Computational Statistics

Keegan Hines, Ph.D.
District Data Labs

Outline

About me

Data Science:

- Data Scientist at IronNet Cybersecurity
- PhD in Computational Neuroscience at University of Texas
- Instructor at General Assembly

Contact

- keegan.hines@gmail.com
- twitter: @keeghin

Resources

Course Slides

http://bit.ly/23PIFL0

Course Repo

https://github.com/keeganhines/computationalStatistics

or

http://bit.ly/20Jwgt3

Introduction

Classical Statistics

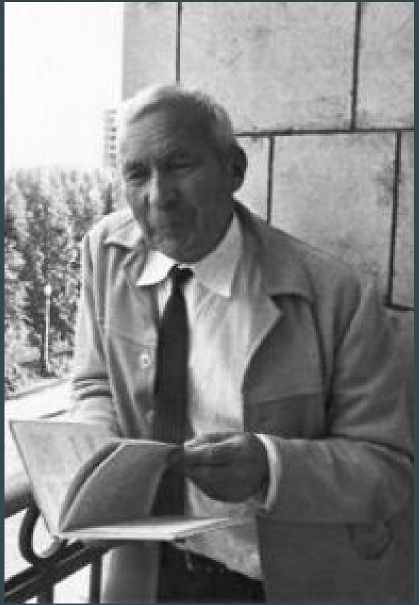
- You probably know a thing or two about ANOVA, Regression, t-tests and the like
- As we'll discuss, these ideas from classical (frequentist) statistics served scientists quite well in a time before computational power was cheap
- However, these methods are accompanied by many assumptions and limitations
- Now that we live in the future, and computers are fast and easy to program, we can approach questions about inference and uncertainty in new ways
- In this course, we'll learn about powerful computational methods that allow us to employ very complex statistical methods

A little history

Probability Theory (18th to 20th centuries)

Laplace

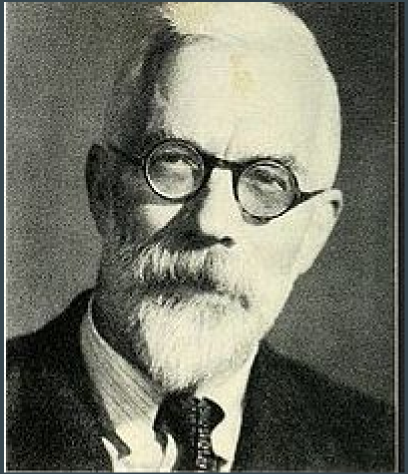
Gauss


Kolmogorov

Imagery from wikipedia

A little history

Classical Statistics (early 20th century) - Quantifying uncertainty about the world


Fisher

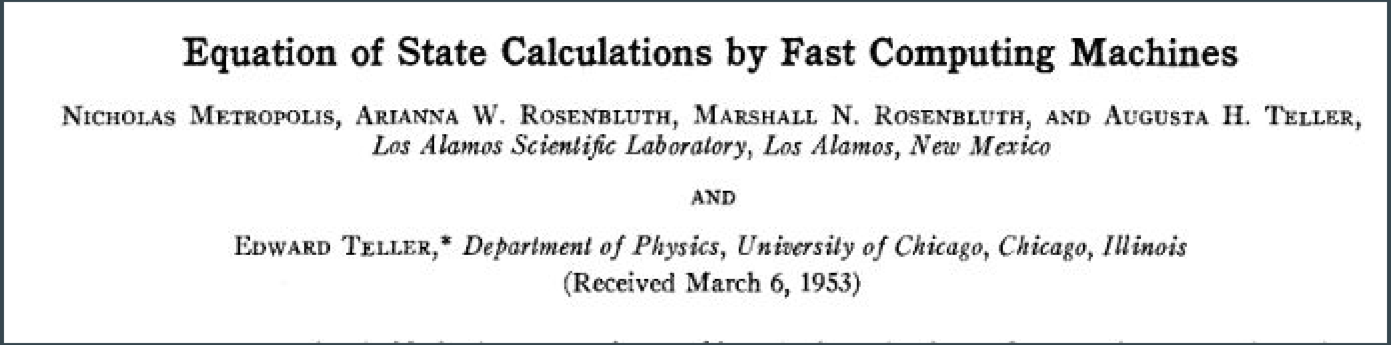
Pearson

Neyman

Imagery from wikipedia

A little history

Monte Carlo methods (mid 20th C) - use computers and random number generators



Equation of State Calculations by Fast Computing Machines

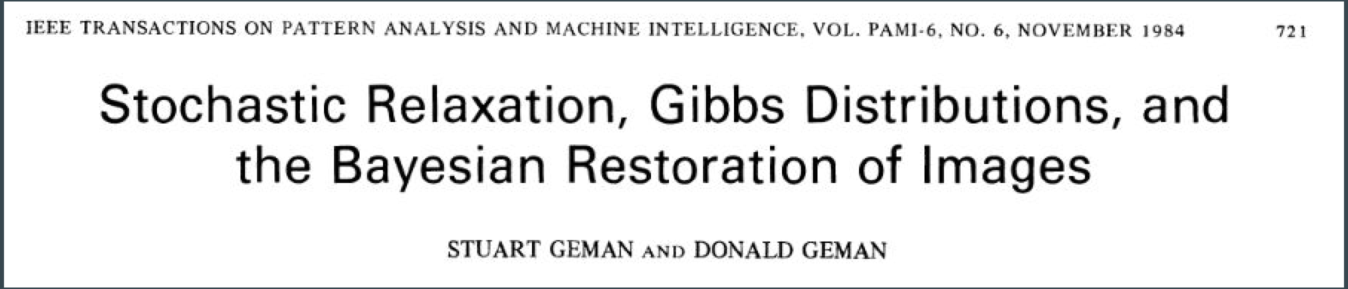
NICHOLAS METROPOLIS, ARIANNA W. ROSENBLUTH, MARSHALL N. ROSENBLUTH, AND AUGUSTA H. TELLER,
Los Alamos Scientific Laboratory, Los Alamos, New Mexico

AND

EDWARD TELLER,* *Department of Physics, University of Chicago, Chicago, Illinois*
(Received March 6, 1953)

A little history

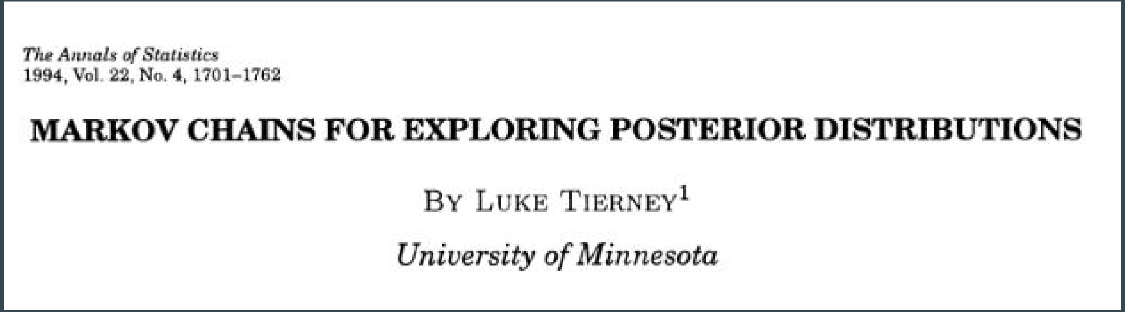
MCMC Bayesian methods (late 20th C) - approximate complex probability distributions



IEEE TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE, VOL. PAMI-6, NO. 6, NOVEMBER 1984 721

Stochastic Relaxation, Gibbs Distributions, and the Bayesian Restoration of Images

STUART GEMAN AND DONALD GEMAN



The Annals of Statistics
1994, Vol. 22, No. 4, 1701–1762

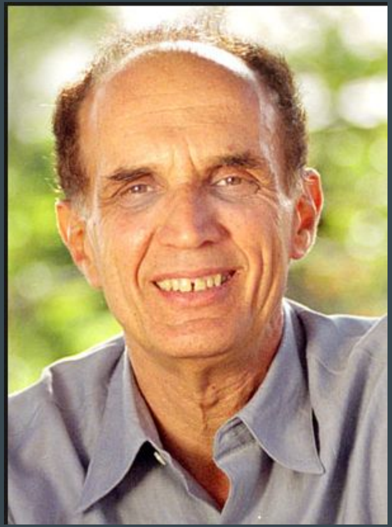
MARKOV CHAINS FOR EXPLORING POSTERIOR DISTRIBUTIONS

BY LUKE TIERNEY[1]

University of Minnesota

A little history

Bootstrapping (late 20th C) - approximate complex probability distributions

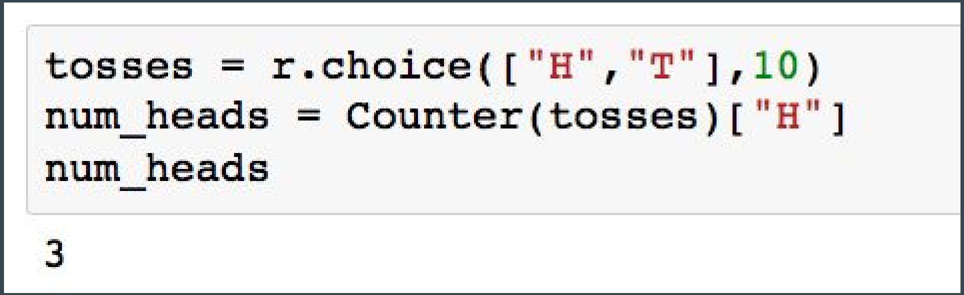

Efron

Imagery from wikipedia

Monte Carlo

Simple Example, coin flipping

Let’s toss a coin ten times, how many heads are there?

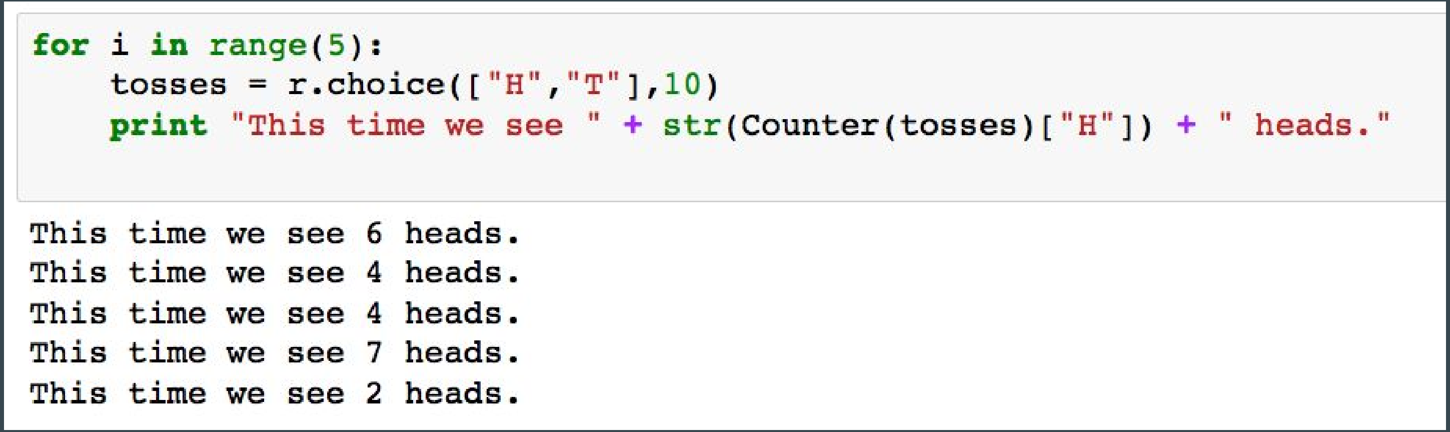


```
tosses = r.choice(["H","T"],10)
num_heads = Counter(tosses)["H"]
num_heads
```

```
3
```

Simple Example, coin flipping

Let’s toss a coin ten times, how many heads are there?

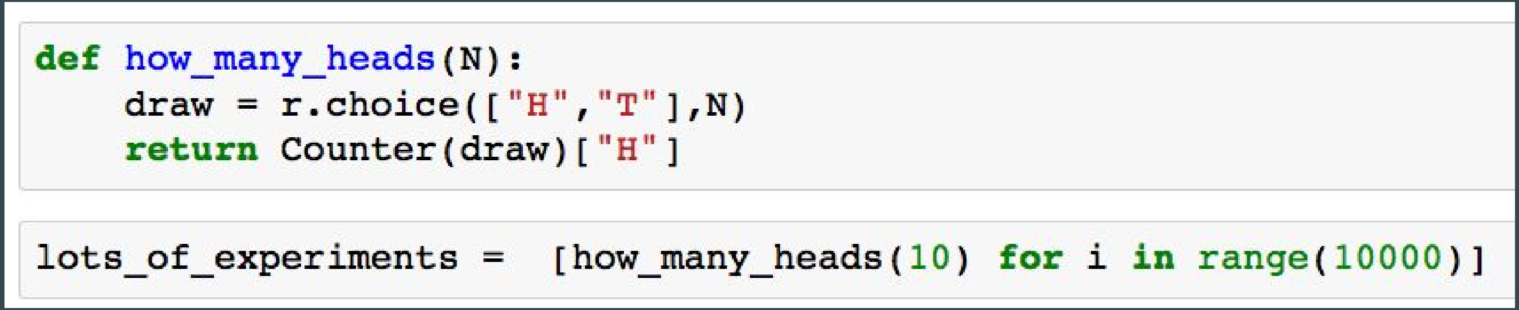


```
for i in range(5):
    tosses = r.choice(["H","T"],10)
    print "This time we see " + str(Counter(tosses)["H"]) + " heads."
```

```
This time we see 6 heads.
This time we see 4 heads.
This time we see 4 heads.
This time we see 7 heads.
This time we see 2 heads.
```

Simple Example, coin flipping

What’s the set of all possible outcomes we might see? Well, let’s just keep doing these experiments. And let’s do a lot of them.

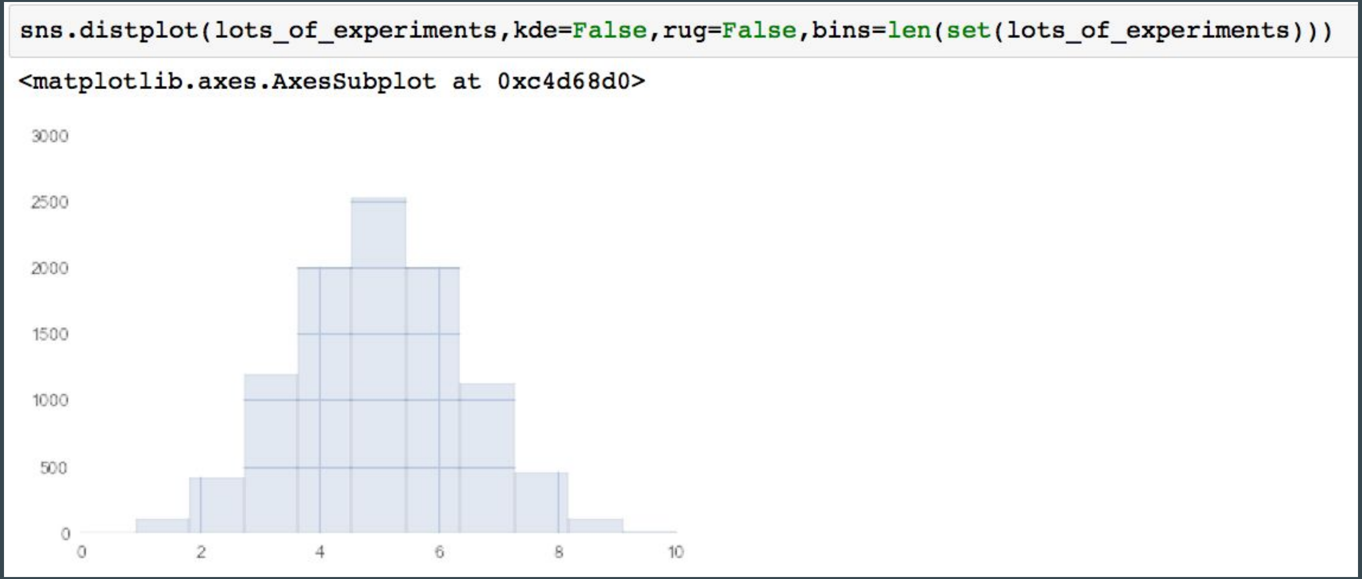


```
def how_many_heads(N):
    draw = r.choice(["H","T"],N)
    return Counter(draw)["H"]
```

```
lots_of_experiments =  [how_many_heads(10) for i in range(10000)]
```

Simple Example, coin flipping

What's the set of all possible outcomes we might see? Well, let's just keep doing these experiments. And let's do a lot of them.



```
sns.distplot(lots_of_experiments,kde=False,rug=False,bins=len(set(lots_of_experiments)))
```

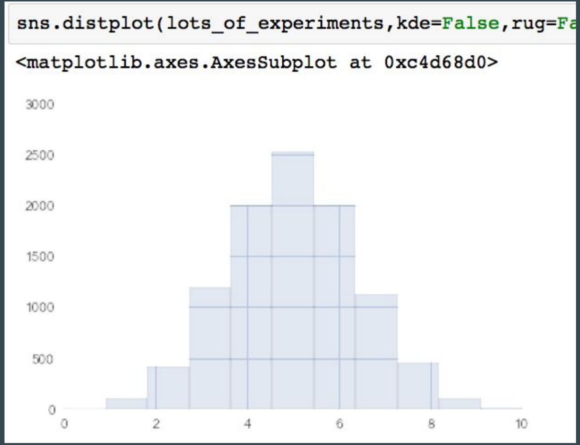
```
<matplotlib.axes.AxesSubplot at 0xc4d68d0>
```

Simple Example, coin flipping

This idea is the frequentist **Sampling Distribution**. If I repeated some experiment over and over, what set of outcomes might I see?

- Repeatedly conduct an experiment
- Summarize the result with some *test statistic*
    - Here it's the number of Heads
- Look at the distribution of the test statistics

Now we have a full notion of what is expected.



```
sns.distplot(lots_of_experiments,kde=False,rug=Fa
```

```
<matplotlib.axes.AxesSubplot at 0xc4d68d0>
```

A simple probability distribution

For a fair coin, the probability of Heads is .5, but we can be more general and consider not just a fair coin, but a coin with any amount of bias, θ.

