INFO 6105 Data Science Eng Methods and Tools, Lecture 8 Day 2
Dino Konstantopoulos, 25 October 2023, with material from Peter Norvig and Chris Fonnesbeck

Bayesian statistical analysis and probabilistic programming

We know how to match analytical parametric pdfs with empirical data in order to zero in on the right parameters. It involves a bit of math (MOM), or a lot of math (MLE) for better estimates.

The parameters are **point estimates**, the **data model** is **frequentist**, and we have little idea of the error we're making. The KS test does give us an idea, but how do we know where to draw the line?

Today, we'll learn about an **algorithmic approach** and a **Bayesian model** to do the same thing. We'll get a ***much better estimate of the error we're making*** in picking our parameters. Next week, we'll also *explore* that algorithm.

A frequentist model is described by an analytic function and its parameters. The method for solving for the model gives us what the most likely values for the parameters. MOM does this. MLE does this, better.

A **Bayesian method** yields parameters too, but it ***also*** yields **uncertainty** about those parameters.

In other words, the model is described as a probability distribution, just like frequentist models, but the uncertainly in its parameters is *also* described as probability distributions. How wild is that?!

The model is a pdf, and its parameters are *also* pdfs!

In [1]:
```
%matplotlib inline
import numpy as np
import pandas as pd
import matplotlib.pylab as plt
import seaborn as sns
sns.set_context('notebook')

RANDOM_SEED = 20090425
```

Bayesian vs Frequentist Statistics: *What's the difference?*

Any statistical inference scheme, Bayesian or otherwise, involves at least the following:

1. Some **unknown quantities**, which we are interested in learning or predicting. These are called the **dependent variables**
2. Some **data** which have been observed, and hopefully contains information leading to the dependent variables. These are called the **independent variables**. Note that some of these may be **correlated** with themselves (linearly or not), so we should be able to throw the correlated ones and only use the ***really independent variables*** for predicting the dependent ones
3. One (or more) **models** that relate the independent variables to the dependent variables via a probablity distribution function (pdf). The pdf will yield **variates** that essentialy statistically ***look like the real data*** and allow us to build scenarios, like our A-B test.

The model is the instrument you use to **learn** about the underlying process that yields the data.

For example, you learn about the real world from the model that your parents build for you then teach you, before you leave home to build your own models.

Machines build models to learn, too. They either learn them from the data, or we (humans) can also teach them the model, like parents to them! For example, we have meteorological models, which predict weather.

Frequentist World View

In a **Frequentist** world view, **data** observed is considered **random**, because it is the realization of random processes and hence will vary each time one goes to observe the system. Model **parameters** are considered **fixed**. A parameter's true value may be as of yet unknown, but it's fixed. You need to wait till you have *all the data* before you can use methods to evaluate it.

For example, Max Verstappen's probability of winning is fixed for the season. That's what you did in your F! homework.

Prophets are central parameter in Religious World Model. Christians will say the world order may be random because of human misgivings, but Jesus Christ and his compassion (the parameter) is fixed and steadfast.

In a frequentist world view, we take Max Verstappen's winning scores from last season and use these parameters as the ground truth for probabilities of wins this season. The wins may not match predictions, but that is because the world is **random**.

Bayesian World View

In a **Bayesian** world view, *data* is considered **fixed**. Model parameters are **random** and treated as probability distributions. Model parameters *change all the time*, and as soon as you observe new evidence, you need to re-evaluate them!

For example, we need to reevaluate Max Verstappen's probability of winning *after every single race* because every single race causes his points total to change. I'm sure you felt that, too, when you did your homework.

Some Christians may postulate that world order is predetermined, however Jesus Christ's compassion may vary because.. *sometimes he gets exasperated by his followers*!

This religious analogy is *mine*, so if it's not very good, I apologize! In my simple mind, it helps me understand what is *fixed*, and what is *variable*.

In a Bayesian world view, the world is not random: There is a perfect scientific explanation for everything. However, the probabilitiy for MV winning a race ***changes*** after every race. Sometimes he's on a winning streak, and other times on a losing streak.

Bayes' Formula as a model parameter estimator

While frequentist statistics uses different estimators for different problems, Bayes formula is the **only estimator** that Bayesians need to obtain estimates of unknown quantities.

