Computing Methods for Particle Physics

Bayesian Inference

Overview

- Today: Computational methods for Bayesian Inference!
- Bayes theorem
- Markov Chain Monte Carlo & Metropolis-Hastings algorithm
- A simple example in Python
- A similar example using PyMC3

[Course and material based on CMPP2017 by Matthew Bass]

Bayes Theorem

$$P(H|d) = \frac{P(d|H)P(H)}{P(d)}$$

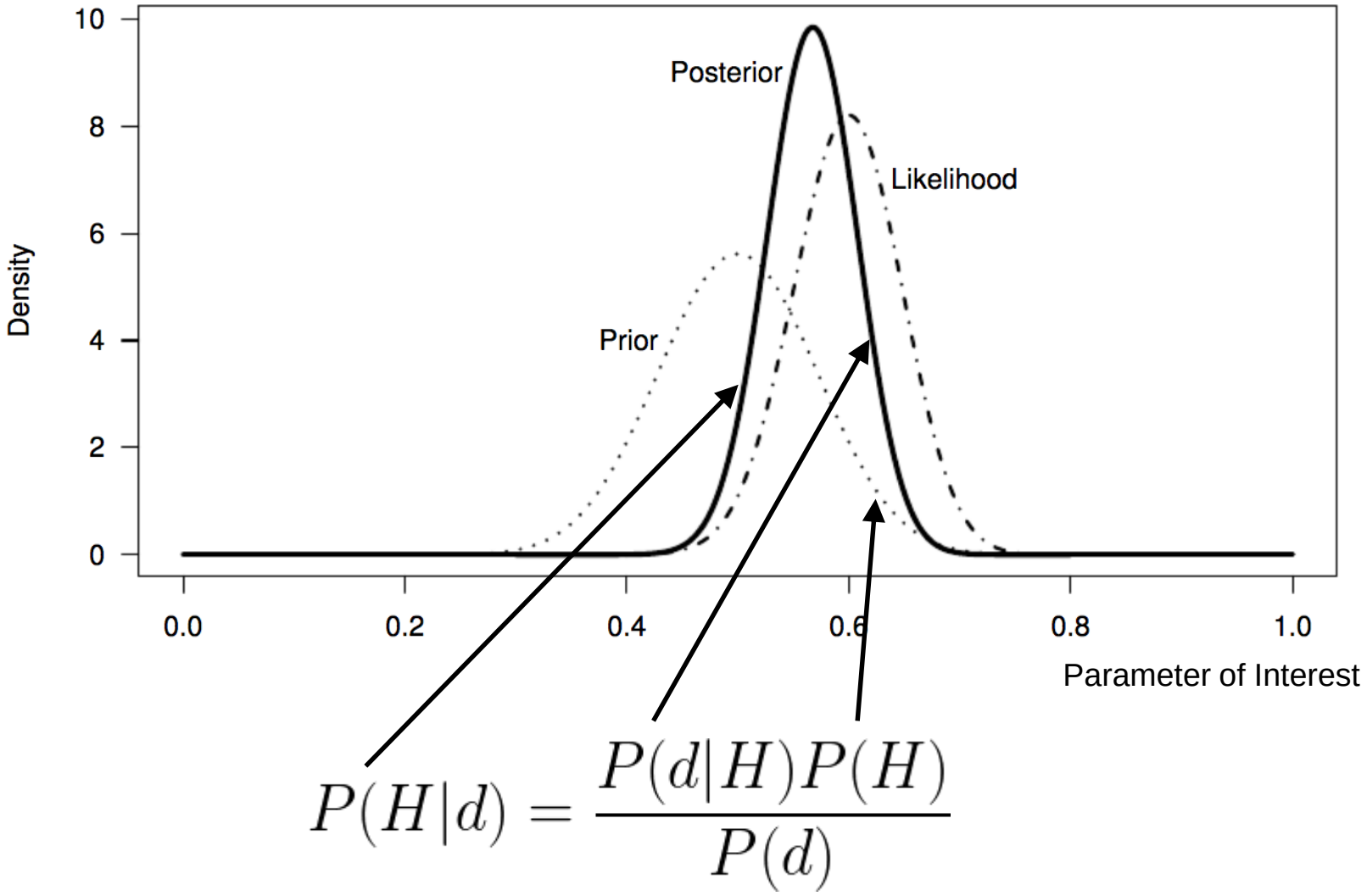
P(H|d): **posterior**; conditional probability of the hypothesis given the data