So p(H) = θ and p(T) = 1 - θ .

$$p(HH) = \theta * \theta$$

$$p(HT) = \theta * (1 - \theta)$$

$$p(TH) = (1 - \theta) * \theta$$

$$p(TT) = (1 - \theta) * (1 - \theta)$$

A simple probability distribution

From the four possible outcomes, we see that there is

1 outcome with 2 Heads

1 outcome with 2 Tails

2 outcomes with 1 Heads

$$p(TwoHeads) = \theta * \theta$$

$$p(OneHeads) = 2 * \theta * (1 - \theta)$$

$$p(TwoTails) = (1 - \theta) * (1 - \theta)$$

A simple probability distribution

We can generalize this pattern to any number of coin tosses and we arrive at the **Binomial Distribution**.

$$p(k|N) = \frac{N!}{k!(N-k)!}\theta^k(1-\theta)^{N-k}$$

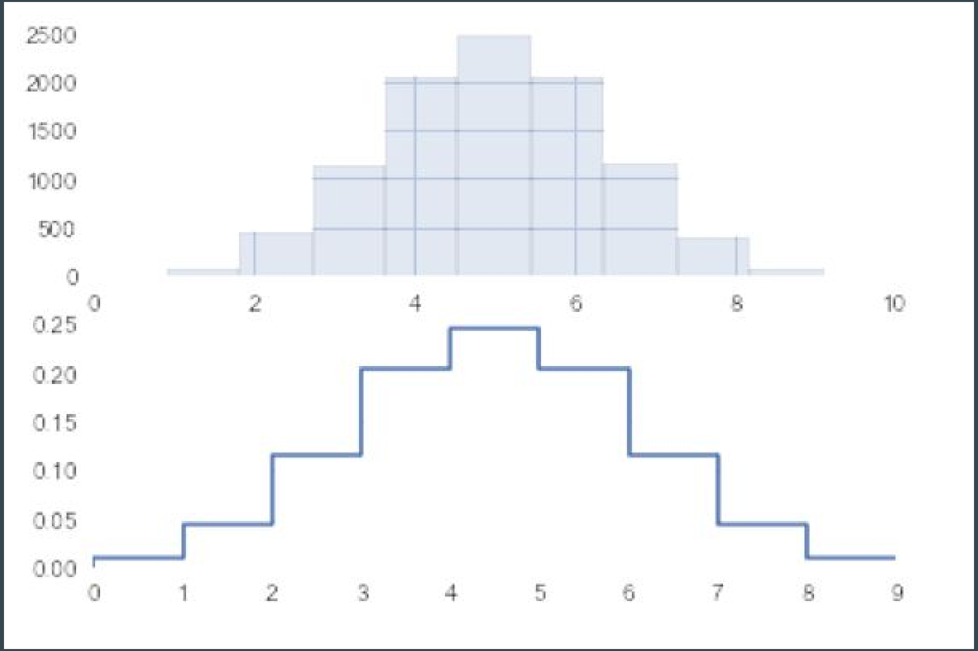
In our coin tossing game, the Binomial distribution happens to be the sampling distribution over the “number of heads” test statistic.

So what is Monte Carlo?

We've arrived at this sampling distribution in two ways

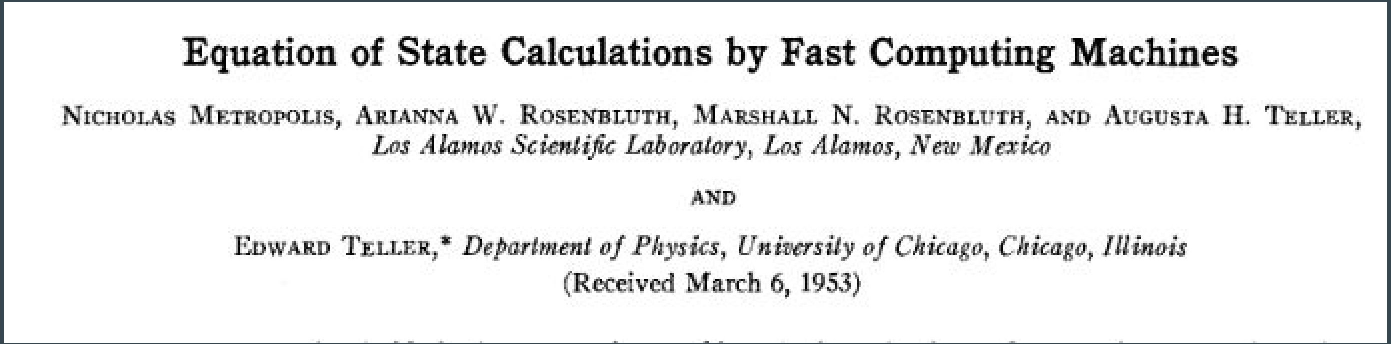
- Simulating repeated experiments

- Deriving the closed form



So what is Monte Carlo?

If we're unable to derive some complicated thing, we can use repeated simulation and random number generation to approximate it.



Equation of State Calculations by Fast Computing Machines

NICHOLAS METROPOLIS, ARIANNA W. ROSENBLUTH, MARSHALL N. ROSENBLUTH, AND AUGUSTA H. TELLER,
Los Alamos Scientific Laboratory, Los Alamos, New Mexico

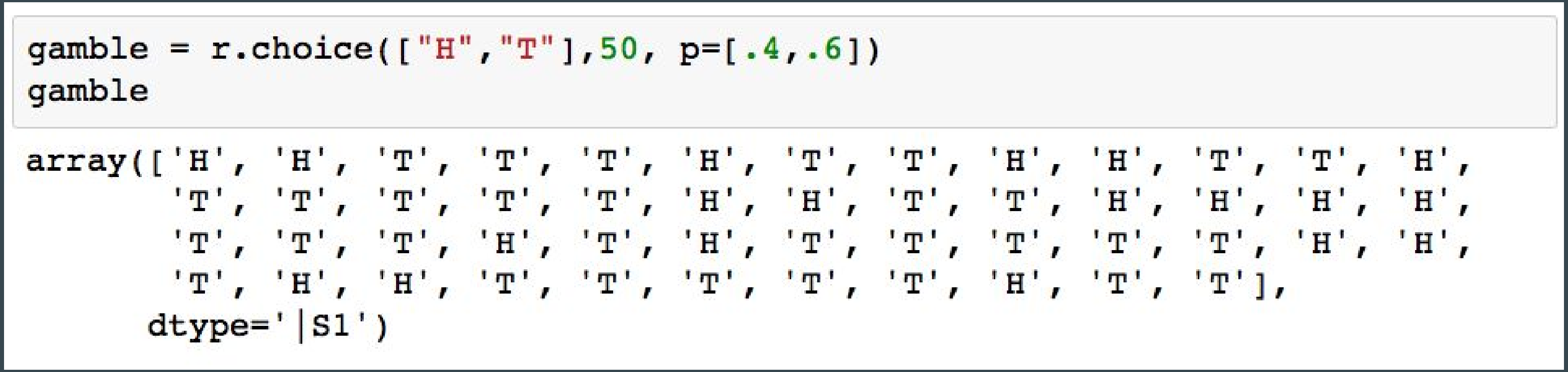
AND

EDWARD TELLER,* *Department of Physics, University of Chicago, Chicago, Illinois*
(Received March 6, 1953)

Computers are fast and for-loops are easy. We know the details of the data-generating process. Just simulate it. Over and over.

A Gambling Game and a Decision

So let's put all these ideas back together. Suppose we're playing some game and we care deeply about the outcomes of the tosses. We start to expect that our opponent is cheating and has an unfair coin. How can we quantify this idea from some observations?

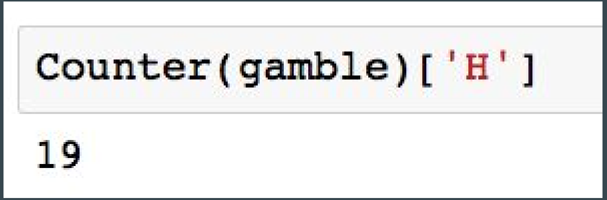


```
gamble = r.choice(["H","T"],50, p=[.4,.6])
gamble
```

```
array(['H', 'H', 'T', 'T', 'T', 'H', 'T', 'T', 'H', 'H', 'T', 'T', 'H',
       'T', 'T', 'T', 'T', 'T', 'H', 'H', 'T', 'T', 'H', 'H', 'H', 'H',
       'T', 'T', 'T', 'H', 'T', 'H', 'T', 'T', 'T', 'T', 'T', 'H', 'H',
       'T', 'H', 'H', 'T', 'T', 'T', 'T', 'T', 'H', 'T', 'T'],
      dtype='|S1')
```

A Gambling Game and a Decision

Here’s how many Heads we saw:

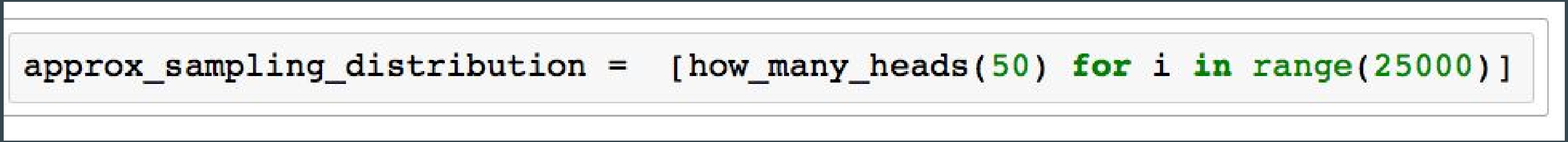


```
Counter(gamble)['H']
```

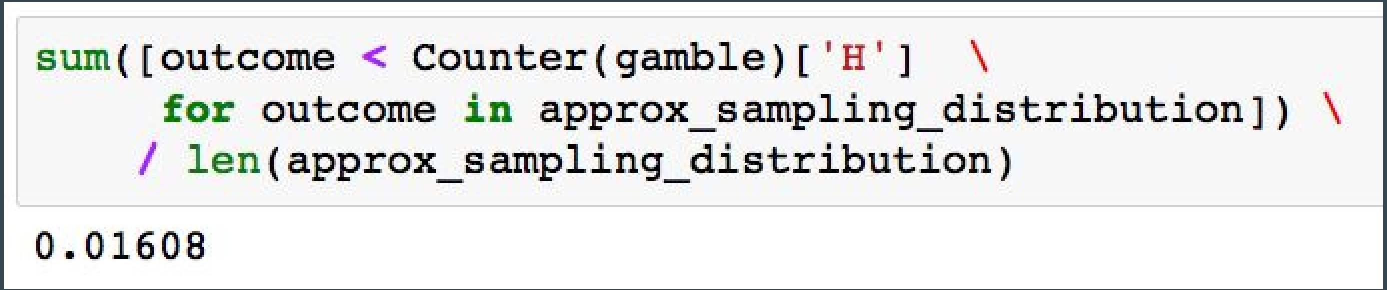
```
19
```

That certainly doesn’t seem like less that we’d expect. But how weird is it really?

A Gambling Game and a Decision



```
approx_sampling_distribution =  [how_many_heads(50) for i in range(25000)]
```



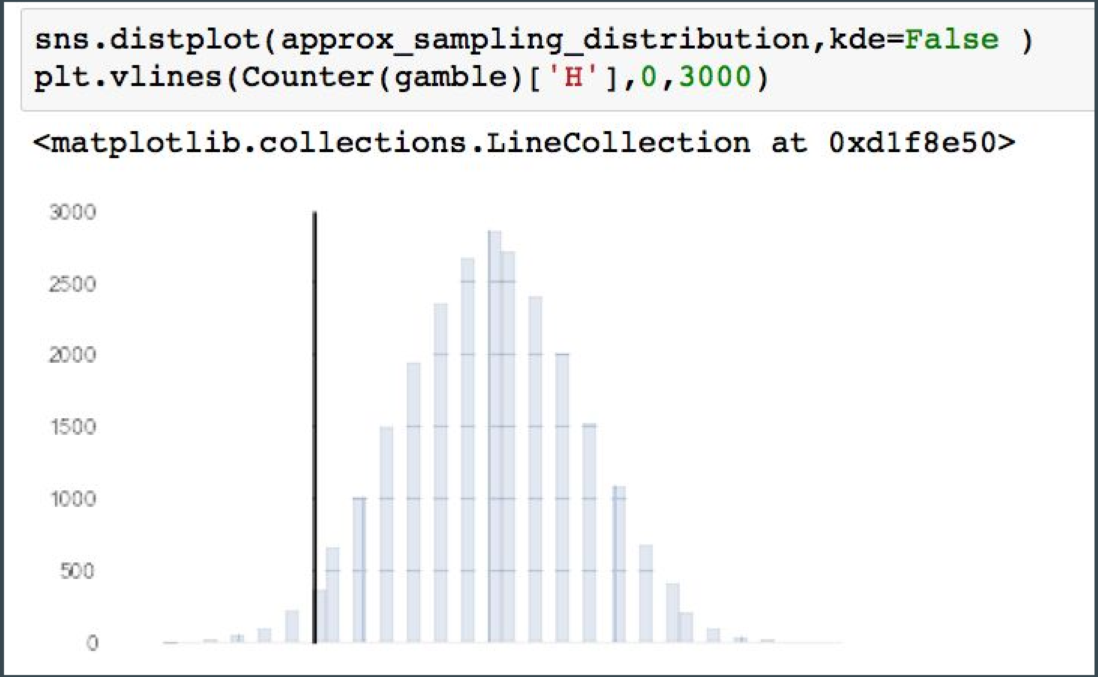
```
sum([outcome < Counter(gamble)['H']  \
     for outcome in approx_sampling_distribution]) \
    / len(approx_sampling_distribution)
```

```
0.01608
```

A Gambling Game and a Decision

So we can see how unlikely our actual observation is compared to the full sampling distribution.

This should sound familiar, this is p-values!



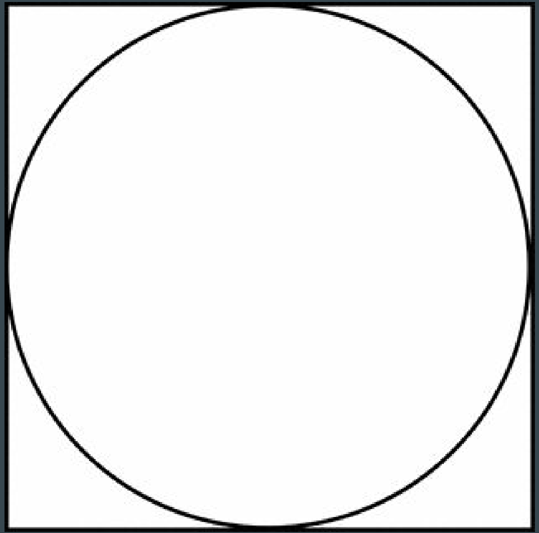
```
sns.distplot(approx_sampling_distribution,kde=False )
plt.vlines(Counter(gamble)['H'],0,3000)
```

```
<matplotlib.collections.LineCollection at 0xd1f8e50>
```

Estimating Pi

Here's a classic challenge in Monte Carlo sampling. We want to use random sampling to estimate the value of π.

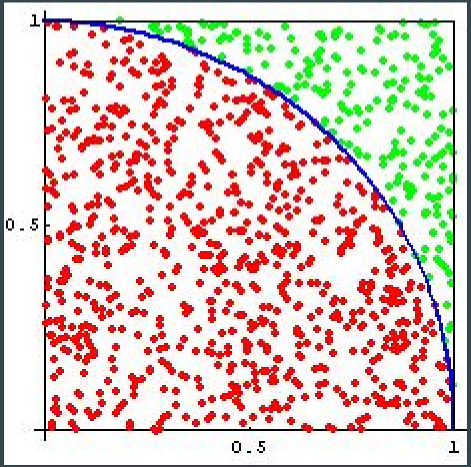
Imagine the unit circle inscribed in the unit square



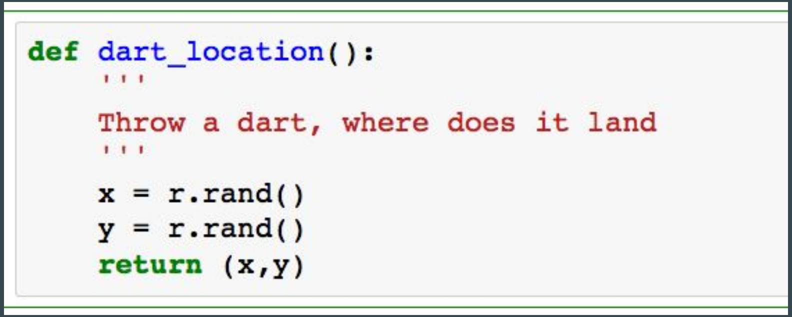
Estimating Pi

We proceed by throwing darts.

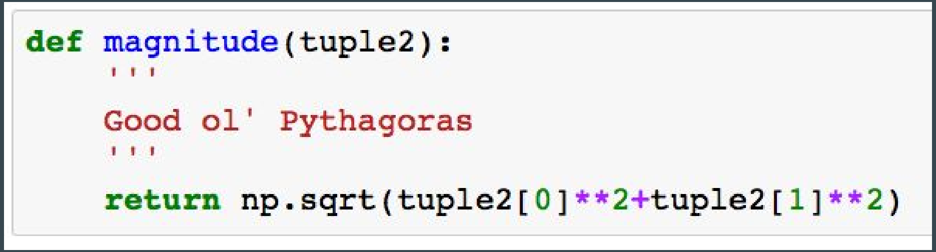
We ask how many darts land inside the circle, and how many land outside. This ratio should approximate π since the area of the unit circle is π.



Estimating Pi

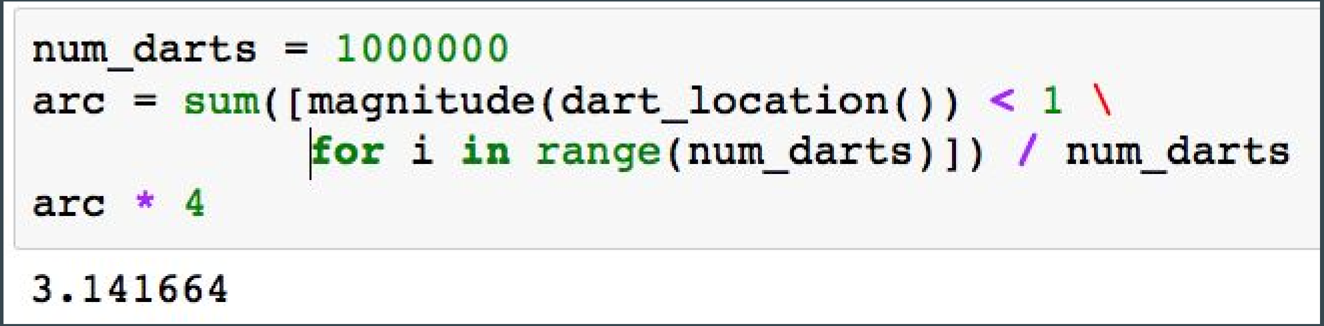


```
def dart_location():
    '''
    Throw a dart, where does it land
    '''
    x = r.rand()
    y = r.rand()
    return (x,y)
```



```
def magnitude(tuple2):
    '''
    Good ol' Pythagoras
    '''
    return np.sqrt(tuple2[0]**2+tuple2[1]**2)
```

Estimating Pi



```
num_darts = 1000000
arc = sum([magnitude(dart_location()) < 1 \
           for i in range(num_darts)]) / num_darts
arc * 4
```

```
3.141664
```

We could have viewed this as a calculus problem, where we're comparing two integrals (areas of two shapes). What we've done is use random sampling to approximate an integral - **Monte Carlo Integration**

$$\int f(x)dx \approx \frac{1}{N}\sum_{i=1}^{N} x_i$$

where $x_i \sim f(x)$

We'll see this idea again later

Summary

Whatever complex thing we're thinking about, whether it's coins and dice, or particle physics, or the stock market, we can pretty easily write a for-loop to get a sense of the **most probable** outcomes of that process.

Bootstrapping

Sampling Distribution

In the last section, we had our first encounter with the sampling distribution: What outcomes might I have seen if I repeated this experiment?

