EXERCISES IN COMPUTATIONAL ANALYTICS

UNIVERSITY OF SAN FRANCISCO

Master of Science in Analytics

TERENCE PARR

parrt@cs.usfca.edu

http://parrt.cs.usfca.edu

A BONKERS THE CAT PRODUCTION

Contents

Introduction

Welcome to MSAN501, the computational analytics boot camp at the University of San Francisco! This exercise book collects all of the labs you must complete by the end of the boot camp in order to pass. The labs start out as very simple tasks or step-by-step recipes but then accelerate in difficulty, culminating with an interesting text analysis project. You will build all projects with Python (version 2, not 3).

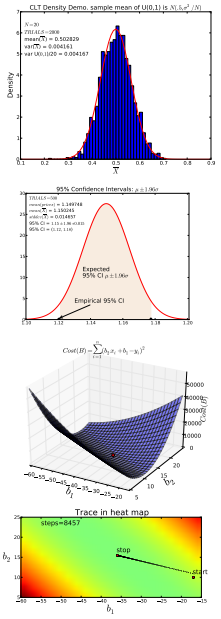
This course is specifically designed as an introduction to analytics programming for those who are not yet skilled programmers. The course also explores many concepts from math and statistics, but in an empirical fashion rather than symbolically as one would do in a math class. Consequently, this course is also useful to programmers who would like to strengthen their understanding of numerical methods.

The exercises are grouped into four parts. We begin with simple programs to compute statistics, build simple data structures, and use libraries to create visualizations. The second part strives to give an intuitive feel for random variables, density functions, the central limit theorem, hypothesis testing, and confidence intervals. It's one thing to learn about their formal definitions, but to get a really solid grasp of these concepts, it really helps to observe statistics in action. All of the techniques we'll use in empirical statistics rely on the ability to generate random values from a particular distribution. We can do it all from a uniform random number generator, which is the first exercise in that part.

The third set of exercises deals with function optimization. Given a particular function, $f(x)$, optimizing it generally means finding its minimum or maximum, which occur when the derivative goes flat: $f'(x) = 0$. When the function's derivative cannot be derived symbolically, we're left with a general technique called *gradient descent* that searches for minima. It's like putting a marble on a hilly surface and letting gravity bring it to the nearest minimum.

Finally, part four has an exercise that introduces text analysis. We will compute something called *TFIDF* that indicates how well that word distinguishes a document from other documents in a corpus. That score is used broadly in text analytics, but our exercise uses it to summarize documents by listing the most important words.

You will work mostly on your own laptops, but you must get familiar with the UNIX command line. It's also important to learn how to install software and execute commands on a remote server; servers or what provide the websites you visit while browsing and they provide services to mobile apps on your phone. We've received an educational grant from Amazon to use their compute cloud called Amazon Web Services (AWS). We also have access to an IBM cluster, housed in the College of arts and sciences at USF.



As you progress through these exercises, you'll learn a great deal about Python and the following libraries: `matplotlib`, `numpy`, `scipy`, and `py.test`. I also recommend that you learn how to use a Python development environment called PyCharm, for which we have been granted a site license.

Part I

Python Programming and Data Structures

Computing Point Statistics

Discussion

The goal of this task is to get familiar with Python function definitions and looping structures, as well as to refresh your memory about a few point statistics.

Stats

This exercise involves writing functions to compute sample mean, variance, and covariance from a data set (list of values). In mathematics notation, the sample estimates are:

$$\bar{x} = \frac{1}{N}\sum_{i=1}^{N} x_i \qquad \text{(Sample mean)}$$

$$s^2 = \frac{1}{N-1}\sum_{i=1}^{N}(x_i - m)^2 \qquad \text{(Unbiased sample variance)}$$

$$cov(x,y) = \frac{1}{N-1}\sum_{i=1}^{N}(x_i - \bar{x})(y_i - \bar{y}) \qquad \text{(Unbiased sample covariance)}$$

In Python, you must define functions `mean(x)`, `var(x)`, `cov(x,y)` where `x` and `y` are objects that behave like a list or iterator. The functions return a floating-point value based on the above mathematics notation. If the length of the incoming vectors to `cov` are not the same, return 0. To test things out, use the `test_stats.py` file provided for you as part of this task.

Libraries

While we're at it let's learn about importing libraries. You'll notice that `test_stats.py` references your code like this:

```
from stats import *
```

That lets us directly access the functions defined in your `stats.py` file.

You can also test the correctness of the functions by using the `numpy` lib, make sure you ask for the sample population statistics by using parameter `ddof=1` for `var()` and `cov()`. E.g., `np.var(data, ddof=1)`. Be careful not to confuse function names; e.g., `numpy` has functions with the same names (although cov() returns a covariance matrix).

```
import numpy as np  # np is an alias for the numpy library
x = ...
y = ...
print np.cov(x,y)[0][1] # np.cov returns cov matrix
```

We now have the kernel of a small statistics library.

Deliverables

- `stats.py`

You may not use sum() or any other built-in functions for this project to compute the point statistics. The whole point of the exercise is to learn to build your own for loops. Obviously.

Approximating $\sqrt{n}$ with the Babylonian Method

Motivation

This lab is really a fancy way to learn about looping in Python and how to quickly prototype something in Excel (if warranted). It also gets you used to encoding mathematical expressions and recurrence relations in Python.

Discussion

To approximate square root, $\sqrt{n}$, the idea is to pick an initial estimate, x_0, and then iterate with better and better estimates, x_i, using the recurrence relation:

$$x_{i+1} = \frac{1}{2}(x_i + \frac{n}{x_i})$$

To see how this works, jump into Excel (yes, a spreadsheet) and crank through a few iterations by defining cells with n and your initial estimate x_0, which can be anything you want. (It's sometimes easier to play around without having to deal with a programming language.) Then you need to define a cell that computes the above better approximation using x_i as the cell above it. I hardcoded the names in column A and the first two rows of column B. Cell B3 should be a formula that computes B4 based upon B3. Then you can extend the formula down and watch it converge on the final (correct) value for $\sqrt{125348}$. My spreadsheet looks like this:

	A	B
1	n	125348
2	x_0	20
3	x_1	3143.7
4	x_2	1591.78638
5	...	835.266564
6		492.668011
7		373.547461
8		354.554285
9		354.04556
10		354.045195
11		354.045195

Try out any nonnegative number and you'll see that it still converges, though at different rates.

There's a great deal on the web you can read to learn more about why this process works but it relies on the average (midpoint) of x and n/x getting us closer to $\sqrt{n}$. It can be shown that if x is above $\sqrt{n}$ then n/x is below $\sqrt{n}$ and the reverse is true if x is below the root. The iteration converges and does so quickly. Informally, as shown in Wikipedia, we can represent the true square root by adding an error term to our estimate:

$$\sqrt{n} = x + \epsilon$$

or,

$$n = (x + \epsilon)^2$$

Expanding, we get:

$$n = x^2 + 2x\epsilon + \epsilon^2$$

Solving for ϵ:

$$n - x^2 = \epsilon(2x + \epsilon)$$

$$\epsilon = \frac{n - x^2}{2x + \epsilon}$$

Because ϵ is much smaller than x, we can drop it from the denominator leaving us with an estimate of epsilon:

$$\epsilon = \frac{n - x^2}{2x}$$

Then we can plug it back into $x + \epsilon$ and get:

$$x := x + \epsilon = x + \frac{n - x^2}{2x} = \frac{2x^2}{2x} + \frac{n - x^2}{2x} = \frac{1}{2}\frac{x^2 + n}{x} = \frac{1}{2}(x + \frac{n}{x})$$

Which gets its back to the Babylonian formula. Since we dropped an ϵ term, this formula for x is inexact but it gets us closer to $\sqrt{n}$.

Now that you understand how this estimate works, your goal is to implement a simple Python method called sqrt() that uses the Babylonian method to approximate the square root. Here is a starter kit for you. Please call the file `sqrt.py`.

```
import math

# Stop iterating when the new approximation is within
# PRECISION of the old value.
PRECISION = 0.000001

# compute square root of n
def sqrt(n):
    x_0 = 1.0 # pick any old value
    ... fill this in ...

def check(n):
    delta = math.sqrt(n) - sqrt(n)
    if math.abs(delta) > PRECISION:
        raise BaseException("Inaccurate sqrt(%f)=%f; estimate is %f" %
            (n, math.sqrt(n), sqrt(n)))
```

```
# check a range of values
check(125348)
check(100)
check(1)
check(0)
```

Deliverables

Please submit:

- a PDF showing a snapshot of your spreadsheet
- the formula you used in B3 **and** B4.
- your `sqrt.py` Python file

You may not use math.sqrt() for implementing your function, but you may use it for testing the results. Obviously.

Generating Uniform Random Numbers

Q: *How to generate pure random string*?
A: *Put a fresh student in front of vi editor and ask him to quit.*
— Emiliano Lourbet (@taitooz)

Discussion

To perform computer-based simulation we need to be able to generate random numbers. Generating random numbers following a uniform distribution are the easiest to generate and are what comes out of the standard programming language "give me a random number" function. Here's a sample Python session:

```
>>> import random
>>> print random.random()
0.635864960246
>>> print random.random()
0.928451783541
>>> print random.random()
0.436722571069
```

We could generate real random numbers by accessing, for example, noise on the ethernet network device but that noise might not be uniformly distributed. We typically generate pseudorandom numbers that aren't really random but look like they are. From Ross' *Simulation*, we see a very easy recursive mechanism that generators values in $[0, m-1]$:

$x_n = ax_{n-1}$ modulo m

That's recursive (or iterative and not *closed form*) because x_n is a function of a prior value:

$x_1 = ax_0$ modulo m
$x_2 = ax_1$ modulo m
$x_3 = ax_2$ modulo m
$x_4 = ax_3$ modulo m
...

To get random numbers between 0 and 1, we use x_n / m.

We must pick a value for a and m that make x_n seem random. Ross suggests choosing a large prime number for m that fits in our integer word size, e.g., $m = 2^{31} - 1$, and $a = 7^5 = 16807$.

Initially we set a value for x_0, called the *random seed* (it is not the first random number). Every seed leads to a different sequence of pseudorandom numbers. In Python, you can set the seed of the standard library by using `random.seed([x])`.

Your goal is to take that simple recursive formula and use it to generate the first 10 random numbers using a `for` loop in Python as part of the "main" code. Please use file `varunif.py` and make function `runif()` that returns a new random value per call. Use $m = 2^{31} - 1$, $a = 7^5 = 16807$, and an initial seed of $x_0 = 666$. Your output should look something like:

```
0.00521236192678
0.604166903349
0.233144581892
0.460987861017
0.822980116505
0.826818094508
0.331714398848
0.1239014343
0.411406287184
0.505468696591
```

Because we are all using the same seed, the sequence of numbers should be the same.

Next, make function `runif_(a,b)` that returns random values between `a` and `b`. Your function definitions should look like:

```
# U(0,1)
def runif():
    ...

# U(a,b)
def runif_(a,b):
    ...
```

Deliverables

You must submit your `varunif.py` containing a main loop and your functions runif and runif_. Also submit your output via Canvas as a text file.

You may not use random.random() or any other built-in random number generators for this project. Obviously.

Histograms Using `matplotlib`

TODO: reuse runif() from previous project for 2015.

Discussion

The goal of this lab is to teach you the basics of using matplotlib to display probability mass functions, otherwise known as histograms. In this lab we will use the uniform distribution. Use filename `hist.py`.

