# Chapter 4

## The greatest theorem never told

This chapter focuses on an idea that is always bouncing around our minds, but is rarely made explicit outside books devoted to statistics. In fact, we've been using this simple idea in every example thus far.

## The Law of Large Numbers

Let $Z_i$ be $N$ independent samples from some probability distribution. According to *the Law of Large numbers*, so long as the expected value $E[Z]$ is finite, the following holds,

$$\frac{1}{N} \sum_{i=1}^{N} Z_i \to E[Z], \quad N \to \infty.$$

In words:

> The average of a sequence of random variables from the same distribution converges to the expected value of that distribution.

This may seem like a boring result, but it will be the most useful tool you use.

### Intuition

If the above Law is somewhat surprising, it can be made more clear by examining a simple example.

Consider a random variable $Z$ that can take only two values, $c_1$ and $c_2$. Suppose we have a large number of samples of $Z$, denoting a specific sample $Z_i$. The Law says that we can approximate the expected value of $Z$ by averaging over all samples. Consider the average:

$$\frac{1}{N} \sum_{i=1}^{N} Z_i$$

By construction, $Z_i$ can only take on $c_1$ or $c_2$, hence we can partition the sum over these two values:

Equality holds in the limit, but we can get closer and closer by using more and more samples in the average. This Law holds for almost *any distribution*, minus some important cases we will encounter later.

## *Example*

Below is a diagram of the Law of Large numbers in action for three different sequences of Poisson random variables.

We sample `sample_size = 100000` Poisson random variables with parameter $\lambda = 4.5$. (Recall the expected value of a Poisson random variable is equal to it's parameter.) We calculate the average for the first $n$ samples, for $n = 1$ to `sample_size`.

```
%pylab inline

figsize( 12.5, 5 )
import pymc as pm

sample_size = 100000
expected_value = lambda_ = 4.5
poi = pm.rpoisson
N_samples = range(1,sample_size,100)
```

```
for k in range(3):

    samples = poi( lambda_, size = sample_size )

    partial_average = [ samples[:i].mean() for i in N_samples ]

    plt.plot( N_samples, partial_average, lw=1.5,label="average \
of  $n$ samples; seq. %d"%k)


plt.plot( N_samples, expected_value*np.ones_like( partial_average),
    ls = "--", label = "true expected value", c = "k" )

plt.ylim( 4.35, 4.65)
plt.title( "Convergence of the average of \n random variables to its \
expected value" )
plt.ylabel( "average of $n$ samples" )
plt.xlabel( "# of samples, $n$")
plt.legend();
```
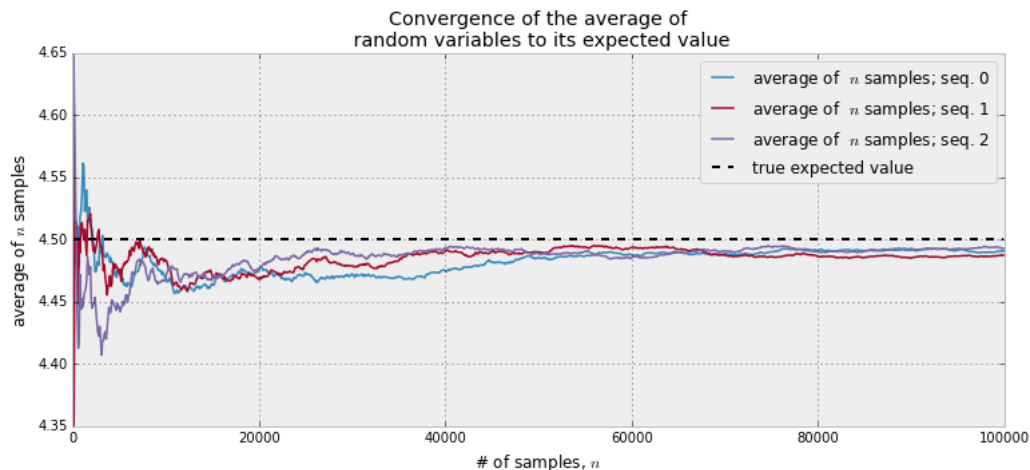
Populating the interactive namespace from numpy and matplotlib

Convergence of the average of
random variables to its expected value



Looking at the above plot, it is clear that when the sample size is small, there is greater variation in the average (compare how *jagged and jumpy* the average is initially, then *smooths* out). All three paths *approach* the value 4.5, but just flirt with it as $N$ gets large. Mathematicians and statistician have another name for *flirting*: convergence.

Another very relevant question we can ask is *how quickly am I converging to the expected value?* Let's plot something new. For a specific $N$, let's do the above trials thousands of times and compute how far away we are from the true expected value, on average. But wait — *compute on average?* This is simply the law of large numbers again! For example, we are interested in, for a specific $N$, the quantity:

$$D(N) = \sqrt{ E\left[ \left( \frac{1}{N} \sum_{i=1}^{N} Z_i - 4.5 \right)^2 \right] }$$

The above formulae is interpretable as a distance away from the true value (on average), for some $N$. (We take the square root so the dimensions of the above quantity and our random variables are the same). As the above is an expected value, it can be approximated using the law of large numbers: instead of averaging $Z_i$, we calculate the following multiple times and average them:

$$Y_k = \left( \frac{1}{N} \sum_{i=1}^{N} Z_i - 4.5 \right)^2$$

By computing the above many, $N_y$, times (remember, it is random), and averaging them:

$$\frac{1}{N_Y} \sum_{k=1}^{N_Y} Y_k \rightarrow E[Y_k] = E\left[ \left( \frac{1}{N} \sum_{i=1}^{N} Z_i - 4.5 \right)^2 \right]$$