- We could make some assumptions about the data and then we arrive at simple expressions for sampling distributions
    - familiar stuff - ANOVA, Regression, t-tests
- If we know the data-generating process, we can directly simulate lots of repeated experiments (Monte Carlo)

In the first case, we make strong assumptions about the data-generating process. In the second case, we can consider more flexible models and just simulate the data.

What if we have totally no idea about the data generating process?

Bootstrapping

We don’t know anything about the data-generating process and we don’t want to make any assumptions. Yet we want to estimate the sampling distribution in order to calculate our confidence about something.

This seemingly impossible task is the purview of **Bootstrap Sampling**.

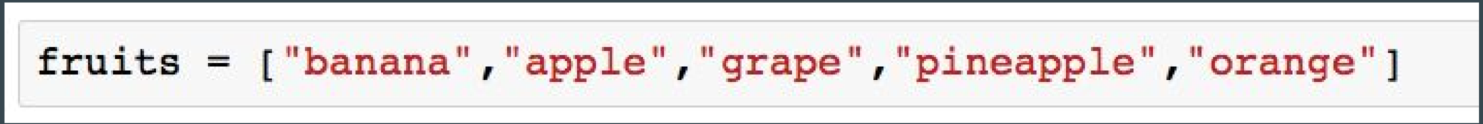
Bootstrapping

- Mimic the notion of repeated experimentation by simply *resampling* the data in a particular way
- We can resample the observed data in order to generate a synthetic data set
- We can do this many times and have a large collection of synthetic data sets
- Then we can easily compute a sampling distribution or confidence intervals or whatever we're interested in

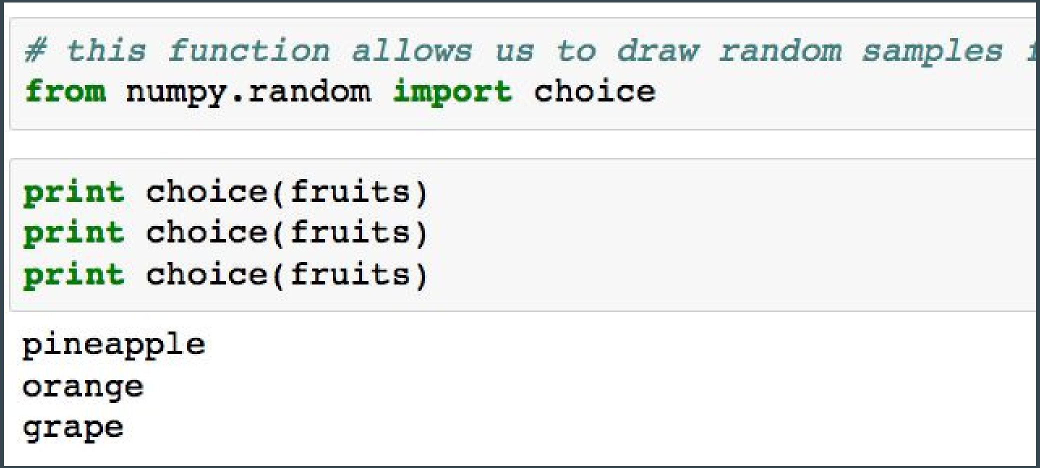
Hard to believe, but this process allows us mimic the "repeated experimentation" idea that's at the heart of the frequentist sampling distribution. And we can do this without making any assumptions about the data.

Resample With Replacement

The primary driver of Bootstrapping is a procedure for random sampling called Sample with Replacement. Let’s start with a toy example



```
fruits = ["banana","apple","grape","pineapple","orange"]
```



```
# this function allows us to draw random samples f
from numpy.random import choice
```

```
print choice(fruits)
print choice(fruits)
print choice(fruits)
```

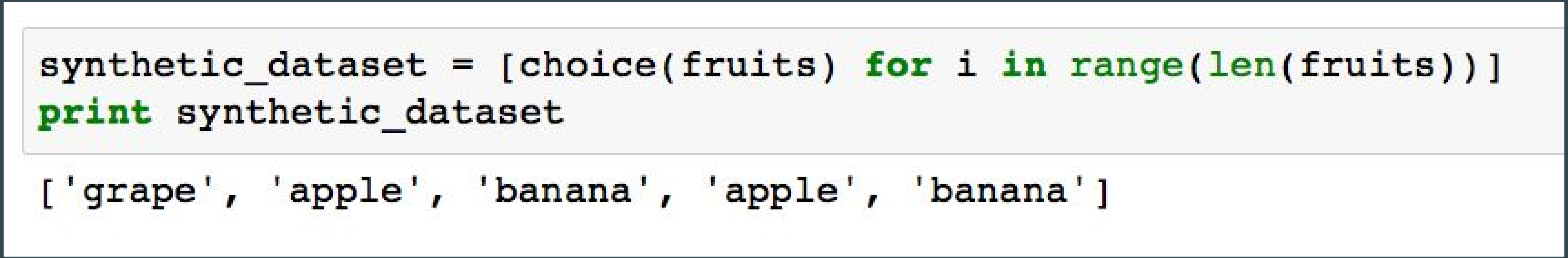
```
pineapple
orange
grape
```

Resample With Replacement

When we want to sample with replacement, we simply need to draw one item at a time, jot down that item, and put it back in the collection. We then redraw and repeat. This process is continued until our “synthetic” collection is the same size as the original.

We can equivalently think about it as simply drawing one sample at a time from the original collection and jotting down that result. Since we never actually change/remove anything from the original collection, this is the same as “replacing” each draw.

Resample With Replacement

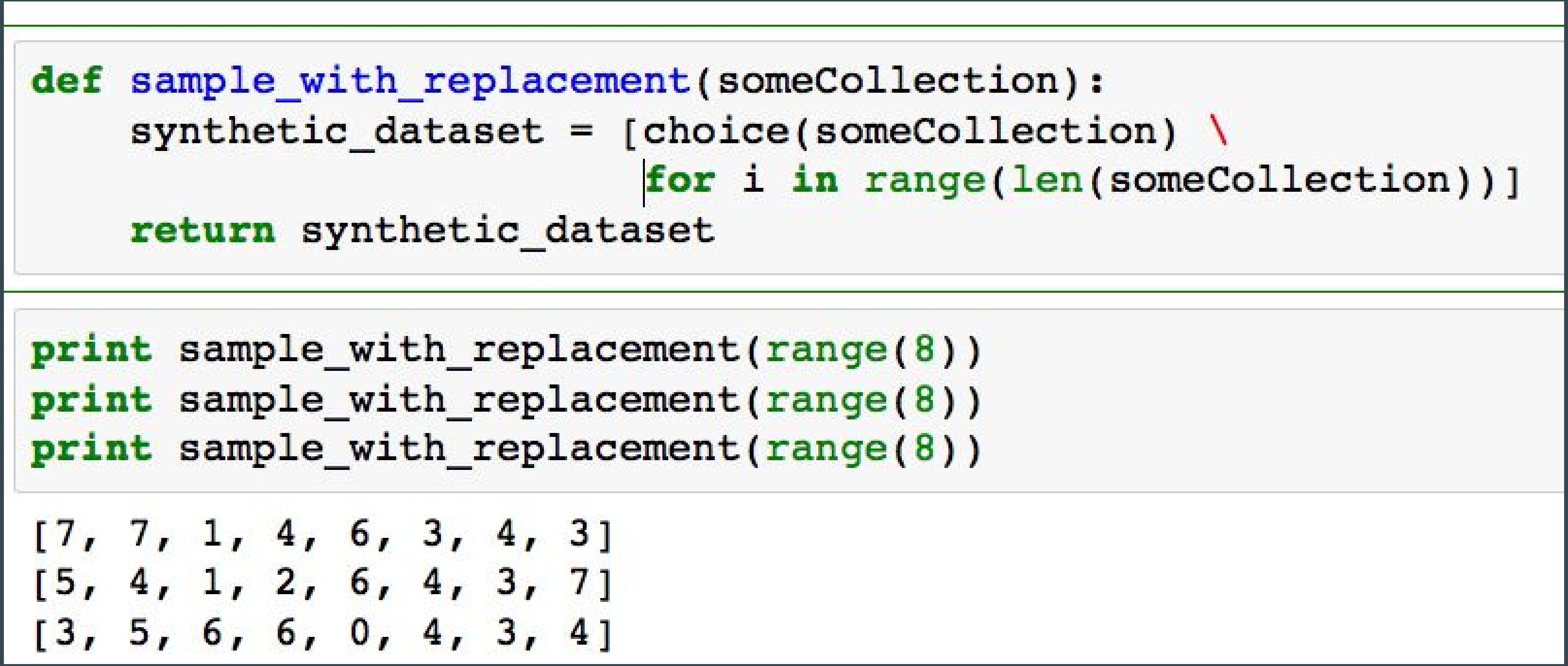


```
synthetic_dataset = [choice(fruits) for i in range(len(fruits))]
print synthetic_dataset
```

```
['grape', 'apple', 'banana', 'apple', 'banana']
```

Note that the synthetic dataset is unlikely to be identical to the original. And it make have repeated occurrences of some elements of the original.

Resample With Replacement



```
def sample_with_replacement(someCollection):
    synthetic_dataset = [choice(someCollection) \
                         for i in range(len(someCollection))]
    return synthetic_dataset
```

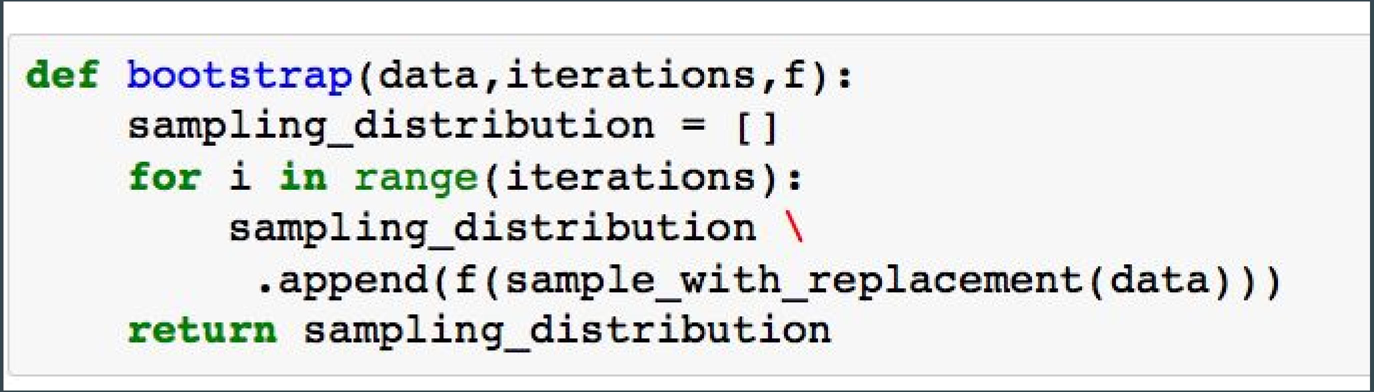
```
print sample_with_replacement(range(8))
print sample_with_replacement(range(8))
print sample_with_replacement(range(8))
```

```
[7, 7, 1, 4, 6, 3, 4, 3]
[5, 4, 1, 2, 6, 4, 3, 7]
[3, 5, 6, 6, 0, 4, 3, 4]
```

Bootstrap

- That’s really all we need for bootstrapping
- By using resample with replacement, we generate new synthetic datasets
- We can repeat this process many times and then compute the test statistic (or whatever we’re interested in) for each time

Bootstrap



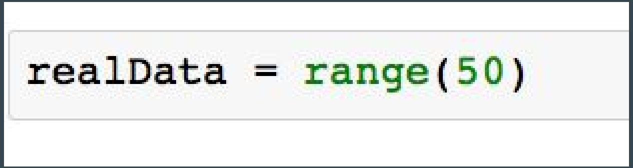
```
def bootstrap(data,iterations,f):
    sampling_distribution = []
    for i in range(iterations):
        sampling_distribution \
         .append(f(sample_with_replacement(data)))
    return sampling_distribution
```

This function takes in three arguments:

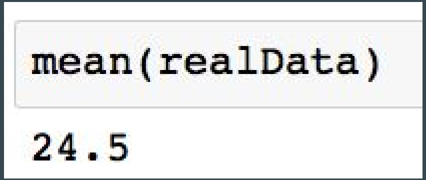
- some data collections
- the desired number of bootstrap samples
- a function that allows us to calc. the test statistics from a dataset

Example

To demonstrate how this function might work, here’s some data



```
realData = range(50)
```

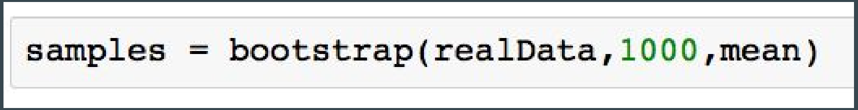


```
mean(realData)
```

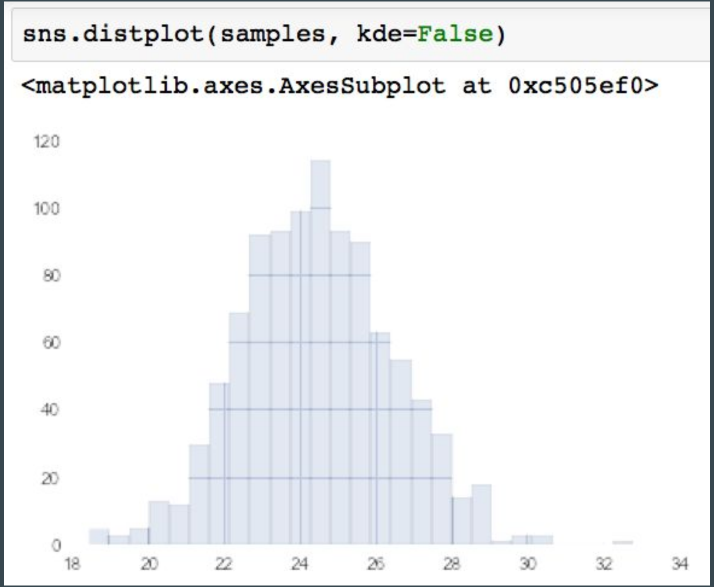
```
24.5
```

Example

Now we ask about the sampling distribution of that mean but draw a bunch of bootstrap samples from this data



```
samples = bootstrap(realData,1000,mean)
```



```
sns.distplot(samples, kde=False)
```

```
<matplotlib.axes.AxesSubplot at 0xc505ef0>
```

Comparison

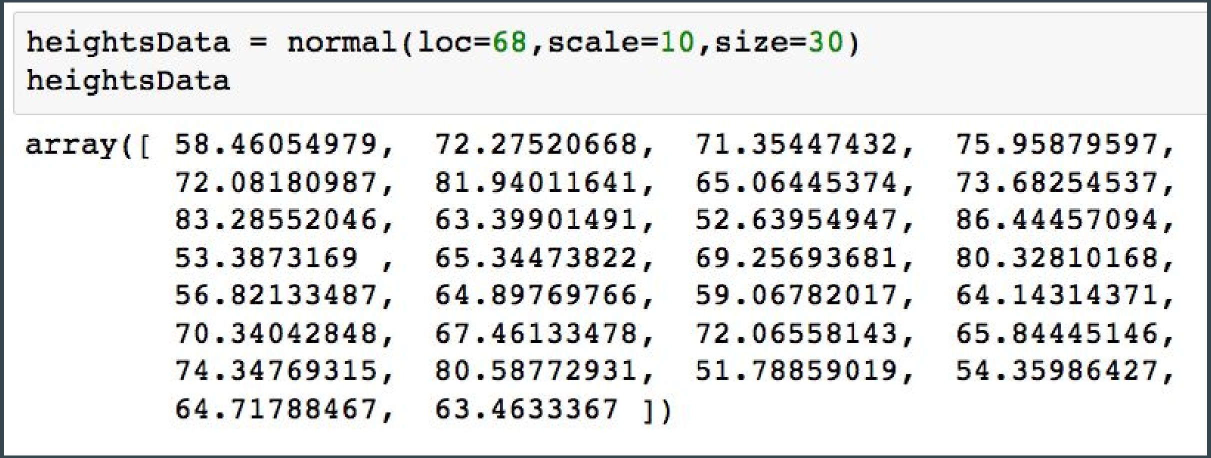
As a another example, let’s look at something simple where we can consult the “true” answer to verify our methods.

We’re going to look the mean of a sample of data points. We can use three methods to estimate the sampling distribution of the sample mean.

- Derived expression from classical stats
- Monte Carlo simulation
- Bootstrap sampling

Comparison

Here's our dataset in this problem - imagine we've gone out into the world and measured the height (in inches) of 30 people.

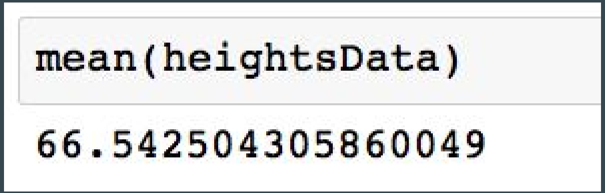


```
heightsData = normal(loc=68,scale=10,size=30)
heightsData
```

```
array([ 58.46054979,  72.27520668,  71.35447432,  75.95879597,
        72.08180987,  81.94011641,  65.06445374,  73.68254537,
        83.28552046,  63.39901491,  52.63954947,  86.44457094,
        53.3873169 ,  65.34473822,  69.25693681,  80.32810168,
        56.82133487,  64.89769766,  59.06782017,  64.14314371,
        70.34042848,  67.46133478,  72.06558143,  65.84445146,
        74.34769315,  80.58772931,  51.78859019,  54.35986427,
        64.71788467,  63.4633367 ])
```

Comparison

This sample has a mean. Easy to compute.

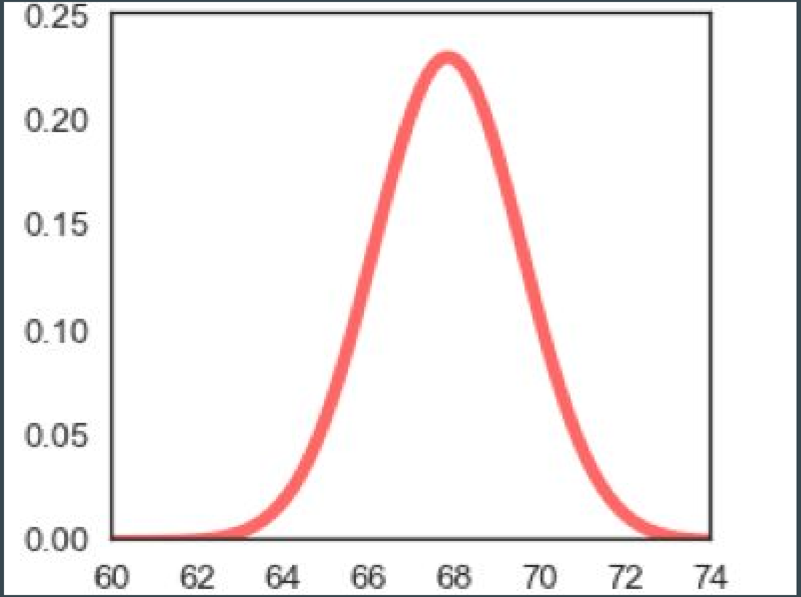


```
mean(heightsData)

66.542504305860049
```

But what if we had measured a different 30 people? How different would this mean be? How confident are we in this sample mean as a good proxy for the true population mean?