Steps

1. import the proper libraries

```
import matplotlib.pyplot as plt
import numpy as np
```

2. Get a sample of uniform random variables in $U(0, 1)$

```
N = 1000
X = [np.random.uniform(0,1) for i in range(N)] # U(0,1)
# or np.random.uniform(0,1,N)
```

3. Display a histogram using matplotlib (in a separate window)

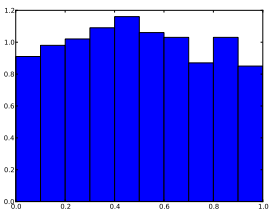
```
plt.hist(X, normed=1)
plt.show()
```

4. Run it.

5. Now, save the image as a PDF to the same directory by inserting a save command in between the histogram and the show method:

```
plt.hist(X, normed=1)
plt.savefig('unif-0-1-density.pdf', format="pdf")
plt.show()
```

6. Run it. Your pdf file should look like



7. Graphs should always have the axes labeled. Let's do that as well as add a title and set the range of the graph. Put this code right before the `savefig()`.

```
plt.title("U(0,1) Density Demo")
plt.xlabel("X", fontsize=16)
plt.ylabel("Density", fontsize=16)
plt.axis([0, 1, 0, 2])
```

8. Run it.

9. It's also common to add some annotations inside the graph to explain more about what we are seeing. First, we need to get access to the figure itself and then has to figure about its axes. (We need this in order to specify coordinates within the graph.) Put the following code before the `hist()` call.

```
fig = plt.figure()        # get a handle on the figure object itself
ax = fig.add_subplot(111) # weird stuff to get the Axes object within figure
```

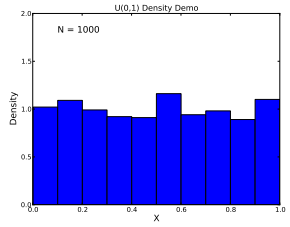
Then, before the `savefig()`, add the following to display some text above the histogram within the graph. The coordinates are from 0..1 where 0 is the left/bottom edge and 1 is the right/top edge.

```
# put N=... at top left
plt.text(.1,.9, 'N = %d' % N,
         fontsize=16,
         transform = ax.transAxes)
```

10. Also, let's change the file name slightly so we can keep our original graph plus our fancy one:

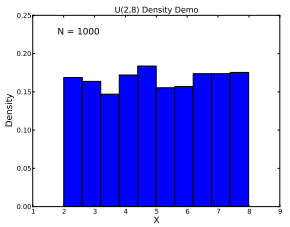
```
plt.savefig('unif-0-1-density-fancy.pdf', format="pdf")
```

11. Run it.



To understand distributions, it's a great idea to start messing around with the parameters of the density or mass function.

12. Change $U(0,1)$ to $U(2,8)$ and examine the results. You will have to alter the axis() to use different ranges. (Or let the plotting software do the work for you and get rid of the axis() call.) Run it. You should see something like the following.



Deliverables

Please submit:

- your `hist.py` Python file
- a PDF of your $U(2, 8)$ graph.

Launching a Virtual Machine at Amazon Web Services

Discussion

The goal of this lab is to teach you to create a Linux machine, login, and copy some data to that machine.

Steps

1. Go to your EC2 dashboard at AWS and click "Launch Instance"; Use the classic Wizard.

2. Select the "Amazon Linux AMI" server, which should be the first one. 64bit.

3. Select instance type "m1.small," which should be the second machine type listed. then click continue.

4. Skip over the advanced instance options by clicking continue.

5. Skip over the storage device configuration

6. For the key named "Name", change the value to something like *youruserID*-windows or something like that so that you can identify it later if you have multiple machines going. click continue.

7. The first time, you will need to create a new key pair. Name it as your user ID then click on the create and download keyfile. It will leave you with a *userid*.pem file, which are your security credentials for getting into the machine. Save that file in a safe spot. If you lose it you will not be able to get into your machine that you create.

8. Choose the default security group, which controls the firewall.

9. Tell it to create the instance. Close that pop up and it will take you to the instances you.

10. We need port 22 open so that `ssh` can get through the firewall to our computer. ssh listens at port 22 for connections from the outside world. Click on security groups in the left gutter. Click on "default" and then select the "inbound" tab. Put 22 in the port range and click Add Rule. Then click apply rule changes. You will see now that it allows port 22 (SSH) connections.

11. Go back to the instances view by clicking instances in the left gutter. Right-click on your instance in the display and select Connect. Click on the "Connect with a standalone SSH client" link and then inside you will see the `ssh` command necessary to connect to your machine. If you have Windows, there is a link to show you how to use an SSH client called PuTTY. For mac and linux users, we will use the direct ssh command from the command line. It will be something like:

```
ssh -i parrt.pem ec2-user@ec2-23-22-115-148.compute-1.amazonaws.com
```

Naturally, you will have to provide the full pathname to your user.pem file.

12. Before we can connect, we have to make sure that the security file is not visible to everyone on the computer (other users). Otherwise ssh will not let us connect because the security file is not secure.

```
$ cd ~/Dropbox/Terence
$ ls -l parrt.pem
-rw-r--r--@ 1 parrt  parrt  1696 Aug  4 15:15 /Users/parrt/Dropbox/Terence/parrt.pem
```

To fixed the permissions, we can use whatever "show information about file" GUI your operating system has or, from the command line, do this:

```
cd ~/Dropbox/Terence
chmod 600 parrt.pem
```

which changes the permissions like this:

```
cd ~/Dropbox/Terence
ls -l parrt.pem
```

Don't worry if you don't understand exactly what's going on there. it's basically saying that the file is only read-write for me, the current user, with no permissions to anybody else. If you don't do this properly, you will see something like this error when you try to connect later:

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@         WARNING: UNPROTECTED PRIVATE KEY FILE!          @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
Permissions 0644 for '/Users/parrt/Dropbox/Terence/parrt.pem' are too open.
```

13. Go to a shell on your mac or linux machine (or PuTTY on windows) and connect to the remote server. On my mac, it looks like this:

```
$ ssh -i ~/Dropbox/Terence/parrt.pem ec2-user@ec2-23-22-115-148.compute-1.amazonaws.com

       __|  __|_  )
       _|  (     /   Amazon Linux AMI
      ___|___|___|

https://aws.amazon.com/amazon-linux-ami/2013.03-release-notes/
There are 6 security update(s) out of 13 total update(s) available
Run "sudo yum update" to apply all updates.
[ec2-user@ip-10-242-203-15 ~]$
```

14. To get data up to the server, you can cut-and-paste if the file is small. For example, cut-and-paste the following data into a file called `coffee` in your home directory. First copy this data from the PDF:

```
3 parrt
2 jcoker
8 tombu
```

then type these commands and paste the data in the sequence:

```
$ cd ~ # get to my home directory
$ cat > coffee
3 parrt
2 jcoker
8 tombu
^D
$ cat coffee # print it back out
3 parrt
2 jcoker
8 tombu
$
```

The `^D` means control-D, which means end of file. `cat` is reading from standard input and writing to the file. The way it knows we are done is when we signal in the file with control-D.

15. For larger files, we need to use the secure copy `scp` command that has the same argument structure

as secure shell `ssh`. From the directory where you have the `coffee` file on your laptop, use the following similar command:

```
$ scp -i ~/Dropbox/Terence/parrt.pem access.log \
   ec2-user@ec2-23-22-115-148.compute-1.amazonaws.com:~ec2-user
access.log                                    100% 1363KB   1.3MB/s   00:00
$
```

Then, from the shell that is connected to the remote server, ask for the directory listing and you will see the new file:

```
[ec2-user@ip-10-242-203-15 ~]$ ls
access.log  coffee
```

16. STOP YOUR INSTANCE WHEN YOU ARE DONE! otherwise will continue to get charged for the use of that machine. If you right-click on the instance and say "STOP", it will stop the machine and you do not get charged but you can restart it without having to go through this whole procedure. If you say "TERMINATE", it will toss the machine out and you will have to go through this procedure again.

Deliverables

None. Please follow along in class.

Graph Adjacency Lists and Matrices

Goal

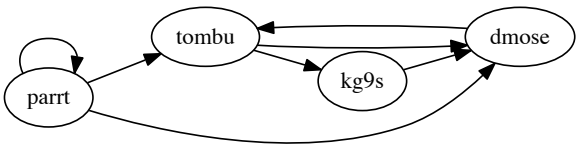
The goal of this task is to teach you about the implementation of graphs in Python, how to implement a few simple related algorithms, and do some simple data loading. As part of this exercise, you will also learn to transform data, which is an important data preparation skill you will need as an analyst.

Discussion

In this project, you will convert a string representation of a graph that looks like this:

```
parrt: tombu, dmose, parrt
tombu: dmose, kg9s
dmose: tombu
kg9s: dmose
```

to an adjacency list representation and ultimately generate a visual representation via graphviz/dot:



For fun, you will also create an edge matrix representation:

$$\begin{array}{c} \\ parrt \\ tombu \\ dmose \\ kg9s \end{array} \begin{array}{c} \begin{array}{cccc} parrt & tombu & dmose & kg9s \end{array} \\ \begin{pmatrix} 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \end{array}$$

where the nodes have the following indexes (all we really care about here is the order):

$$\begin{bmatrix} 0 : parrt \\ 1 : tombu \\ 2 : dmose \\ 3 : kg9s \end{bmatrix}$$

The following sections describe the functions you must create in `graph.py`. See a `graph.py` starter kit I've built for you at github.

Processing an adjacency list string

First, you have to process a string representation of an adjacency list and create an internal data structure:

```
def adjlist(adj_list):
    """
    Read in adj list and store in form of dict mapping node
    name to list of outgoing edges. Preserve the order you find
    for the nodes.
    """
    ...
```

You will use an ordered dictionary (`OrderedDict`) that maps node name x to a list of target nodes. x will be a string and the target list will be a list of strings. For example, from line in string adj_list

`parrt: tombu, dmose, parrt`

you will create an entry in the dictionary with key `parrt` and value:

```
['tombu', 'dmose', 'parrt']
```

To process the text, you must split the incoming string into lines and then process them one at a time as each line represents an adjacency list. You will use string functions `split` and (likely) `strip` to process the text. The goal here is to learn how to process text so don't look for built-in functions that do all of this for you.
Printing the adjacency list dictionary from `adjlist`, we should all get the following output:

```
OrderedDict([('parrt', ['tombu', 'dmose', 'parrt']),
 ('tombu', ['dmose', 'kg9s']),
 ('dmose', ['tombu']),
 ('kg9s', ['dmose'])])
```

Adjacency list to adjacency matrix

Given an adjacency list stored as a dictionary per adjlist(), create a function that converts it to an adjacency matrix:

```
def adjmatrix(adj):
    """
    From an adjacency list, return the adjacency matrix with entries in {0,1}.
    The order of nodes in adj is assumed to be same as they were read in.
    """
    ...
```

The matrix should look like the one shown above.

Getting a list of all nodes

A very useful function to have is the following that returns a list of all nodes visited starting at a particular node in a graph.

```
def nodes(adj, start_node):
    """
    Walk every node in graph described by adj list starting at start_node
    using a breadth-first search.  Return a list of all nodes found (in
    any order). Include the start_node.
    """
    ...
```

Do not build a recursive function as you must do a breadth-first search. (Recursive functions are much more useful when doing a depth-first search.) The basic algorithm looks like this:

```
visited = [];
add the start node to a work list;
while more work do
  node = remove a node from work list;
  add node to visited list;
  targets = adjacency_list[node];
  add all unvisited targets to work list;
end
return visited;
```

Generating DOT output

In order to visualize the graph you have read in, create the following function that dumps valid Graphviz DOT code, given an adjacency list. Then cut-and-paste the output and put it into Graphviz to display it.

```
def gendot(adj):
    """
    Return a string representing the graph in Graphviz DOT format
    with all p->q edges. Parameter adj is an adjacency list.
    """
    ...
```

Or, to amaze your family and friends, you can directly from the command line on a mac or unix box:

```
python test_dot.py | dot -Tpdf > /tmp/graph.pdf; open /tmp/graph.pdf
```

Here is a simple test rig, `test_dot.py`, that translates an input string description to DOT and prints it out.

```
from graph import *

# test dot
g = \
"""
parrt: tombu, dmose, parrt
tombu: dmose, kg9s
dmose: tombu
```

```
kg9s: dmose
"""
list = adjlist(g)
dot = gendot(list)
print dot
```

For the adjacency list shown at the start of this assignment, you should to generate the following DOT code:

```
digraph g {
  rankdir=LR;
  parrt -> tombu;
  parrt -> dmose;
  parrt -> parrt;
  tombu -> dmose;
  tombu -> kg9s;
  dmose -> tombu;
  kg9s -> dmose;
}
```

Testing

I have provided `test_graph.py` and `test_dot.py` test rigs that exercise the required functions using the sample adjacency list described above. Please make sure that your library works with this test rig at minimum.

Deliverables

Please submit the following via canvas:

- `graph.py` (the functions inside should emit no output at all, just return data as specified)
- a text file with the output of running `test_dot.py`, showing the DOT output.
- a PDF showing the visual representation of the graph as generated by graphviz/dot

Part II

Empirical statistics

Generating Binomial Distributions

Goal

The goal of this lab is to simulate a binomial distribution using repeated Bernoulli trials and then compare it against the theoretical binomial distribution. Use filename `rbinomial.py`.

Discussion

1. First, import your uniform random number generator library and set the seed of the random number generator. (Otherwise you will always get the same Bernoulli trials.)

