**Bayesian Statistics 101**

**Love it or hate it, you will never look at statistics the same way again**

**Introduction**

Bayesian statistics differs from classical statistics (also known as frequentist) basically in its interpretation of probability. The former sees it as a “*degree of belief*”, whereas the latter sees it as the “*relative frequency observed during many trials*”.

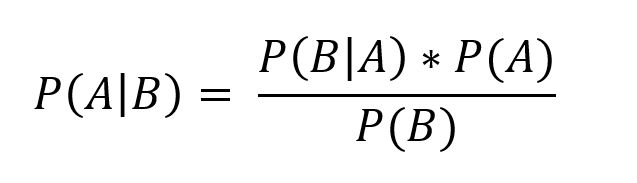
This difference might seem too abstract, but it has great practical impact on the methods developed afterwards.

The frequentist view is the most common one and, therefore, has influenced a greater number of statistical techniques. However, many modern methods rely on the Bayesian approach and can yield great results, providing you with alternatives when there isn’t much data to work with.

The methods we’ll see now will give you a formal framework through which you can add subjective judgment to your data science problems, which can be specially helpful when you don’t have much data available, or when you know that data is flawed somehow. It will also help you understand the reasoning that originated some famous machine learning algorithms, such as Naive Bayes Classifier and Bayesian Neural Networks.

We’ll start by taking a quick look at the the Bayes’ theorem — the core of Bayesian statistics — and then move on to some of the techniques that stem from that and how they can be used to solve all sorts of statistical problems. We’ll be using Python, by the way.

**Bayes’ Theorem**



Bayes’ Theorem

The equation above is quite simple, but understanding it requires knowing some notations from probability theory:

* **P(A)**: probability of an event A
* **P(A|B)**: probability of an event A *given that an event B occurred*

An event can be pretty much anything. For instance, **P(A|B)** can mean “the probability (**P**) of having COVID (**A**) given that (**|**) your PCR test came positive (**B**)”. To calculate that probability using the above equation, we would need:

* **P(A)**: probability of having COVID (regardless of test results)
* **P(B)**: probability of having a positive result in a PCR test (regardless of whether you have COVID or not)
* **P(B|A)**: probability of having a positive result in a PCR test, given that you have COVID

This theorem can be easily derived from the definition of conditional probability and it’s not exclusively used in Bayesian statistics, nor it is disputed by Frequentists. Up to this point it’s just basic probability theory.

**Bayesian statistics**

What Bayesian statistics does from here on is turn that theorem into the following mantra:

You have a prior belief about how the world works. Once you get data, you update this belief accordingly.

That means that you start the solution to any problem by defining a prior distribution (this step is quite subjective), and then you use the likelihood of the observed data to update that prior, creating a posterior distribution.

Not clear? Let’s look at an example of a problem solution and you will see how this happens in practice.

**The Monty Hall Problem**

*Let’s Make a Deal* was a popular TV game show that started in the ‘60s, in the United States and whose original host was called Monty Hall. A famous probability puzzle based on it became famous afterwards, with the following format:

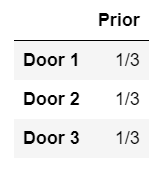
You are on the game show’s stage, where there are 3 doors. Behind one of them there’s a car, and behind 2 of them there’s a goat. You have to pick one of the doors and your prize will be whatever’s behind it (obviously you want the car).

You pick Door 1. The host then opens Door 3, revealing there was a goat behind it. Then, he gives you the choice of sticking to Door 1 or switching to Door 2. What do you do?

Here is where this problem has raised a lot of furor: our intuition tells us there is no difference between switching doors or sticking to our choice, both hold a 50% of hiding the car. A closer look reveals, however, that it is indeed beneficial to switch doors — it will actually double your chances of winning.

Let’s look at the problem using a Bayesian perspective.

We start with a prior probability distribution of 1/3 for each door. This means each door has, a priori, 1 chance in 3 of being the “right” door (there are 3 doors and we have no reason to believe one is more likely than the other). The prior is the P(A) in the Bayes’ theorem.