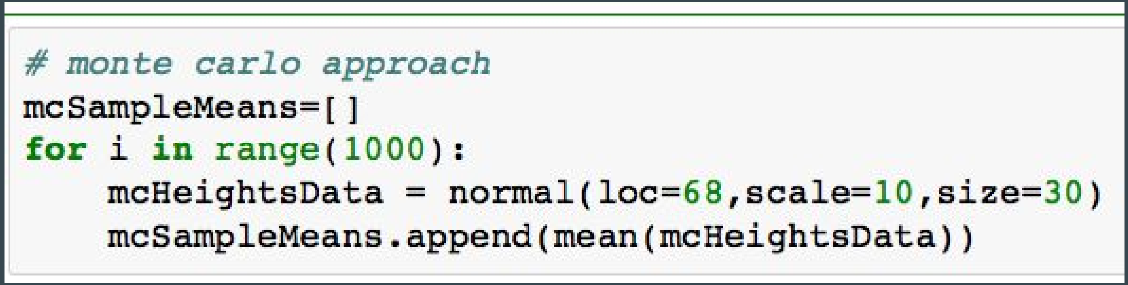
Comparison

Classical statistics and Normal theory says that the sampling distribution of the sample mean is a Normal distribution centered at the observed sample mean and with variance given by $\sigma_M^2 = \frac{\sigma^2}{N}$

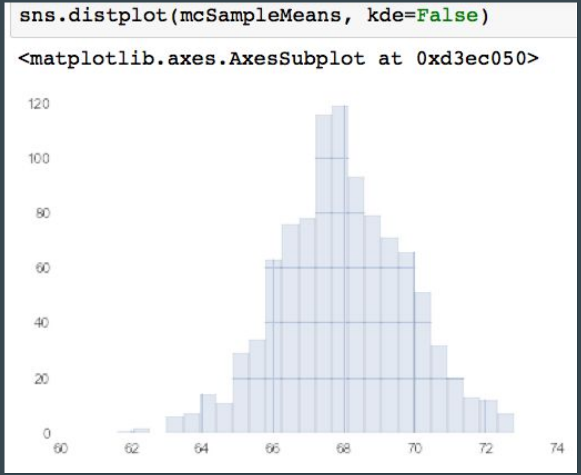


Comparison

We can use Monte Carlo as well. In this case, we just redo that draw of 30 samples. We do that draw over and over and again and jot down the sample mean each time.



```
# monte carlo approach
mcSampleMeans=[]
for i in range(1000):
    mcHeightsData = normal(loc=68,scale=10,size=30)
    mcSampleMeans.append(mean(mcHeightsData))
```

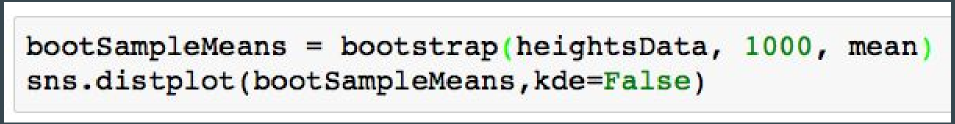


```
sns.distplot(mcSampleMeans, kde=False)
```

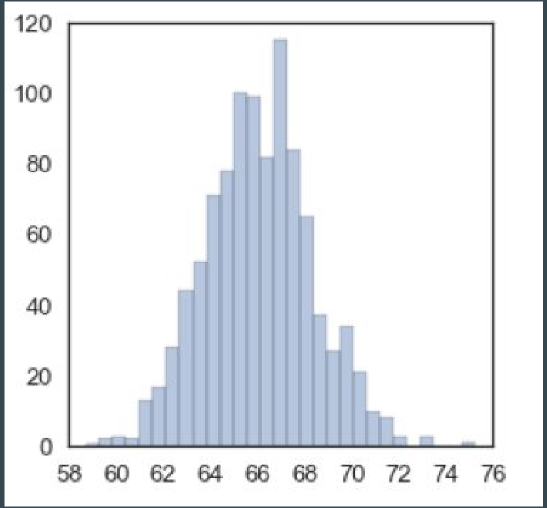
```
<matplotlib.axes.AxesSubplot at 0xd3ec050>
```

Comparison

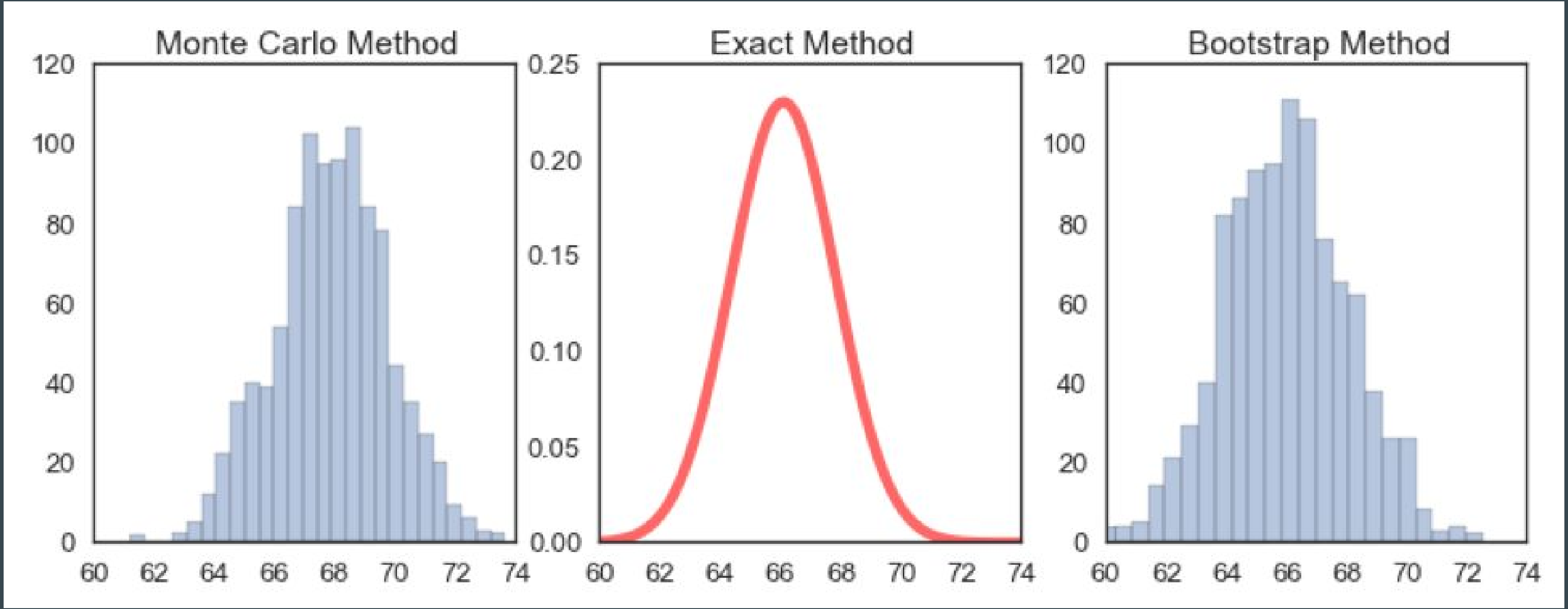
We can use Bootstrapping as well. In this case, we never resample new datasets like we did with Monte Carlo. Instead, we accept the dataset as fixed and use sample with replacement to sample from *it*. This is how we generate a large collection of synthetic datasets.



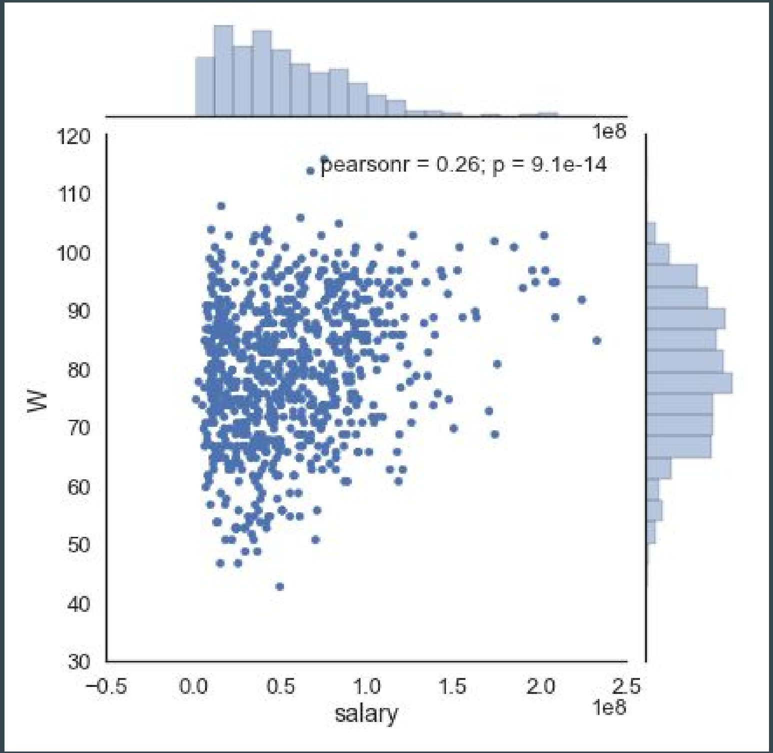
```
bootSampleMeans = bootstrap(heightsData, 1000, mean)
sns.distplot(bootSampleMeans,kde=False)
```



Comparison


Monte Carlo Method
Exact Method
Bootstrap Method
0
20
40
60
80
100
120
0.00
0.05
0.10
0.15
0.20
0.25
60
62
64
66
68
70
72
74

Another Example - Regression



With this baseball data, is there any meaningful correlation between total salary spend and yearly number of wins?

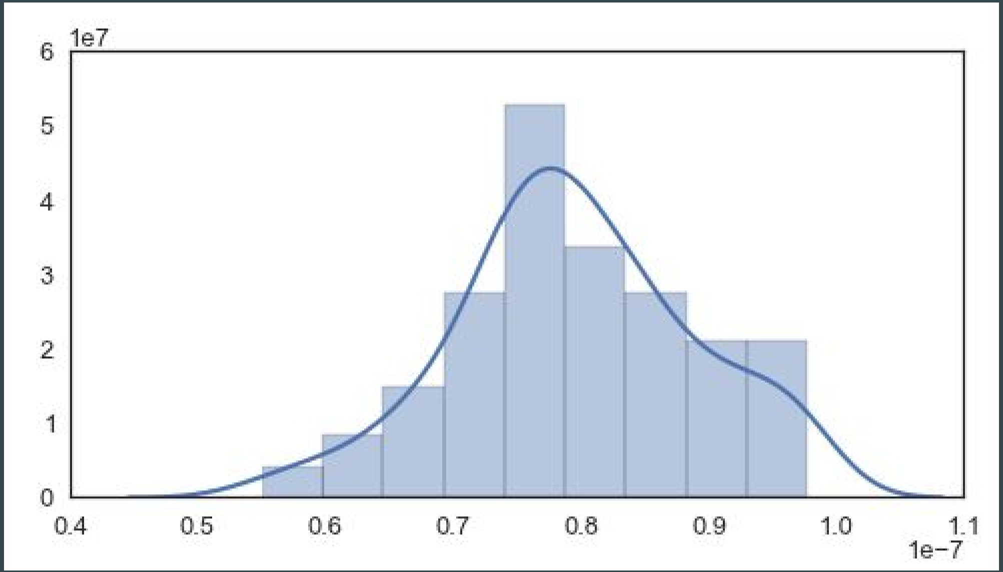
A linear regression yields a coefficient of 8e-08, a number just barely above 0.

This data is certainly non-Normal, can we trust that coefficient?

We use bootstrap to resample the rows of the dataset and perform a new regression each time.

Another Example - Regression

There is a tiny, but non-zero, effect here.



Bootstrapping Conclusions

- No model of the data-generating process

- We simply resample the observed data in order to estimate the sampling distribution.

- Can be applied to lots of kinds of models, no matter how complex (Decision Tree Bagging).

Bayesian Inference

Outline

- Frequentist vs Bayes
- A Motivating Example
- Bayes' Rule
- Conjugate Priors

Frequentist and Bayes

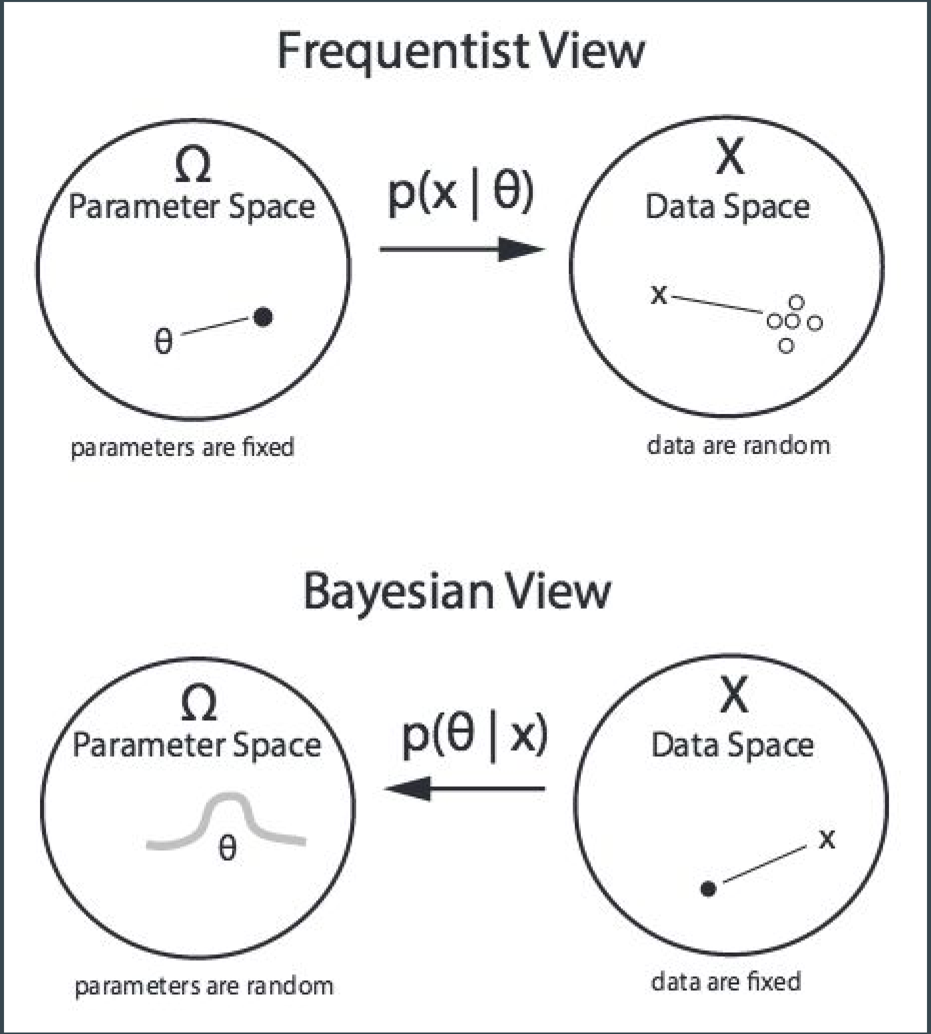
At this point, we're going to take a slight departure in our terminology as we focus on the alternative philosophy of statistics: Bayesian Inference.

- **Frequentist** - if I repeated this experiment again, what might I see?
  sampling distribution
- **Bayesian** - I don't care about hypothetical repeated experiments. This is all the data I have, so what can I conclude about the world?
  posterior distribution

Frequentist and Bayes

- **Frequentist** - if I repeated this experiment again, what might I see?
  sampling distribution

- **Bayesian** - I don't care about hypothetical repeated experiments. This is all the data I have, so what can I conclude about the world?
  posterior distribution



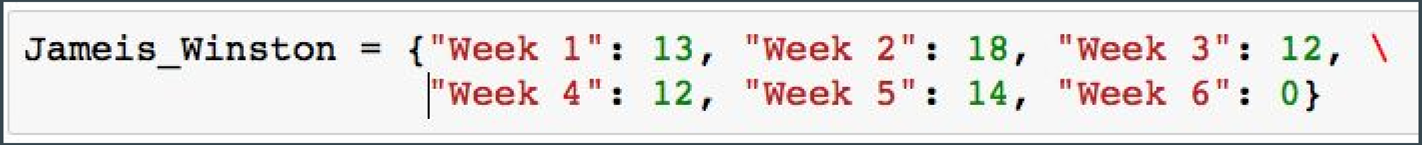
Imagery from Jonathan Pillow lecture notes

An Ongoing Example

As an introduction to Bayesian ideas, let's think about the game of Fantasy Football

- I have a "team" of players who score various points each week
- I can switch up my roster in order to maximize points

For example, quarterback Jameis Winston

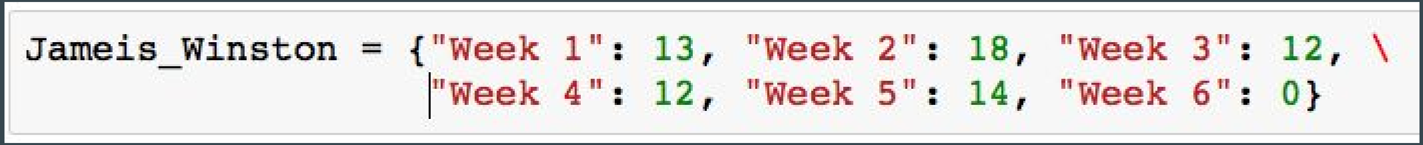


```
Jameis_Winston = {"Week 1": 13, "Week 2": 18, "Week 3": 12, \
                  "Week 4": 12, "Week 5": 14, "Week 6": 0}
```

Fantasy Football

From this data, what can say about Winston? What can we guess and estimate?

- A pretty good source of about a dozen points each week
- Sometimes more (like 18), sometimes less (like 0)



```
Jameis_Winston = {"Week 1": 13, "Week 2": 18, "Week 3": 12, \
                  "Week 4": 12, "Week 5": 14, "Week 6": 0}
```

We'd like to be able to *predict* how many points he'll get next week. It probably won't be 50, and it probably won't be 0 again, so how can we quantify this?

Fantasy Football

We need to come up a *generative* model of how points are scored. We need to specify our guess about the data-generating-process.

- Normal distribution? doesn't quite seem right
- Coin flipping? also not a good fit

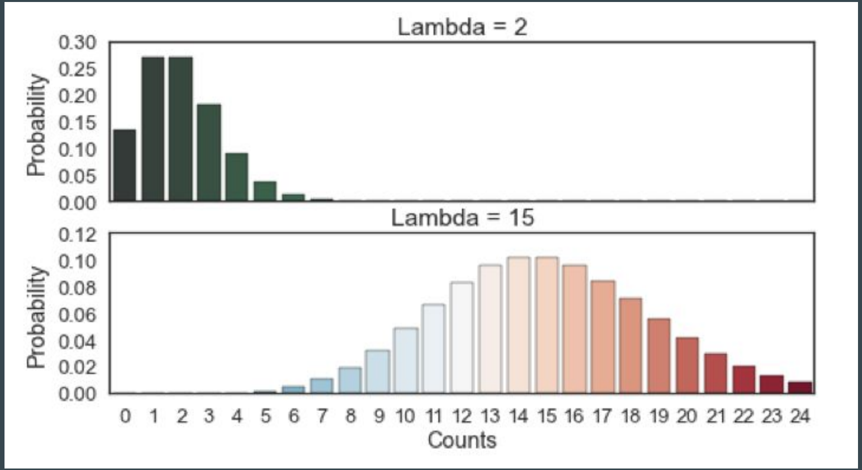
What we're looking for is a probability distribution over *count data*.