Finally, taking the square root:

$$\sqrt{\frac{1}{N_Y} \sum_{k=1}^{N_Y} Y_k} \approx D(N)$$

```
figsize( 12.5, 4)


N_Y = 250 #use this many to approximate D(N)
N_array = np.arange( 1000, 50000, 2500 ) #use this many samples in the approx. to the variance.
D_N_results = np.zeros( len( N_array ) )


lambda_ = 4.5
expected_value = lambda_ #for X ~ Poi(lambda) , E[ X ] = lambda

def D_N( n ):
    """
    This function approx. D_n, the average variance of using n samples.
    """
    Z = poi( lambda_, size = (n, N_Y) )
    average_Z = Z.mean(axis=0)
    return np.sqrt( (  (average_Z - expected_value)**2  ).mean() )


for i,n in enumerate(N_array):
    D_N_results[i] =  D_N(n)


plt.xlabel( "$N$" )
plt.ylabel( "expected squared-distance from true value" )
plt.plot(N_array, D_N_results, lw = 3,
            label="expected distance between\n\
expected value and \naverage of $N$ random variables.")
plt.plot( N_array, np.sqrt(expected_value)/np.sqrt(N_array), lw = 2, ls = "--",
        label = r"$\frac{\sqrt{\lambda}}{\sqrt{N}}$" )
plt.legend()
plt.title( "How 'fast' is the sample average converging? " );
```
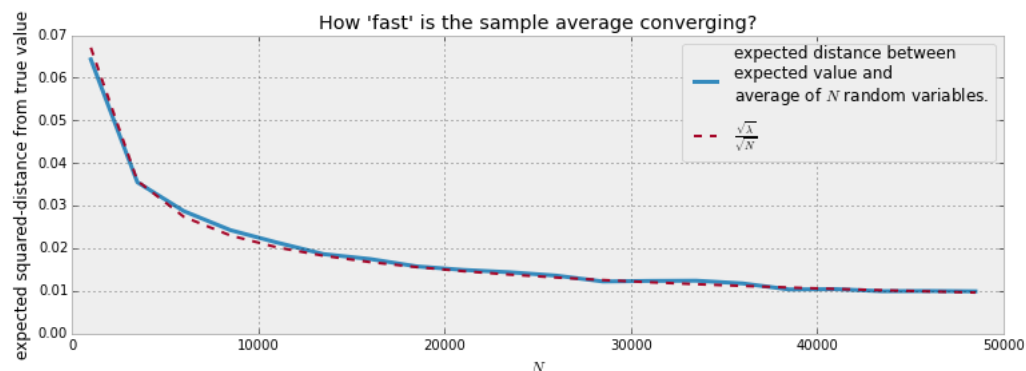


As expected, the expected distance between our sample average and the actual expected value shrinks as $N$ grows large. But also notice that the *rate* of convergence decreases, that is, we need only 10 000 additional samples to move from 0.020 to 0.015, a difference of 0.005, but *20 000* more samples to again decrease from 0.015 to 0.010, again only a 0.005 decrease.

It turns out we can measure this rate of convergence. Above I have plotted a second line, the function $\sqrt{\lambda}/\sqrt{N}$. This was not chosen arbitrarily. In most cases, given a sequence of random variable distributed like $Z$, the rate of converge to $E[Z]$ of the Law of Large Numbers is

$$\frac{\sqrt{Var(Z)}}{\sqrt{N}}$$

This is useful to know: for a given large $N$, we know (on average) how far away we are from the estimate. On the other hand, in a Bayesian setting, this can seem like a useless result: Bayesian analysis is OK with uncertainty so what's the *statistical* point of adding extra precise digits? Though drawing samples can be so computationally cheap that having a *larger $N$* is fine too.

# How do we compute $Var(Z)$ though?

The variance is simply another expected value that can be approximated! Consider the following, once we have the expected value (by using the Law of Large Numbers to estimate it, denote it $\mu$), we can estimate the variance:

$$\frac{1}{N} \sum_{i=1}^{N} (Z_i - \mu)^2 \rightarrow E[\ (Z - \mu)^2\ ] = Var(Z)$$

## Expected values and probabilities

There is an even less explicit relationship between expected value and estimating probabilities. Define the *indicator function*

$$\mathbb{1}_A(x) = \begin{cases} 1 & x \in A \\ 0 & else \end{cases}$$

Then, by the law of large numbers, if we have many samples $X_i$, we can estimate the probability of an event $A$, denoted $P(A)$, by:

$$\frac{1}{N} \sum_{i=1}^{N} \mathbb{1}_A(X_i) \rightarrow E[\mathbb{1}_A(X)] = P(A)$$

Again, this is fairly obvious after a moments thought: the indicator function is only 1 if the event occurs, so we are summing only the times the event occurs and dividing by the total number of trials (consider how we usually approximate probabilities using frequencies). For example, suppose we wish to estimate the probability that a $Z \sim Exp(.5)$ is greater than 10, and we have many samples from a $Exp(.5)$ distribution.

$$P(Z > 10) = \sum_{i=1}^{N} \mathbb{1}_{z>10}(Z_i)$$

```
import pymc as pm

N = 10000
print np.mean( [ pm.rexponential( 0.5 )>10 for i in range(N) ] )
```

```
0.0073
```

## What does this all have to do with Bayesian statistics?

*Point estimates*, to be introduced in the next chapter, in Bayesian inference are computed using expected values. In more analytical Bayesian inference, we would have been required to evaluate complicated expected values represented as multi-dimensional integrals. No longer. If we can sample from the posterior distribution directly, we simply need to evaluate averages. Much easier. If accuracy is a priority, plots like the ones above show how fast you are converging. And if further accuracy is desired, just take more samples from the posterior.

When is enough enough? When can you stop drawing samples from the posterior? That is the practitioners decision, and also dependent on the variance of the samples (recall from above a high variance means the average will converge slower).

We also should understand when the Law of Large Numbers fails. As the name implies, and comparing the graphs above for small $N$, the Law is only true for large sample sizes. Without this, the asymptotic result is not reliable. Knowing in what situations the Law fails can give us *confidence in how unconfident we should be*. The next section deals with this issue.

## The Disorder of Small Numbers

The Law of Large Numbers is only valid as $N$ gets *infinitely* large: never truly attainable. While the law is a powerful tool, it is foolhardy to apply it liberally. Our next example illustrates this.

### Example: Aggregated geographic data

Often data comes in aggregated form. For instance, data may be grouped by state, county, or city level. Of course, the population numbers vary per geographic area. If the data is an average of some characteristic of each the geographic areas, we must be conscious of the Law of Large Numbers and how it can *fail* for areas with small populations.