Wow!

Let me repeat this: We use Bayes' formula to estimate our model parameters.

Bayes formula can express our belief about the value of θ, the model parameter(s), as a reallocated **prior distribution** $P(\theta)$ following the observation of new data y:

For **discrete random variables**:

$$Pr(\theta \mid y) = \frac{Pr(\theta \cap y)}{Pr(y)} = \frac{Pr(y \mid \theta)Pr(\theta)}{\sum_\theta Pr(y \mid \theta)Pr(\theta)}$$

The denominator is actually the expression in the numerator integrated over all possible discrete model parameters θ.

For **continuous random variables**, the denominator usually cannot be computed directly:

$$Pr(\theta \mid y) = \frac{Pr(y \mid \theta)Pr(\theta)}{\int Pr(y \mid \theta)Pr(\theta)d\theta}$$

The denominator is the expression in the numerator integrated over all possible continuous model parameters θ

The **intractability** of the integral in the denominator was the reason for the under-utilization of Bayesian methods by statisticians for many years. But with the advent of computers and clever algorithms like Metropolis-Hastings (https://en.wikipedia.org/wiki/Metropolis%E2%80%93Hastings_algorithm), this has changed!

Recalling Bayes using M&Ms experiment

Remember this?

> The blue M&M was introduced in 1995. Before then, the color mix in a bag of plain M&Ms was (30% Brown, 20% Yellow, 20% Red, 10% Green, 10% Orange, 10% Tan). Afterward it was (24% Blue , 20% Green, 16% Orange, 14% Yellow, 13% Red, 13% Brown). A friend of mine has two bags of M&Ms, and he tells me that one is from 1994 and one from 1996. He won't tell me which is which, but he gives me one M&M from each bag. One is yellow and one is green. What is the probability that the yellow M&M came from the 1994 bag? Well, the old M&M bags' yellow count was higher, so it must be higher, right? But how to count?

In a frequentist world view, you pick M&Ms the way we picked balls from our urns in our first lecture on probabilities. We just count all favorable outcomes and divide by all outcomes. No new experiment will change anything about that.

In a Bayesian world view, experiments change *everything*.

We are asked about the probability of an event (yellow_M&M94 + green_M&M96) given the evidence (M&M94 + M&M96 = {yellow, green}). The probability of the event is not readily available. However the probability of the evidence, given the event, is readily available!

Before (prior) we see the colors of the M&Ms, there are two hypotheses, `A` and `B`, both with equal probability:

```
A: first M&M from 94 bag, second from 96 bag
B: first M&M from 96 bag, second from 94 bag
P(A) = P(B) = 0.5
```

Then (posterior) we get some evidence:

```
E: first M&M yellow, second green
```

We want to know the ***new*** (posterior) probability of hypothesis `A`, given the evidence:

```
P(A | E)
```

That's not easy to calculate, except by enumerating the sample space (frequentist world view). But Bayes Theorem says:

```
P(A | E) = P(E | A) * P(A) / P(E)
```

The quantities on the *right-hand-side* are easier to calculate:

```
P(E | A) = 20/100 * 20/100 = 0.04
P(E | B) = 10/100 * 14/100 = 0.014
P(A)     = 0.5
P(B)     = 0.5
P(E)     = P(E | A) * P(A) + P(E | B) * P(B)
         = 0.04     * 0.5  + 0.014    * 0.5   =   0.027
```

Where did the probability of the evidence P(E) formula come from?

There are two possibilities of getting the evidence: A and B, a *union* and so we sum their probabilities. The joint probability of the evidence *and* case A is a succession or *intersection*, so it must be a product of their probabilities: P(E|A).P(A). Likewise for the case B: P(E|B).P(B)

And so here is how Bayes helps us get a final answer:

```
P(A | E) = P(E | A) * P(A) / P(E)
         = 0.04     * 0.5  / 0.027
         = 0.7407407407
```

This is the **posterior** probability that A came from the '94 bag, given an event, which changed

In search of the model parameter θ

Suppose we are given some data and we are told that there is a process that yields this data, and which we must try to model. So we must

- 1: Pick the right pdf from the catalogue
- 2: Determine the right θs for the data

More specifically, we are concerned with *beliefs* about what the θs might be:

> Rather than evaluating the θs exactly, we reason about what θs are ***likely to be*** by assigning a probability distribution to them!

