots \hat{x}Tm) = (x0 \cdots xd-1)$.

Topic 15: Compression via PCA

The main topic of this lesson is a data analysis method referred to as Principal Components Analysis (PCA). The method requires computing the eigenvectors of a certain matrix; one way to compute those eigenvectors is to use a special factorization from linear algebra called the Singular Value Decomposition (SVD).

Lesson 0: Motivation

Motivation: data "compression." In previous lessons, we've looked at a few of the major tasks in data analysis: ranking, regression, classification, and clustering. Beyond these, the last problem you'll consider in our class is what we'll call compression.

At a high level, the term compression simply refers to finding any compact representation of the data. Such representations can help us in two ways. First, it can make the data set smaller and therefore faster to process or analyze. Secondly, choosing a clever representation can reveal hidden structure. As a concrete example, consider the problem of dimensionality reduction: given a d-dimensional data set, we wish to transform it into a smaller k-dimensional data set where k≤d Choosing the kk dimensions in a clever way might even reveal structure that is hard to see in all d original dimensions.

Lesson 1: Projections

Basic definitions

Input data matrix, centered.

Per our usual conventions, let $\hat{x}\,0,\ldots,\hat{x}\,m-1$ be the m data points, where each $xi\in\mathbb{R}d$ is a single observation. Each observation is represented by a d -dimensional real-valued vector corresponding to d measured predictors. As usual, we can stack these into a data matrix, denoted $X \equiv (\hat{x}0T \vdots \hat{x}m-1T)$

However, we'll add one more important assumption: these data should be centered about their mean, i.e., $1/m \sum i:0,m-1 \hat{x}\, i = 0$. If the observations are not centered initially, then preprocess them accordingly.

Projections

Let $\varphi\in\mathbb{R}d$ be a vector of unit length, i.e., $// \varphi // 2^2 = \varphi T\varphi = 1$. The projection of a data point $\hat{x}i$ onto φ is $\hat{x}iT\varphi$, which measures the length of the projected vector.

Lesson 2: PCA

Maximized projections

If the length of a projected data point is large, then intuitively, we have "preserved" its shape. So let's think of the total length of projections of all the data points as a measure of cost, which we can then try to maximize.

Projection cost.

Let J(φ) be a cost function that is proportional to the mean squared projections of the data onto φ:

J(ϕ)≡1/2m ∑i:0,m−1 (x̂i Tφ)^2.

The additional factor of "1/2" is for aesthetic reasons. (It cancels out later on.)

Let's also apply some algebra-fu to the right-hand side to put it into a more concise matrix form: J(ϕ) ≡ ½ φT (1/m ∑i:0,m−1 x̂i x̂iT) φ = ½ φT(1/mXTX)φ = ½ φTCφ.

In the last step, we defined C≡1/m XTX. In statistics, if X represents mean-centered data, then the matrix C is also known as the sample covariance matrix of the data.

Principal components via maximizing projections.

There are several ways to formulate the PCA problem. Here we consider the one based on maximizing projections.

Start by defining a principal component of the data X to be a vector, φ, of unit length that maximizes the sum of squared projections.

To convert this definition into a formal problem, there is a technique known as the method of Langrange multipliers, which may be applied to any minimization or maximization problem that has equality constraints. The idea is to modify the cost function in a certain way that effectively incorporates each constraint: for each constraint you will add to the cost function a term proportional to a dummy parameter times some form of the constraint.

Huh? It's easiest to see this formulation by example. In the case of a principal component, the modified cost function is

Ĵ (φ,λ) ≡ J(φ) + λ2(1−φTφ),

where the second term captures the constraint: it introduces a dummy optimization parameter, λ, times the constraint that φ has unit length, i.e., // φ // 2^2 = φTφ =1 or 1−φTφ = 0

The reason to add the constraint in this way should become clear momentarily.

As before, the factor of "1/2" is there solely for aesthetic reasons and will "cancel out," as you'll soon see.

The optimization task is to find the $\varphi*$ and $\lambda*$ that maximize $\hat{J}$:

$(\varphi*,\lambda*) \equiv \text{argmax}_{\varphi,\lambda}\ \hat{J}\ (\varphi,\lambda)$.

To solve this optimization problem, you just need to "take derivatives" of $\hat{J}$ with respect to φ and λ and then set these derivatives to 0.

$\nabla_{\varphi}\hat{J} = C\varphi - \lambda\varphi$
$\partial/\partial\lambda\ \hat{J} = 1/2(1 - \varphi T\varphi)$

Setting these to zero and solving yields the following computational problem:

$C\varphi = 1/m\ XTX\varphi = \lambda\varphi$
$// \varphi // 2\text{^}2 = 1.$

Is it now clear why the constraint was incorporated into $\hat{J}$ as it was? Doing so produces a second equation that exactly captures the constraint!

This problem is an eigenproblem, which is the task of computing an eigenvalue and its corresponding eigenvector of $C = 1/m\ XTX$

The matrix C will usually have many eigenvalues and eigenvectors. So which one do you want? Plug the eigenvector back into the original cost function. Then,

$J(\varphi) = ½\ \varphi TC\varphi = \lambda/2\ \varphi T\varphi = \lambda/2$. In other words, to maximize $J(\varphi)$ you should pick the φ with the largest eigenvalue λ.