We will observe this on a toy dataset. Suppose there are five thousand counties in our dataset. Furthermore, population number in each state are uniformly distributed between 100 and 1500. The way the population numbers are generated is irrelevant to the discussion, so we do not justify this. We are interested in measuring the average height of individuals per county. Unbeknownst to the us, height does not vary across county, and each individual, regardless of the county he or she is currently living in, has the same distribution of what their height may be:

$$height \sim Normal(150, 15)$$

We aggregate the individuals at the county level, so we only have data for the *average in the county*. What might our dataset look like?

```
figsize( 12.5, 4)
std_height = 15
mean_height = 150

n_counties = 5000
pop_generator = pm.rdiscrete_uniform
norm = pm.rnormal

#generate some artificial population numbers
population = pop_generator(100, 1500, size = n_counties )

average_across_county = np.zeros( n_counties )
for i in range( n_counties ):
    #generate some individuals and take the mean
    average_across_county[i] = norm(mean_height, 1./std_height**2,
                                    size=population[i] ).mean()

#located the counties with the apparently most extreme average heights.
i_min = np.argmin( average_across_county )
i_max = np.argmax( average_across_county )

#plot population size vs. recorded average
plt.scatter( population, average_across_county, alpha = 0.5, c="#7A68A6")
plt.scatter( [ population[i_min], population[i_max] ],
            [average_across_county[i_min], average_across_county[i_max] ],
            s = 60, marker = "o", facecolors = "none",
            edgecolors = "#A60628", linewidths = 1.5,
             label="extreme heights")

plt.xlim( 100, 1500 )
plt.title( "Average height vs. County Population")
plt.xlabel("County Population")
plt.ylabel("Average height in county")
plt.plot( [100, 1500], [150, 150], color = "k", label = "true expected \
height", ls="--" )
plt.legend(scatterpoints = 1)
```
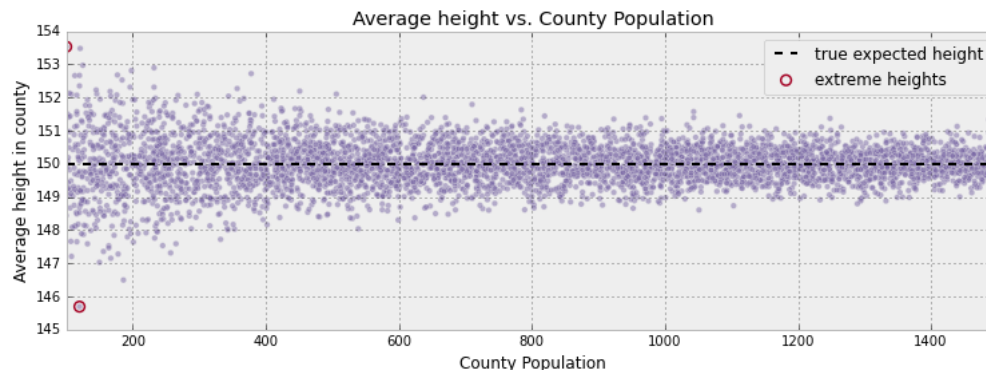
```
<matplotlib.legend.Legend at 0x115640f10>
```

What do we observe? *Without accounting for population sizes* we run the risk of making an enormous inference error: if we ignored population size, we would say that the county with the shortest and tallest individuals have been correctly circled. But this inference is wrong for the following reason. These two counties do *not* necessarily have the most extreme heights. The error results from the calculated average of smaller populations not being a good reflection of the true expected value of the population (which in truth should be $\mu = 150$). The sample size/population size/$N$, whatever you wish to call it, is simply too small to invoke the Law of Large Numbers effectively.

We provide more damning evidence against this inference. Recall the population numbers were uniformly distributed over 100 to 1500. Our intuition should tell us that the counties with the most extreme population heights should also be uniformly spread over 100 to 4000, and certainly independent of the county's population. Not so. Below are the population sizes of the counties with the most extreme heights.

```
print "Population sizes of 10 'shortest' counties: "
print population[ np.argsort( average_across_county )[:10] ]
print
print "Population sizes of 10 'tallest' counties: "
print population[ np.argsort( -average_across_county )[:10] ]
```

```
Population sizes of 10 'shortest' counties:
[120 186 150 167 108 178 257 156 164 164]

Population sizes of 10 'tallest' counties:
[100 121 125 232 198 378 177 120 113 184]
```

Not at all uniform over 100 to 1500. This is an absolute failure of the Law of Large Numbers.

## Example: Kaggle's U.S. Census Return Rate Challenge

Below is data from the 2010 US census, which partitions populations beyond counties to the level of block groups (which are aggregates of city blocks or equivalents). The dataset is from a Kaggle machine learning competition some colleagues and I participated in. The objective was to predict the census letter mail-back rate of a group block, measured between 0 and 100, using census variables (median income, number of females in the block-group, number of trailer parks, average number of children etc.). Below we plot the census mail-back rate versus block group population:

```
figsize( 12.5, 6.5 )
data = np.genfromtxt( "./data/census_data.csv", skip_header=1,
                        delimiter= ",")
plt.scatter( data[:,1], data[:,0], alpha = 0.5, c="#7A68A6")
plt.title("Census mail-back rate vs Population")
plt.ylabel("Mail-back rate")
plt.xlabel("population of block-group")
plt.xlim(-100, 15e3 )
plt.ylim( -5, 105)

i_min = np.argmin( data[:,0] )
i_max = np.argmax( data[:,0] )

plt.scatter( [ data[i_min,1], data[i_max, 1] ],
             [ data[i_min,0],  data[i_max,0] ],
             s = 60, marker = "o", facecolors = "none",
             edgecolors = "#A60628", linewidths = 1.5,
             label="most extreme points")

plt.legend(scatterpoints = 1);
```
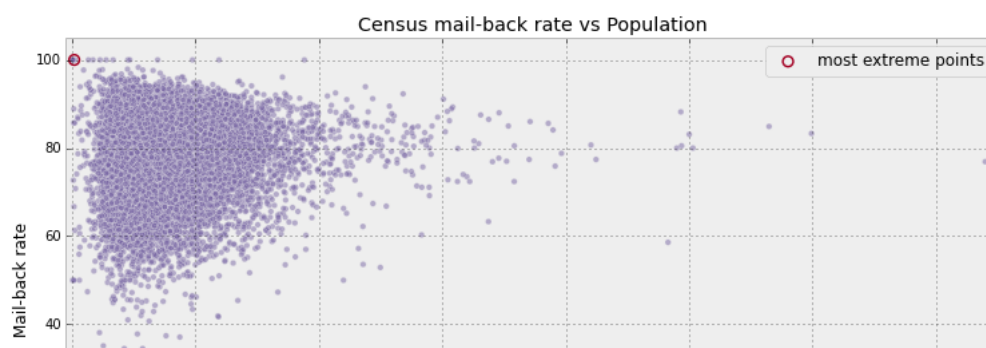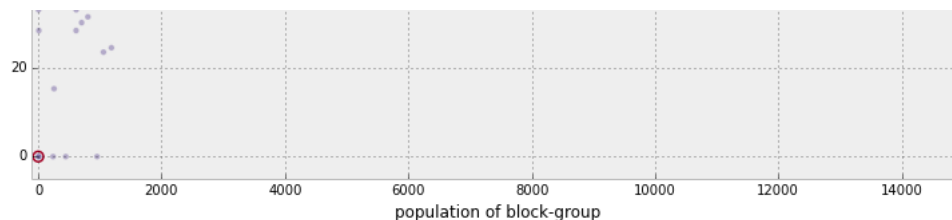
The above is a classic phenomenon in statistics. I say *classic* referring to the "shape" of the scatter plot above. It follows a classic triangular form, that tightens as we increase the sample size (as the Law of Large Numbers becomes more exact).