But these probability distributions are hidden from us. We see only see the data, and must ***go backwards in time*** to try and determine the θs to build the best possible model that we can use to reconstruct our data. This problem is **difficult** because there is no one-to-one mapping from the data to the θs.

In classical statistics we use the Method of Moments (MOM), and Maximum Likelihood Estimation (MLE), to get **point estimates** (not pdfs) for θ.

In the Bayesian approach, we use **probabilistic programming** to solve this problem, with **Markov Chain Monte Carlo** (MCMC) simulations and variational inference.

Use Case: Bayesian Model for 2 groups with continuous outcome

Let's do statistical inference for two groups (two statistics), with continuous random variables.

We'll use the fictitious example from Kruschke (2012) (http://www.indiana.edu/~kruschke/articles/KruschkeAJ2012.pdf) concerning the evaluation of a clinical trial for drug evaluation. The trial aims to evaluate the efficacy of a "*smart drug*" that is supposed to increase intelligence by comparing IQ scores of individuals in a treatment arm (those receiving the drug) to those in a control arm (those recieving a placebo). There are 47 individuals and 42 individuals in the treatment (`drug`) and control (`placebo`) arms, respectively, and these are their post-trial IQs. An IQ between 90 and 110 is considered average; over 120, superior. Let's look at the histograms of our data, first thing you should always do.

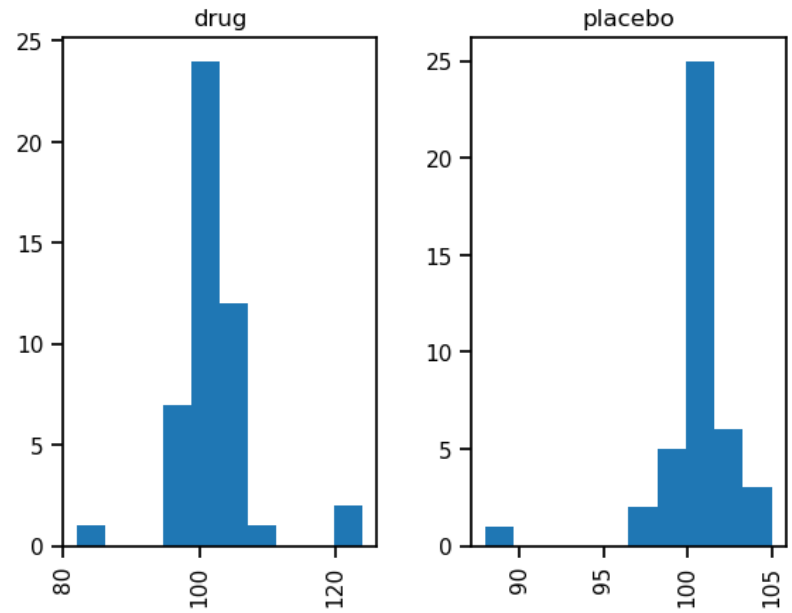
Note that although our IQ data is integer type, our datasets here could easily be real-valued, and so we consider our random variable to be continuous.

Please plot histograms using `pd.concat([drug, placebo], ignore_index=True)` , and then `.hist('iq', by='group')` on the pandas dataframe.

In [2]:
```
drug = pd.DataFrame(dict(iq=(101,100,102,104,102,97,105,105,98,101,100,123,105
        109,102,82,102,100,102,102,101,102,102,103,103,97,97,103,101,97,104,
        96,103,124,101,101,100,101,101,104,100,101),
                         group='drug'))
placebo = pd.DataFrame(dict(iq=(99,101,100,101,102,100,97,101,104,101,102,102,
           104,100,100,100,101,102,103,97,101,101,100,101,99,101,100,100,
           101,100,99,101,100,102,99,100,99),
                            group='placebo'))

trial_data = pd.concat([drug, placebo], ignore_index=True)
trial_data.hist('iq', by='group')
```

Out[2]:
```
array([<Axes: title={'center': 'drug'}>,
       <Axes: title={'center': 'placebo'}>], dtype=object)
```



Introducing the Student-T distribution

Hmm... there appears to be extreme ("outlier") values in the data (bins on the left and/or right that are distant from where most of the data lies, we say the data "*has wings*). That is a good indicator to pick a new pdf from our catalogue of all possible pdfs, called the Student-t (https://en.wikipedia.org/wiki/Student%27s_t-distribution) distribution, to describe the distributions of the scores in each group.