P(d|H): **likelihood**; conditional probability of the data given the hypothesis

P(H): **prior**; probability of a particular value for the hypothesis

P(d): **marginal** distribution; normalization factor

Graphically...



$$P(H|d) = \frac{P(d|H)P(H)}{P(d)}$$

Likelihood

The **likelihood** introduces our data and our model. It tells us how plausible the **data** are **given the parameters of the model**.

Example: if we think our data is distributed as a Gaussian and we want to know the mean and standard deviation then we would use a likelihood:

$$p(d|\mu,\sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} \prod_i^N \exp\left(\frac{-(d_i-\mu)^2}{2\sigma^2}\right)$$

Notice that **if our data is far away** from the distribution suggested by our model then **this likelihood is small**.

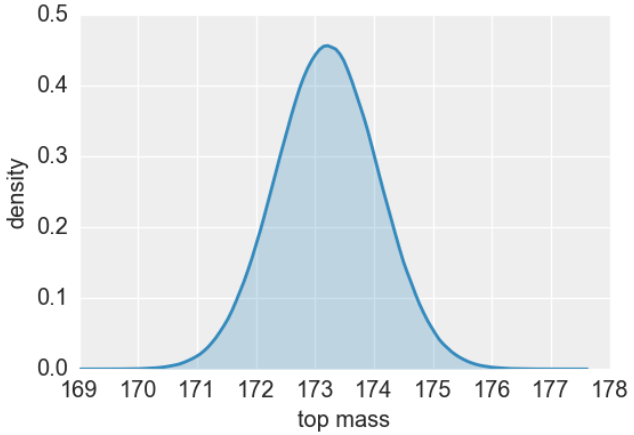
Bayes theorem tells us that this likelihood, along with the **prior**, determines the **posterior** distribution.

Priors

A **prior** is the **degree of belief** we have in a **hypothesis before the data comes in**. We must choose the priors for our particular problem carefully.

Criticism: A criticism that is often leveled at Bayesian analysis is that the priors are **subjective**. In general, but for particle physics in particular, the priors aren't all that subjective (e.g. PDG values are often used).

- Example: a parameter to our model might be the mass of the top particle, m_t, we might give this a prior of norm(173.21 GeV, 0.87 GeV).



Note: **When the data is informative** the **posterior distribution is <u>insensitive</u>** to which prior distribution is used. *I.e.* It is only when we lack data that we have to rely on our prior beliefs.

Finding the Posterior

In the simple, Gaussian example we could just integrate and solve for the posterior **analytically**. In general, this is **not possible**. Want an approach that can be applied across high dimensional parameter spaces, without taking too long!

In principle we can build an algorithm to just scan the parameter space, computing the likelihood at each point, and multiplying by the prior, to solve for the posterior. But this can be **computationally prohibitive** with large dimensional parameter spaces!

A Markov Chain Monte Carlo (**MCMC**) is an approach that solves this problem by **efficiently sampling from the posterior distribution**.

Markov Chain Monte Carlo (MCMC)

A **Markov Chain** is the result of any algorithm that takes **only** the current state of the chain, X^i, as input to generate a new proposed step, X^{i+1}. That is

$$p(X^{(i+1)}) = t(X^{(i+1)}|X^{(i)})$$

Where t(x|y) represents the **transition probability** from one state to another. So the probability that the new state is some particular value depends on the conditional transition probability of the new state given the current state.

If we define the Markov chain algorithm in a clever way we can efficiently sample from the posterior distribution!