```
from varunif import runif
from varunif import setseed

# I defined a function in runif.py to hide implementation details
setseed( int(round(time.time() * 1000)) )
```

In this case, we're using the current time in milliseconds as the random seed so that it is different every time you run the program. (remember this trick.)

2. Next, define a function that performs n Bernoulli trials with probability p of success. It should return the number of successes out of n:

```
def binomial(n,p):
    "Sim with prob p, n bernoulli trials; return number of successes"
    ...
```

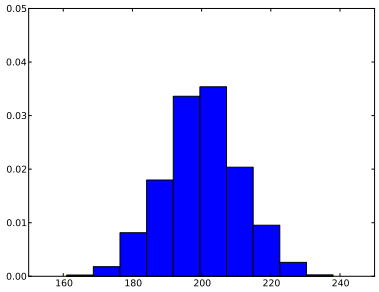
The pseudocode is just a loop that goes around n times and uses a variable from $U(0,1)$, using your `runif()` function, to check for success or failure. For example, my solution assumes failure if the uniform random variable is greater than p.

3. Now that we can know how to get a binomial random variable, we can examine the binomial distribution. All we have to do is grab a vector of, say, $SAMPLES$ binomial random values and the plot a histogram. The density function at k is just how many successes out of $SAMPLES$ there were ($k/SAMPLES$, actually).

Let me introduce you to something called a *list comprehension* in Python, which is a for loop that results in a list. It's also considered a *map* function ala *map-reduce*. Get list X as $SAMPLES$ binomial values with parameters $n = 500$ and $p = 0.4$. Do that by simply calling the `binomial` function N times.

```
X = [binomial(n,p) for t in range(SAMPLES)]
```

4. Plot the histogram normalized (normed=1) and run it. (You'll need `hist()`.) You should see a graph similar to the following:



5. We could use the built-in binomial mass function but let's define our own since it's easy:

$$\binom{n}{k} p^k (1-p)^{n-k} \qquad \text{(Binomial mass function)}$$

That's the probability that there are k successes in n trials with probability p of success. Define a function like this:

```
def binom(n, k, p):
    """
    If we run n trials with p prob for each trial of success,
    how many have k successes?
    """
    ...
```

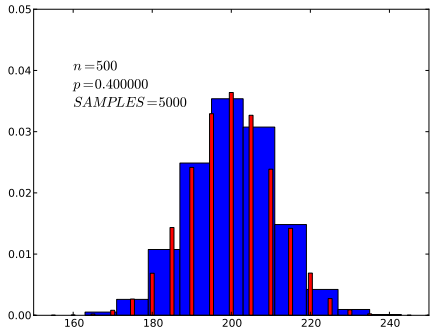
You may use function `scipy.misc.comb()` to compute $\binom{n}{k}$, but otherwise do the arithmetic yourself. (There is no loop in this function.)

6. To show the real distribution on top, we need to iterate k across the range $0..n$ used in our empirical simulation above. Since this is a mass function not a smooth density function, we can use every fifth value in the range. Let's also add some text to describe the parameters.

```
Y = [binom(n, k, p) for k in range(0,n+1,5)]
plt.bar(range(0,n+1,5), Y, color='red', align='center', width=1)
plt.axis([150,250,0,.05]) # set the axes so that we get a close-up
plt.text(160,0.04, '$n = %d$' % N, fontsize=16)
plt.text(160,0.037, '$p = %f$' % P, fontsize=16)
plt.text(160,0.034, '$SAMPLES = %d$' % SAMPLES, fontsize=16)
```

In this case I am not using 0..1 for the axes coordinates of the text; the default is the values of the graph itself. sometimes this is useful.

7. Run it and you should see something like the following:



Note that we use a bar chart for the binomial theoretical distribution and not a smooth graph because this is a mass function not a density function.

Deliverables

Please submit:

- your `rbinomial.py` file with values $n = 500, p = 0.4, SAMPLES = 5000$. Include the code to show the histogram but only run it as "main".
- submit a PDF of your final graph.

Generating Exponential Random Variables

Discussion

The goal of this lab is to generate random values from the exponential distribution using the inverse transform method. You will show the histogram of the random values and then show the theoretical exponential distribution on top to verify your results. You will reuse your exponential distribution random variable generator for the central limit theorem lab. Use filename `rexp.py`.

Steps

1. First, create a function called `rexp(lambduh)` that returns a random value from the exponential distribution using the inverse transform method. To do that, you need the inverse *cumulative distribution function* (CDF) for the exponential distribution $Exp(\lambda)$. The *probability density function* for the exponential distribution is:

$$p = F(x;\lambda) = \lambda e^{-\lambda x}$$

Therefore the inverse function to get the x value associated with a probability p, we use

$$x = F^{-1}(p;\lambda) = -\frac{ln(1-p)}{\lambda}$$

Your function should look like the following:

```
def rexp(lambduh): # lambduh mispelled to avoid clash with lambda in python
    # u = get value from U(0,1) then
    # return F^-1(u) for exp cdf F^-1
```

Use your `runif()` function from previous labs.

2. To plot things, you will need the usual libraries:

```
import math
import matplotlib.pyplot as plt
import numpy as np
```

3. Get a sample of exponential random variables into variable `X` of size N from $Exp(1.5)$ using your `rexp()`. I usually define constants to make the code more readable:

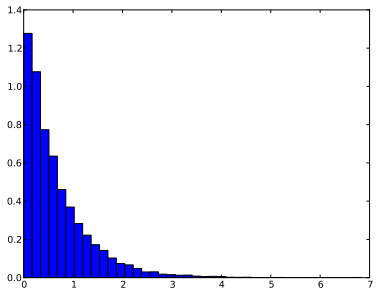
```
N = 1000
LAMBDUH = 1.5
```

then I can call `rexp(LAMBDUH)` and change LAMBDUH everywhere in my code by just changing the constant. In this case, there's no real need but it's good practice.

4. Verify that your exponential random variable behaves properly by displaying a histogram using matplotlib:

```
X = # N exponential random variables
plt.hist(X, bins=40, normed=1) # use bins option to get better resolution
plt.show()
```

You should see something like this:



How do we know that this accurately represents the exponential distribution? We plot the theoretical distribution on top with a red line.

5. Since it's easy, let's define our own exponential probability density function as follows:

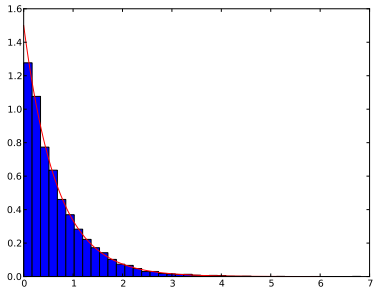
```
def exp_pdf(x, lambduh):
    ...
```

When you call it, make sure use the same lambduh.

6. Now, before the `show()`, plot the theoretical distribution so that we can see both at once:

```
# Show real distribution
x = np.arange(0,6, 0.01) # get a set of values from 0..6 stepping by 0.01
y = [exp_pdf(v, LAMBDUH) for v in x]
plt.plot(x,y, color='red')
```

You should see the following.



Deliverables

Please submit:

- your `rexp.py` file and please use the usual "if main" gate so that I can import your code for testing without creating the graph:

  ```
  if __name__ == '__main__':
  ```

- a PDF of your final graph.

The Central Limit Theorem in Action

Discussion

The goal of this lab is to observe how the sample means of uniform and exponential random variables have normal distributions with $N(\mu, \sigma^2/n)$ where σ^2 is the variance of the underlying distribution and n is the sample size whose mean we compute. Use filenames `clt_unif.py`, `clt_exp.py` for this lab.

Discussion

The CLT in a nutshell says that the sample mean, $X\_$, of samples X of size n from lots of distributions follows the normal distribution,specifically, $N(\mu, \sigma^2/n)$ for sample size n. In this lab will use $U(0,1)$ and the exponential distribution with $\lambda = 1.5$ and verify that using the mean as a random variable, the histogram shows a normal distribution of $N(0.5, 1/12)$ for the uniform and $N(\lambda^{-1}, \lambda^{-2}/n)$ for the exponential distribution. The mean of the uniform distribution is $\frac{a+b}{2}$ and the variance is $\frac{(b-a)^2}{12}$. The mean of the exponential distribution is $\mu = \lambda^{-1}$ and its variance is $\sigma^2 = \lambda^{-2}/n$.

The key thing here is to note that not only is the distribution of the mean random variable normal, but its variance gets tighter as we increase the sample size.

The law of large numbers says that the average of a large number of trials should approach the theoretical mean. That means that our sample mean:

$$\bar{X} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

will converge to μ as n approaches infinity with probability 1.

Also note that the number of trials we do improves the resolution of our normal distribution but doesn't change the variance.

CLT applied to uniform random variables

Steps

1. Import the usual libraries for plotting and then define these constants:

```
N = 4  # sample size (i.e, array size len(X))
TRIALS = 500 # how many samples we will take from the uniform distribution
```

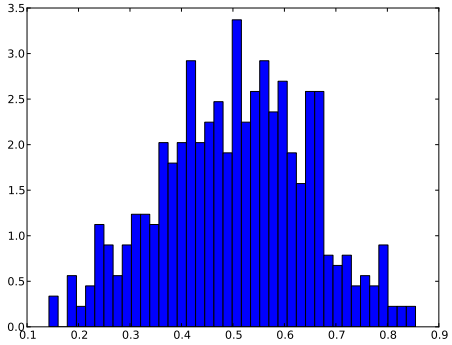
Now, we need to build a loop that gets `TRIALS` X vectors of size N with values from $U(0,1)$. Use your `runif()` function. Compute the mean of each X vector and add it to the end of a different array $X\_$.

2. Plot the histogram of `X_`:

```
# plot density of means (normalized histogram of means)
# WARNING: bins=40 is to show changes in resolution
```

```
#          where normally it's best to let the hist()
#          choose the bins for smoother view
plt.hist(X_, bins=40, normed=1)
```

3. Your histogram should look like this



Cool. It looks kind of like a normal distribution to me. Let's add the theoretical normal distribution on top. To do that we need the appropriate parameters of $Normal(\mu, \sigma^2/n)$. The mean μ of uniform samples should be midway between a and b from $U(a,b)$. In our case, that's 0.5 since we are doing $U(0,1)$. The variance of the uniform distribution is $(b-a)^2/12$ and we need the variance divided by sample size N. Define a function that returns the variance of uniform distribution $U(a,b)$:

```
def unifvar(a, b):
    ...
```

4. To get the theoretical distribution, let's define it ourselves:

```
def normpdf(x, mu, sigma): # sigma is the standard deviation, sigma^2 is the variance
    """
    Accept either a floating-point number or a numpy ndarray, such as what you get
    from arange().  You do not need a loop in the code does not change here
    because 2 * ndarray is another ndarray automatically. In this respect,
    numpy is very convenient and behaves like R.
    """
    ...
```

The function in math notation is:

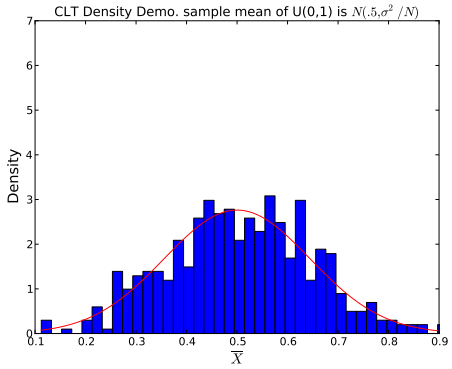
$$P(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/2\sigma^2}$$

5. Then, plot the theoretical normal distribution on top of the histogram and set the axes so that we can use the same range throughout the next series of tests to see how the distribution changes. Note that the usual normal density function provided above expects the **standard deviation not the variance** and so we need to pass `normpdf()` the square root of the expected variance.