It was developed by William Sealy Gosset (https://en.wikipedia.org/wiki/William_Sealy_Gosset) under the pseudonym `Student` . That's because William worked for Guiness, and Guiness was very worried about its secret Beer formula. Another researcher at Guinness had previously published a paper containing trade secrets of the Guinness brewery. To prevent further

disclosure of confidential information, Guinness prohibited its employees from publishing any papers regardless of the contained information. So William published his results with a pseudonym.

Another researcher, Ronald Fisher (https://en.wikipedia.org/wiki/Ronald_Fisher) introduced a new form of that statistic, denoted `t`. The t-form was adopted because it fit in with Fisher's theory of degrees of freedom. And so now we're stuck with the mysteriously-named `Student-t` distribution, which works great modeling histograms ***with outliers*** (like financial data!).

This sampling distribution adds **robustness** to the analysis, as a `T distribution` is less sensitive to outlier observations, relative to a `normal` distribution. In other words, if you have ***a lot of outliers*** in your data and attempt to model your data with a normal distribution, your outliers will *skew* your model so that it does not fit the non-outlier data very well.

The **three-parameter** Student-t distribution allows for the specification of the following θs: A mean μ, a precision (inverse-variance) λ, and a degrees-of-freedom parameter ν:

$$f(x \mid \mu, \lambda, \nu) = \frac{\Gamma(\frac{\nu+1}{2})}{\Gamma(\frac{\nu}{2})}\left(\frac{\lambda}{\pi\nu}\right)^{\frac{1}{2}}\left[1+\frac{\lambda(x-\mu)^2}{\nu}\right]^{-\frac{\nu+1}{2}}$$

where Γ denotes the Gamma function (https://en.wikipedia.org/wiki/Gamma_function), an extension of the factorial function (with its argument shifted down by 1) to real and complex numbers, and not to be confused with the (lower-case) maximum entropy γ distribution (https://en.wikipedia.org/wiki/Gamma_distribution) used to model rainfalls and insurance claims. This class will, if anything, teach you the greek alphabet!

The degrees-of-freedom parameter essentially specifies the "***normality***" of the data, since larger values of ν make the distribution converge to a normal distribution, while small values (close to zero) result in heavier tails.

Thus, the likelihood functions of our model will be specified as follows (since we seem to have outliers in our observations of IQ):

$$y_i^{(drug)} \sim T(\nu, \mu_1, \sigma_1)$$
$$y_i^{(placebo)} \sim T(\nu, \mu_2, \sigma_2)$$

As a simplifying assumption, we will assume that the degree of normality ν is the same for both groups (both groups with similar outlier statistics).

In-class Lab

Windows/Mac-intel:

```
pip install pymc3
```

Mac-M1/M2:

```
pip install pymc
```

Instructions:

- Draw 10,000 samples from a Student-T distribution (`StudentT` in PyMC3) with parameter `nu=3` and compare the distribution of these values to a similar number of draws from a `Normal` distribution with parameters `mu=0` and `sd=1`.

The distribution is denoted `StudentT` in `pymc3`, while the normal distribution is denoted by `Normal`. So, `StudentT.dist(nu=3)` and `Normal.dist(0,1)`.

Getting a random sampling of 10,000 datapoints can be achieved with `.random(size=10000)`.

- Plot histogram with `seaborn` using `.distplot()`, from -10 to 10.

You should find that the Student-T is more spread out and has a lower peak than the Gaussian.

- Import the Student-T distribution and the Normal distribution from pymc3: `from pymc3 import StudentT, Normal` and draw 10,000 random variates from Student-T: `StudentT.dist(nu=3).random(size=10000)`, and 10,000 random variates from a gaussian: `Normal.dist(0,1).random(size=10000)`.
- Use `seagram` to plot both histograms on top of each other.