Poisson Distribution

$$p(k|\lambda) = \frac{\lambda^k}{k!}e^{-\lambda}$$

The Poisson Distribution is a distribution over count data, for example:

- How many people will attend my party?
- How many times will I sneeze tomorrow?
- How many points will Jameis Winston get next week?



Poisson Distribution

This is a good start, but how can we use it? Here's the idea

- The number of points Winston gets each week is a random draw from some Poisson distribution with unknown λ
- We use the data we've seen in order to estimate λ properly
- Then we can have a full quantification about our predictions for next week

Bayes' Rule

We have observed the first week of the season, let's call this y_1.

We need to quantify our uncertainty about λ, ie $p(\lambda \mid y_1)$.

Bayes' Rule says

$$p(\lambda | y_1) = \frac{p(y_1 | \lambda) p(\lambda)}{p(y_1)}$$

Bayes' Rule has four components

The posterior : $p(\lambda | y_1)$

The likelihood : $p(y_1 | \lambda)$

The prior : $p(\lambda)$

The marginal : $p(y_1)$

Bayes' Rule

We can usually ignored the marginal (since it is independent of the parameters), so Bayes rule is often seen in a form like

$$p(\lambda \mid y) \propto p(y \mid \lambda)\, p(\lambda)$$

So what are these components all about?

Bayes' Rule

$p(\lambda \mid y)$ - the posterior - this is the thing we want to know, this is the point of Bayesian inference

$p(y \mid \lambda)$ - the likelihood - this is how data arises in our model, this is our data-generating process

$p(\lambda)$ - the prior - this part is new and unfamiliar. This is our guess about the value of λ before we ever gather any observations.

- For example, we have prior guesses that Winston is unlikely to score 100 points in a game, because no one does that.

Conjugate Priors

We know about likelihood and prior and that they form the basis of the posterior, but how do we pick the prior? Or how do we even pick what family of functions to use for the prior?

If we pick the prior with some care and cleverness, then the posterior turns out to be super simple and useful. The prior will combine very easily with the likelihood and the posterior will have the same functional form as the prior.

Such a prior is called a Conjugate Prior

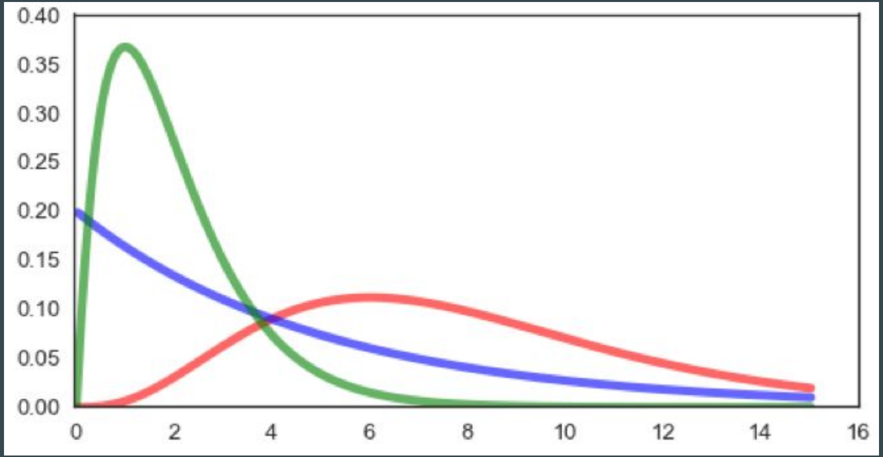
(Useful table here https://en.wikipedia.org/wiki/Conjugate_prior)

Gamma Distribution

$$p(\lambda|\alpha,\beta) = \frac{\beta^\alpha}{\Gamma(\alpha)}\lambda^{\alpha-1}e^{-\beta\lambda}$$

Here's a new distribution to learn about. As you might guess, it forms a useful conjugate prior when we have a Poisson likelihood.

The Gamma distribution is defined over the positive real numbers and has two parameters (a shape parameter and a scale parameter). Here's a few different Gamma distributions



Conjugate Prior

Returning to Bayes Rule,

$$p(\lambda|y) \propto p(y|\lambda)p(\lambda)$$

Now we can just fill in the components for likelihood (Poisson) and prior (Gamma)

$$p(\lambda|y) \propto \frac{\lambda^y}{y!}e^{-\lambda}\,\frac{\theta^{\alpha-1}e^{-\theta/\beta}}{\beta^{\theta}\Gamma(\theta)}.$$

With some (unpleasant) algebra, see that there's some terms from the prior and the likelihood that we can combine. This allows us to reduce things.

$$p(\lambda|y) \propto \lambda^{y+\alpha-1}e^{-\lambda(1+1/\beta)}$$

Conjugate Prior

You might notice that this form

$$p(\lambda|y) \propto \lambda^{y+\alpha-1} e^{-\lambda(1+1/\beta)}$$

is exactly the form of the Gamma distribution. Thus, our posterior distribution is also a Gamma distribution, but with new parameters..

$$p(\lambda|y) = \text{Gamma}(A, B)$$
$$\text{where } A = y + \alpha, B = \tfrac{\beta}{1+\beta}.$$

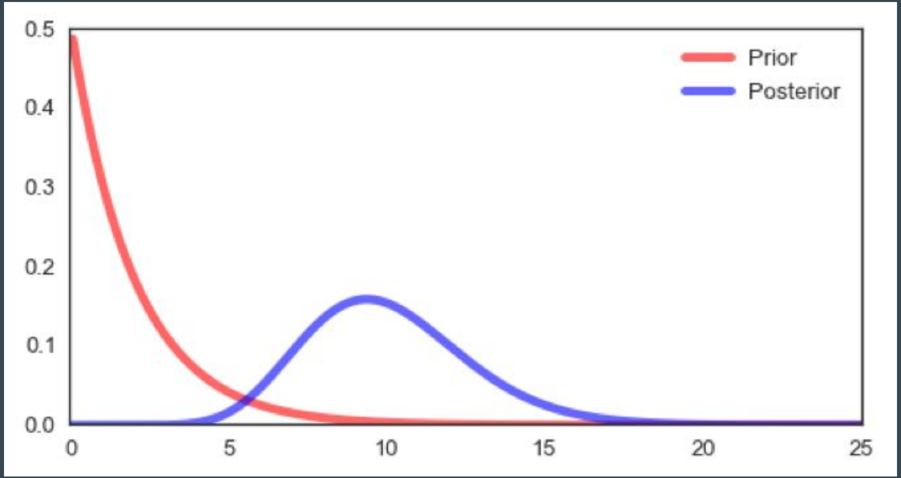
This is the point of conjugate priors - the posterior has a simple form with parameters that are easily calculated from the data.

Conjugate Prior

Now we can put this knowledge to work with our football data. Say after week one, Winston scored 13 points and we want to estimate λ. For our prior, let's say p (λ) = Gamma(1,2) - this gives us a broad distribution.

Now our posterior is

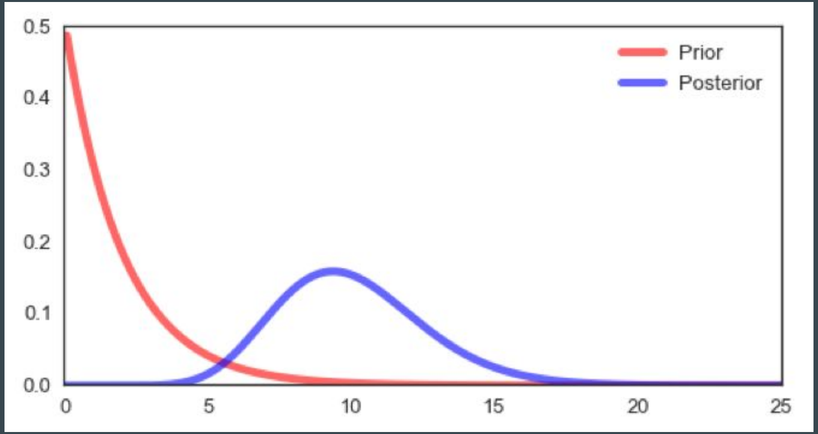
$$p(\lambda|y_1) = \text{Gamma}(y_1 + 1)(\tfrac{2}{1+2}).$$
$$= \text{Gamma}(14)(\tfrac{2}{3})$$



Bayes Rule - Before and After

Bayes rule is all about using newly observed data to update our beliefs about the world. This is how we update a prior into a posterior.

This process can be repeated as new things are observed and we constantly update our posterior.

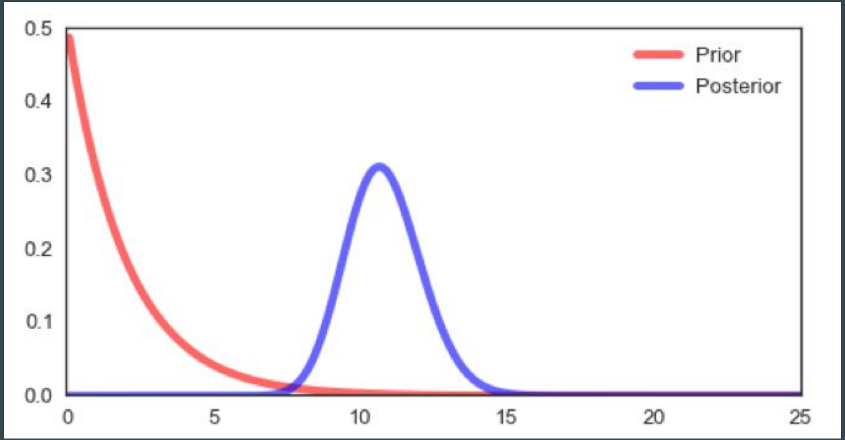


Conjugate Prior

We can also use all of our data at once to update the posterior in one step (this is more common). So the more sensible form for our Poisson-Gamma posterior would look at all 6 weeks of data.

$$p(\lambda | y_N) = \text{Gamma}(A)(B)$$

$$\text{where } A = \alpha + \textstyle\sum_i y_i, B = \frac{\beta}{N\beta+1}$$



Posterior Predictive Distribution

Now we have a pretty good estimate of λ, and the probility that it takes on various values. Let's predict the probability distribution over the number of points Winston scores next week.

This is called the Posterior Predictive Distribution, and in many cases there is a simple form for it.

However, let's combine what we learned earlier (Monte Carlo) to do this.

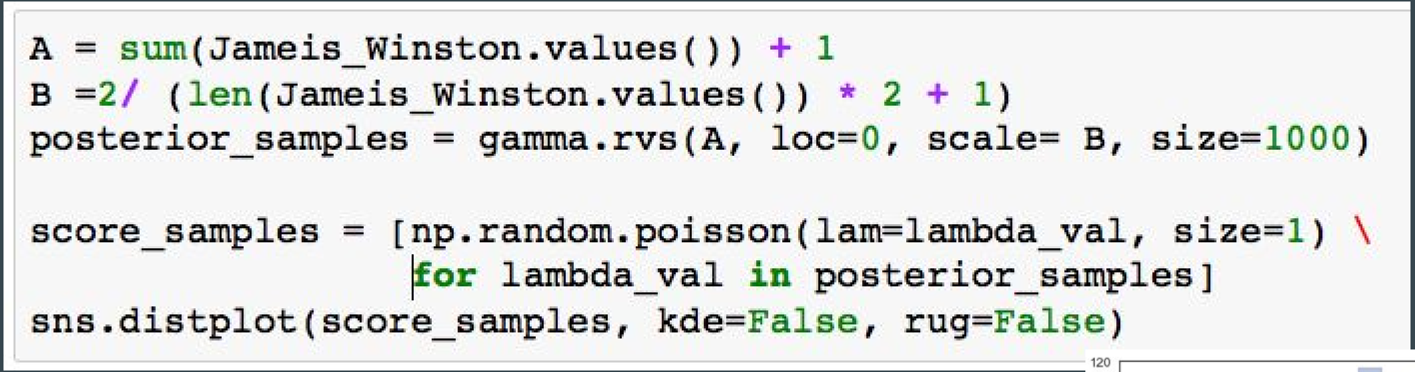
Posterior Predictive Distribution

We're going to simulate the data-generating process in the Monte Carlo way.

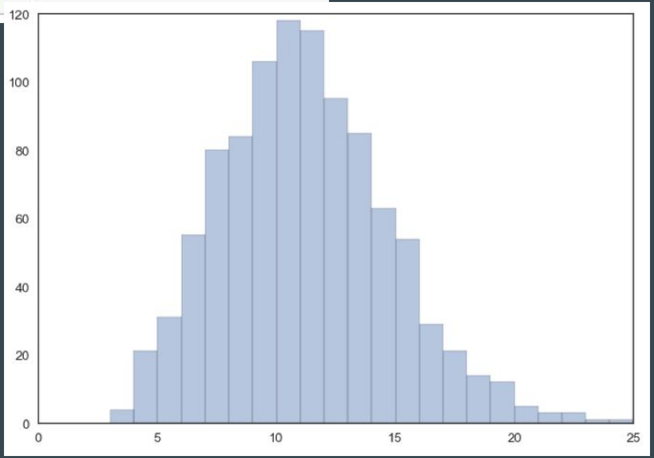
Since we know the distribution $p(\lambda|y_N)$, let's draw random samples from it. Then for each of those sample, called it λ_i, let's take a random draw from a Poisson distribution with parameter λ_i. This is equivalent to simulating the data-generating process for how many points Winston scores.

Then we aggregate all the Poisson draws in order to understand our full predictions over next week's points.

Posterior Predictive Distribution



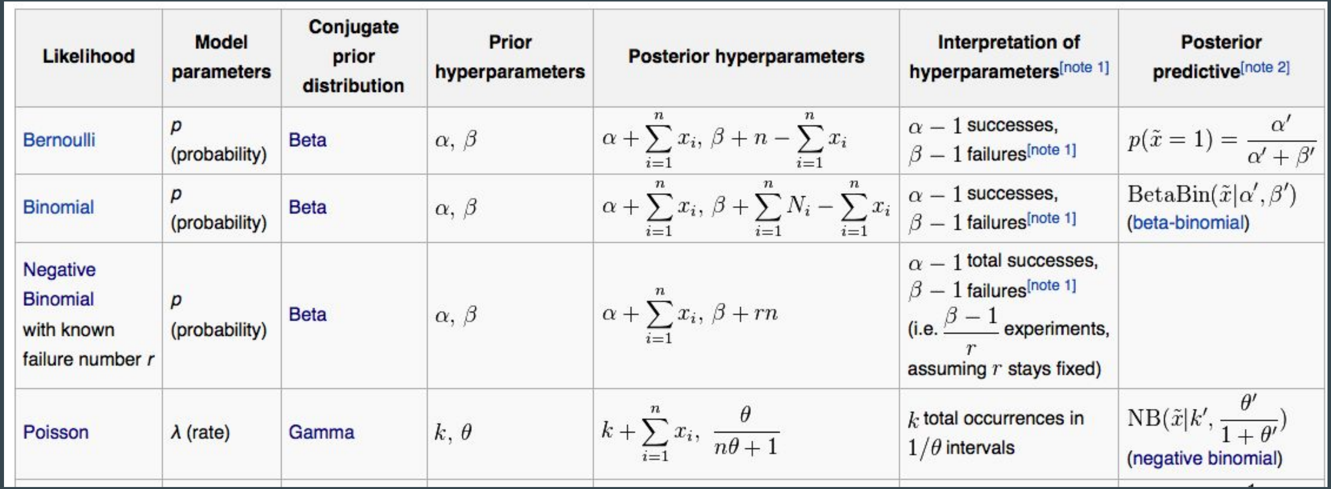
```
A = sum(Jameis_Winston.values()) + 1
B =2/ (len(Jameis_Winston.values()) * 2 + 1)
posterior_samples = gamma.rvs(A, loc=0, scale= B, size=1000)

score_samples = [np.random.poisson(lam=lambda_val, size=1) \
                 for lambda_val in posterior_samples]
sns.distplot(score_samples, kde=False, rug=False)
```



Conjugate Priors - Conclusion

Makes posterior inference super simple and straightforward.

For most basic models (and model components), a good conjugate prior has already been thought of. This table is hugely helpful https://en.wikipedia.org/wiki/Conjugate_prior



Likelihood	Model parameters	Conjugate prior distribution	Prior hyperparameters	Posterior hyperparameters	Interpretation of hyperparameters[note 1]	Posterior predictive[note 2]
Bernoulli	*p* (probability)	Beta	$\alpha,\, \beta$	$\alpha + \sum_{i=1}^n x_i,\, \beta + n - \sum_{i=1}^n x_i$	$\alpha - 1$ successes, $\beta - 1$ failures[note 1]	$p(\tilde{x} = 1) = \frac{\alpha'}{\alpha' + \beta'}$
Binomial	*p* (probability)	Beta	$\alpha,\, \beta$	$\alpha + \sum_{i=1}^n x_i,\, \beta + \sum_{i=1}^n N_i - \sum_{i=1}^n x_i$	$\alpha - 1$ successes, $\beta - 1$ failures[note 1]	$\operatorname{BetaBin}(\tilde{x}\|\alpha', \beta')$ (beta-binomial)
Negative Binomial with known failure number *r*	*p* (probability)	Beta	$\alpha,\, \beta$	$\alpha + \sum_{i=1}^n x_i,\, \beta + rn$	$\alpha - 1$ total successes, $\beta - 1$ failures[note 1] (i.e. $\frac{\beta - 1}{r}$ experiments, assuming r stays fixed)	
Poisson	λ (rate)	Gamma	$k,\, \theta$	$k + \sum_{i=1}^n x_i,\ \frac{\theta}{n\theta + 1}$	k total occurrences in $1/\theta$ intervals	$\operatorname{NB}(\tilde{x}\|k', \frac{\theta'}{1+\theta'})$ (negative binomial)

MCMC

Outline

- Posterior sampling
- Markov chains
- Conditional posteriors
- Gibbs sampling
- Metropolis-Hastings
- Python Libraries

So Now What?

With conjugate priors, with a pretty good arsenal of tools to use to tackle a variety of problems in the Bayesian way.

But as you might guess, we don’t have to stray too far before we devise some model or problem domain where conjugate priors are not enough.

To move forward, we’re going to discuss a family of computational methods, called Markov chain Monte Carlo, that allow us to approximate posterior distributions, no matter how complicated.

Posterior Sampling

Here’s the big idea about MCMC generally:

- We have some posterior distribution that is complicated and has no simple form (and has hundreds or thousands of parameters)
- If we could draw a bunch of random samples from the posterior, then we could use this finite set of samples to approximate the true posterior (this is Monte Carlo integration again)
- But if we don’t know anything about the posterior, how could possibly draw random samples from it?
- Big Idea: We construct a Markov chain whose limiting distribution is the posterior. Then we simply simulate this chain for a while.

We’re going to learn about two different ways for constructing such Markov chains.

Markov Chains

If we’re already in unfamiliar territory, let’s take a step back and review Markov chains.