I am perhaps overstressing the point and maybe I should have titled the book *"You don't have big data problems!"*, but here again is an example of the trouble with *small datasets*, not big ones. Simply, small datasets cannot be processed using the Law of Large Numbers. Compare with applying the Law without hassle to big datasets (ex. big data). I mentioned earlier that paradoxically big data prediction problems are solved by relatively simple algorithms. The paradox is partially resolved by understanding that the Law of Large Numbers creates solutions that are *stable*, i.e. adding or subtracting a few data points will not affect the solution much. On the other hand, adding or removing data points to a small dataset can create very different results.

For further reading on the hidden dangers of the Law of Large Numbers, I would highly recommend the excellent manuscript The Most Dangerous Equation (http://nsm.uh.edu/~dgraur/niv/TheMostDangerousEquation.pdf).

## Example: How to order Reddit comments

You may have disagreed with the original statement that the Law of Large numbers is known to everyone, but only implicitly in our subconscious decision making. Consider ratings on online products: how often do you trust an average 5-star rating if there is only 1 reviewer? 2 reviewers? 3 reviewers? We implicitly understand that with such few reviewers that the average rating is not a good reflection of the true value of the product.

This has created flaws in how we sort items, and more generally, how we compare items. Many people have realized that sorting online search results by their rating, whether the objects be books, videos, or online comments, return poor results. Often the seemingly top videos or comments have perfect ratings only from a few enthusiastic fans, and truly more quality videos or comments are hidden in later pages with *falsely-substandard* ratings of around 4.8. How can we correct this?

Consider the popular site Reddit (I purposefully did not link to the website as you would never come back). The site hosts links to stories or images, and a very popular part of the site are the comments associated with each link. Redditors can vote up or down on each comment (called upvotes and downvotes). Reddit, by default, will sort comments by Top, that is, the best comments.



How would you determine which comments are the best? There are a number of ways to achieve this:

1. *Popularity*: A comment is considered good if it has many upvotes. A problem with this model is that a comment with hundreds of upvotes, but thousands of downvotes. While being very *popular*, the comment is likely more controversial than best.
2. *Difference*: Using the *difference* of upvotes and downvotes. This solves the above problem, but fails when we consider the temporal nature of comments. Comments can be posted many hours after the original link submission. The difference method will bias the *Top* comments to be the oldest comments, which have accumulated more upvotes than newer comments, but are not

necessarily the best.

3. *Time adjusted*: Consider using Difference divided by the age of the comment. This creates a *rate*, something like *difference per second*, or *per minute*. An immediate counter example is, if we use per second, a 1 second old comment with 1 upvote would be better than a 100 second old comment with 99 upvotes. One can avoid this by only considering at least t second old comments. But what is a good t value? Does this mean no comment younger than t is good? We end up comparing unstable quantities with stable quantities (young vs. old comments).

4. *Ratio*: Rank comments by the ratio of upvotes to total number of votes (upvotes plus downvotes). This solves the temporal issue, such that new comments who score well can be considered Top just as likely as older comments, provided they have many upvotes to total votes. The problem here is that a comment with a single upvote (ratio = 1.0) will beat a comment with 999 upvotes and 1 downvote (ratio = 0.999), but clearly the latter comment is *more likely* to be better.

I used the phrase *more likely* for good reason. It is possible that the former comment, with a single upvote, is in fact a better comment than the later with 999 upvotes. The hesitation to agree with this is because we have not seen the other 999 potential votes the former comment might get. Perhaps it will achieve an additional 999 upvotes and 0 downvotes and be considered better than the latter, though not likely.

What we really want is an estimate of the *true upvote ratio*. Note that the true upvote ratio is not the same as the observed upvote ratio: the true upvote ratio is hidden, and we only observe upvotes vs. downvotes (one can think of the true upvote ratio as "what is the underlying probability someone gives this comment a upvote, versus a downvote"). So the 999 upvote/1 downvote comment probably has a true upvote ratio close to 1, which we can assert with confidence thanks to the Law of Large Numbers, but on the other hand we are much less certain about the true upvote ratio of the comment with only a single upvote. Sounds like a Bayesian problem to me.

One way to determine a prior on the upvote ratio is that look at the historical distribution of upvote ratios. This can be accomplished by scraping Reddit's comments and determining a distribution. There are a few problems with this technique though:

1. Skewed data: The vast majority of comments have very few votes, hence there will be many comments with ratios near the extremes (see the "triangular plot" in the above Kaggle dataset), effectively skewing our distribution to the extremes. One could try to only use comments with votes greater than some threshold. Again, problems are encountered. There is a tradeoff between number of comments available to use and a higher threshold with associated ratio precision.

2. Biased data: Reddit is composed of different subpages, called subreddits. Two examples are *r/aww*, which posts pics of cute animals, and *r/politics*. It is very likely that the user behaviour towards comments of these two subreddits are very different: visitors are likely friend and affectionate in the former, and would therefore upvote comments more, compared to the latter, where comments are likely to be controversial and disagreed upon. Therefore not all comments are the same.

In light of these, I think it is better to use a `Uniform` prior.