Intel:

```
from pymc3 import StudentT, Normal

t = StudentT.dist(nu=3).random(size=10000)
n = Normal.dist(0,1).random(size=10000)

sns.distplot(t, label='Student-T')
sns.distplot(n, label='Gaussian')
plt.legend()
plt.xlim(-10,10)
```

Mac M1/M2:

```
import numpy as np
import pymc as pm
from pymc import StudentT, Normal
import matplotlib.pyplot as plt
%matplotlib inline

with pm.Model() as model:
    t = pm.StudentT.dist(nu=3)
    samples_t = pm.draw(t, draws=10000)

    n = pm.Normal.dist(0,1)
    samples_n = pm.draw(t, draws=10000)

sns.distplot(samples_t, label='Student-T')
sns.distplot(samples_n, label='Gaussian')
```

In [ ]:

Introducing PyMC3/PyMC

`PyMC3` and `PyMC` are Python libraries for programming Bayesian analysis; see here (https://doi.org/10.7717/peerj-cs.55). It's a wonderful package. Looky here (https://docs.pymc.io/) for its API and docs. It helps us solve tough inverse problems and extract a model from the data.

We will model our problem using PyMC3. This type of programming is called ***probabilistic programming***, and it is probabilistic in that we create probability models ***using programming variables as the model's components***. Model components are first-class primitives within the PyMC3 framework.

> Another way of thinking about this: unlike a traditional program, which only runs in the forward direction, a probabilistic program runs in ***both*** forward and backward directions. It runs forward to compute consequences of assumptions it contains about the model, but also backward from the data to constrain possible explanations. In practice, many probabilistic programming systems will cleverly interleave forward and backward operations to efficiently home in on the best explanations. - Cronin, Beau. "Why Probabilistic Programming Matters." 24 Mar 2013. Google, Online Posting to Google . Web. 24 Mar. 2013 (https://plus.google.com/u/0/107971134877020469960/posts/KpeRdJKR6Z1)

PyMC3 used to rely on **theano** (https://en.wikipedia.org/wiki/Theano_(software)), a a graph computational library that allows one to define, optimize, and evaluate mathematical expressions involving multi-dimensional arrays efficiently. Theano is the brainchild of Yoshua Benjio (https://en.wikipedia.org/wiki/Yoshua_Bengio) of the University of Montreal's MILA (https://mila.quebec/en/) laboratory. In my opinion, it's the most famous university lab associated to artificial neural networks and deep learning. It's pretty well-funded (http://nouvelles.umontreal.ca/en/article/2017/09/15/facebook-invests-over-7m-u.s.-in-mila-and-ai-research-in-montreal/).

`theano` is now deprecated because other libraries like facebook's `Torch` and Google's `TensorFlow` now include the same features. Older PyMC3 versions still uses theano, but the newer versions don't, they use **tensorflow** (https://en.wikipedia.org/wiki/TensorFlow) instead.

Theano was used for the first deep neural networks. After Theano announced plans to discontinue development in 2017, the PyMC3 team evaluated `TensorFlow Probability` as a computational backend: PyMC4 was based on TensorFlow Probability, where Models must be defined as `generator functions` using a `yield` keyword for each random variable, But the team decided in 2020 to abandon Tensorflow Probability and to fork Theano under the name `Aesara`. Large parts of the Theano codebase have been refactored. The PyMC team has released the revised computational backend under the name `PyTensor` and continues the development of PyMC5 under the name `pymc`.

For probabilistic programming, you write a program in Python that builds expressions on a computational graph. You still have to declare variables a, b, c and give their types (int, int, int), build expressions for how to put those variables together e.g. $a^b + c$, and compile expression graphs to functions $f(a, b, c)$ in order to use them for computation. What theano builds in return is a super-fast callable object from a purely symbolic graph, optimizes the graph, and even compiles some or all of it into native machine instructions. More on Theano here (http://www.deeplearning.net/software/theano/).