Metropolis-Hastings

The **Metropolis-Hasting** (MH) algorithm is an MCMC method for sampling the posterior distribution. It builds a **chain** that has visited regions of parameter space with **high posterior probability** *relatively* more often that those with low posterior probability

Steps:

1. Start with an initial value (possibly random) of model parameters
2. Choose new **proposal** parameters (random walk).
3. **Accept the proposal with probability p_{accept}**, based on the ratio of the posteriors.
4. Store into chain if accepted; otherwise store the previous step
   - Note: you **always** store a value!
5. Repeat

$$p_{\text{accept}} = min\left[1, \frac{\frac{P(x|\mu)P(\mu)}{P(x)}}{\frac{P(x|\mu_0)P(\mu_0)}{P(x)}} = \frac{P(x|\mu)P(\mu)}{P(x|\mu_0)P(\mu_0)}\right]$$

→ The result is a chain of values that samples the posterior distribution!

Video demonstration from
https://www.youtube.com/watch?v=OTO1DygELpY

An MCMC-MH example

Let's look at a simple **example** of the MH algorithm for computing a posterior using an MCMC with as little code as possible (but no MCMC libraries, yet).

It generates a chain for a Gaussian model (μ only, σ is fixed) to a data set drawn from a Gaussian to obtain a posterior distribution.

Will point out a few features using this simple example.

```
import numpy as np
from scipy.stats import norm

## create some data
np.random.seed(123) #for reproducibility
data = np.random.randn(100) #sample from normal dist (mean=0, sigma=1)
to get some data

prop_width=0.2
chain=[1.]
rejected=0

while len(chain)<1000:
    ## Compute proposal for new posterior value
    # sample from norm distribution with mean 1 and width prop_width
    prop=norm(chain[-1],prop_width).rvs() #propose step
    # get probability density for normal distribution with mean prop
      and sigma 1 for each data point and compute the product of those
      (this is the likelihood / no prior is computed it's assumed to
      be flat)
    pprop=norm(prop,1).pdf(data).prod() #posterior for proposal
    pcurr=norm(chain[-1],1).pdf(data).prod() #current posterior

    ## decide whether or not to accept proposal
    acceptance=pprop/pcurr  #ratio of posteriors
    # compute acceptance probability. We will always accept the event
      if the probability is greater than one.
    accept= np.random.rand() < acceptance #acceptance
    if accept:
        chain.append(prop) #add accepted proposal to chain
    else:
        rejected+=1
        chain.append(chain[-1]) #note: still always store a value!

print("rejection rate:%f"%(rejected/len(chain)))

##plotting
import matplotlib.pyplot as plt
plt.plot(chain)
plt.show()

plt.hist(chain[200:], normed=1, alpha=0.5, color="red")
plt.show()
```

Underflow

$$p(d|\mu,\sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} \prod_i^N \exp\left(\frac{-(d_i-\mu)^2}{2\sigma^2}\right)$$

Beware of **machine precision!!**

Consider a likelihood that is being computed way out on the tails of the distribution. We **might get an extremely small value!**

And if we have 1000 data points, using a likelihood of this form will **multiply 1000 very small numbers** yielding a number **too small to be represented as a floating point number**. This is known as **underflow**.

The usual method for avoiding this is to take the **negative of the natural log** of the likelihood. So that multiplying many small numbers becomes a sum of reasonable sized negative numbers.

MH Parameters

Step size

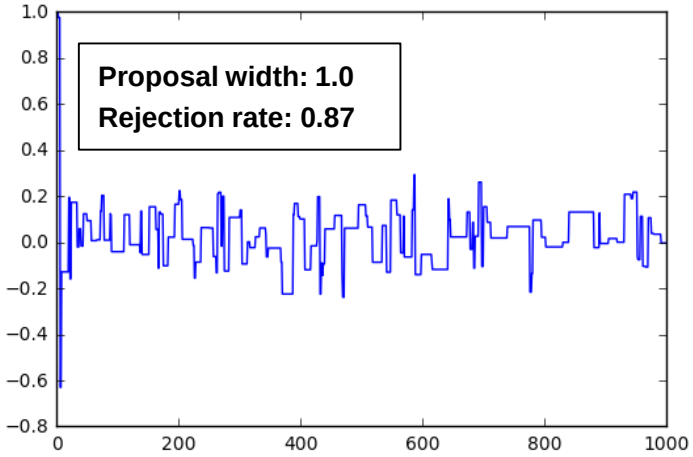
Note that in this simple example we drew the next step from a normal pdf centered around the previous step value and with width equal to the proposal_width or “jumping width” (but this increases the rejection probability and therefore the number of iterations you need.