With our prior in place, we can find the posterior of the true upvote ratio. The Python script `comments_for_top_reddit_pic.py` will scrape the comments from the current top picture on Reddit. Below is the picture, and some comments:

```
from IPython.core.display import Image
#adding a number to the end of the %run call with get the ith top photo.
%run top_pic_comments.py 2

Image(top_post_url)
```

```
Title of submission:
Happy halloween filthy animals.
http://i.imgur.com/Ecery9G.jpg
```
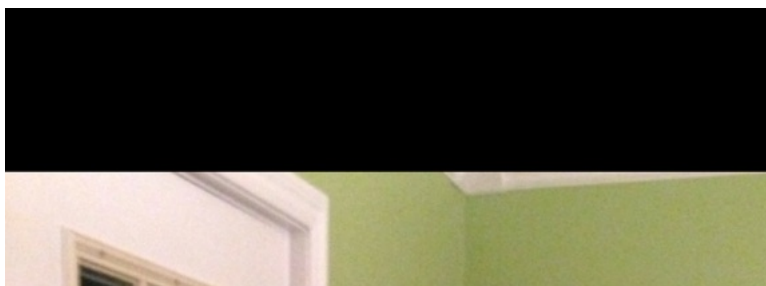
```
"""
contents: an array of the text from all comments on the pic
votes: a 2d numpy array of upvotes, downvotes for each comment.
"""
n_comments = len(contents )
comments = np.random.randint( n_comments, size=4)
print "Some Comments (out of %d total) \n-----------"%n_comments
for i in comments:
    print '"' + contents[i] + '"'
    print"upvotes/downvotes: ",votes[i,:]
    print
```

```
Some Comments (out of 117 total)
-----------
"Dat iron mark...this is some award winning work right here"
upvotes/downvotes:  [43  6]

"Harry: [talking to Kate as she drags Kevin upstairs] Don't worry about me. I spoke with your hus
band already, and don't worry about your home. It's in good hands."
upvotes/downvotes:  [1 0]

"You win. "
upvotes/downvotes:  [15  3]

"Brilliant. "
upvotes/downvotes:  [8 0]
```

For a given true upvote ratio $p$ and $N$ votes, the number of upvotes will look like a Binomial random variable with parameters $p$ and $N$. (This is because of the equivalence between upvote ratio and probability of upvoting versus downvoting, out of $N$ possible votes/trials). We create a function that performs Bayesian inference on $p$, for a particular comment's upvote/downvote pair.

```
import pymc as pm

def posterior_upvote_ratio( upvotes, downvotes, samples = 20000):
    """
    This function accepts the number of upvotes and downvotes a particular comment recieved,
    and the number of posterior samples to return to the user. Assumes a uniform prior.
    """
    N = upvotes + downvotes
    upvote_ratio = pm.Uniform( "upvote_ratio", 0, 1 )
    observations = pm.Binomial( "obs",  N, upvote_ratio, value = upvotes, observed = True)
    #do the fitting; first do a MAP as it is cheap and useful.
    map_ = pm.MAP([upvote_ratio, observations ]).fit()
    mcmc = pm.MCMC([upvote_ratio, observations ])
    mcmc.sample(samples, samples/4)
    return mcmc.trace("upvote_ratio")[:]
```

Below are the resulting posterior distributions.

```
figsize( 11., 8)
posteriors = []
colours = ["#348ABD", "#A60628", "#7A68A6", "#467821", "#CF4457"]
for i in range(len(comments)):
    j = comments[i]
    posteriors.append( posterior_upvote_ratio( votes[j, 0], votes[j,1] ) )
    plt.hist( posteriors[i], bins = 18, normed = True, alpha = .9,
            histtype="step",color = colours[i%5], lw = 3,
            label = '(%d up:%d down)\n%s...'%(votes[j, 0], votes[j,1], contents[j][:50]) )
    plt.hist( posteriors[i], bins = 18, normed = True, alpha = .2,
            histtype="stepfilled",color = colours[i], lw = 3, )


plt.legend(loc="upper left")
plt.xlim( 0, 1)
plt.title("Posterior distributions of upvote ratios on different comments");
```

```
[****************100%******************]  20000 of 20000 complete
```
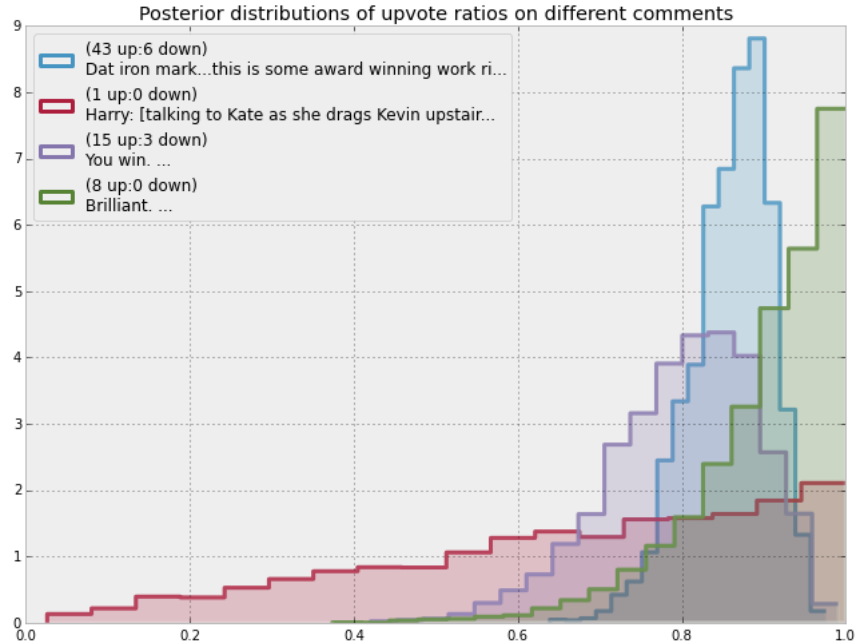


Some distributions are very tight, others have very long tails (relatively speaking), expressing our uncertainty with what the true upvote ratio might be.

## Sorting!

We have been ignoring the goal of this exercise: how do we sort the comments from *best to worst*? Of course, we cannot sort distributions, we must sort scalar numbers. There are many ways to distill a distribution down to a scalar: expressing the distribution through its expected value, or mean, is one way.

Choosing the mean bad choice though. This is because the mean does not take into account the uncertainty of distributions.