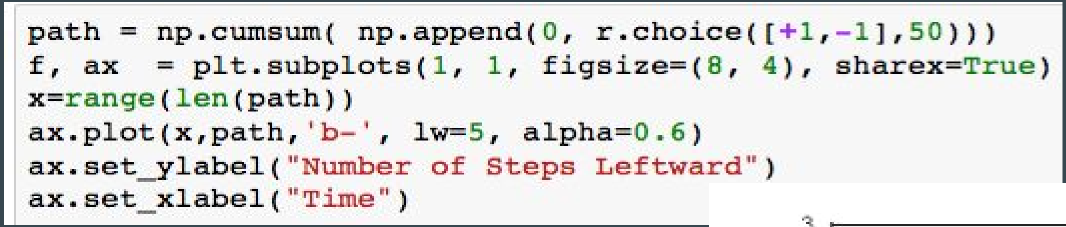
A (first-order) Markov chain is a random process whose future depends only on its present value. That is, given the present state, the past doesn’t matter (it doesn’t matter how we got to the present).

Random Walk

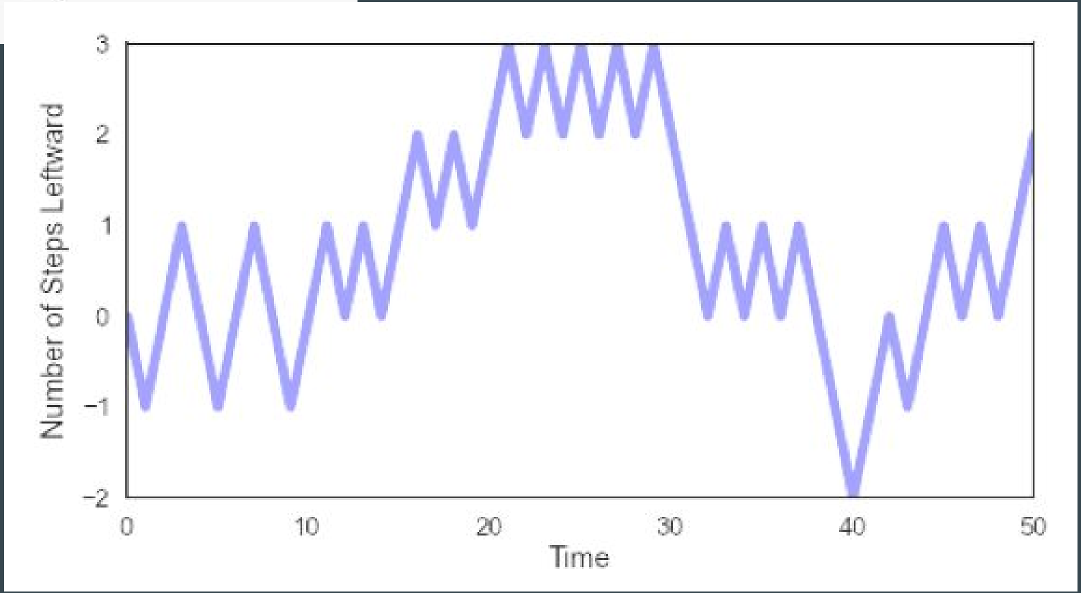
Physicists call this the Drunkard's Walk :

- We start at location 0
- At each time step, we move one step to the left or to the right with equal probability
- At each time step, we repeated this random choice from wherever our current position is

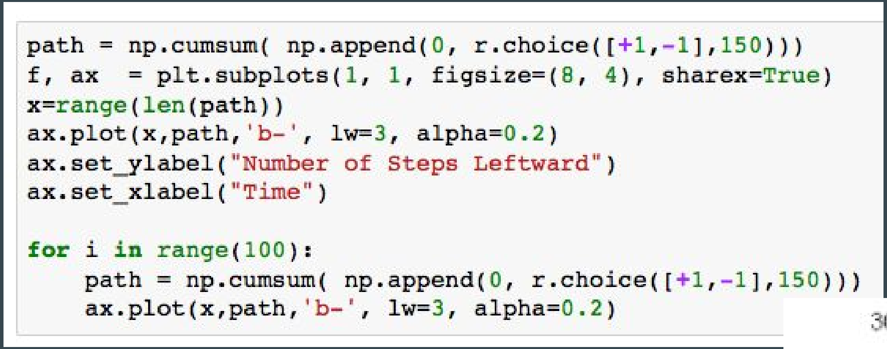
Random Walk



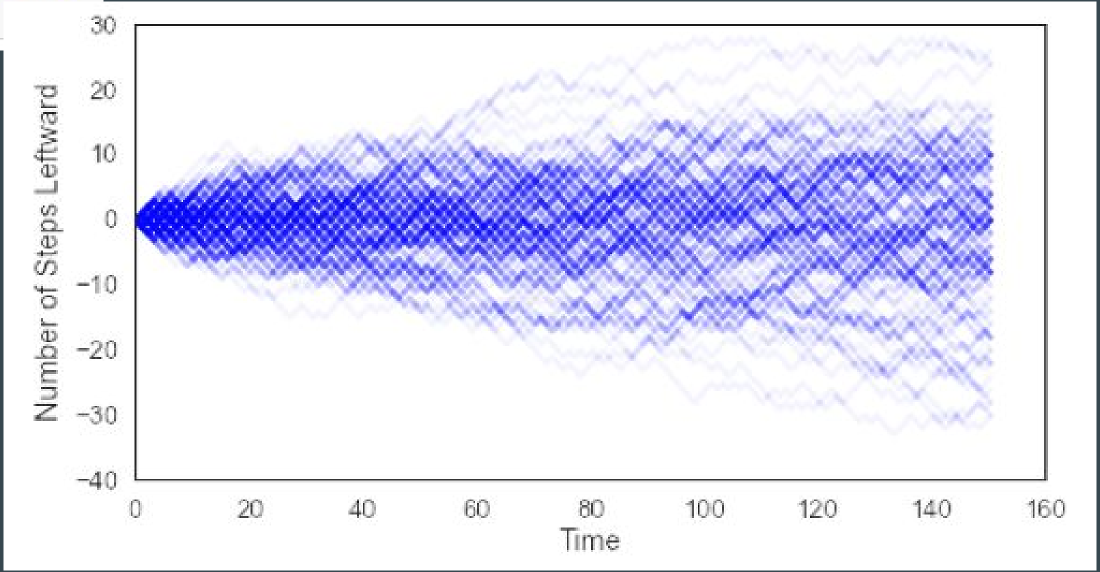
```
path = np.cumsum( np.append(0, r.choice([+1,-1],50)))
f, ax  = plt.subplots(1, 1, figsize=(8, 4), sharex=True)
x=range(len(path))
ax.plot(x,path,'b-', lw=5, alpha=0.6)
ax.set_ylabel("Number of Steps Leftward")
ax.set_xlabel("Time")
```



Random Walk

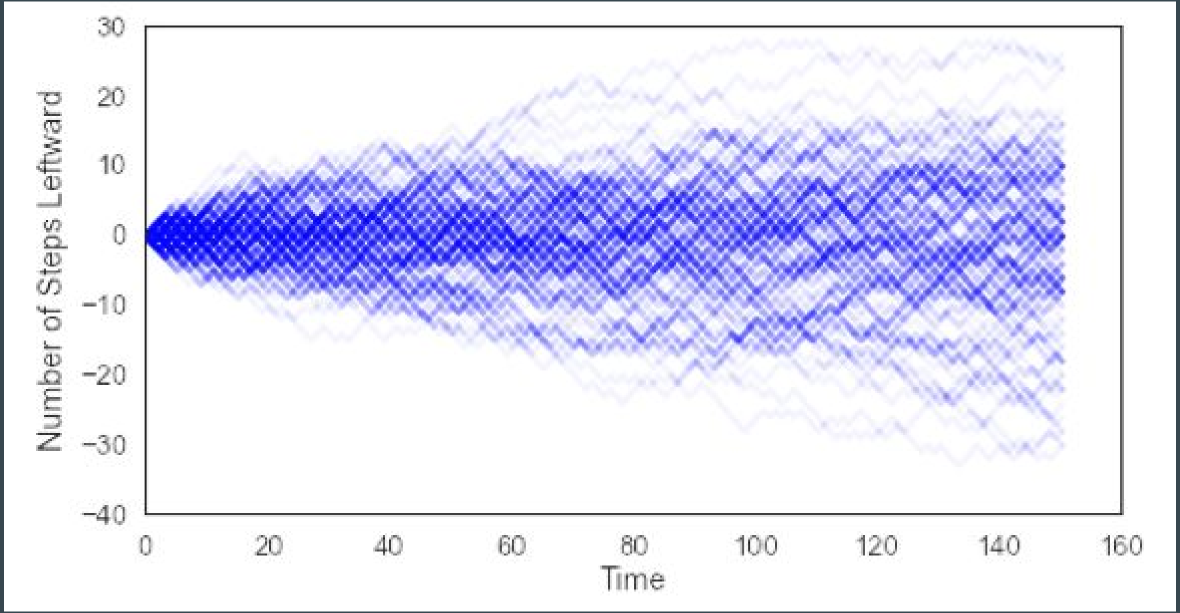


```
path = np.cumsum( np.append(0, r.choice([+1,-1],150)))
f, ax  = plt.subplots(1, 1, figsize=(8, 4), sharex=True)
x=range(len(path))
ax.plot(x,path,'b-', lw=3, alpha=0.2)
ax.set_ylabel("Number of Steps Leftward")
ax.set_xlabel("Time")

for i in range(100):
    path = np.cumsum( np.append(0, r.choice([+1,-1],150)))
    ax.plot(x,path,'b-', lw=3, alpha=0.2)
```



Random Walk

- Highest density of path around horizontal 0 line. On average, we shouldn't go anywhere.
- The *spread* depends on how many steps have occurred.
- Limiting distribution is infinitely wide



The Random Walk has no fixed and finite limiting distribution, but other Markov chains we'll encounter will.

A Harder Problem

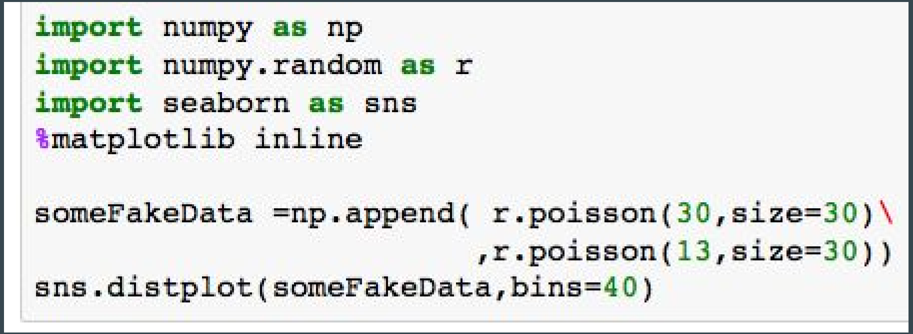
Let’s return to our fantasy football idea and imagine a new problem.

Imagine we’ve observed a bunch of points that were earned, but lost track of the position played by each player. We have a hunch that quarterbacks perform similarly to quarterbacks, and kickers to kickers.

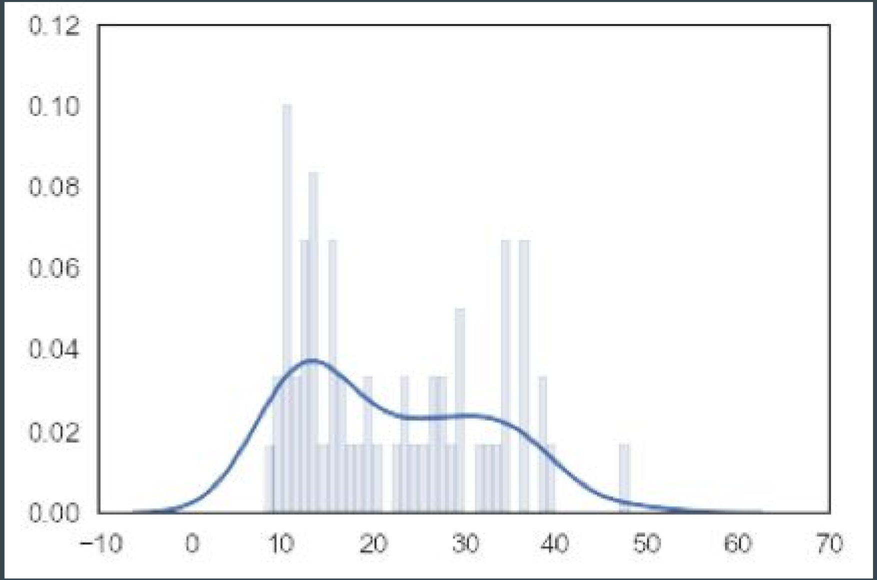
Our goal is to understand and estimate the groupings within the data, even though we don’t have the ground truth. This problem might be referred to as unsupervised clustering or as latent mixture modelling.

A Harder Problem

Here's some fake data:



```
import numpy as np
import numpy.random as r
import seaborn as sns
%matplotlib inline

someFakeData =np.append( r.poisson(30,size=30)\
                        ,r.poisson(13,size=30))
sns.distplot(someFakeData,bins=40)
```



A mixture of two Poisson processes with different λ values

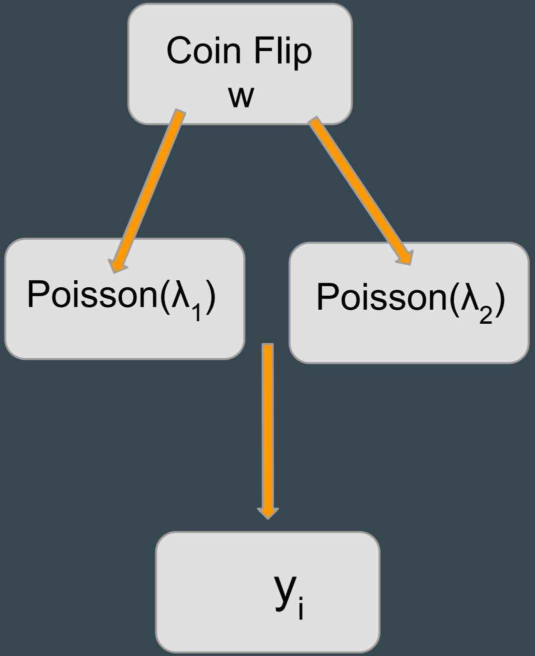
Mixture Model

When we see this data, we get the sense that it is drawn from two distinct groups. This notion is what informs our model of the data generating process.

Each observation y_i is a draw from one of two Poisson distributions.

The propensity to draw from each distribution needn't be the same, so we imagine flipping a coin with bias **w** in order to decide which distribution to draw from.

$$p(y_i|\lambda_1, \lambda_2, w) = w\text{Poiss}(\lambda_1) + (1 - w)\text{Poiss}(\lambda_1)$$
$$= w\frac{\lambda_1^{y_i}}{y_i!}e^{-\lambda_1} + (1 - w)\frac{\lambda_2^{y_i}}{y_i!}e^{-\lambda_2}$$



Mixture Model

From this data-generating-process we have jotted down the data likelihood and thus can piece together the posterior distribution.

$$p(\lambda_1, \lambda_2, w | y_N) \propto p(y_N | \lambda_1, \lambda_2, w) p(\lambda_1, \lambda_2, w)$$
$$= \left(\prod_{i=1}^{N} w \frac{\lambda_1^{y_i}}{y_i!} e^{-\lambda_1} + (1 - w) \frac{\lambda_2^{y_i}}{y_i!} e^{-\lambda_2} \right) p(\lambda_1, \lambda_2, w)$$

But we're not going to be able to come up with conjugate priors here. The parameter space is just too complex ($\boldsymbol{\lambda}_1$,$\boldsymbol{\lambda}_2$,w).

Data Augmentation

First thing we're going to do is a trick call Data Augmentation, which is misnomer, because what we're really doing is augmenting the parameters and not the data.

This will (seemingly) make our model much worse, but will in fact make our lives easier.

We add new parameters. We add latent indicator variables (s_i), one for each datapoint, which point to which mixture component was likely to have generated that datapoint.

So $s_1,\dots,s_N$ are just labels, either the value 1 or 2, that simply serve to label each datapoint according to cluster membership.

Data Augmentation

Now we have a lot more parameters, and our posterior seems much much worse.

$$p(\lambda_1, \lambda_2, w, s_1, s_2, \ldots, s_N | y_N)$$

But in the course of MCMC sampling, we *integrate* out those parameters (Monte Carlo integration), thus returning to our original posterior

$$\int p(\lambda_1, \lambda_2, w, s_1, s_2, \ldots, s_N | y_N) ds_1, ds_2, \ldots, ds_N = p(\lambda_1, \lambda_2, w | y_N)$$

And there's a reason we do this and that's to make the MCMC real simple.

Gibbs Sampling

Here’s some straightforward and basic facts about probabilities:

$$
\begin{aligned}
p(A|B) &= \frac{p(A,B)}{p(B)}, \\
p(A,B) &= p(B)p(A|B), \\
p(A,B) &\propto p(A|B)
\end{aligned}
$$

$$
\begin{aligned}
p(B|A) &= \frac{p(A,B)}{p(A)}, \\
p(A,B) &= p(A)p(B|A), \\
p(A,B) &\propto p(B|A)
\end{aligned}
$$

Combining these, we see that the joint probability p(A,B) is linearly proportional to both of its conditionals p(A|B) and p(B|A). This silly little fact is the basis of Gibbs sampling.

Gibbs Sampling

Consider that we have some complicated poster that we don't know how to handle, such as $p(\lambda_1,\lambda_2,w|y_N)$.

We do know that each *univariate conditional* posterior is proportional to the full joint posterior, so

- $p(\lambda_1| \lambda_2 =\lambda^*, w = w^*,y_N) \propto p(\lambda_1,\lambda_2,w|y_N)$
- $p(\lambda_2| \lambda_1=\lambda^*, w = w^*,y_N) \propto p(\lambda_1,\lambda_2,w|y_N)$
- $p(w| \lambda_1=\lambda^*,\lambda_2=\lambda^*,y_N) \propto p(\lambda_1,\lambda_2,w|y_N)$

With the conditioning, we fix all the variables to some temporary value, and then we have the univariate conditional posteriors.

Gibbs Sampling

Because of the linear proportionality in

- $p(\lambda_1 | \lambda_2 = \lambda^*, w = w^*, y_N) \propto p(\lambda_1, \lambda_2, w | y_N)$
- $p(\lambda_2 | \lambda_1 = \lambda^*, w = w^*, y_N) \propto p(\lambda_1, \lambda_2, w | y_N)$
- $p(w | \lambda_1 = \lambda^*, \lambda_2 = \lambda^*, y_N) \propto p(\lambda_1, \lambda_2, w | y_N)$

If we can draw a sample from each conditional, it would be proportional to a sample from the full joint posterior, and thus a valid random draw from the posterior.

The full posterior is complex and scary. Each univariate conditional is simple and friendly. We simply cycle through each one, updating the (temporarily) fixed values of the other parameters.

Gibbs Sampling

Back to our mixture problem, we have this complex posterior $p(\lambda_1, \lambda_2, w, s_1, \ldots, s_N | y_N)$

But notice that this breaks into three distinct types of conditional posteriors

$$p(\lambda_j | \ldots),$$
$$p(w | \ldots),$$
$$p(s_i | \ldots)$$

These are the three ingredients we need for a Gibbs sampler here. And that's it.