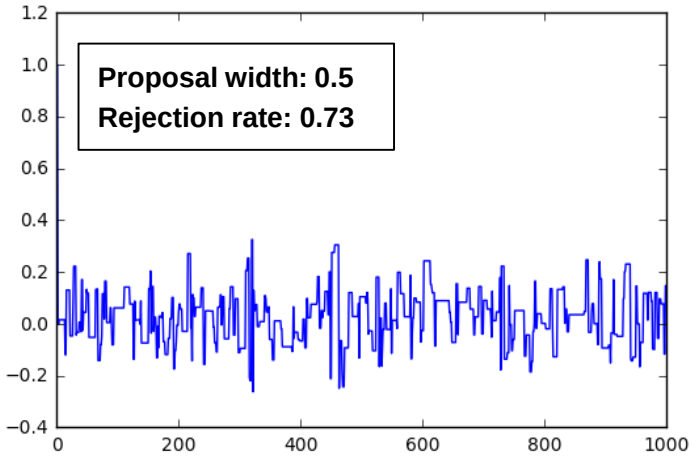
Autocorrelation!

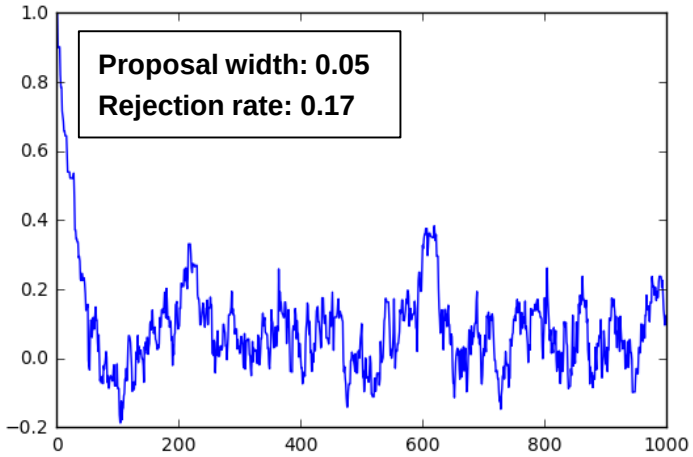
Starting value

The starting value takes a certain amount of time to be “forgotten” by the chain. This depends on the step size and the strength of the data.

Burn-in!