I suggest using the *95% least plausible value*, defined as the value such that there is only a 5% chance the true parameter is lower (think of the lower bound on the 95% credible region). Below are the posterior distributions with the 95% least-plausible value plotted:

```python
N = posteriors[0].shape[0]
lower_limits = []

for i in range(len(comments)):
    j = comments[i]
    plt.hist( posteriors[i], bins = 20, normed = True, alpha = .9,
            histtype="step",color = colours[i], lw = 3,
            label = '(%d up:%d down)\n%s...'%(votes[j, 0], votes[j,1], contents[j][:50]) )
    plt.hist( posteriors[i], bins = 20, normed = True, alpha = .2,
            histtype="stepfilled",color = colours[i], lw = 3, )
    v = np.sort( posteriors[i] )[ int(0.05*N) ]
    #plt.vlines( v, 0, 15 , color = "k", alpha = 1, linewidths=3 )
    plt.vlines( v, 0, 10 , color = colours[i], linestyles = "--",  linewidths=3  )
    lower_limits.append(v)
    plt.legend(loc="upper left")

plt.legend(loc="upper left")
plt.title("Posterior distributions of upvote ratios on different comments");
order = argsort( -np.array( lower_limits ) )
print order, lower_limits
```

[0 3 2 1] [0.77284791921589524, 0.21606229171548902, 0.63518378151985821, 0.7259241799341366]
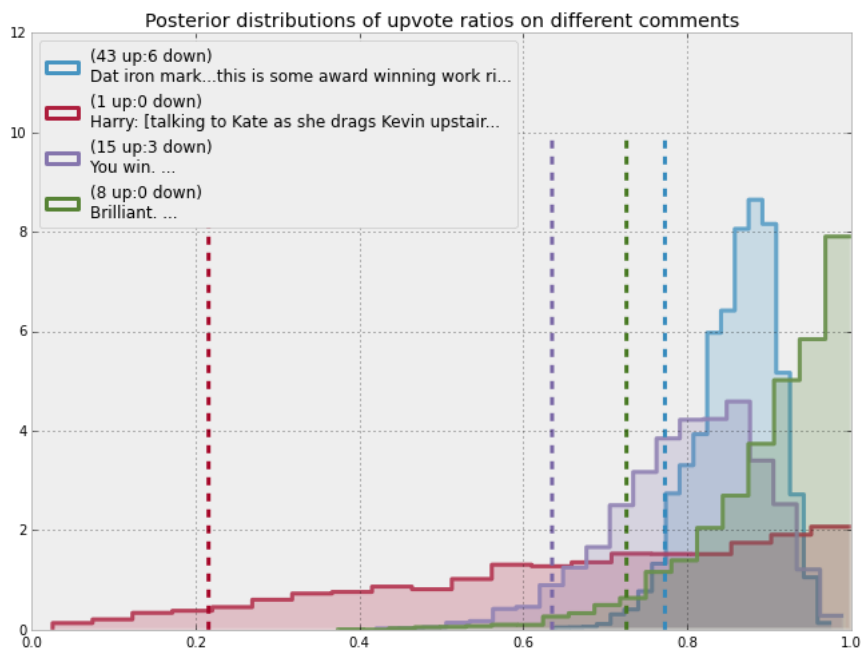


The best comments, according to our procedure, are the comments that are *most-likely* to score a high percentage of upvotes. Visually those are the comments with the 95% least plausible value close to 1.

Why is sorting based on this quantity a good idea? By ordering by the 95% least plausible value, we are being the most conservative with what we think is best. That is, even in the worst case scenario, when we have severely overestimated the upvote ratio, we can be sure the best comments are still on top. Under this ordering, we impose the following very natural properties:

1. given two comments with the same observed upvote ratio, we will assign the comment with more votes as better (since we are more confident it has a higher ratio).
2. given two comments with the same number of votes, we still assign the comment with more upvotes as *better*.

## But this is too slow for real-time!

I agree, computing the posterior of every comment takes a long time, and by the time you have computed it, likely the data has changed. I delay the mathematics to the appendix, but I suggest using the following formula to compute the lower bound very fast.

$$\frac{a}{a + b} - 1.65\sqrt{\frac{ab}{(a + b)^2 (a + b + 1)}}$$

where

$u$ is the number of upvotes, and $d$ is the number of downvotes. The formula is a shortcut in Bayesian inference, which will be further explained in Chapter 6 when we discuss priors in more detail.

```
def intervals(u,d):
    a = 1. + u
    b = 1. + d
    mu = a/(a+b)
    std_err = 1.65*np.sqrt( (a*b)/( (a+b)**2*(a+b+1.) ) )
    return ( mu, std_err )

print "Approximate lower bounds:"
posterior_mean, std_err  = intervals(votes[:,0],votes[:,1])
lb = posterior_mean - std_err
print lb
print
print "Top 40 Sorted according to approximate lower bounds:"
print
order = np.argsort( -lb )
ordered_contents = []
for i in order[:40]:
    ordered_contents.append( contents[i] )
    print  votes[i,0], votes[i,1], contents[i]
    print "-------------"
```

```
Approximate lower bounds:
[ 0.84053422  0.80282754  0.81932714  0.84117301  0.84126989  0.80968254
  0.78400654  0.79853422  0.83882784  0.78547082  0.68548406  0.7609129
  0.80556899  0.7908931   0.75075188  0.75075188  0.65597619  0.6931046
  0.6931046   0.60100251  0.63905875  0.6931046   0.6530083   0.6931046
  0.635       0.68704735  0.47201974  0.60091591  0.47201974  0.60100251
  0.53055613  0.53055613  0.60091591  0.53055613  0.60091591  0.43047887
  0.43047887  0.53055613  0.43047887  0.43047887  0.43047887  0.43047887
  0.43047887  0.53055613  0.43047887  0.43047887  0.27        0.53055613
  0.43047887  0.43047887  0.45074913  0.45093892  0.3726793   0.3726793
  0.27775794  0.27775794  0.27775794  0.27775794  0.27775794  0.27775794
  0.43047887  0.27775794  0.27775794  0.27775794  0.27775794  0.27775794
  0.13104878  0.27775794  0.27775794  0.27775794  0.43047887  0.27775794
  0.27775794  0.27775794  0.27775794  0.27775794  0.27775794  0.13104878
  0.27775794  0.27775794  0.27775794  0.27775794  0.27775794  0.27775794
  0.27775794  0.27775794  0.27775794  0.27775794  0.27775794  0.27775794
  0.27775794  0.27775794  0.27775794  0.43047887  0.27775794  0.27775794
  0.27775794  0.27775794  0.27775794  0.27775794  0.27775794  0.27775794
  0.27775794  0.27775794  0.27775794  0.27775794  0.27775794  0.27775794
  0.27775794  0.27775794  0.27775794  0.27775794  0.27775794  0.27775794
  0.27775794  0.27775794  0.27775794]

Top 40 Sorted according to approximate lower bounds:

205 28 "SUCK BRICK KID!"
-------------
362 54 "Why'd you take your shoes off?"
 "Why are you dressed like a chicken?"
-------------
856 140 There really is only one thing a bartender could say to you two:

&gt; You guys give up, or ya thirsty for more?
-------------
127 16 You guys fucking win any contest in my book.  Amazingly done.
-------------
160 25 Are you scared Marv? Are you afraid?
-------------
125 20 Marv: Kids are scared of the dark.

Harry: You're afraid of the dark, too, Marv.
-------------

11 0 So awesome.
```

"SMASH his face with an iron"

-------------

95 15 Really weird because Home Alone 2 was on TV tonight. It's not even the Christmas season yet
 but I am pumped for classic movies like this to come back on.