Finding an eigenpair via the SVD

So how do you find the eigenvectors of C? That is, what algorithm will compute them?
One way is to form C explicitly and then call an off-the-shelf eigensolver. However, forming C explicitly from the data X may be costly in time and storage, not to mention possibly less accurate. (Recall the condition number blow-up problem in the case of solving the normal equations.)

Instead, we can turn to the "Swiss Army knife" of linear algebra, which is the singular value decomposition, or SVD. It is an extremely versatile tool for simplifying linear algebra problems. It can also be somewhat expensive to compute accurately, but a lot of scientific and engineering

effort has gone into building robust and reasonably efficient SVD algorithms. So let's assume these exist -- and they do in both Numpy and Scipy -- and use them accordingly.

The SVD.

Every real-valued matrix $X \in \mathbb{R}^{m \times d}$ has a singular value decomposition. Let $s = \min(m,d)$, i.e., the smaller of the number of rows or columns. Then the SVD of X is the factorization, $X = U\Sigma V^T$, where U, Σ, and V^T are defined as follows.

The matrices $U \in \mathbb{R}^{m \times s}$ and $V \in \mathbb{R}^{d \times s}$ are orthogonal matrices, meaning $U^TU=I$ and $V^TV=I$; and the matrix Σ is an $s \times s$ diagonal matrix.

Note that V is taken to be $d \times s$, so that the V^T that appears in $U\Sigma V^T$ is $s \times d$.

The columns of U are also known as the left singular vectors, and the columns of V are the right singular vectors of X. Using our usual "column-view" of a matrix, these vectors are denoted by ui and vi:

$U = [u_0\ u_1 \cdots u_{s-1}]$

$V = [v_0\ v_1 \cdots v_{s-1}]$

Regarding the diagonal matrix Σ, its entries are, collectively, called the singular values of X:

$[\sigma_0\ \sigma_1\ \ddots\ \sigma_{s-1}]$.

From these definitions, the SVD implies that $XV = U\Sigma$. This form is just a compact way of writing down a system of independent vector equations,

$Xv_i = \sigma_i u_i$

Recall that in PCA, you want to evaluate $C = 1/m\ X^TX$. In terms of the SVD,

$X^TX = V\Sigma^TU^TU\Sigma V^T = V\Sigma^{\wedge}2V^T$,

or

$X^TXV=V\Sigma 2$.

This relation may in turn be rewritten as the system of vector equations,

$X^TX\ v_i = \sigma_i^{\wedge}2\ v_i$.

In other words, every pair $(\varphi,\lambda) \equiv (v_i,\sigma 2_{im})$ is a potential solution to the eigenproblem,

Cφ = 1/m XTX φ =λφ

The pair with the largest eigenvalue is (v0,σ0^2/m)

Lesson 3: Truncated SVD

Rank-k approximations: the truncated SVD

We motivated PCA by asking for a single vector φ, which effectively projects the data onto a one-dimensional subspace (i.e., a line). You might instead want to represent the original d-dimensional data points on a k-dimensional surface or subspace, where k ≤ s ≤ d. As the previous discussion suggests, you could choose the top-k right singular vectors of X, v0,…,vk−1.

Indeed, there is another "principled" reason for this choice. Let A∈ℝm×d be any matrix with an SVD given by A=UΣVT. Per the notation above, let s ≡ min(m,d).

Then, define the k - truncated SVD as follows. Consider any k ≤ s, and let Uk, Σk, and Vk consist of the singular vectors and values corresponding to the k largest singular values. That is, Uk is the first k columns of U, Vk is the first k columns of V, and Σk is the upper k×k submatrix of Σ. The k-truncated SVD is the product UkΣkVTk

Now consider the following alternative way to write the SVD:

A = UΣVT = ∑i:0,s−1 uiσiviT

Each term, uiσiviT is known as a rank-1 product. So the existence of the SVD means that A may be written as a sum of rank-1 products.
It would be natural to try to approximate A by truncating the SVD after k terms, i.e.,

A ≈ UkΣkVkT = ∑i:0,k−1 uiσiviT.

And in fact, there is no rank-k approximation of A that is better than this one!

In particular, consider any pair of k column vectors, Yk∈ℝm×k and Zk∈ℝd×k; their product, YkZk has rank at most k. Then there is a theorem that says the smallest difference between A and the rank-k product YkZk, measured in the Frobenius norm, is

minYk,Zk // A−YkZkT // F^2 = // A−UkΣkVkT // F^2 = σk^2 + σk+1^2 + σk+2^2 +⋯+σs−1^2

In other words, the truncated SVD gives the best rank-k approximation to A in the Frobenius

norm. Moreover, the error of the approximation is the sum of the squares of all the smallest s−k singular values.

Applied to the covariance matrix, we may conclude that C = 1/m XTX ≈ 1/m VkΣk^2VkT is in fact the best rank-k approximation of C, which justifies choosing the k eigenvectors corresponding to the top k eigenvalues of C as the principal components.

Summary: The PCA algorithm

Based on the preceding discussion, here is the basic algorithm to compute the PCA, given the data X and the desired dimension k of the subspace.

If the data are not already centered, transform them so that they have a mean of 0 in all coordinates, i.e., $1/m \sum_{i:0,m-1} \hat{x}_i = 0$
Compute the k - truncated SVD, $X \approx U_k \Sigma_k V_k^T$.
Choose $v_0, v_1, \ldots, v_{k-1}$ to be the principal components.