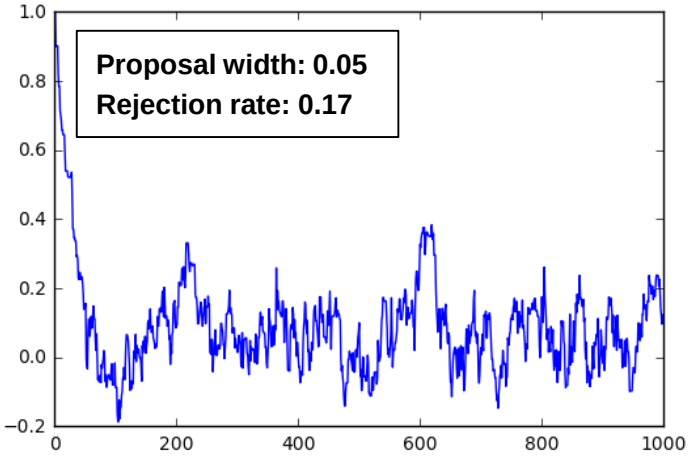


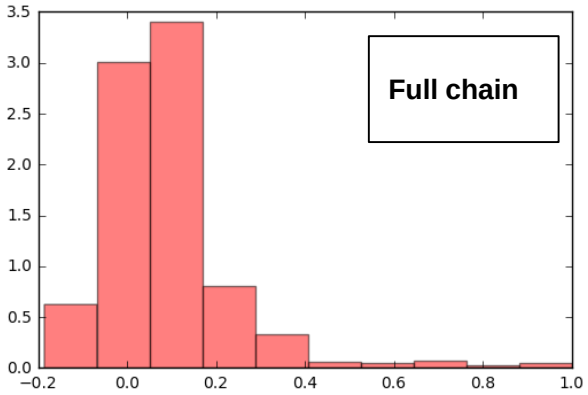
Burn-in

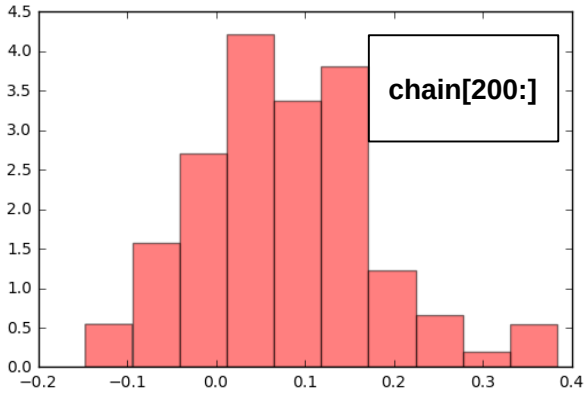
From the simple example, notice the behaviour of the trace.

The Markov Chain starts of with some memory of the initial state of the chain. This can lead to a bias in our posterior distribution.

We choose some cutoff to throw away earlier portions of the chain. This is known as **burn-in**.







PyMC3 is a python package for Bayesian modeling and machine learning.

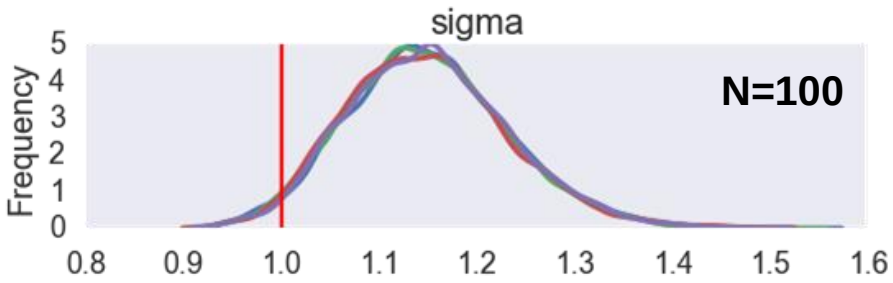
- Does the MCMC & MH for you! Along with more advanced implementations!
- Can handle thousands of parameters
- Not slow! Relies on Theano which dynamically compiles code before running MCMC.
- Has a suite of analysis tools and plots to optimize your MCMC.

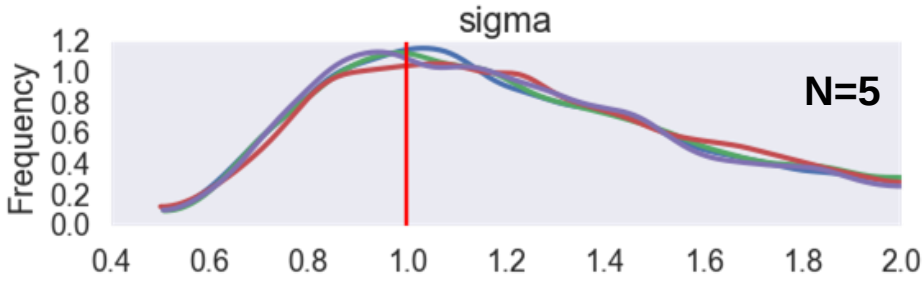
Let's **revisit** our **example**, but now using PyMC3 and also fitting for the SD.

Uninformative Priors?

Let’s revisit the example with lower statistics (N=5)...

Notice how the shape of the posterior wrt to σ changes drastically:





We used a **uniform** prior on σ. Why is the posterior **biased** upwards in σ when we rely on the prior?

Uninformative Priors

An **uninformative prior** (or regular, or reference) is one that provides no prior information or bias to the parameter.

We used a **uniform** prior on sigma, which is not an **uninformative** prior! It biases the likelihood distribution towards higher values of sigma.

A prior that is **invariant under reparameterization** is known as a **Jeffreys prior**.