```
# plot real normal curve N(0.5, sigma^2/n)
x = np.arange(min(X_), max(X_), 0.01)
y = normpdf(x, 0.5, FILL THIS IN))
```

```
plt.axis([.1,.9,0,7])
plt.plot(x,y, color='red')
```

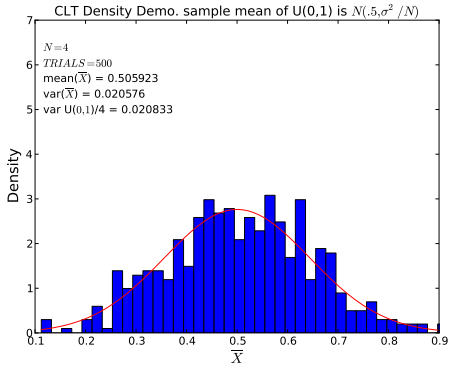
6. Run it. The resulting graph should look like this



7. Now, display some important parameters in the graph using `text()`. You will need to do that `fig.add_subplot(111)` thing again early in your script. The text in between the $ symbols is latex and lets us display nice math symbols (e.g., the title), although I'm not doing much with it here.

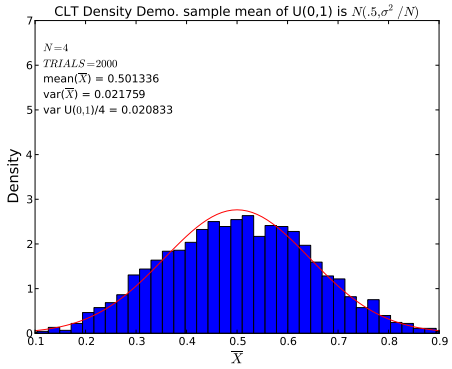
```
plt.text(.02,.9, '$N = %d$' % N, transform = ax.transAxes)
plt.text(.02,.85,'$TRIALS = %d$' % TRIALS, transform = ax.transAxes)
plt.text(.02,.8, 'mean($\\overline{X}$) = %f' % np.mean(X_), transform = ax.transAxes)
plt.text(.02,.75,'var($\\overline{X}$) = %f' % np.var(X_), transform = ax.transAxes)
plt.text(.02,.7, 'var U($0,1$)/%d = %f' % (N,varunif(0,1)/N), transform = ax.transAxes)

plt.title("CLT Density Demo. sample mean of U(0,1) is $N(.5, \sigma^2/N)$")
plt.xlabel("$\\overline{X}$", fontsize=16)
plt.ylabel("Density", fontsize=16)
```

8. Run it. The resulting graph should look like this

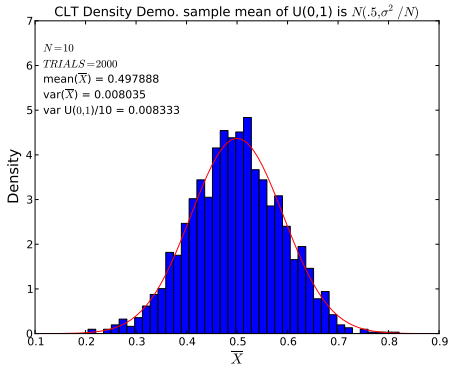


Notice how the mean is close to the expected 0.5 and that the variance of the sample mean is close to the theoretical variance.

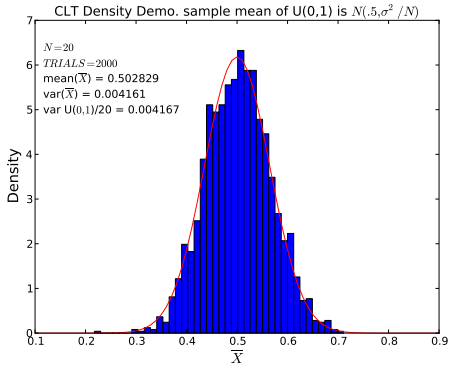
9. Increasing the number of trials two 2000 shows much higher resolution but does not change the variance/tightness of the distribution at all. Run it and see the following:



10. Now, if we increase the sample size to $N = 10$, we get a much tighter variance on the mean of $\overline{X}$. Run it:



11. Increasing to 20 we get:



CLT applied to exponential random variables

Now let's look at how the central limit theorem still gives us a normal distribution even when we pull random variables from a skewed distribution like the exponential. Create and edit a new file `clt_exp.py`.

Steps

12. Import the `rexp(lambduh)` function you wrote for the previous lab to get exponential random variables and start out with the following constants:

```
N = 10
TRIALS = 4000
LAMBDUH = 1.5
```

13. Repeat the loop we did above to get the mean of a bunch of samples into $X\_$, but this time from the exponential distribution `rexp(LAMBDUH)` instead of the uniform distribution function. Plot the histogram of $X\_$ as you did before, using a bin size of 40.

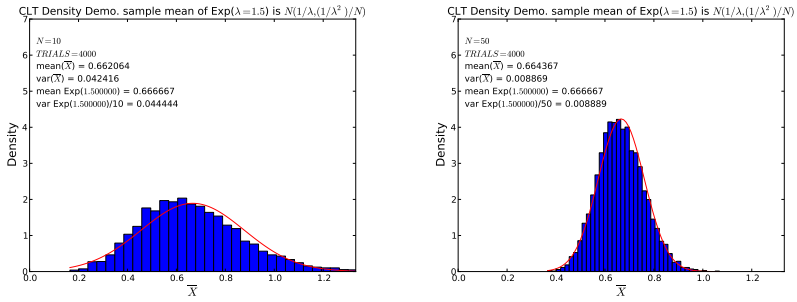
14. Plot the theoretical normal distribution on top using your `normpdf()` (you can cut/paste it into `clt_exp.py`). The mean of the exponential distribution is $\mu = \lambda^{-1}$ and its variance is $\sigma^2 = \lambda^{-2}$.

```
# plot real normal curve N(lambda^-1, sigma^-2 / N)
x = np.arange(min(X_), max(X_), 0.01)
y = normpdf(x, FILL IN MEAN, FILL IN STDDEV)
plt.plot(x,y, color='red')
```

15. Here are the appropriate text annotations:

```
plt.text(.02,.9, '$N = %d$' % N, transform = ax.transAxes)
plt.text(.02,.85, '$TRIALS = %d$' % TRIALS, transform = ax.transAxes)
plt.text(.02,.8,   'mean($\\overline{X}$) = %f' % np.mean(X_), transform = ax.transAxes)
plt.text(.02,.75, 'var($\\overline{X}$) = %f' % np.var(X_), transform = ax.transAxes)
plt.text(.02,.7, 'mean Exp($%f$) = %f' % (LAMBDUH,1/LAMBDUH), transform = ax.transAxes)
plt.text(.02,.65, 'var Exp($%f$)/%d = %f' % (LAMBDUH,N,(1/LAMBDUH**2)/N), transform = ax.transAxes)

plt.title("CLT Density Demo. sample mean of Exp($\lambda=1.5$) is $N(1/\lambda, (1/\lambda^2)/N)$")
plt.xlabel("$\\overline{X}$", fontsize=16)
plt.ylabel("Density", fontsize=16)
plt.axis([0,1.333,0,5])
plt.savefig('clt_exp-'+str(TRIALS)+'-'+str(N)+'.pdf', format="pdf")
```

16. Run it and you should see the following two graphs according to the value of N:



Notice that there is a slight leftward bias in that the normal distribution is a little bit to the right it looks like. This is to be expected. You really need to bump up N before you see it converge to the proper alignment.

17. Play around with other values of lambda and N.

Deliverables

Please submit:

- both `clt_unif.py`, `clt_exp.py` files
- a PDF for $N = 20$, $TRIALS = 2000$ for CLT $U(0, 1)$ demo
- a PDF for $N = 50$, $TRIALS = 4000$, $\lambda = 1.5$ for CLT $Exp(\lambda)$ demo

Generating Normal Random Variables

Discussion

The goal of this lab is to generate normal random variables but using the Central limit theorem instead of the inverse transform or the accept reject method. I'm not recommending this as the most efficient method, but it is a great practical application of the central limit theorem. The hard part about all of this is using the right variance and shifting from $N(0,1)$ to the general $N(\mu, \sigma^2)$. Use filename `rnorm.py`.

Steps

1. First, let's define some constants and the variance of a uniform variable (you should have this from the CLT lab already):

```
N = 100
TRIALS = 4000

def unifvar(a,b):
    return ((b-a)**2)/12.0
```

2. To define a function that generates normal random variables in $N(0,1)$, we rely on the fact that the sample mean, $\overline{X}$ from a sample, X, of uniform distribution values is normal. This gives us as many normal random values as we want, one per sample X. We just have to tweak things so that the mean of the distribution is zero-centered and has variance 1. That shifted and scaled value is what we return from `rnorm01()`:

```
def rnorm01():
    "return a value from N(0,1)"
    ...
```

The process looks like this:

A. Get N uniform random values from $U(0,1)$ into X using your `runif()` function.

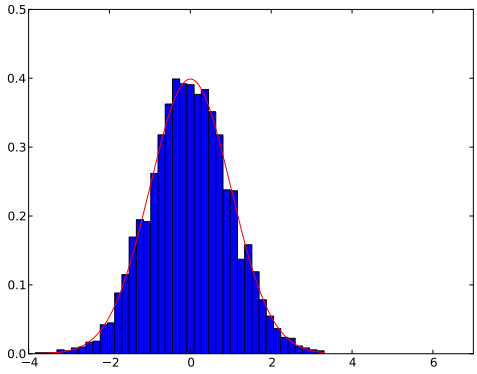
B. Then compute the mean $\overline{X}$.

C. Shift that value so that is zero-centered and call it rv.

D. We know from the CLT lab that the variance of random variable $\overline{X}$ is σ^2/N, where σ^2 is the variance of the underlying distribution $U(0,1)$, but we need the variance to be 1. Scale rv so that it has variance 1. Note that a "standard normal" variable can be created from an arbitrary normal X via $Z = (X - \mu)/\sigma$. Z is effectively a shifted and scaled version of the original. Interestingly, it really just measures how many standard deviations X is from $N(0,1)$.

3. Now, let's fill in the code we need to draw a histogram and the theoretical distribution on top using the `normpdf()` from the CLT labs:

```
# Get X taken from TRIALS trials, plot histogram normalized to density func
X = [rnorm01() for i in range(TRIALS)]
plt.axis([-4, 7, 0, 0.5]) # let's keep the same access across plot for this lab
plt.hist(X, bins=40, normed=1) # histogram should look standard normal

# plot real normal curve
x = np.arange(min(X),max(X), 0.01)
MEAN = 0
VARIANCE = 1
y = normpdf(x, MEAN, math.sqrt(VARIANCE)) # recall our normpdf takes standard deviation as variance
plt.plot(x,y, color='red')
plt.savefig('rnorm01-%d-%d.pdf' % (TRIALS,N), format="pdf")
plt.show()
```



4. Now define a more general method that accepts a desired mean and variance (*not the mean and the standard deviation*):

```
def rnorm(mean, variance):
    "return a value from N(mean,variance)"
    ...
```

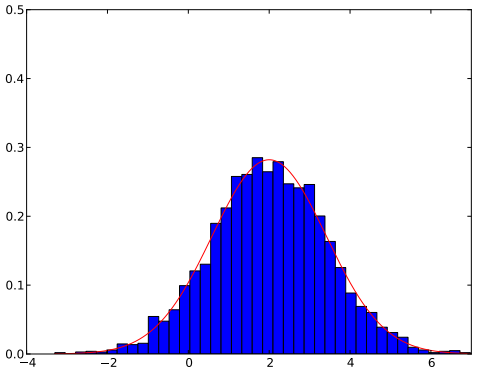
We know how to get a standard normal random variable, Z, as we just defined `rnorm01()`. To get a normal random variable with different mean and variance, we reverse the process we used to get a standard normal via $Z = (X - \mu)/\sigma$. Dust off your high school algebra and solve for X. That tells you how to shift and scale properly: $X = \mu + Z\sigma$.

5. And test as before but this time use $\mu = 2$ and $\sigma^2 = 2$:

```
MEAN = 2.0
VARIANCE = 2.0
# Get X taken from TRIALS trials, plot histogram normalized to density func
X = [rnorm(MEAN,VARIANCE) for i in range(TRIALS)]
plt.hist(X, bins=40, normed=1) # histogram should look gaussian

# plot real normal curve
```

```
x = np.arange(min(X),max(X), 0.01)
y = normpdf(x, MEAN, math.sqrt(VARIANCE))
plt.plot(x,y, color='red')
plt.savefig('rnorm-%d-%d-%d-%d.pdf' % (MEAN,VARIANCE,TRIALS,N), format="pdf")
plt.show()
```

You should get the following graph:



Deliverables

Please submit:

- your `rnorm.py` file
- a PDF of the graphs shown above for $N(0,1)$ and $N(\mu = 2, \sigma^2 = 2)$.

Confidence Intervals for Price of Hostess Twinkies

Goal

The goal of this lab is to learn how to compute an empirical 95% confidence interval for sample means using an awesome technique called *bootstrapping*. As part of this lab, you will also learn to read in a file full of numbers. In this case, we are going to read in the price of Hostess Twinkies, a tasty snack recently returned from the dead, from around the US.

Discussion

A sample mean confidence interval of 95% tells us the range in which most (95% or 1.96σ) of the sample means fall. All we have to do is create a number of samples, X, and compute the means $\overline{X}$. If we do this lots of times (trials) then 95% of the time, we would expect the sample mean to fall within the range of 95% of the samples. We just have to order the $\overline{X}$ values and strip off the lower and top 2.5%. Then, the lowest and highest value in that stripped list represent the boundaries of the confidence interval. Cool, right?

From the central limit theorem, we know that the distribution of $\overline{X}$ is $N(\overline{X}, \sigma^2/n)$ for sample size n (not the number of trials). In this case, though we don't know what the underlying distribution is because we just got a bunch of prices from a file. We could assume that it's normally distributed, but there's no point. The central limit theorem works on any underlying distribution we care about here but we do need the variance. For that, we can use the sample variance as an estimate of the variation in the overall Hostess Twinkies price population.

The question is how do we get lots of trials from an underlying distribution that we cannot identify? By repeated sampling from our single sample *with replacement*. This is called *bootstrapping*, which you could also call *resampling*. The idea is to randomly select N values from our known data set of size N. That gives us one trial. We can then repeatedly compute our test statistic, the mean, on each sample.

To verify that we are doing the right thing, we will draw the theoretical normal distribution expected by the Central limit theorem and then shade in the 95% theoretical confidence interval, which we know is 1.96 standard deviations on either side of the mean: $\mu \pm 1.96\sigma$.

Please do your work in filename `conf.py`.

Steps

1. First, we have to get the data into a file called `prices.txt`:

```
prices = []
f = open("prices.txt")
for line in f:
    v = float(line.strip())
    prices.append(v)
```

When debugging or during development, you can print those numbers out to verify they look okay.

2. Now, we need a function that lets us sample *with replacement* from that raw data set. In other words, we need a function that gets n values at random from a data parameter (a list of numbers). It should allow repeated grabbing of the same value (that's what we call with replacement).