Gibbs Sampling

Let’s first think about the latent indicators, s_i. Why do they do for us? They label each data point as belonging to one of two groups.

New notation - let’s call A_j the set of all y_i where i=j. So A_1 is the set of data points in cluster number 1 and A_2 is the set of data points in cluster number 2.

Keeping this in mind, it’ll soon be clear why these extra indicator parameters are useful.

Gibbs Sampling

For our first univariate conditional, let's tackle the Poisson rate parameters $p(\lambda_j|\ldots)$

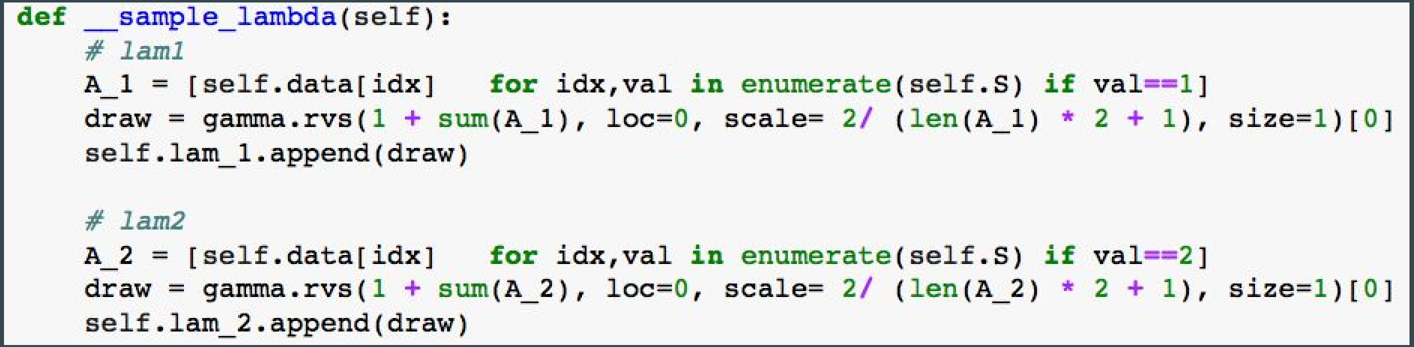
We already know a good conjugate prior for this *kind* of thing: the Poisson-Gamma. Here's the trick: with the s_i in place, we just use the Poisson-Gamma model and only make our estimates using the datapoints currently assigned to group j.

$$p(\lambda_j|\ldots) \propto \prod_{i\in A_j} p(y_i|\lambda_j)p(\lambda_j)$$
$$= \text{Gamma}(A, B)$$
where
$$A = a + \sum_{i\in A_j} y_i,$$
$$B = b + |A_j|$$

Gibbs Sampling

For our first univariate conditional, let's tackle the Poisson rate parameters $p(\lambda_j | \ldots)$

$$p(\lambda_j | \ldots) \propto \prod_{i \in A_j} p(y_i | \lambda_j) p(\lambda_j)$$
$$= \text{Gamma}(A, B)$$
where
$$A = a + \sum_{i \in A_j} y_i,$$
$$B = b + |A_j|$$


```
def __sample_lambda(self):
    # lam1
    A_1 = [self.data[idx]   for idx,val in enumerate(self.S) if val==1]
    draw = gamma.rvs(1 + sum(A_1), loc=0, scale= 2/ (len(A_1) * 2 + 1), size=1)[0]
    self.lam_1.append(draw)

    # lam2
    A_2 = [self.data[idx]   for idx,val in enumerate(self.S) if val==2]
    draw = gamma.rvs(1 + sum(A_2), loc=0, scale= 2/ (len(A_2) * 2 + 1), size=1)[0]
    self.lam_2.append(draw)
```

Gibbs Sampling

Now how about the labels s_i? This label asks - Given the current estimates of all other model parameters, what’s my best guess about which Poisson component generated the datapoint y_i.

We can directly calculate the likelihood of y_i under each Poisson component, that’s just $p(y_i| \lambda_1)$ which we’ll call $L_{i,1}$ and $p(y_i| \lambda_2)$ which we’ll call $L_{i,2}$.

We compute relative likelihoods

$$p(s_i = 1|\ldots) = \frac{L_{i,1}}{L_{i,1}+L_{i,2}} = \frac{p(y_i|\lambda_1)}{p(y_i|\lambda_1)+p(y_i|\lambda_2)}$$

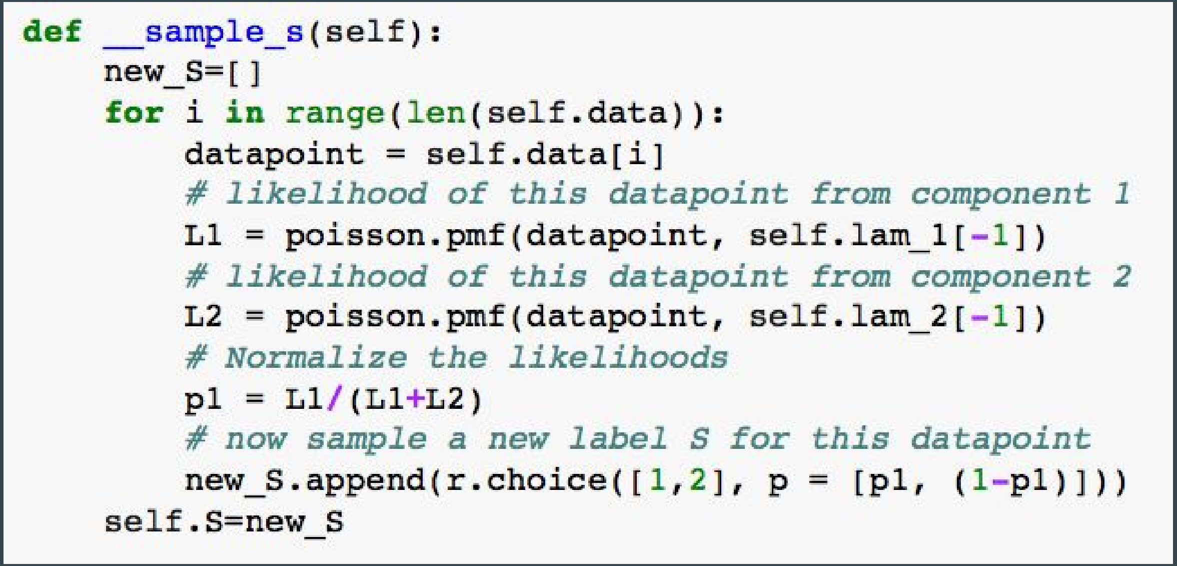
$$p(s_i = 2|\ldots) = \frac{L_{i,2}}{L_{i,1}+L_{i,2}} = \frac{p(y_i|\lambda_2)}{p(y_i|\lambda_1)+p(y_i|\lambda_2)}.$$

Then we just flip a biased coin: $s_i \sim \text{Bernoulli}(prob = p(s_i = 1|\ldots)).$

Gibbs Sampling

Now how about the labels s_i?

We just flip a biased coin $s_i \sim \text{Bernoulli}(prob = p(s_i = 1|\ldots))$.

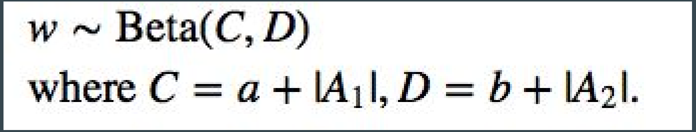


```
def __sample_s(self):
    new_S=[]
    for i in range(len(self.data)):
        datapoint = self.data[i]
        # likelihood of this datapoint from component 1
        L1 = poisson.pmf(datapoint, self.lam_1[-1])
        # likelihood of this datapoint from component 2
        L2 = poisson.pmf(datapoint, self.lam_2[-1])
        # Normalize the likelihoods
        p1 = L1/(L1+L2)
        # now sample a new label S for this datapoint
        new_S.append(r.choice([1,2], p = [p1, (1-p1)]))
    self.S=new_S
```

Gibbs Sampling

And finally, the mixture weight w. This also falls out pretty easily since we have the s_i.

Here we use the Beta-Binomial conjugate prior



$w \sim \text{Beta}(C, D)$

where $C = a + |A_1|$, $D = b + |A_2|$.

Gibbs Sampling

Our whole Gibbs sampler has these three simple ingredients:

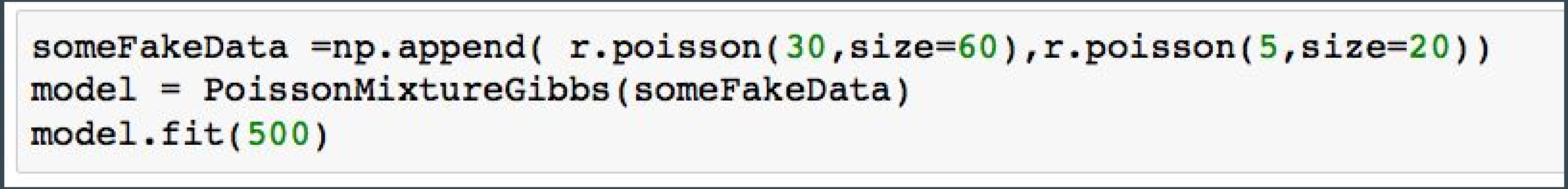
$$\begin{aligned}
&\lambda_j \sim \text{Gamma}(A, B),\\
&s_i \sim \text{Bernoulli}(prob = p(s_i = 1|\ldots)),\\
&w \sim \text{Beta}(C, D)
\end{aligned}$$

And we just cyclically sample through these and keep updating everything.

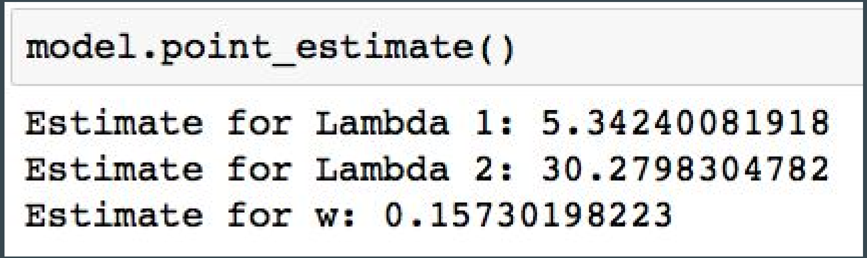
Magically, this strange process forms a Markov chain that explores the posterior distribution.

Gibbs Sampling, Mixture Model

Let’s try it out

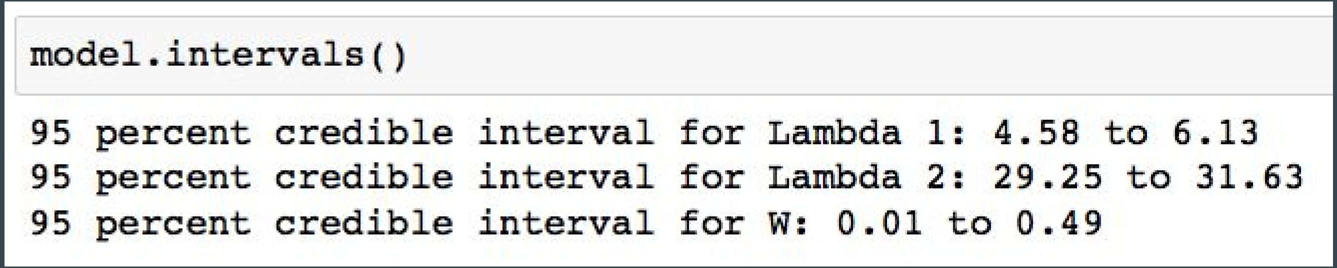


```
someFakeData =np.append( r.poisson(30,size=60),r.poisson(5,size=20))
model = PoissonMixtureGibbs(someFakeData)
model.fit(500)
```



```
model.point_estimate()
```

```
Estimate for Lambda 1: 5.34240081918
Estimate for Lambda 2: 30.2798304782
Estimate for w: 0.15730198223
```

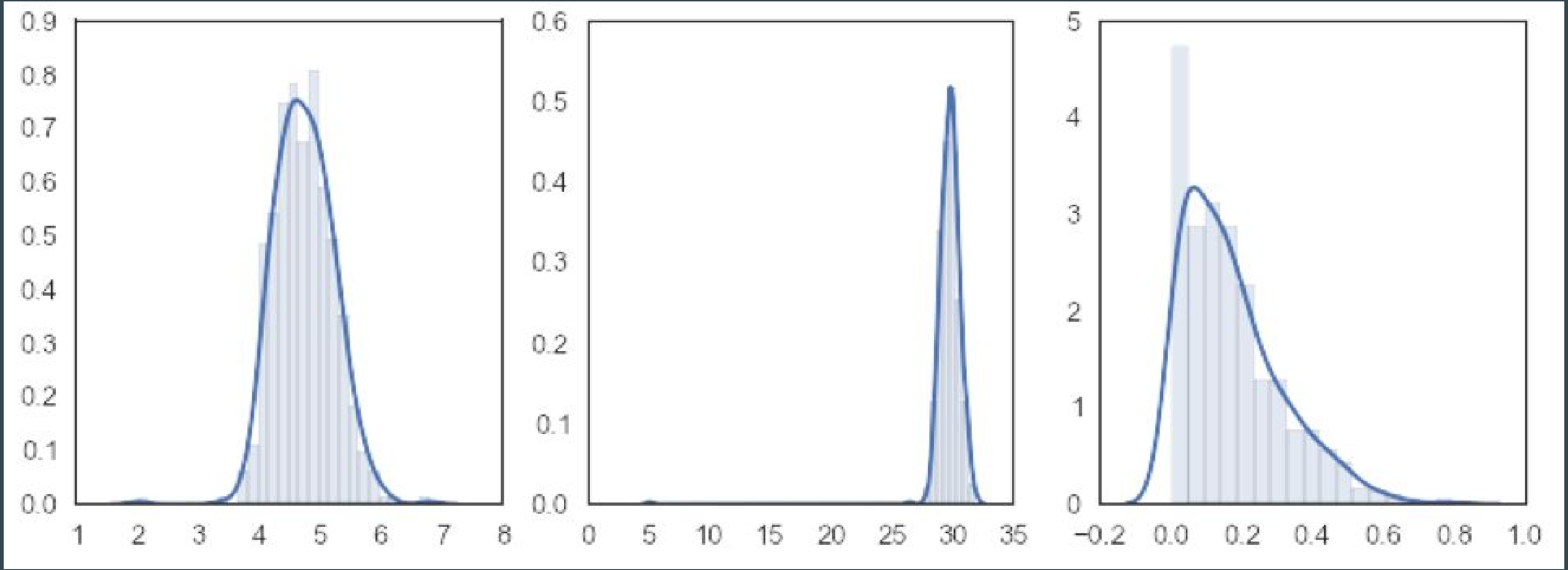


```
model.intervals()
```

```
95 percent credible interval for Lambda 1: 4.58 to 6.13
95 percent credible interval for Lambda 2: 29.25 to 31.63
95 percent credible interval for W: 0.01 to 0.49
```

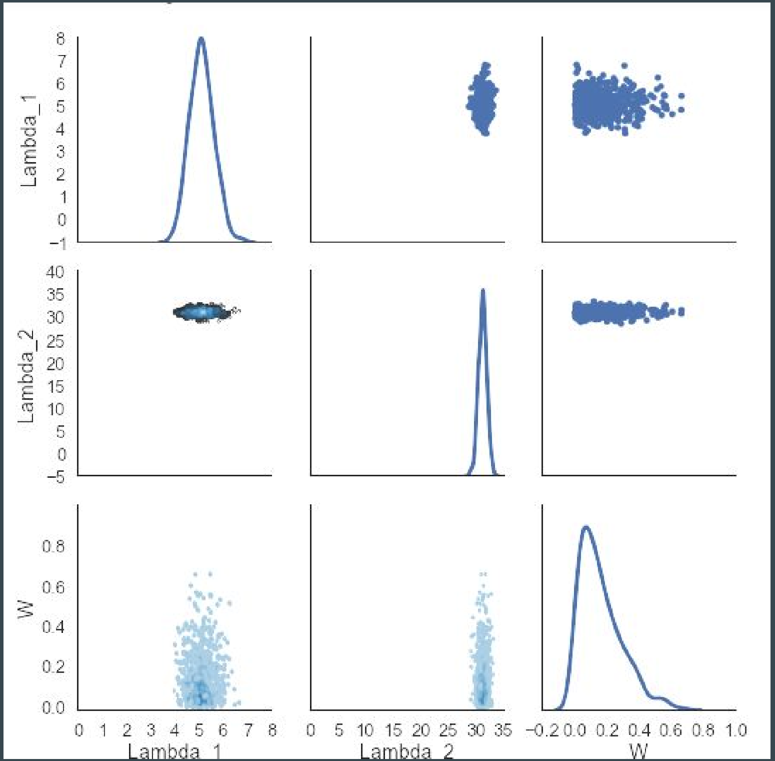
Gibbs Sampling, Mixture Model

Each marginal posterior



Gibbs Sampling, Mixture Model

2D joint posteriors



Gibbs Sampling Summary

The goal of MCMC is to approximate a complex posterior by drawing samples from it using Markov chain methods.

Gibbs sampling allows us to break down at high-dimensional posterior in a bunch of simple one-dimensional posterior. Drawing samples from each of these in turn will result in a Markov chain whose limiting distribution is the full posterior.

By break the problem into little pieces, we were able to easily reuse what we learned about conjugate priors.

Metropolis-Hastings

With Gibbs sampling, we need to have a good handle on the problem. We need to have a simple and usable expression for the univariate conditional posteriors. This won't always be possible or simple. It might be the case that our likelihood and prior simply don't combine in a nice way.

In such situations, we turn to a much more general methods called the Metropolis-Hastings algorithm.

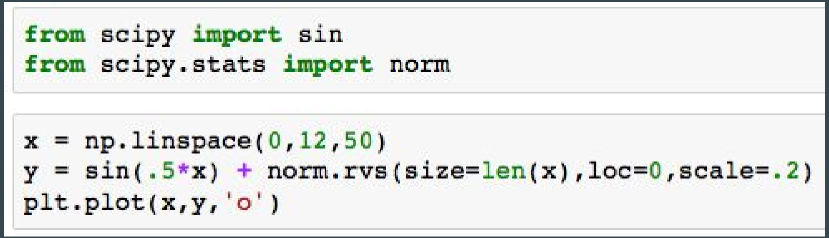
Simple Example: Curve Fitting

Let’s take a simple motivating example from a task that’s pretty common in science and engineering: curve fitting.

We imagine that the data we observe comes from some known scientific model and by fitting the measured data, we gain an estimate of some unknown scientific parameters.