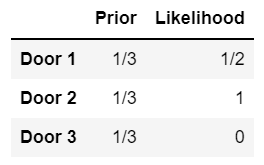


Our priors

Then, we move on to calculating the likelihoods, given the new data we have. What is the likelihood the host would open Door 3 (as he did), if the car was behind:

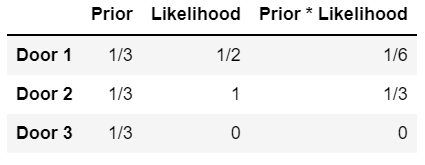
* **Door 1**: the host would choose between Doors 2 and 3 at random, so the probability he opens Door 3 in this case is **1/2**
* **Door 2**: the host would have to open Door 3 (since the car is behind Door 2 and you already chose Door 1), so the probability he opens Door 3 in this case is **1**
* **Door 3**: the host could not have opened Door 3 if the car was behind it, so the probability he opens Door 3 in this case is **0**

This leaves us with:



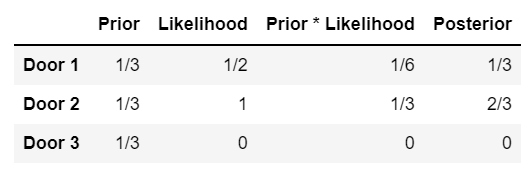
The prior and the likelihood for each door

The likelihood is equivalent to P(B|A) in the Bayes’ theorem. Now, let’s calculate the product of “Prior” and “Likelihood”:



Prior \* Likelihood is just the product of the 2 previous columns

You will notice that the 3 values (1/6, 1/3 and 0) do not add up to 1. That’s because we are missing our last element of the Bayes’ theorem: P(B). This happens to be the sum of those 3 values. By dividing each by that sum, we’ll scale them in a way that they add to 1. In our problem, the 3 values add to 1/2 (1/6 + 1/3 = 1/2). In order to find our posterior, we only have to divide the last column by 1/2:



Finally, our posterior!

As expected, the probability that the car is behind Door 3 is 0, since the host has already opened it. We see also that Door 2 is 2x more likely to hide the car than Door 1.

If you are particularly interested in this problem, or if you find the result too counter-intuitive, there’s a whole article about it [here](https://towardsdatascience.com/the-monty-hall-problem-9c4053ef0640).

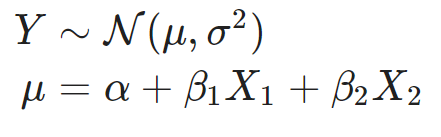
Ok, I know this was a toy problem, and that we used a very simple distribution as our prior. What happens when we face real-life, complex problems, with many variables, and where the priors are not that easily defined and the likelihoods are hard to calculate? Luckily, there’s a Python library to help us with that: PyMC3.

With PyMC3, you can use Bayesian inference for many things, including estimating parameters for regression and classification. You can do it using the built-in functions that come with the library or use it to build custom models from scratch. The first option is easier, but less flexible, while the second will give you more freedom on choosing your parameters.

Let’s take a look at how to use both options for classical problems such as regression and classification.

**Regression**

**Linear regression “from scratch”**



Bayesian linear regression definitions

If you are familiar with linear regressions, you will spot a few differences from the traditional models. The main one is that, in Bayesian regression, we don’t look at the parameters α, β1, β2 and σ² as fixed values, but rather as variables following distributions.

And it’s these distributions that we will try to estimate, by using the same method as before: define a prior distribution and update it with data to get a posterior distribution.

While writing this part, I hesitated between using real world data or generating fake data to run the regression on. I’ve decided to use fake data because that will allow us to compare our results with an actual ground truth: the parameters we used to generate the data.

**[IN]:**import arviz as az  
import matplotlib.pyplot as plt  
import numpy as np  
import pandas as pd  
import pymc3 as pm**# Setting a random seed for reproducibility**  
np.random.seed(23)**# True parameter values**  
alpha = -0.8  
beta = [-1, 1.5]  
sigma = 1**# Number of observations**  
size = 500**# Predictor variables**  
X1 = np.random.randn(size)  
X2 = np.random.randn(size) \* 0.8**# Outcome variable**  
Y = alpha + beta[0] \* X1 + beta[1] \* X2 + np.random.randn(size) \* sigma**# Putting our original data into a dataframe (we'll use it later on)**  
df = pd.DataFrame(  
 data = np.array([X1, X2, Y]),  
 index=['X1', 'X2', 'Y']  
).T

For now, we haven’t done much, except for importing the needed libraries and generating our data. Let’s now create our first model using PyMC3:

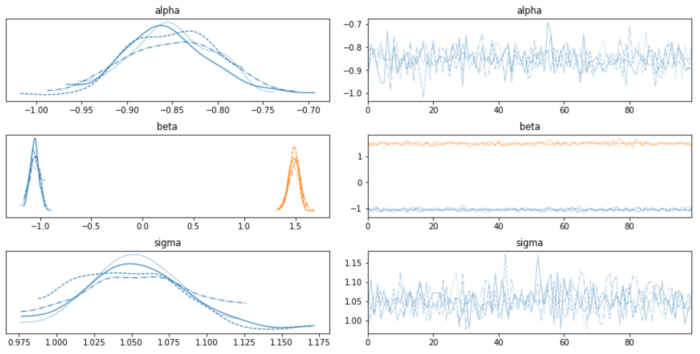
**[IN]:**  
with pm.Model() as model\_1:  
 **# Data**  
 X\_1 = pm.Data('X1', X1)  
 X\_2 = pm.Data('X2', X2) **# Priors**  
 alpha = pm.Normal("alpha", mu=0, sigma=10)  
 beta = pm.Normal("beta", mu=0, sigma=10, shape=2)  
 sigma = pm.HalfNormal("sigma", sigma=10) **# Likelihood**  
 mu = alpha + beta[0] \* X\_1 + beta[1] \* X\_2  
 Y\_obs = pm.Normal("Y\_obs", mu=mu, sigma=sigma, observed=Y)  
   
 **# Posterior**  
 trace = pm.sample(100, return\_inferencedata=False, chains=4)  
 az.plot\_trace(trace)

First, notice the format of the model construction, which is standard for PyMC3: you put everything (data, priors, likelihood and posterior) inside a “with” statement, where you name your model (in our case, “model\_1”). This model can then be accessed later on.

Again, you might be wondering why these priors. In this specific case, for alpha and beta we are using normal distributions centered around 0, since we don’t have any prior knowledge that could indicate a strong relationship between X and Y. Regarding the sigma=10 inside each prior, it is setting the standard deviation for the Normal distribution, so the higher this number, the more variance there is in our priors and, therefore, the less informative they will be (which can be a good thing if you are not very certain of those priors). The *HalfNormal* distribution for the sigma parameter will keep it positive.

Try testing different priors and see how much the end results change, which is actually a good thing to do, to see how robust your model is to having bad priors (the more it changes, the more you are relying on your priors).

Let’s take a look at the output of our model:



First model output

Each “row” in this image represents one of our parameters (alpha, beta, sigma), the left part represents the posterior distributions and the right part represents their convergence over time.

Take the top left plot, for instance: it shows the estimated posterior distribution for alpha. But wait, there are actually 4 distributions there, not one! That is because pm.sample() has a parameter called *chains*, set to 2 or the number of CPUs in your system, whichever is larger. In my case, that was equal to 4, but I decided to explicitly set it for illustrations purposes. *Chains* will set the number of independent chains running in parallel, which allows us to have some convergence statistics.

You can also see the notion of chains on the plots on the right side, which have one line per chain. What are those lines showing, though? Do you see the 100 in pm.sample()? That is setting the number of samples to draw. So those lines are basically showing the estimated most likely value for that parameter, in each of the 100 samples, for each of the 4 chains. For alpha, you can see that it turns around -0.8 (with a slight bias towards lower values, though).

In the middle left and right plots, you have the same information for beta 1 and beta 2 (you can tell them apart by the different colors).

As you can see, the estimated distributions are more or less coherent with the ground truth (some of the bias might come from the priors we set).

We have posterior distributions and their plots, but you might be wondering how do we make predictions with that. PyMC3 has a function called *fast\_sample\_posterior\_predictive*, that allows us to do exactly that:

**[IN]:**  
with model\_1:pm.set\_data({  
 'X1': [0.6, 0.02],  
 'X2': [-1.3, 0.3]  
 })  
 y\_test = pm.fast\_sample\_posterior\_predictive(trace, samples=100)print(y\_test['Y\_obs'].mean(axis=0))**[OUT]:**  
[-3.27941019 -0.31231568]

The output are the predictions for the 2 observations we put in our example, generated by taking 100 samples using our posterior distributions.