```
def sample(data):
    """
    Return a random sample of data values with replacement.
    The returned array has same length as data.
    """
```

The idea is to get an array of random numbers from $U(0, n)$ for `n=len(data)`. These then are a set of indices into the data array so just loop through this index array grabbing values from data according to the index. For example if you have indexes = [3,9] for a 2-element data array, then return a new array [data[3], data[9]. My solution has two lines in it.

3. Now define TRIALS=20 and perform that many samplings of prices. For each sample, create the sample mean and add it to an X_ list.

4. Sort that list and get the values from TRIALS*0.025..TRIALS*0.975 in X_ and call it `inside`.

5. Print the first and last value of the inside array as that will tell you what the bounds of your 95% confidence interval are

```
print inside[0], inside[-1]
```

You might get something like (there will be a lot of variation):

1.12295362319 1.16113333333

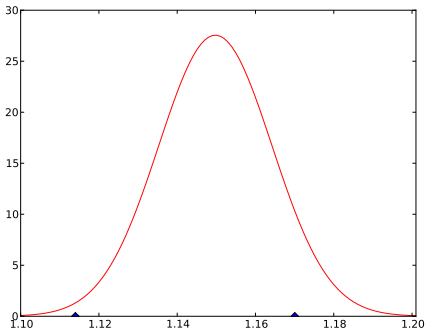
6. Add code to plot diamonds on the graph at those locations:

```
plt.plot(inside[0], 0, 'bD')
plt.plot(inside[-1], 0, 'bD')
```

7. Now plot the normal curve using your amazing new understanding of the central limit theorem. Use the following range and also set the overall graph range:

```
x = np.arange(1.05, 1.25, 0.001)
plt.axis([1.10, 1.201, 0, 30])
```

8. Run it and you should get the following graph:

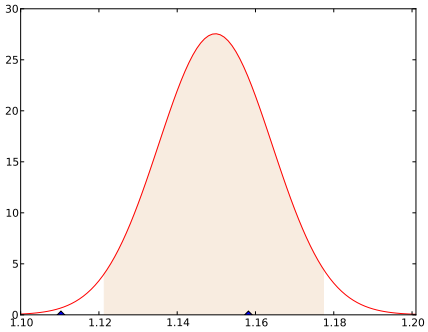


Ok, that's great but we have no idea if this is correct or not. Now, let's go nuts and show lots of stuff on the graph.

9. First, let's shade in the theoretical 95% confidence interval using your `normpdf()`.

```
mean = ...
stddev = ...
# redraw normal but only shade in 95% CI
left  = FILL IN
right = FILL IN

ci_x = np.arange(left, right, 0.001)
ci_y = normpdf(ci_x,mean,stddev)
# shade under (ci_x,ci_y) curve
plt.fill_between(ci_x,ci_y,color="#F8ECE0")
```

Run it again to see how it shades in the graph.

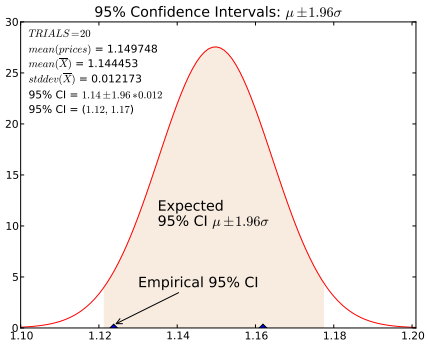


10. Now let's annotate with lots of information. Please read through and figure out what all of that stuff does to draw the nice arrows and so on.

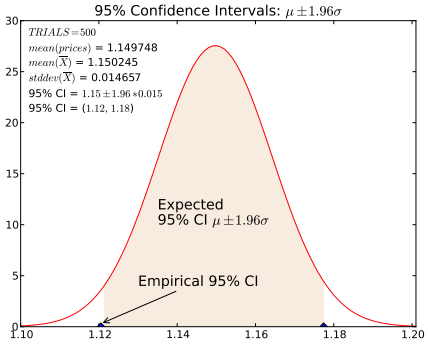
```
plt.text(.02,.95, '$TRIALS = %d$' % TRIALS, transform = ax.transAxes)
plt.text(.02,.9,  '$mean(prices)$ = %f' % np.mean(prices), transform = ax.transAxes)
plt.text(.02,.85, '$mean(\\overline{X})$ = %f' % np.mean(X_), transform = ax.transAxes)
plt.text(.02,.80, '$stddev(\\overline{X})$ = %f' %
    np.std(X_), transform = ax.transAxes)
plt.text(.02,.75, '95%% CI = $%1.2f \\pm 1.96*%1.3f$' %
   (np.mean(X_),np.std(X_)), transform = ax.transAxes)
plt.text(.02,.70, '95%% CI = ($%1.2f,\\ %1.2f$)' %
                  (np.mean(X_)-1.96*np.std(X_),
                   np.mean(X_)+1.96*np.std(X_)),
         transform = ax.transAxes)

plt.text(1.135, 11.5, "Expected", fontsize=16)
plt.text(1.135, 10, "95% CI $\\mu \\pm 1.96\\sigma$", fontsize=16)
plt.title("95% Confidence Intervals: $\\mu \\pm 1.96\\sigma$", fontsize=16)
```

```
ax.annotate("Empirical 95% CI",
            xy=(inside[0], .3),
            xycoords="data",
            xytext=(1.13,4), textcoords='data',
            arrowprops=dict(arrowstyle="->",
                            connectionstyle="arc3"),
            fontsize=16)
```

11. Run it and you should get the following graph:



12. We don't have to increase the number of trials very much before the confidence interval tightens up nicely. Try 500:



Deliverables

Please submit:

- your `conf.py` file with TRIALS=500
- a PDF of the graph with TRIALS=500 shown above.

Is Free Beer Good For Tips?

Goal

The goal of this lab is to test a hypothesis using a variety of techniques: "eyeball" test, t-test, and bootstrapping. Use filename `hyp.py`.

Discussion

Here is a typical statistics question (derived from one by Jeff Hamrick) that we will solve in multiple ways.

Q. *Psychologists studied the size of the tip in a restaurant when the waiter/waitress gave the patron a free beer. Here are tips from 20 patrons, measured in percent of the total bill: 20.8, 18.7, 19.1, 20.6, 21.9, 20.4, 22.8, 21.9, 21.2, 20.3, 21.9, 18.3, 21.0, 20.3, 19.2, 20.2, 21.1, 22.1, 21.0, and 21.7. Does a beer-inspired tip exceed 20 percent? Use a significance level equal to $\alpha = 0.06$.*

Side note: Always pick the significance level before you run your experiment. It is really bad mojo to pick your significance after you know what the p-value is.

Before starting on this, let's interpret that question: It asks whether the mean of the specified sample differs significantly from the usual 20% tip. By "significantly" we refer to the likelihood that the usual population (with mean 20.0) could yield a sample with the observed sample mean. By "usual" we mean our control of approximately: $N(20.0, s^2/n)$ where s is the sample variance of the sample tips and $n = len(tips)$. (We can reasonably assume that tips follow a normal distribution.)

While the population mean is 20.0, the means of any resamples we take will bounce around left and right of 20.0. The question is: does this particular test sample's mean, $m = 20.725$, fall outside of the typical variability of the sample means?

More formally, we would say the following: The **null hypothesis** is that the mean for the specified sample does not differ significantly from $\mu = 20.0$. I think of this as the *control* in my experiment. The **alternate hypothesis** is that the sample mean differs significantly above or below the population mean. Formally,

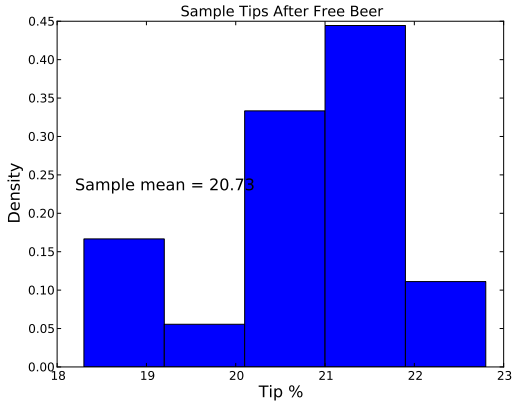
$H_0 : m = 20.0$ (non-free beer situation)
$H_1 : m \neq 20.0$ (free beer situation; two-sided alternative hypothesis)

We could also say that $H_0 : m - \mu = 0$ and $H_1 : |m - \mu| > 0$.

Steps

Eyeballing it

1. First, just draw a histogram of the tips to see what it looks like. For this exercise, create a file called `hyp.py`.

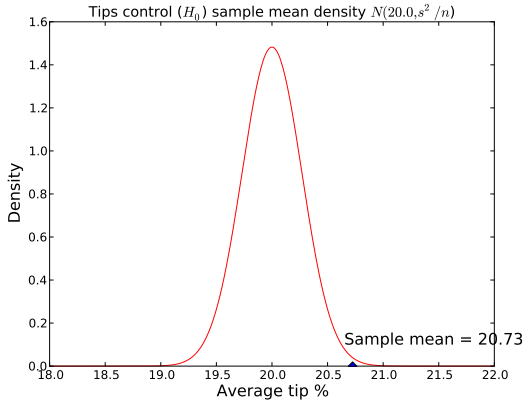


For your convenience, here are the tips in python format:

```
tips = [20.8, 18.7, 19.1, 20.6, 21.9, 20.4, 22.8,
        21.9, 21.2, 20.3, 21.9, 18.3, 21.0, 20.3,
        19.2, 20.2, 21.1, 22.1, 21.0, 21.7]
```

(Use your awesome new skills from previous labs to generate the histogram.) To me, there is a lot of "mass" to the right of the usual 20% tip but my eyeball is not a rigorous significance mechanism.

2. To get a better idea, let's simply plot the distribution of the sample means given our H_0 assumption: $N(20.0, s^2/n)$. We need to use the sample variance s^2 from our test sample because we don't know the variance of the original distribution. It safe to assume that the variance is similar. This is our "control" or the usual tipping distribution: the distribution of the set of average tips per day if H_0, the control, is true.



Looking at that graph, it seems that a sample mean of 20.73 is pretty far in the right tail of a normal curve centered at the control average 20% tip. It looks to be a few standard deviations away from the mean. My gut says that it's pretty likely that giving people a free beer increases tips significantly.

t-test

1. Let's use a *t-test* now to test for significance, just like we would do in statistics class. The t value measures the number of standard deviations a sample mean, m, is away from our presumed population mean μ:

$$t = \frac{m - \mu}{s/\sqrt{N}} \quad \text{(t-value)}$$

It's just the difference between the means scaled to be in units of standard deviations. Write some code to compute the t-value. When computing s, the sample standard deviation, note that the numpy std() function returns a biased estimate of the standard deviation. Use `np.std(tips, ddof=1)` instead of just `std(tips)`. Print out the value of t.

I get $t = 2.69417199392$. That means that m is about 2.7 standard deviations away from μ, which is a very significant departure.

2. To get a p-value, likelihood that we would see such a t value in the nonfree beer situation, look up t in a t-distribution c.d.f. using `1-scipy.stats.t.cdf(t,N-1)`. You should get 0.0071844. Since we need to check both tails, the probability is actually $2x$ that, or, p-value=0.014369 (1.4%). The definition of significance is $\alpha = 0.06$, which means that our sample mean is definitely significant since $1.4\% < 6\%$. There is only a 1.4% chance that the control could generate a value that extreme or beyond.

We must conclude that m differs significantly from $\mu = 20.0$ based upon the significance of $\alpha = 0.06$ and, therefore, we reject H_0 in favor of H_1. Giving out free beers is extremely likely to have increased the average tip in that experiment.

Boostrapping for empirical hypothesis testing

Ok, now, let's use bootstrapping to estimate a *p-value*. A p-value for some point statistic or value is the probability that the control (null hypothesis H_0) could generate that statistic or value. In our case, a p-value can tell us the likelihood that a normal distribution centered around $\mu = 20.0$ with $s^2 = var(tips)$ could generate a sample mean of 20.725. (We approximate the population variance with our sample variance.) *Note and we are sampling from $N(\mu = 20.0, s^2)$ to conjure up samples from the control situation. We are not resampling from the tips list as we are trying to see how the observed sample mean, 20.725, fits within the control distribution not the test distribution. We are also not generating samples from the distribution of a mean random variable, $N(\mu = 20.0, s^2/n)$.*

1. Bootstrap TRIALS=5000 samples of size $n = len(tips)$ from $N(\mu, s^2)$. It's very important that we use the same sample size as $len(tips)$ so we are comparing the same thing. Compute the mean of each sample, X, an add to $\overline{X}$ as you generate samples from the normal distribution.

2. Compute how many means in $\overline{X}$ are greater than or equal to mean(tips):