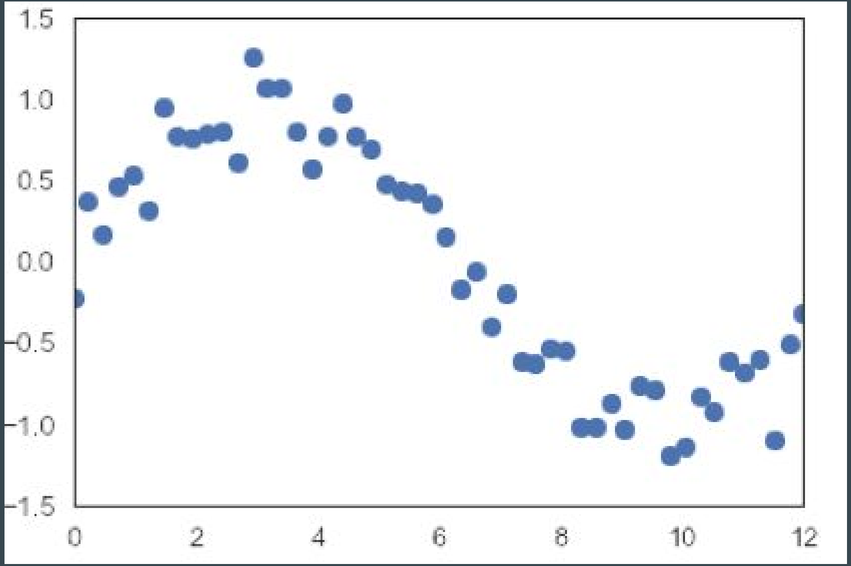
Simple Example: Curve Fitting

Imagine our scientific model is a sinusoid and we want to use some noisy data to estimate the parameters of that sinusoid. Here's our fake data:



```
from scipy import sin
from scipy.stats import norm
```

```
x = np.linspace(0,12,50)
y = sin(.5*x) + norm.rvs(size=len(x),loc=0,scale=.2)
plt.plot(x,y,'o')
```



We want to estimate the frequency parameter θ, which in this case is 0.5.

Simple Example: Curve Fitting

In Bayesian language, we want to know $p(\theta|y_N)$.

What's the likelihood? Think through our data-generating process (sine curve plus Gaussian noise):

$$y_i \sim \sin(\theta * x_i) + \text{Normal}(0,\sigma^2)$$

Said differently, the likelihood is a Normal distribution centered on the prediction from sine curve.

$$p(y_i|\theta, \sigma) = \text{N}(y_i; f(x_i), \sigma^2), \text{ where } f(x_i) = sin(\theta * x_i)$$

$$p(\theta, \sigma|y_N) \propto \textstyle\prod_i \text{N}(y_i; f(x_i), \sigma^2)p(\theta)p(\sigma).$$

What’s the problem?

Unfortunately, no matter what priors we pick, they won’t be conjugate to our final likelihood, since the likelihood is related to the model parameters (θ) only through some non-linear function.

We simply will not be able to use Gibbs sampling here.

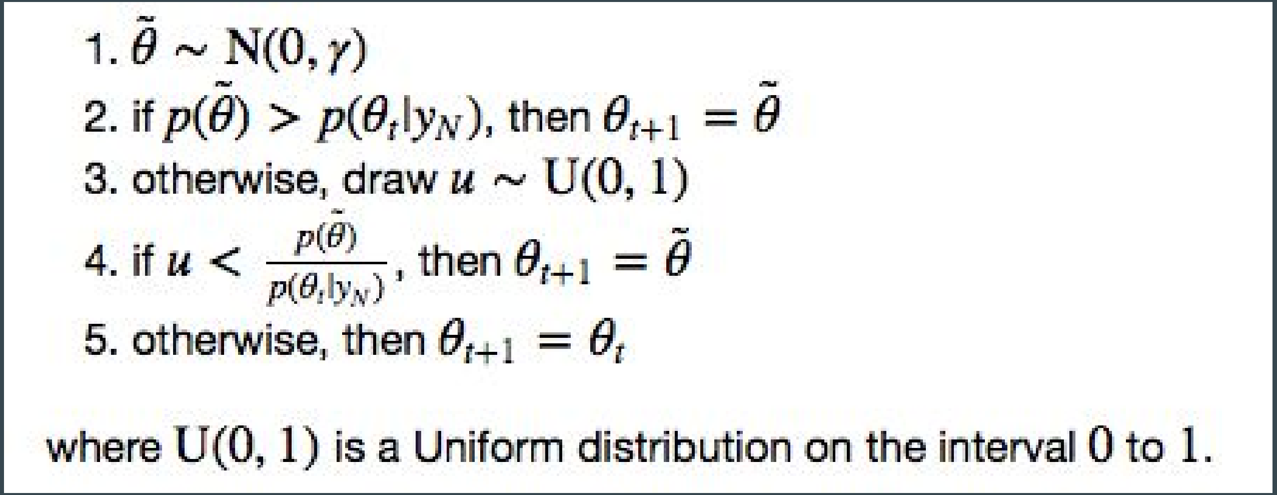
Accept and Reject

The Metropolis Random Walk algorithm was the first MCMC method historically, and is based on some surprisingly simple steps. We create a Markov chain that evolves with the following rules:

- At iteration *t* the Markov chain is at location θ_t
- We take a random walk to a new location $\underline{\theta}$
- **If** $\underline{\theta}$ is better than θ_t we keep it, and $\theta_{t+1} = \underline{\theta}$
- Otherwise, we probabilistically accept $\underline{\theta}$ in proportion to its posterior prob
- Otherwise, we reject $\underline{\theta}$ and extend chain with $\theta_{t+1} = \theta_t$

Accept and Reject

Said more compactly:



1. $\tilde{\theta} \sim \mathrm{N}(0, \gamma)$
2. if $p(\tilde{\theta}) > p(\theta_t | \mathrm{y}_N)$, then $\theta_{t+1} = \tilde{\theta}$
3. otherwise, draw $u \sim \mathrm{U}(0, 1)$
4. if $u < \frac{p(\tilde{\theta})}{p(\theta_t | \mathrm{y}_N)}$, then $\theta_{t+1} = \tilde{\theta}$
5. otherwise, then $\theta_{t+1} = \theta_t$

where $\mathrm{U}(0, 1)$ is a Uniform distribution on the interval 0 to 1.

Accept and Reject

What's going on here?

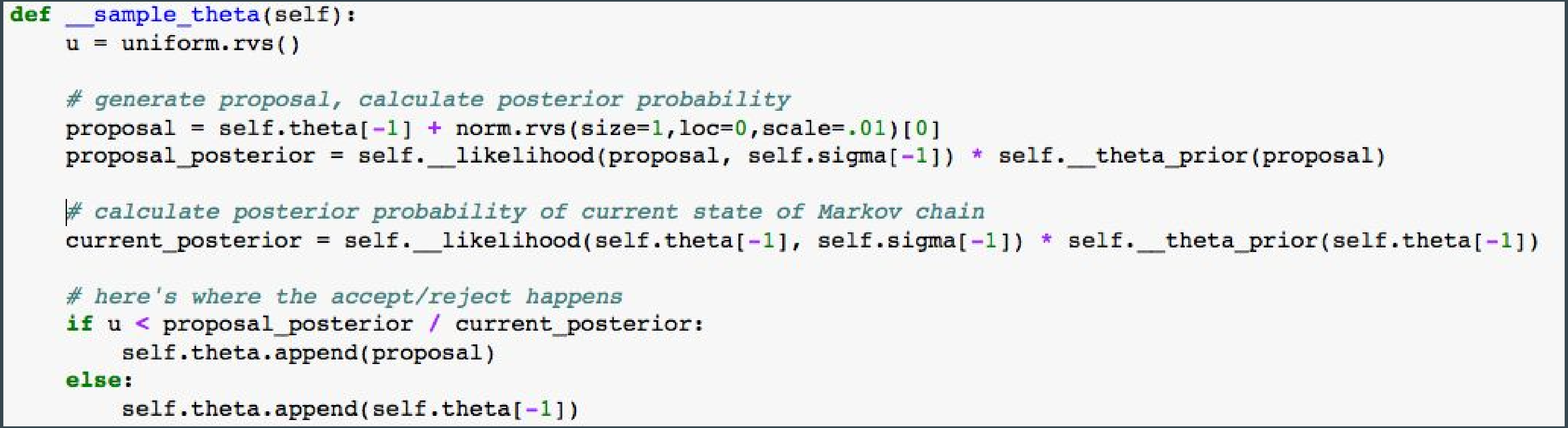
- We random-walk around the parameter space in search of improvements in our parameter estimates
- When we find a better estimate we keep it
- When we find a worse estimate we *might* keep it
- This allows our chain to explore the full posterior distribution (it's not just optimization)
- The probability that the chain occupies a certain location in the parameter space is exactly equal to the posterior probability

Accept and Reject

That's too easy

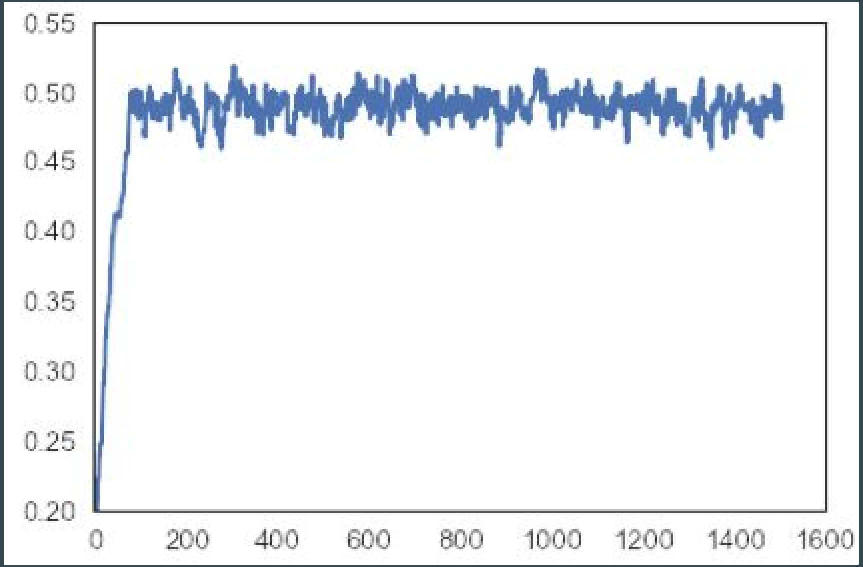
- We decided we would be unable to use Gibbs sampling in this problem
- We turn to the Metropolis random walk
- All we have to do is be able to compute the posterior probability for any particular value of $\underline{\theta}$ (that's easy)
- Then we just walk around the parameter space and play this accept/reject game

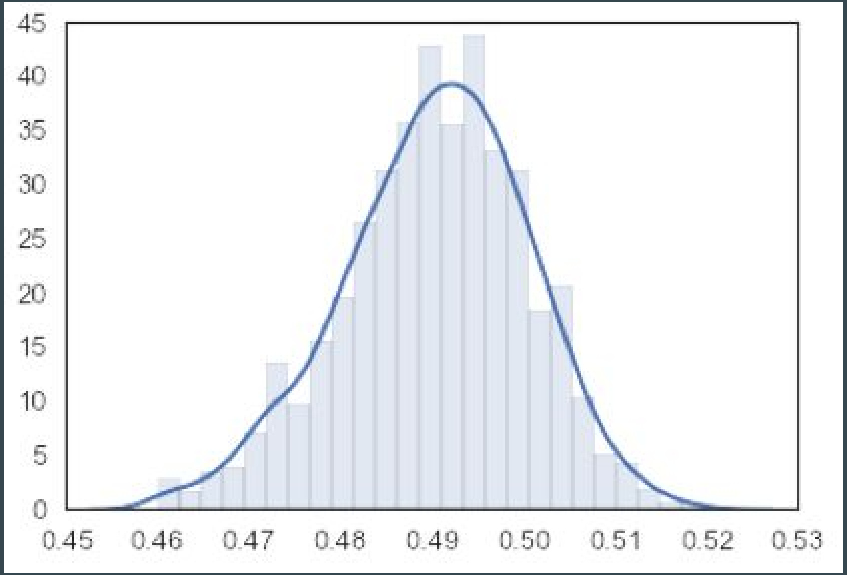
Curve Fitting with Metropolis Random Walk



```python
def __sample_theta(self):
    u = uniform.rvs()

    # generate proposal, calculate posterior probability
    proposal = self.theta[-1] + norm.rvs(size=1,loc=0,scale=.01)[0]
    proposal_posterior = self.__likelihood(proposal, self.sigma[-1]) * self.__theta_prior(proposal)

    # calculate posterior probability of current state of Markov chain
    current_posterior = self.__likelihood(self.theta[-1], self.sigma[-1]) * self.__theta_prior(self.theta[-1])

    # here's where the accept/reject happens
    if u < proposal_posterior / current_posterior:
        self.theta.append(proposal)
    else:
        self.theta.append(self.theta[-1])
```

Curve Fitting with Metropolis Random Walk



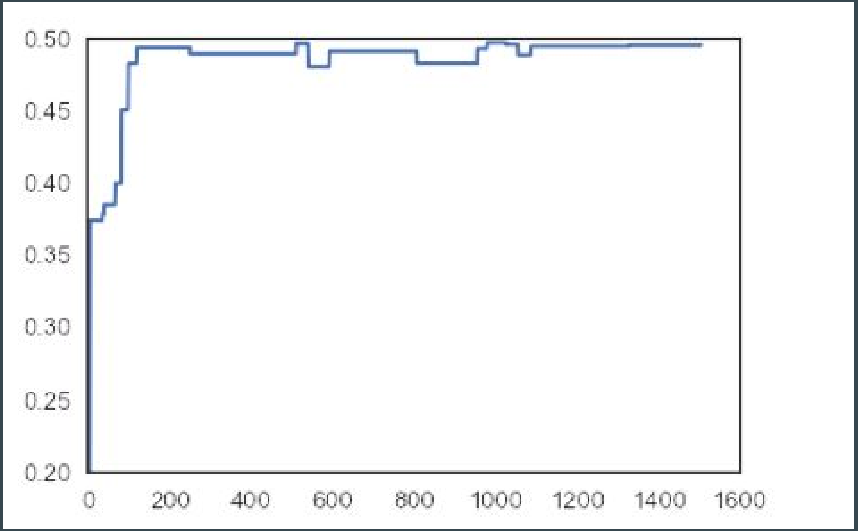


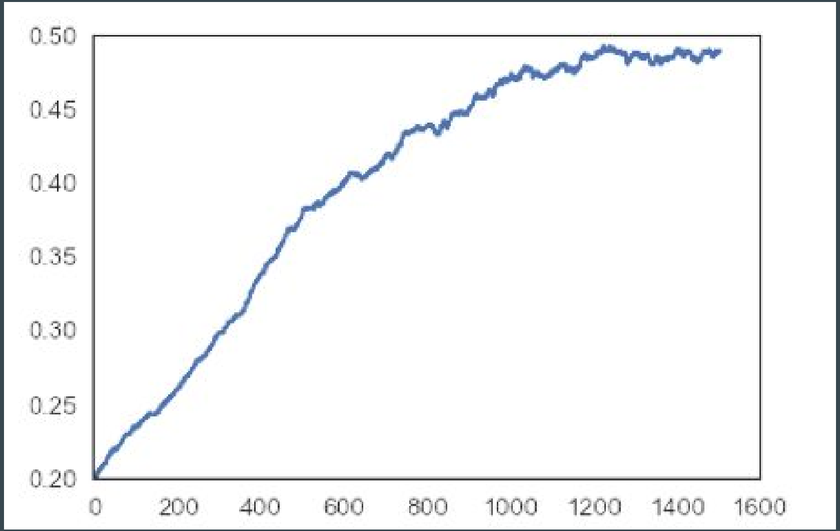
Burn-in and Chain Mixing

The random-walk transition kernel has an effect on the Markov chain

- If the step sizes are too big, then every proposal is way far away and probably a bad guess, so there's lots of rejections
- If the step sizes are too small, then every proposal gets accepted, but it takes **forever** to move around the parameter space
- Just need to explore this question and make sure the chain is "mixing well"

Burn-in and Chain Mixing

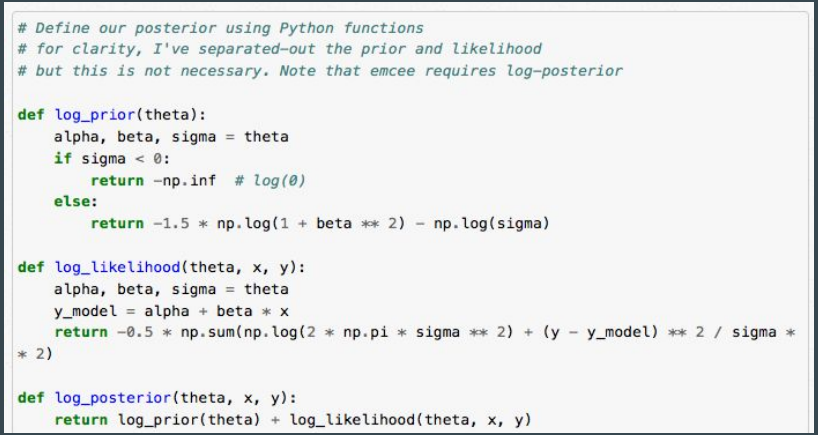




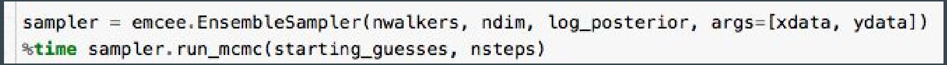
Python Libraries to Explore

PyMC2 (http://pymc-devs.github.io/pymc/)

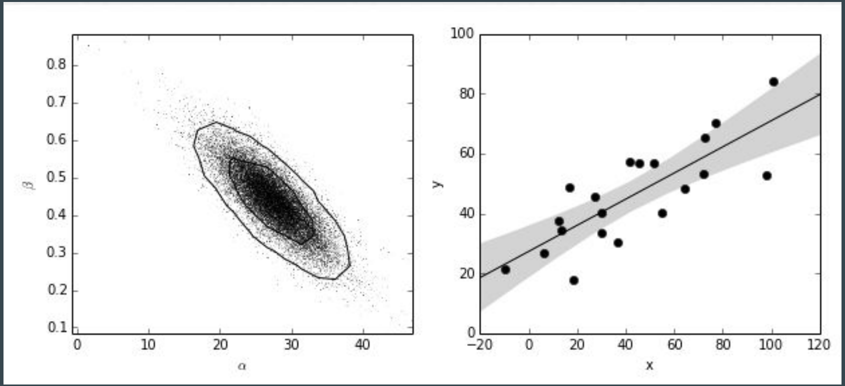
emcee (http://dan.iel.fm/emcee/current/)



```
# Define our posterior using Python functions
# for clarity, I've separated-out the prior and likelihood
# but this is not necessary. Note that emcee requires log-posterior

def log_prior(theta):
    alpha, beta, sigma = theta
    if sigma < 0:
        return -np.inf  # log(0)
    else:
        return -1.5 * np.log(1 + beta ** 2) - np.log(sigma)

def log_likelihood(theta, x, y):
    alpha, beta, sigma = theta
    y_model = alpha + beta * x
    return -0.5 * np.sum(np.log(2 * np.pi * sigma ** 2) + (y - y_model) ** 2 / sigma *
* 2)

def log_posterior(theta, x, y):
    return log_prior(theta) + log_likelihood(theta, x, y)
```



```
sampler = emcee.EnsembleSampler(nwalkers, ndim, log_posterior, args=[xdata, ydata])
%time sampler.run_mcmc(starting_guesses, nsteps)
```



http://jakevdp.github.io/blog/2014/06/14/frequentism-and-bayesianism-4-bayesian-in-python/

Conclusions & Summary

- Frequentist Methods
  - Monte Carlo
  - Bootstrap
- Bayesian Methods
  - Bayesian Inference
  - Markov chain Monte Carlo

Thanks!