PyMC3/PyMC code is easy to read. The only novel thing is the syntax: Simply remember that we are representing the model's components (τ, λ_1, λ_2) as **probabilistic variables** (a.k.a. **stochastic variables**). And the way we represent *continuous* probabilistic variables is with a probability density function (pdf): A **function**, not a **dictionary** anymore.

> **Note**: Do not confuse this/these function/s with the histogram of the dataset or the pdf that we will use to model it. That is a *different* function.

To figure out these functions, we will first assume they have a certain shape, which we will call the **prior shape**.

Then, we will run a simulation and try to reconstruct our data.

This will allow us to refine the prior shapes into **posterior shapes**, in accordance with Bayes' theorem.

Ready? *You* work, professor rests!

Picking our Priors

We have an idea about the means of our IQ data: The data for both drug and placebo group seem to be centered around IQ = 100.

So let's center the Student-T priors for μ, the mean, at 100, using a Normal distribution, and a standard deviation that is wide enough to account for plausible deviations from this population mean:

$$\mu_k \sim N(100, 10^2)$$

- *Craaaaazy*, right? I'm modeling data using a **Student-T pdf model** with 3 parameters and I model each parameter using a **normal distribution** (pdf).

Please do this below using:

For Intel

```
from pymc3 import Model, Uniform

with Model() as drug_model:
    μ_0 = Normal('μ_0', 100, sd=10)
    μ_1 = Normal('μ_1', 100, sd=10)
```

For Mac M1/M2

```
from pymc import Model, Uniform

with Model() as drug_model:
    μ_0 = Normal('μ_0', 100, 10)
    μ_1 = Normal('μ_1', 100, 10)
```

In [22]:

Now use a Uniform prior for the standard deviations σ_k of the Student-T, with a lower bound of 0 and an upper bound of 20, here below:

```
with drug_model:
    σ_0 = Uniform('σ_0', lower=0, upper=20)
    σ_1 = Uniform('σ_1', lower=0, upper=20)
```

- *Craaaazy*! I use a **uniform distribution** (pdf) to model the second parameter of the pdf-based model (standard deviation).

In [11]:

For the degrees-of-freedom parameter ν, use an exponential distribution with a mean of 30.

Intel

```
from pymc3 import Exponential
sns.distplot(Exponential.dist(1/29).random(size=10000), kde=False);
```

Mac M1/M2

```
from pymc import Exponential

with pm.Model() as model:
    e = pm.Exponential.dist(1/29)
    samples_e = pm.draw(e, draws=10000)

sns.distplot(samples_e, kde=False);
```

- *Craaazy*! I use an **exponential distribution** to model the third parameter of my **student-T model**.

In [ ]:

This allocates ***high prior probability*** over the regions of the parameter that describe the range from normal to heavy-tailed data under the Student-T distribution:

```
with drug_model:
    ν = Exponential('ν_minus_one', 1/29.) + 1
```

Now go ahead and model both datasets in `pymc3` / `pymc` :

In [14]:

Turn your attention now to tracking the **posterior *quantities of interest***. Namely, calculate the difference in means between the drug and placebo groups: `diff_of_means = Deterministic('difference of means', μ_1 - μ_0)` .

As a joint measure of the groups, also estimate the **effect size** (https://en.wikipedia.org/wiki/Effect_size), which is the difference in means scaled by the pooled (square root of the squares) estimates of standard deviation: `Deterministic('effect size', diff_of_means / np.sqrt((σ_1**2 + σ_0**2) / 2))` .

The effect size resembles the computation for a t-test statistic, with the critical difference that the t-test statistic includes a factor of $\sqrt{n}$ that does not appear in the Bayesian definition of effect size. Including a factor of $\sqrt{n}$ means that for a given effect size, the significance level increases with the sample size. Unlike the t-test statistic, our effect size here aims to estimate a population parameter and is not affected by the sample size (yay!). Effect size can be hard to interpret, since it is no longer in the same units as our data, but it is a function of all four estimated parameters.

We need to specify `Deterministic` for non-probabilistic variables (a.k.a. **stochastic variables**) because all PyMC3 variables are by default **probabilistic**, unless we set them to be **deterministic**.