Whenever my dad will hurt himself he will do the voice of Marv when he's in the basement of the a
partment building in HA2 and go "Haaaawwwy..."

-------------

145 26 The wet bandits!

-------------

23 2 You guys actually resemble the actors, too.

-------------

72 12 You know what I'll give you Snakes... I'll give you to the count of ten, to get your ugly,
yella, no-good kester, off my property before I pump your guts full of lead!

One..
Two..
TEN!

*unleashes like a billion rounds while laughing like a psycho*

And keep the change ya filthy animal.

-------------

43 6 Dat iron mark...this is some award winning work right here

-------------

29 4 Keep the change.

-------------

8 0 right in the shnauz!

-------------

8 0 Brilliant.

-------------

6 0 We're being scammed by a kindy-garner!

Well done boys. Well done. i wish I was there to be your kevin.

-------------

6 0 A legitimate scary costume from that movie would've been the old man who salted the sidewalk.
 He freaked me the hell out a as a kid. Granted it wouldn't be very topical but you get the idea.

-------------

6 0 That's the sound of a tool chest coming down the stairs....adjust nose

-------------

6 0 http://i.imgur.com/kPuetee.jpg

-------------

10 1 Marry Christmas *tiwnkle*Smile*

-------------

17 3 macaulay culkin is now the same age as daniel stern was when he was in home alone.

-------------

15 3 You win.

-------------

5 0 Hey guys I'm callin' the cops!

-------------

14 3 Snakes, Snakes. I don't know no Snakes.

-------------

11 2 Excellent detail work!

-------------

7 1 Upvote from me you big horse's ass.

-------------

7 1 The Wet Bandits!

-------------

4 0 Haha, brings back great childhood memories!

-------------

4 0 And a happy new year!!

-------------

4 0 Im upvoting because of my childhood appreciates you.

-------------

3 0 Shut up, Marv! Geez

-------------

3 0 Bald guy needs more of a shit-eating grin displaying a little *gold* crown.

EDIT: Now we're talkin': http://whiggles.landofwhimsy.com/hdcaptures/homealone3.jpg

-------------

3 0 This gave me a nostalgia rush

-------------

3 0  What's the brand mark on his hand from? I must not have remembered that part.

-------------

3 0 Gotta agree with most people here. Awesome costumes!

```
ed agree with most people here. Awesome costumes.
-------------
3 0 So much fucking win.
-------------
6 2 I have such a strong holiday association of Christmas for this movie that I'm having trouble
processing that it's a Halloween costume. But the fact that your costume makes me think of Christ
mas means there must be an incredible likeness.
-------------
6 2 This leads me to believe Michael Rapaport would be excellent in a remake.
-------------
7 3 I wonder if the youngins are going to know who you guys are?
-------------
4 1 A, I'm not that lucky. Two, we use smoke detectors and D, we live on the most boring street i
n the whole United States of America, where nothing even remotely dangerous will ever happen. Per
iod.
-------------
```

We can view the ordering visually by plotting the posterior mean and bounds, and sorting by the lower bound. In the plot below, notice that the left error-bar is sorted (as we suggested this is the best way to determine an ordering), so the means, indicated by dots, do not follow any strong pattern.
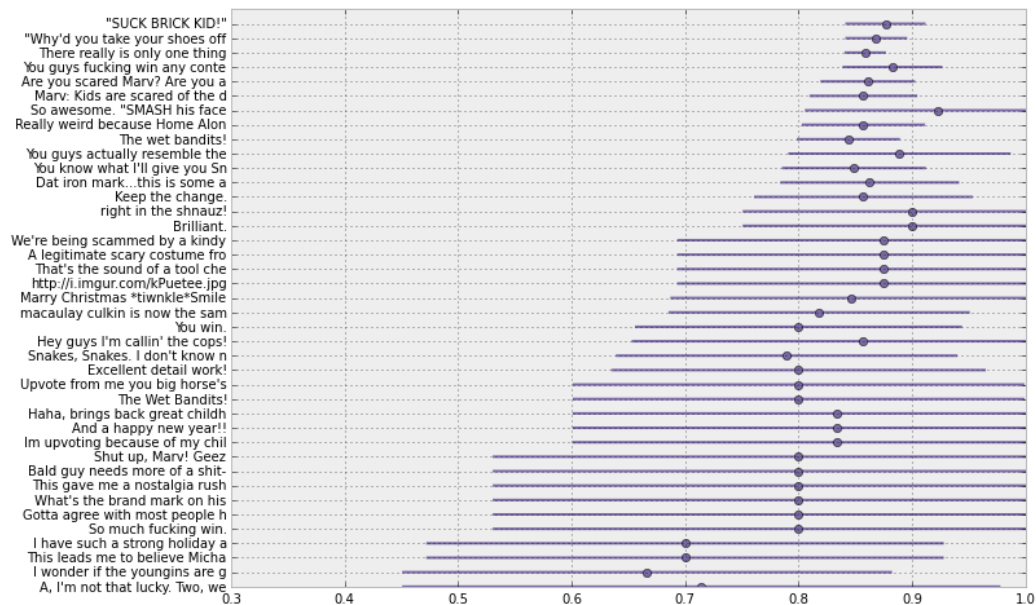
```
r_order
```

```
array([ 77,  66,  46, 116,  69,  82,  81,  80,  79,  78,  76,  75,  74,
        73,  72,  71,  67,  68,  84,  65,  64,  63,  62,  61,  59, 115,
        57,  56,  55,  54,  83,  58,  85, 100,  86, 113, 112, 111, 110, 109])
```

```python
r_order = order[::-1][-40:]
plt.errorbar( posterior_mean[r_order], np.arange( len(r_order) ),
             xerr=std_err[r_order],xuplims=True, capsize=0, fmt="o",
              color = "#7A68A6")
plt.xlim( 0.3, 1)
plt.yticks( np.arange( len(r_order)-1,-1,-1 ), map( lambda x: x[:30].replace("\n",""), ordered_c
ontents) );
```



In the graphic above, you can see why sorting by mean would be sub-optimal.

## Extension to Starred rating systems

The above procedure works well for upvote-downvotes schemes, but what about systems that use star ratings, e.g. 5 star rating systems. Similar problems apply with simply taking the average: an item with two perfect ratings would beat an item with thousands of perfect ratings, but a single sub-perfect rating.