```
greater = np.sum(X_ > np.mean(tips))
```

or

```
greater = sum([x>np.mean(tips) for x in X_]) # the number of true values
```

3. The (one-sided) p-value is just the ratio of values above the observed mean, mean(tips), to the number of trials. Double that because we're doing a two-sided test. With 5000 trials, I see just 13 values greater than $m = 20.725$. That gives us a p-value of 2*13/5000 = 0.0052 or .52%. That means that, empirically, we find that there is an extremely small probability that the control could generate an extreme value like $m = 20.725$. Certainly the likelihood is less than the required 6% significant value.

Note: we would expect the empirical p-value (.52%) and the p-value derived from the t-test (1.4%) to be very close to each other when the number of trials is large with bootstrapping. Our resident statistician,

Jeff Hamrick, explains that the difference is not a problem with our bootstrapping solution and is ok.

"*A student t distribution with dof=19, is pretty close to a normal. But the differences are most greatly felt in the tails, and we're in the tails (rejection H_0), thus casting a little bit of sketchiness or your choice to draw the simulated raw data from a normal random variable. If we were performing this exact same operation on a data set with reasonably large size (say, 40 or 50 or 75) the differences would still exist but be even more minute.*"

Again, we easily reject the control and conclude that giving out free beers increases tips.

Deliverables

Please submit:

- `hyp.py`
- A text file that gives your t-value, and p-value from the t-test. Also give your empirical p-value from bootstrapping with TRIALS=5000.

Part III

Optimization and Prediction

Iterative Optimization Via Gradient Descent

Goal

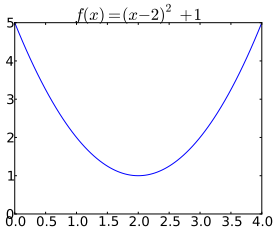
The goal of this task is to increase your programming skill by solving an iterative computation problem with nontrivial iteration and termination conditions: *gradient descent function minimization*.

Discussion

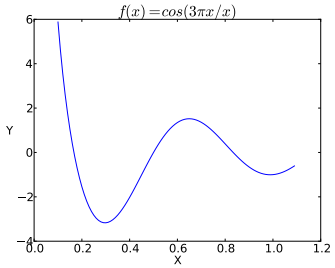
Finding x that minimizes function $f(x)$ (usually over some range) is an incredibly important operation as we use it to minimize risk and, for machine learning, to learn the parameters of our classifiers or predictors. Generally x will be a vector but we will assume x is a scalar for this task. If we know that it is convex, there is a unique solution and we can simply set the derivative equal to zero and solve for x:

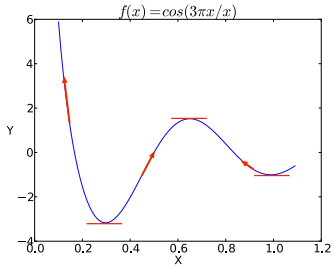
$$f'(x) = 0 \qquad \text{(Analytic solution to convex optimization)}$$

For example, the function $f(x) = (x-2)^2 + 1$ has $f'(x) = 2x - 4$ whose zero is $x = 2$.



We prefer to find the *global minimum* but generally have to be satisfied with a *local minimum*, which we hope is close to the global minimum. A decent approach to finding the global minimum is to find a number of local minima via random starting x_0 and just choose the minimum local minimum discovered. For example, the function $f(x) = cos(3\pi x)/x$ has two minima in $[0, 1.1]$, with one obvious global minimum:





If the function has lots of zeros or is very complicated, there may be no easy analytic solution. There are many approaches to finding function minima iteratively (i.e., non-analytically), but we will use a well-known technique called *gradient descent* or *method of steepest descent*.

Gradient descent

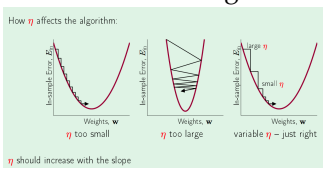
This technique can be used to train everything from *linear regression* models to *neural networks*. Gradient descent requires a starting position, x_0, the function, $f(x)$, and its derivative $f'(x)$. Recall that the derivative is just the slope of a function at a particular point. In other words, as x increases does y go up or down, and by how much? E.g., the derivative of x^2 is $2x$, which gives us a positive slope when $x > 0$ and a negative slope when $x < 0$. Gradient descent uses the derivative to iteratively pick a new value of x that gets us closer and closer to the minimum $f(x)$. Negating the derivative tells us the direction of the nearest minimum. For example the graph to the right above shows a number of vectors representing derivatives at particular points. Note that the derivative is zero, i.e. flat, at the minima (same is true for maxima). The recurrence relation for updating our estimate of x that minimizes $f(x)$ is then just:

$$x_{i+1} = x_i - \eta f'(x_i)$$

where η is called the *learning rate*, which we'll discuss below. The $\eta f'(x_{i-1})$ term represents the size of the step we take towards the minimum. The basic algorithm is:

1. Pick an initial x_0, let $x = x_0$
2. Let $x_{i+1} = x_i - \eta f'(x_i)$ until $f'(x_i) = 0$

Stopping the algorithm is problematic when dealing with the finite precision of computers. Specifically, no two floating-point numbers are ever equal really. So $f'(x) = 0$ is always false. Usually we do something like $abs(x_{i+1} - x_i) < precision$ or when $abs(f(x_{i+1}) - f(x_i)) < precision$ where precision is some very small number like 0.0000001. The steps we take are scaled by the learning rate η. Yaser S. Abu-Mostafa has some great slides and videos that you should check out. Here is his description on slide 21 of how the learning rate can affect convergence:



The domain of x also affects the learning rate magnitude. This is all a very complicated finicky business. You can start out with a low learning rate and crank it up to see if you still converge without oscillating around the minimum. An excellent description of gradient descent and other minimization techniques can be found in Numerical Recipes.

Approximating derivatives with finite differences

Sometimes, however, the derivative is hard, expensive, or impossible to compute. To get around this and to reduce the input requirements, we can approximate the derivative in the neighborhood of a particular x value. That way we can optimize any reasonably well behaved function (left and right continuity would be nice). To approximate the derivative, we can take several approaches. The simplest involves a comparison. Since we really just need a direction, all we have to do is compare the current $f(x_i)$ with values a small step, h, away in either direction: $f(x_i - h)$ and $f(x_i + h)$. If $f(x_i - h) < f(x_i)$, we can move x_{i+1} to the left of x_i. $f(x_i + h) < f(x_i)$, we move x_{i+1} to the right. Which is really just the same formula but replacing the derivative with the finite (forward) difference:

$$x_{i+1} = x_i - \eta \frac{f(x_i + h) - f(x_i)}{h} \text{ where } f'(x) \approx \frac{f(x_i + h) - f(x_i)}{h}$$

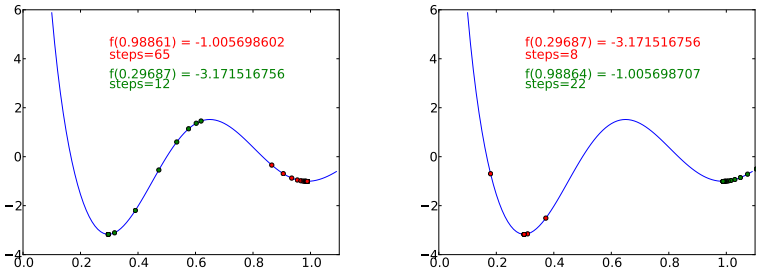
To simplify things, we can roll the step size h into the learning rate η constant as we are going to pick that anyway.

$$x_{i+1} = x_i - \eta(f(x_i + h) - f(x_i))$$

Something important to notice is that the step size is bigger when the slope is bigger and gets smaller as we approach the minimum (since the region is flatter). This helps us converge more quickly. Abu-Mostafa indicates in his slides that η should increase with the slope whereas we are keeping it fixed and allowing the finite difference to increase the step size. We are not normalizing the derivative/difference to a unit vector like he does (see his slides).

Your task

You will use gradient descent to minimize $f(x) = cos(3\pi x)/x$. To increase chances of finding the global minimum, pick **two** random locations in the range [0.1,1.2] and perform gradient descent with both of them. As part of your final submission, you must provide a plot of $f(x)$ with traces that indicate the steps taken by your gradient descent; use a different color for each descent. Here are two sample descents where the x and $f(x)$ values are displayed as well as the minimum of those two:



To create the dots you just need to add the x values two an array as you search for the minimum and then plot the x and $f(x)$ values with red or green dots:

```
tracey = [f(x) for x in tracex]
plt.plot(tracex, tracey, 'ro')
```

Please show the information as I have shown in the graphs to make it easier to compare results and for me to grade.

Define a function called `minimize` that takes the indicated parameters and returns a trace of all x values visited including the initial guess:

```
def minimize(f, x0, eta, h, precision):
    tracex = []
    tracex.append(x0)  # add starting position
    ...
    return tracex
```

As an example, I call that function like this:

```
tracex = optimize(f, x0, ETA, STEP, PRECISION)
```

for an appropriate `f()` definition per the above cosine function. Note that Python allows us to pass a function just like any other object. For parameter `f`, we can call that function from within `optimize()` with the usual syntax `f()`.

So that we all have the same graph structure, please use the following code to plot the cosine function:

```
import matplotlib.pyplot as plt

graphx = np.arange(.1,1.1,0.01)
graphy = f(graphx)
plt.plot(graphx,graphy)
plt.axis([0,1.1,-4,6])
```

You will have to pick an appropriate step value h to get a decent approximation of the derivative through finite differences that is large enough to avoid faulty results from lack of precision (subtracting two floating-point numbers in the computer results in a number with much less precision than the original numbers). What you want that number to be small enough so that your algorithm does not oscillate around the minimum. If the number is too big it will compute a finite difference by leaping across the minimum to the other wall of the function. You must pick a learning rate η that allows you to go as fast as you can but not so fast that it overruns the minimum back and forth. When I crank up my learning rate too far, I also see the algorithm oscillate:

```
...
f(0.491296576641) = -0.166774773584 , delta = 2.05763033375622805821
f(0.296744439739) = -3.171512867583 , delta = -3.00473809399913660556
f(0.297092626880) = -3.171512816769 , delta = 0.00000005081414267138
...
```

To help you understand what your program is doing, print out x, $f(x)$, and any other value you think is helpful to see how your program explores the curve.

To give you some idea about how fast your minimization function should converge, over many trials it my implementation seems to converge in less than 70 steps.

Deliverables

Please submit the following via canvas:

- A PDF of your graph with two visible traces (sometimes they will overlap and you can't see one of them). It doesn't matter if they both are converging to the same minimum or two different ones. The graph should include the text I have on mine for x, $f(x)$, number of steps, etc...
- Your `descent.py` code

Predicting Murder Rates With Gradient Descent

Goal

The goal of this task is to extend the techniques you learned in the one-dimensional gradient descent task to a two-dimensional domain space, solving a *linear regression problem*. This problem is also known as *curve fitting*. You'll learn how to compute with vectors instead of scalars.

Discussion

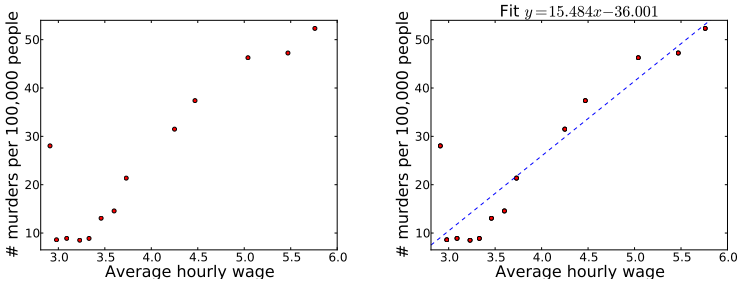
Problem statement

Given training data (x_i, y_i) for $i = 1..n$ samples with dependent variable y_i, we would like to predict y for some x not in our training set. If we assume there is a linear relationship between x and y, then we can draw a line through the data and predict future values with that line function. To do that, we need to compute the two parameters of our model: a slope and a y intercept.

For example, if we compare the number of murders per 100,000 people in Detroit to the average hourly wage, our eyeballs easily detect a correlation. Here is the data suitable to copy and paste into Python:

```
HOURLY_WAGE = [2.98, 3.09, 3.23, 3.33, 3.46, 3.6, 3.73, 2.91, 4.25, 4.47, 5.04, 5.47, 5.76]
MURDERS = [8.6, 8.9, 8.52, 8.89, 13.07, 14.57, 21.36, 28.03, 31.49, 37.39, 46.26, 47.24, 52.33]
```

and here is a scatter plot and best fit line as determined by numpy (using `np.polyfit(HOURLY_WAGE, MURDERS,1)`).



This might be a good point to remind everyone that correlation does not equal causation. I hardly think that paying people more makes them murderous, although I could see the opposite. ;) Correlation is a *necessary* but not *sufficient* condition for causation. When you find a correlation, that gives you a candidate to check for cause-and-effect.

Best fit line that minimizes squared error

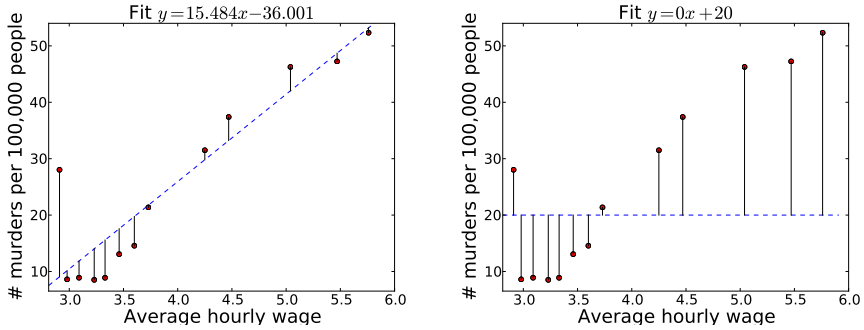
Recall the formula for a line from high school: $y = mx + b$. We normally rewrite that using elements of vector $\vec{B}$ in preparation for describing it with vector notation from linear algebra. For simplicity, though, we'll stick with scalar coefficients for now:

$$y = b_1 + b_2 x$$

The "best line" is one that minimizes some cost function that compares the known y values at x to the predicted y of the linear model that we conjure up using parameters b_1, b_2. A good measure is the *sum of squared residuals*. The cost function adds up all of these squared errors to tell us how good of a fit our linear model is:

$$Cost(B) = \sum_{i=1}^{n} (\underbrace{b_1 + b_2 x_i}_{\text{linear model}} - \overbrace{y_i}^{\text{true value}})^2$$

As we change the linear model parameters, the cost function will change. The following graph shows the residuals which are squared and summed to get the overall cost for that line.



The costs are 533.82 for the left and 3563.50 for the right.

The good news is that we know the cost function is a quadratic, which is convex and has an exact solution. All we have to do is figure out where the partial derivatives of the cost function are both zero; i.e., where the cost function flattens out (at the bottom).

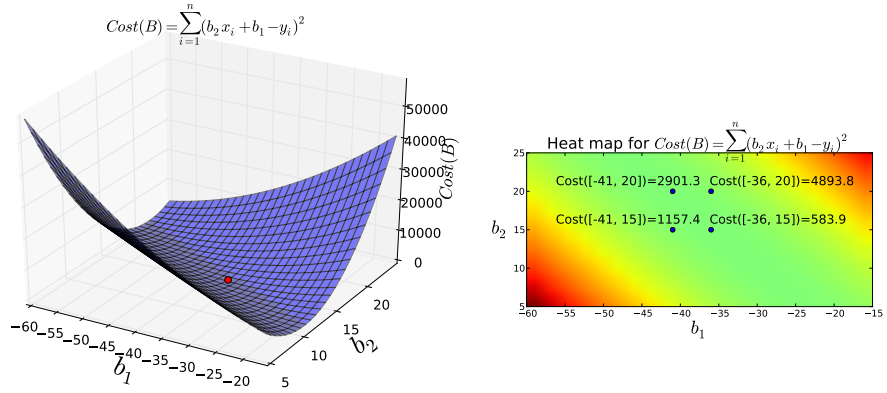
$$\nabla Cost(B) = 0 \qquad \text{(Analytic solution to convex optimization)}$$

For our purposes, though, we'll reuse gradient descent to minimize the cost function.

To show our prediction model in action, we can ask how many murders would there be in Detroit if the average salary were $4.7? (Obviously, these wages are from 30 years ago.) To make a prediction, all we have to do is plug $x = 4.7$ into $y = -36.001 + 14.484x$, which gives us 32.074 murders.

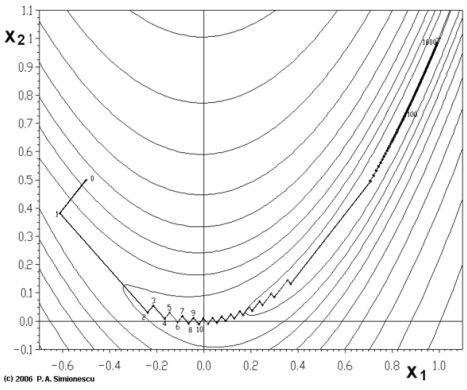
Gradient descent in 3D

Before trying to minimize the cost function, it's helpful to study what the surface looks like in three dimensions, as shown in the following two graphs.



What surprised me is that changes to the slope of the linear model, coefficient b_2, away from the optimal $b_2 = 15.484$ cost much more than tweaks to the y intercept, b_1. Regardless, the surface is convex and a unique solution exists.

Unfortunately, based upon the deep trough that grows slowly along the diagonal of (b_1, b_2), gradient descent takes a while to converge on the minimum. We will examine the path of gradient descent for a few initial starting point. Wikipedia says that the Rosenbrock function is a pathological case for traditional gradient descent and it looks pretty similar to our surface with its shallow valley:



The recurrence relation for updating our estimate of $\vec{B} = [b_1, b_2]$ that minimizes $Cost(\vec{B})$ is the same as our previous task but with a vector instead of a scalar x_i.

$$\vec{B}_{i+1} = \vec{B}_i - \eta \nabla Cost(\vec{B}_i)$$

where we will approximate vectors of partial derivatives with partial finite differences defined generically as:

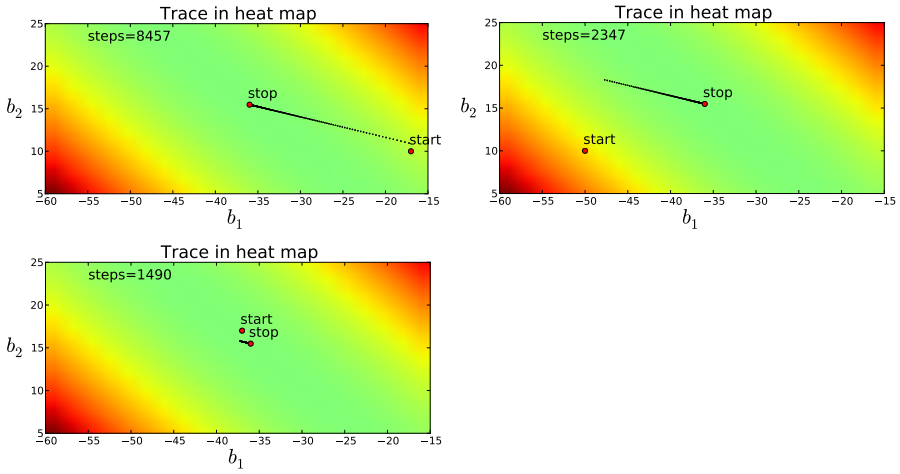
$$\nabla F(\vec{X}) = \begin{bmatrix} \frac{\delta}{x_1} F(\vec{X}) \\ \frac{\delta}{x_2} F(\vec{X}) \end{bmatrix} \approx \begin{bmatrix} \frac{F(\begin{bmatrix} x_1+h \\ x_2 \end{bmatrix}) - F(\vec{X})}{h} \\ \frac{F(\begin{bmatrix} x_1 \\ x_2+h \end{bmatrix}) - F(\vec{X})}{h} \end{bmatrix}$$

In our case, we will compute the components of a finite difference vector C' ignoring the division by the step h as we want to accelerate when the difference is large.

The algorithm looks like:

1. Pick an initial B_0
2. let $B = B_0$
3. $C' = \begin{bmatrix} Cost(\begin{bmatrix} b_1+h \\ b_2 \end{bmatrix}) - Cost(B_i) \\ Cost(\begin{bmatrix} b_1 \\ b_2+h \end{bmatrix}) - Cost(B_i) \end{bmatrix}$
4. Let $B_{i+1} = B_i - \eta C'$
5. goto step 3 until $Cost(B_{i+1}) - Cost(B_i) = 0$

Using a low learning rate, my solution takes 4843 steps starting from coordinate (-45,10) using a very small step size, which gives me a fairly decent approximation of the minimum: [-36.00066933 15.48414587] compared to the analytic solution [-36.000625 15.484375]. Starting from about the same distance away in the shallow valley at (-45,25), my solution takes 5142 steps. Cutting my step size by 5x, takes 30463 steps but gives a slightly more precise result [-36.00063497 15.48432944]. Starting at (-45,25) takes 31958 steps. Surprisingly, when I start very close to the minimum at (-36,15), my solution takes 47350 steps and does not exactly give a more accurate result. "*Your mileage may vary.*"



When I crank up the learning rate and leave the very small step size, my solution converges must faster and with the same accuracy. For example, with 10 times the learning rate as before, (-45,25) converges in 3193 steps instead of 31958 steps.

Your task

You will use gradient descent to solve the linear regression problem above, using the same data. As part of your final submission, you must provide heat maps with traces that indicate the steps taken by your gradient descent as I have shown above. Choose two random starting B_0 vectors to produce your heat

maps. I used pylab.imshow() to draw the heat map whose b_1 are the y intercepts, b_2 coordinates are the murders, and heat value is the cost of x, y. It took me a while to figure out all of the crazy methods to draw the heat maps. Please show the information as I have shown in the graphs to make it easier to compare results and for me to grade.

You must tweak the step size and other parameters so that your results agree with the first four decimal points of the analytic solution [-36.000625000000007, 15.484375].

Define a function called `minimize` that takes the indicated parameters and returns the minimum B parameters of your linear model, the number of steps, and the trace array of intermediate B_i values.