For the normal distribution the Jeffreys priors are:

$\mu \sim 1$ (uniform)

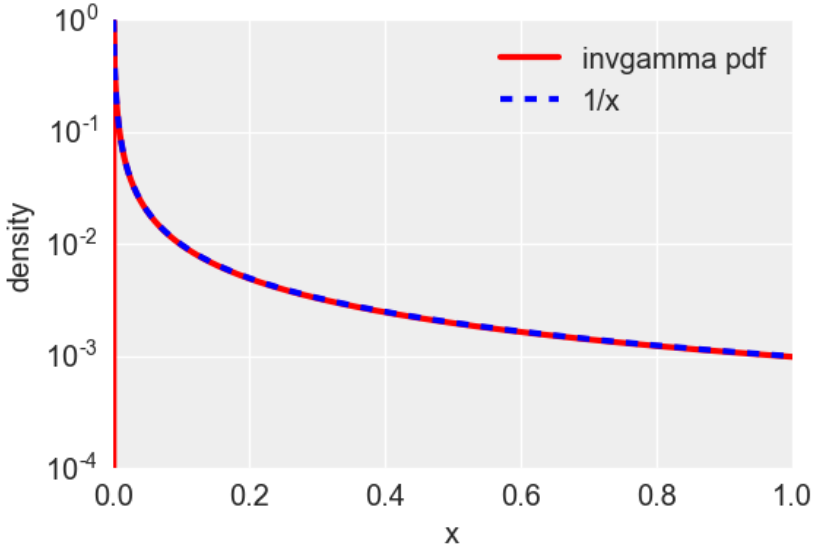
$\sigma \sim 1/\text{sigma}$

Uninformative Priors!

We can use the **Inverse Gamma** distribution to approximate the 1/x behavior and implement an uninformative prior on σ.

(This also allows us to use built in PyMC3 distributions. You could also code a custom 1/x based distribution, but we will keep things simple.)

Intuitively this is reweighting the probability of the σ prior toward lower values.



Deciding on priors

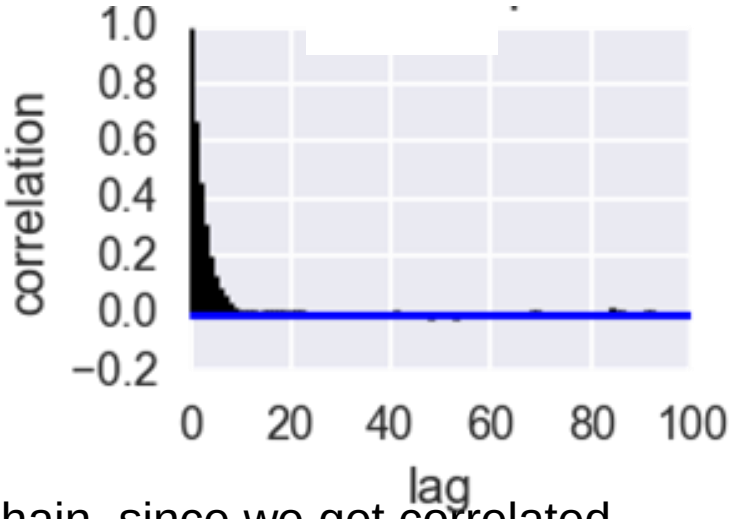
In practice, you will often be fitting data to models that have well defined priors:

- Particle masses
- MPV for Landau dist for dE/dx of particle in medium
- Neutrino oscillation parameters of the PMNS matrix

That is, an uninformative prior is rarely needed: either you have an implicit prior in your problem, or enough data for the prior not to matter. Part of a complete Bayesian inference is **establishing** and **communicating** that one of these is the case.

Autocorrelation

A chain, particular one generated by the MH algorithm, will tend to have samples correlated with their previous chain entries. An **autocorrelation** plot shows this effect:



This reduces the effective sampling rate of the chain, since we get correlated chain elements. In the past, MCMC practitioners have performed thinning (taking every n^{th} chain element) to avoid autocorrelation.

Generally unnecessary. Sufficient to just ensure the chain is long enough. **Suggestion**: run multiple chains to look for convergence.

Convergence

We can run multiple chains to test many important features of a chain.

Is our chain long enough?

Is autocorrelation biasing our posterior?

One test, built-in to PyMC3, is the **Gelman-Rubin Test.** This test checks for convergence by comparing the variance of each model parameter between multiple chains. The value of this test, R-hat, should be near 1.