We can consider the upvote-downvote problem above as binary: 0 is a downvote, 1 if an upvote. A $N$-star rating system can be seen as a more continuous version of above, and we can set $n$ stars rewarded is equivalent to rewarding $\frac{n}{N}$. For example, in a 5-star system, a 2 star rating corresponds to 0.4. A perfect rating is a 1. We can use the same formula as before, but with $a, b$ defined differently:

$$\frac{a}{a+b} - 1.65\sqrt{\frac{ab}{(a+b)^2(a+b+1)}}$$

where

where $N$ is the number of users who rated, and $S$ is the sum of all the ratings, under the equivalence scheme mentioned above.

## Example: Counting Github stars

What is the average number of stars a Github repository has? How would you calculate this? There are over 6 million respositories, so there is more than enough data to invoke the Law of Large numbers. Let's start pulling some data. TODO

## Conclusion

While the Law of Large Numbers is cool, it is only true so much as its name implies: with large sample sizes only. We have seen how our inference can be affected by not considering *how the data is shaped*.

1. By (cheaply) drawing many samples from the posterior distributions, we can ensure that the Law of Large Number applies as we approximate expected values (which we will do in the next chapter).
2. Bayesian inference understands that with small sample sizes, we can observe wild randomness. Our posterior distribution will reflect this by being more spread rather than tightly concentrated. Thus, our inference should be correctable.
3. There are major implications of not considering the sample size, and trying to sort objects that are unstable leads to pathological orderings. The method provided above solves this problem.

## Appendix

## Derivation of sorting comments formula

Basically what we are doing is using a Beta prior (with parameters $a = 1, b = 1$, which is a uniform distribution), and using a Binomial likelihood with observations $u, N = u + d$. This means our posterior is a Beta distribution with parameters $a' = 1 + u, b' = 1 + (N - u) = 1 + d$. We then need to find the value, $x$, such that 0.05 probability is less than $x$. This is usually done by inverting the CDF (Cumulative Distribution Function (http://en.wikipedia.org/wiki/Cumulative_Distribution_Function)), but the CDF of the beta, for integer parameters, is known but is a large sum [3].

We instead using a Normal approximation. The mean of the Beta is $\mu = a'/(a' + b')$ and the variance is

$$\sigma^2 = \frac{a'b'}{(a' + b')^2(a' + b' + 1)}$$

Hence we solve the following equation for $x$ and have an approximate lower bound.

$$0.05 = \Phi\left(\frac{(x - \mu)}{\sigma}\right)$$

$\Phi$ being the cumulative distribution for the normal distribution (http://en.wikipedia.org/wiki/Normal_distribution#Cumulative_distribution)

## Exercises

1. How would you estimate the quantity $E[\cos X]$, where $X \sim \text{Exp}(4)$? What about $E\big[\cos X | X < 1\big]$, i.e. the expected value *given* we know $X$ is less than 1? Would you need more samples than the original samples size to be equally accurate?

```
## Enter code here
import scipy.stats as stats
exp = stats.expon( scale=4 )
N = 1e5
X = exp.rvs( N )
## ...
```

2. The following table was located in the paper "Going for Three: Predicting the Likelihood of Field Goal Success with Logistic Regression" [2]. The table ranks football field-goal kickers by their percent of non-misses. What mistake have the researchers made?

Kicker Careers Ranked by Make Percentage

| Rank | Kicker | Make % | Number of Kicks |
|---|---|---|---|
| 1 | Garrett Hartley | 87.7 | 57 |
| 2 | Matt Stover | 86.8 | 335 |
| 3 | Robbie Gould | 86.2 | 224 |
| 4 | Rob Bironas | 86.1 | 223 |
| 5 | Shayne Graham | 85.4 | 254 |
| ... | ... | ... | |
| 51 | Dave Rayner | 72.2 | 90 |
| 52 | Nick Novak | 71.9 | 64 |
| 53 | Tim Seder | 71.0 | 62 |
| 54 | Jose Cortez | 70.7 | 75 |
| 55 | Wade Richey | 66.1 | 56 |

In August 2013, a popular post (http://bpodgursky.wordpress.com/2013/08/21/average-income-per-programming-language/) on the average income per programmer of different languages was trending. Here's the summary chart: (reproduced without permission, cause when you lie with stats, you gunna get the hammer). What do you notice about the extremes?

Average household income by programming language

| Language | Average Household Income ($) | Data Points |
|---|---|---|
| Puppet | 87,589.29 | 112 |
| Haskell | 89,973.82 | 191 |
| PHP | 94,031.19 | 978 |
| CoffeeScript | 94,890.80 | 435 |
| VimL | 94,967.11 | 532 |
| Shell | 96,930.54 | 979 |
| Lua | 96,930.69 | 101 |
| Erlang | 97,306.55 | 168 |
| Clojure | 97,500.00 | 269 |
| Python | 97,578.87 | 2314 |
| JavaScript | 97,598.75 | 3443 |
| Emacs Lisp | 97,774.65 | 355 |
| C# | 97,823.31 | 665 |
| Ruby | 98,238.74 | 3242 |
| C++ | 99,147.93 | 845 |
| CSS | 99,881.40 | 527 |
| Perl | 100,295.45 | 990 |
| C | 100,766.51 | 2120 |
| Go | 101,158.01 | 231 |

| | | |
|---|---|---|
| Scala | 101,460.91 | 243 |
| ColdFusion | 101,536.70 | 109 |
| Objective-C | 101,801.60 | 562 |
| Groovy | 102,650.86 | 116 |
| Java | 103,179.39 | 1402 |
| XSLT | 106,199.19 | 123 |
| ActionScript | 108,119.47 | 113 |

## References

1. Wainer, Howard. *The Most Dangerous Equation*. American Scientist, Volume 95.
2. Clarck, Torin K., Aaron W. Johnson, and Alexander J. Stimpson. "Going for Three: Predicting the Likelihood of Field Goal Success with Logistic Regression." (2013): n. page. Web (http://www.sloansportsconference.com/wp-content/uploads/2013/Going%20for%20Three%20Predicting%20the%20Likelihood%20of%20Field%20Goal%20Success%20wit 20 Feb. 2013.
3. http://en.wikipedia.org/wiki/Beta_function#Incomplete_beta_function

```python
from IPython.core.display import HTML
def css_styling():
    styles = open("../styles/custom.css", "r").read()
    return HTML(styles)
css_styling()
```