Intel

```
from pymc3 import Deterministic

with drug_model:
    diff_of_means = Deterministic('difference of means', μ_1 - μ_0)
    effect_size = Deterministic('effect size', diff_of_means / np.sqr
t((σ_1**2 + σ_0**2) / 2))
```

Mac M1/M2

```
from pymc import Deterministic

with drug_model:
    diff_of_means = Deterministic('difference of means', μ_1 - μ_0)
    effect_size = Deterministic('effect size', diff_of_means / np.sqr
t((σ_1**2 + σ_0**2) / 2))
```

In [15]:

All right, all right, all right! Now our model is **fully specified** and we are ready to track our posteriors.

We will fit the model using **variational inference** (https://en.wikipedia.org/wiki/Variational_Bayesian_methods). This will estimate all our posterior distributions using an optimized approximation, and then draw 1,000 samples from it. Be *patient* now, we are running **probabilistic regressions**.

Intel

```
from pymc3 import fit

with drug_model:
    drug_trace = fit(random_seed=RANDOM_SEED).sample(1000)
```

Mac M1/M2

```
from pymc import fit

with drug_model:
    drug_trace = fit(random_seed=RANDOM_SEED).sample(1000)
```

In [ ]:

Waaaaait for the probabilistic computation to finish!

> **Note**: You *may* have to specify the extra argument (inside the `fit()` API): `cores = 1` if your laptop does not have the power to run all 4 cores (the *old* API analog was `njobs = 1`)

Now plot all your posterior distributions, throwing away the first 100 samples. You typically always throw away from 10% to 20% of your simulation samples, because they start off *wrong* before converging to the *right solution* (it's like when you start off to climb a mountain: First you wander around to find the trainlead):

Intel

To plot the probability distribution for each parameter:

```
from pymc3 import plot_posterior

plot_posterior(drug_trace[100:],
               varnames=['μ_0', 'μ_1', 'σ_0', 'σ_1', 'v_minus_one'],
               color='#87ceeb');
```

and in the next cell:

```
plot_posterior(drug_trace[100:],
               varnames=['difference of means', 'effect size'],
               ref_val=0,
               color='#87ceeb');
```

if you have import errors on `plot_posterior`, chances are the APIs got moved to another library. So do this instead:

```
pip install arviz
from arviz import plot_posterior
```

Also, possible that `varnames` was renamed to `var_names`.

In any case, any error, please google the error. If no solution, let us know :-)

Mac M1/M2

First:

```
# explore
drug_trace
```

Then, to remove the forst 100 draws for one of the model parameters:

```
drug_trace.posterior.μ_0.draw[100:]
```

To plot the probability distribution for all parameters:

```
import arviz as az
az.plot_posterior(drug_trace.posterior)
```

In [ ]:

Now, *conclude* please..

- Conclusion: *The posterior probability that the mean IQ of the subjects in the treatment group is greater than that of the placebo group is...*

Homework

- Repeat your experiment with MOM on a single-humped dataset but with MLE instead!
- Repeat your MOM/MLE experiment but with Bayesian simulation using PyMC3/PyMC instead!
- Which method is more correct, which was the most natural to you?

References and Resources

- Goodman, S. N. (1999). Toward evidence-based medical statistics. 1: The P value fallacy. Annals of Internal Medicine, 130(12), 995–1004. http://doi.org/10.7326/0003-4819-130-12-199906150-00008 (http://doi.org/10.7326/0003-4819-130-12-199906150-00008)
- Johnson, D. (1999). The insignificance of statistical significance testing. Journal of Wildlife Management, 63(3), 763–772.
- Gelman, A., Carlin, J. B., Stern, H. S., Dunson, D. B., Vehtari, A., & Rubin, D. B. (2013). Bayesian Data Analysis, Third Edition. CRC Press.
- Norvig, Peter. 2009. The Unreasonable Effectiveness of Data (http://static.googleusercontent.com/media/research.google.com/en//pubs/archive/35179.pdf
- Salvatier, J, Wiecki TV, and Fonnesbeck C. (2016) Probabilistic programming in Python using PyMC3. *PeerJ Computer Science* 2:e55 https://doi.org/10.7717/peerj-cs.55 (https://doi.org/10.7717/peerj-cs.55)
- Cronin, Beau. "Why Probabilistic Programming Matters." 24 Mar 2013. Google, Online