```
def minimize(f, B0, eta, h, precision):
    trace = []
    B = B0
    steps = 0
    while True:
        steps += 1
        if steps % 10 == 0: # only capture every 10th value
            trace.append(B)
        ...
    ...
    return (B, steps, trace)
```

As an example, I call that function like this:

```
def f(B): # a helper function that simply adds in the default arguments of the data
    return Cost(B, HOURLY_WAGE, MURDERS)
# or, if you are one of the cool kids:
f = lambda B : Cost(B, HOURLY_WAGE, MURDERS)
(m,steps,trace) = minimize(f, B0, LEARNING_RATE, h, PRECISION)
```

I plot the trace using:

```
plot(p[0], p[1], "ko", markersize=1)
```

You will have to pick an appropriate step value h to get a decent approximation of the derivative through finite differences that is large enough to avoid faulty results from lack of precision (subtracting two floating-point numbers in the computer results in a number with much less precision than the original numbers). You want that number to be small enough that your algorithm does not oscillate around the minimum. If the number is too big it will compute a finite difference by leaping across the minimum to the other wall of the function. You must pick a learning rate η that allows you to go as fast as you can but not so fast that it overruns the minimum back and forth. When I crank up my learning rate too far, I also see the algorithm go off into the weeds and stops with a minimum of [-4.86000929e+10 -2.85744570e+12].

Resources

There is a lot of material out there on the web that can be helpful.

- Finite difference at Wikipedia
- Stochastic Gradient Descent Tricks

- Numerical recipes (See Chap 10 on minimization of functions)
- Single verbal minimization in line searches
- Andrew Ng's CS229 Lecture notes
- Data analysis with Python

Deliverables

Please submit the following via canvas:

- A PDF of your graph with two visible traces on two heat maps or the same heat map if the traces are clear. The graph should include the start, stop location and the number of steps as I have done on mine. As part of your PDF, please indicate the B parameters you compute with your minimize function.
- Your `regression_descent.py` code.

Part IV

Text Analysis

Summarizing Reuters Articles with TFIDF

Goal

The goal of this task is to learn a core technique used in text analysis called *TFIDF* or *term frequency, inverse document frequency*. We will use what is called a *bag-of-words* representation where the order of words in a document don't matter—we care only about the words and how often they are present. A word's TFIDF value is often used as a feature for document clustering or classification. We will use it simply as a summarization tool for document. The more a term helps to distinguish its enclosing document from other documents, the higher its TFIDF score.

This task is also an opportunity learn how to organize Python code as a set of functions rather than an unstructured script (blob) with a bunch of global variables. You will also learn how to translate some simple algorithms written in pseudocode to Python code. As a practical matter, you will learn how to process XML files in Python.

Discussion

One way to summarize a text document is to list, say, the top 25 words that seem most important. That could also be used to compare documents to see if they're talking about the same thing. For example, I had to solve a problem 15 years ago to reduce noise in the forums of a Java developer's website. Users were posting stupid posts about movies and were also putting database questions in the forum on GUIs. The goal was to detect non-Java posts and also to detect misplaced posts. What does it mean to "talk about Java"? How do I know when someone is talking about databases versus GUIs? My solution was to identify the words important to Java as a whole ("Java-speak"), database, and GUI posts. Any posts that did not have words important to Java, were tossed out as irrelevant after giving them a mild smack on the snout. Similarly, posts without words relevant to databases were compared to vocabularies associated with other topics to see if another forum would be more appropriate. To make this work, I needed a precise definition of "important words." As I did for that project, you will use a classic text analysis technique called TFIDF in this project.

Certainly a word is important to a document if it's used a lot, but that would also include words like "the" so we need to discount words used frequently among our *corpus* (set of documents). So, we boost words used frequently in a document but attenuate if that word is used in a lot of documents. For more on this topic, see Introduction to Information Retrieval.

The *term frequency* is just the term count within a document divided by the number of words in that document (some people use "frequency" to mean "count" but that is an affront to the gods):

$$tf(t,d) = \frac{count(t), t \in d}{|d|} \qquad \text{(Term frequency of term } t\text{, document } d\text{)}$$

A term's *document frequency* is the count of documents containing that term divided by the total number of documents:

$$df(t, N) = \frac{|\{d_i : t \in d_i,\ i = 1..N\}|}{N} \qquad \text{(Document frequency of } t \text{ in } N \text{ documents)}$$

In order to reduce the TFIDF score for terms with high document frequencies, we need the document frequency in the denominator:

$$tfidf(t, d, N) = \frac{tf(t, d)}{df(t, N)} \qquad \text{(First approximation to TFIDF)}$$

This formula is meaningful but gives a poor weight because the document frequency tends to overwhelm the term frequency fractions in the numerator so we take the log of the denominator first. The log also prevents division by 0 errors when a term does not exist in a corpus (e.g., in search applications where we pass random terms). Here's the formula slightly rewritten as it is normally shown:

$$tfidf(t, d, N) = tf(t, d) \times log(\frac{1}{df(t, N)}) \qquad \text{(TFIDF with attenuated document frequency)}$$

To summarize a document, we can order its terms by *tfidf* in reverse order and look at the top 20 words, for example. To get the lexicon of a topic like databases, we can collect a known set of database posts into a single document and compute the tfidf in association with the aggregated documents of the other topics. Any word below a certain threshold, that we find by eyeballing it, is considered not relevant to that particular topic.

For example, here is a set of terms and the associated *tfidf* computed from a sample Reuters article. It's clear that it's talking about Nielsen ratings for news programs, without even looking at the original article.

Term	*tfidf*
rating	0.12332931962551781
fox	0.11911646171233138
nbc	0.11911646171233138
homes	0.11408838230482544
cbs	0.0794109744748876
audiences	0.0794109744748876
neilsen	0.0794109744748876
evening	0.06678324701893232
abc	0.06678324701893232
watching	0.0634765565309808

To compute TFIDF, we need an overall index that maps term t to document frequency $df(t, N)$ for all t in all N documents and an index that maps document d to another index that maps each $t \in d$ to $tf(t, d)$. From that, we can compute all of the TFIDF scores. That is what you will do for this project, as described in the next section.

Your task

To implement this project, you have six key functions to implement. For the first four, you need to translate the following pseudocode to Python in a file called `tfidf.py`.

Function: *words*(document d)
Input: Document d
Result: non-unique list of words $wordlist$
  $wordlist$ = Split d into words, removing numbers and punctuation
  Normalize $w \in wordlist$ to lowercase
  Strip out $w \in wordlist$ smaller than 3 letters
  return *wordlist*

Function: *create_indexes*(list of $files$)
Input: List of filenames $files$
Result: Map document name to Counter object mapping term to frequency map *tf_map*
Result: Counter object mapping term to document count df
  df = {}; *tf_map* = {}
  foreach f *in files* **do**
    $d = get\_text(f)$
    $words = words(d)$
    $n = len(words)$
    tf = {}
    foreach $t \in words$ **do** $tf[t] = count(t)/n$
    $tf\_map[f] = tf$
    $df[t]$ += 1 # not currently a frequency; it's a count
  end
  return *(tf_map,df)*

Function: *doc_tfidf*(tf, df, N)
Input: Term to frequency map tf
Input: Term to document count map df
Input: Number of documents N
Result: Map of each term in doc (tf) to TFIDF score
  $tfidf$ = {}
  foreach $t \in tf$ **do**
    $df_t = df[t]/N$
    $idf_t = 1/df_t$
    $tfidf[t] = tf[t] \times log(idf_t)$
  end
  return $tfidf$

Function: *create_tfidf_map*($files$)
Input: List of xml filenames $files$
Result: Map from file to map of term to TFIDF score
  $(tf\_map, df) = create\_indexes(files)$
  tfidf_map = {}
  foreach $f \in files$ **do**
    $tfidf = doc_t fidf(tf\_map[f], df)$
    $tfidf\_map[f] = tfidf$
  end
  return *tfidf_map*

You must also provide a function called `filelist(pathspec)` that returns a list of all files that match `pathspec` and that *have non-zero file sizes*:

```
def filelist(pathspec):
    ...
    return files
```

For example, I might pass in `../data/reuters-vol1-disk1/*.xml`. Naturally, if this doesn't work, then the rest of the code will not work as it won't get the proper data. That function should only consider the files in the specified directory, not subdirectories. You will probably want to use Python function `glob.glob()`.

To process XML files, you should also create the following function:

```
def get_text(fileName):
    """
    read an xml file and return the text from <title> and <text>
    """
    ...
    return text
```

As part of your development work you will use lots of maps that look like {dog: 36, cat: 19, ...}. Those integers, such as term counts, are easy to compute yourself but Python has an object that is effectively a histogram called `Counter`. For example, if you give it a list of words, it will return an object that maps terms to their count. When you print them out, it will do so in reverse order of term count, which is very handy for testing. Further, the unit tests I provide expect Counter objects.

For what it's worth, my implementation is just 60 lines including the `import` statements. This is not a huge project but it is tricky when messing around with all of these maps of maps and lists of things. Start by understanding the problem and working a few TFIDF examples manually. Then, build a simple functions and test them individually before moving on to the more complex functions. For example, you should start by building `filelist()` and then probably `get_text()`. My typical strategies to design from the top down and test from the bottom up.

XML Input

As part of this project, I will provide you with a set of Reuters news articles in XML format, which will be the input to your program. From it, you will create the appropriate indexes and I will test those values against what I computed with my solution.

The format of the files doesn't matter much except that you need to pull out the `title` and `text` tags. The `p` paragraph tags inside `text` need to be collected. All of this text is what you will return from `get_text()`.

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<newsitem itemid="131701" ...>
<title>German consumer confidence rises in Aug/Sept</title>
<text>
<p>German consumer confidence rose...</p>
<p>The Icon index, which...</p>
</text>
...
</newsitem>
```

The collection of Reuters articles is considered proprietary to Reuters and, to get access to the data, the faculty had to promise Reuters the data would not be made available on a public website or given to anyone else. Please treat this data with care, do not posted to github, etc.

Testing

In computer science, programmers recognize two primary kinds of tests: *unit tests* and *functional tests*. A unit test is really just testing a function or a few functions whereas functional tests test the overall functionality of the program. In file `test_tfidf.py`, I have provided a set of unit and functional tests that you can use for basic sanity checking of your TFIDF project. I would typically test your projects with a different set of unit tests but, in this case, we will define success as getting the correct answers for the large corpus of Reuters articles that I will provide to you.

To make the unit tests work, make sure that you install py.test, which is usually just a matter of:

```
easy_install -U pytest
```

I will test your code using the following command line (with your `tfidf.py` is in the same directory):

```
$ python -m pytest test_tfidf.py
============================= test session starts ==============================
platform darwin -- Python 2.7.7 -- py-1.4.20 -- pytest-2.5.2
collected 5 items

test_tfidf.py .....

=========================== 5 passed in 0.04 seconds ===========================
```

If you don't see all tests passing, and there is a problem at a basic level with your software.

Note that the test file imports your file with:

```
from tfidf import *
```

If you name it incorrectly, the program won't work.

To test the entire corpus of Reuters articles, I will run your program as follows, potentially with a different path specification.

```
python test_corpus.py '../data/reuters-vol1-disk1/*.xml'
```

Note that the quotations around the path specification are required to prevent the command line from expanding `*.xml`. You want that path specification to go in as a single argument, not a list of files. The core of `test_corpus.py` is:

```
(file_to_histo, word_to_numdocs) = create_indexes(files)
for f in files:
    pairs = doc_tfidf(file_to_histo[f], word_to_numdocs, N)
    # convert map to a Counter object so we can use most_common()
    term_pair = Counter(tfmap).most_common(1)[0]
    print os.path.basename(f), "(\%s, \%1.4f)" \% (term_pair[0], term_pair[1])
```

The output, which I have provided in file `corpus_output.txt.7z`, starts with:

```
81880 files
131674newsML.xml (ewe, 0.3792)
131675newsML.xml (telefonica, 0.1701)
131676newsML.xml (ingenico, 0.1645)
131677newsML.xml (lisbon, 0.1397)
131678newsML.xml (warrant, 0.0490)
131679newsML.xml (billion, 0.1192)
13167newsML.xml (cents, 0.2657)
131680newsML.xml (tightness, 0.0628)
131681newsML.xml (telefonica, 0.1767)
131682newsML.xml (rate, 0.1994)
131683newsML.xml (asset, 0.0921)
131684newsML.xml (india, 0.0953)
131685newsML.xml (crowns, 0.0952)
131686newsML.xml (crowns, 0.1355)
131687newsML.xml (ireland, 0.1857)
...
```

My implementation takes about 1 minute 30 seconds to compute TFIDF scores for 81,880 XML files loaded from an SSD on a fast machine. Loading those files takes just 5 seconds.

Resources

I provide for you the following files:

- `test_tfidf.py`: some simple tests using py.test.
- `test_corpus.py`: prints out the file, term, and TFIDF score for the highest scoring term in each file.
- `corpus_output.txt.7z`: compressed output from running test_corpus.py
- `reuters-vol1-disk1-subset.7z`: compressed directory full of XML files—the corpus. It is 385M when uncompressed.

Deliverables

Please submit the following file via canvas:

- Your `tfidf.py` file. *I will deduct a full point if your library is not executable exactly in the fashion mentioned in this project; that is, method names and filename must be exactly right. For you PC folks, note that case is significant for file names on unix! All projects must run properly under linux or OS X.*

Extra credit — Search Engine

In this project, we created an index from term to the number of documents that contain that term. If we extend that to be an index from term to the list of documents containing the term, we can get the same results as we did before. The benefit would be that we could create a search engine.

Given a query such as "consumer confidence," we could merge the list of files containing those two terms and display those to a user. It's fast and works great! The only problem is that we might get 1000 documents back and we'd really like to show the most relevant documents first. Using the tf_map index, we can compute a relevance score for a query, relative to a document, by summing the TFIDF scores for each term in the query that is present in the document. The document with the highest two TFIDF scores for "consumer confidence," would be the first document we displayed.

You need to modify the df map from above and then implement the following function.

```
def search(query):
    docs = []
    # find list of documents for each term in query, put in docs
    # compute sum of TFIDF scores for each term in query relative to each document in docs
    # sort documents by reverse score
    return docs
```