An alternative approach...

Could also solve this problem with a frequentist approach.

See here for a good discussion, using Python examples, of when they differ.

- This really helped me! Looking at and modifying code has a way of making things more intuitive.

Summary

Bayesian inference is a huge area and we've barely scratched the surface.

Bayes Theorem, Priors, MCMC, MH, PyMC3

You should have enough information to get started on implementing a Bayesian fit to a simple model of your chosen data set.

Suggested Exercises

1. Read through Frequentism and Bayesianism: A Python-driven Primer and try to understand the edge cases where these approaches give different results.
2. For the simple MCMC-MH implementation, change the number of data points to 1000 (instead of 100), what happens? Why? How could the code be modified to prevent this?
3. For the PyMC example:
    a. Explore the strength of the prior relative to the likelihood by varying the size of the generated data set
    b. Verify that the σ parameter is biased at low statistics. Implement the inverse gamma function as an uninformative prior.

More advanced inference

We only discussed the basics of MCMC mapping of the posterior distribution using the MH algorithm. There are many more advanced topics you should be aware of and explore depending on your analysis needs:

- Other MCMC samplers
- Parallel chains
- Hierarchical models
- Bayes factors (for comparing one model to another)

Extra Slides

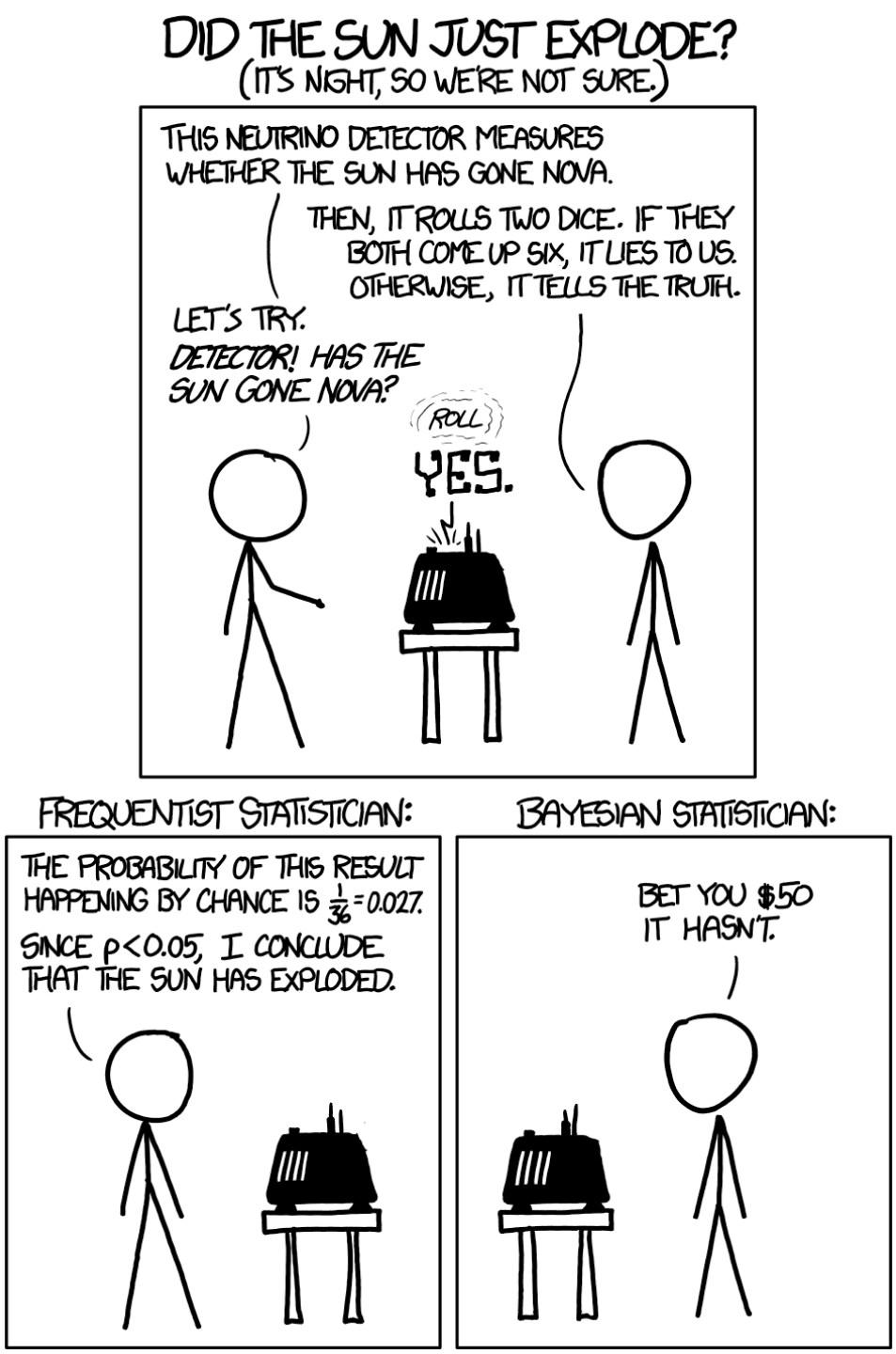
Theano Issues

I had compilation issues with running pymc3 on my PP linux desktop. It appears to be a bug in g++. See here.

Solution was to

1. Purge the theano cache: theano-cache purge
2. Add to .theanorc (in $HOME directory)

   ```
   [gcc]
   cxxflags=-march=corei7
   ```
3. Retry.


DID THE SUN JUST EXPLODE?
(IT'S NIGHT, SO WE'RE NOT SURE.)
THIS NEUTRINO DETECTOR MEASURES WHETHER THE SUN HAS GONE NOVA.
THEN, IT ROLLS TWO DICE. IF THEY BOTH COME UP SIX, IT LIES TO US. OTHERWISE, IT TELLS THE TRUTH.
LET'S TRY.
DETECTOR! HAS THE SUN GONE NOVA?
ROLL
YES.
FREQUENTIST STATISTICIAN:
THE PROBABILITY OF THIS RESULT HAPPENING BY CHANCE IS 1/36 = 0.027.
SINCE p<0.05, I CONCLUDE THAT THE SUN HAS EXPLODED.
BAYESIAN STATISTICIAN:
BET YOU $50 IT HASN'T.

Frequentist Approach to Flux Measurement

We will start with the classical frequentist maximum likelihood approach. Given a single observation $D_i = (F_i, e_i)$, we can compute the probability distribution of the measurement given the true flux F given our assumption of Gaussian errors:

$$P(D_i|F) = \left(2\pi e_i^2\right)^{-1/2} \exp\left(\frac{-(F_i - F)^2}{2e_i^2}\right).$$

This should be read "the probability of D_i given F equals ...". You should recognize this as a normal distribution with mean F and standard deviation e_i. We construct the *likelihood* by computing the product of the probabilities for each data point:

$$\mathscr{L}(D|F) = \prod_{i=1}^{N} P(D_i|F)$$

Here $D = \{D_i\}$ represents the entire set of measurements. For reasons of both analytic simplicity and numerical accuracy, it is often more convenient to instead consider the log-likelihood; combining the previous two equations gives

$$\log \mathscr{L}(D|F) = -\frac{1}{2} \sum_{i=1}^{N} \left[\log(2\pi e_i^2) + \frac{(F_i - F)^2}{e_i^2}\right].$$

We would like to determine the value of F which maximizes the likelihood. For this simple problem, the maximization can be computed analytically (e.g. by setting $d \log \mathscr{L}/dF|_{\hat{F}} = 0$), which results in the following point estimate of F:

$$\hat{F} = \frac{\sum w_i F_i}{\sum w_i}; \quad w_i = 1/e_i^2$$

The result is a simple weighted mean of the observed values. Notice that in the case of equal errors e_i, the weights cancel and $\hat{F}$ is simply the mean of the observed data.

Generate some data and the corresponding probability distribution for a fixed true flux assuming a normal distribution

For a given set of measurements compute the likelihood of the data for a given true flux (a single number for all measurements)

Put it all together and take the log for simplicity of calculation

Find the F for which the likelihood is maximised and compute error on it.