Additionally, if you need point estimates and not just distributions, you can get it by running *find\_MAP* and use these parameters to build a classical linear equation:

**[IN}:**  
pm.find\_MAP(model=model\_1)**[OUT]:**  
{'alpha': array(-0.85492716),  
 'beta': array([-1.05603871, 1.48859634]),  
 'sigma\_log\_\_': array(0.04380838),  
 'sigma': array(1.04478214)}

Even thouhg PyMC3 can take a while to run, it definitely makes our lives a lot easier. But what does it do, precisely?

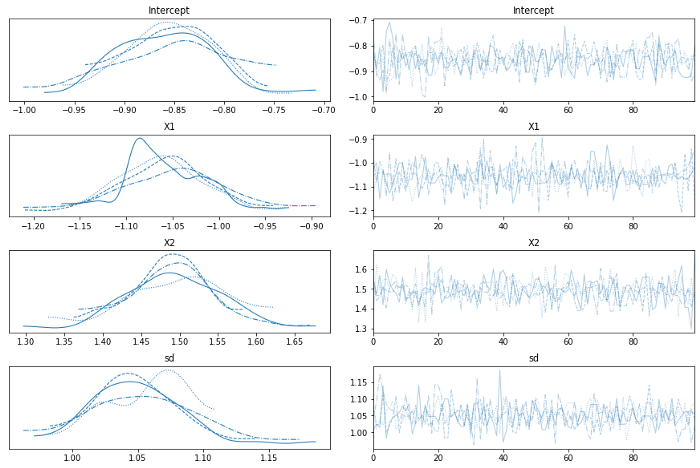
PyMC3 does not work exactly the same way we built our Monty Hall example, in the sense that it does not compute the exact posterior distributions. Instead, it makes a smart move by actually *sampling* data from the posterior in order to estimate it (that’s why you see the “sample” function in the “Posterior” part of our code).

Although this will not yield the exact distribution we are looking for, it gives us a good approximation in a reasonable amount of time. It fits models using methods based mostly on MCMC (Markov Chain Monte Carlo) and VI (variational inference) algorithms. If you want more mathematical detail on both methods, I suggest you read [this article](https://towardsdatascience.com/bayesian-inference-algorithms-mcmc-and-vi-a8dad51ad5f5).

**Linear regression using GLM**

Now, let’s see how we can use the GLM class from PyMC3 to get our posterior:

**[IN]:**  
from pymc3.glm import GLM# Creating our model  
with pm.Model() as model\_glm:  
 GLM.from\_formula('Y ~ X1 + X2', df)  
 trace = pm.sample(100)  
 pm.traceplot(trace)  
 plt.show()**[OUT]:**



GLM output

The construction of the model is almost self-explanatory, in the sense that all you have to do is write the formula for the GLM, and the rest goes on more or less as previously. The difference, obviously, lies in what is happening behind the scenes when you run the code. For instance, you will notice that, this time, you didn’t set priors for your parameters. Instead, PyMC3 is using default, non-informative priors, such as wide uniform and diffuse normal distributions, trying to add as little prior information as possible.

The interpretation of the results is the same as the previous example.

**Classification**

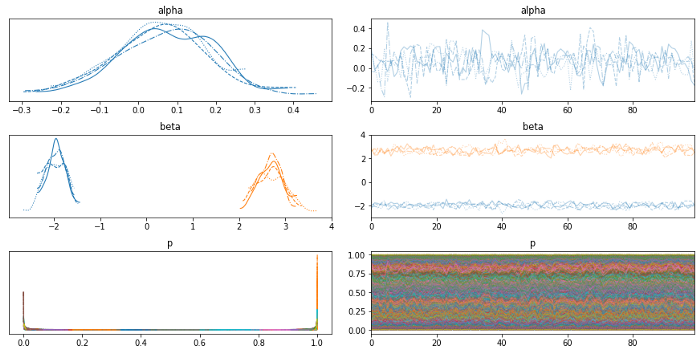
**Logistic regression “from scratch”**

For this example, let’s re-use the same data as before, but creating two classes for Y: those who are above the mean, and those which are not.

**[IN]:**  
**# Creating a binary variable**  
df['Y\_bin']=df['Y']>df['Y'].mean()

We’ll now apply a similar method as the one used for our linear regression, but using different functions for the likelihood:

**[IN]:**  
with pm.Model() as model\_log: **# Priors**  
 alpha = pm.Normal("alpha", mu=0, sigma=10)  
 beta = pm.Normal("beta", mu=0, sigma=10, shape=2) **# Likelihood**    
 p = pm.Deterministic('p', pm.math.sigmoid(alpha + beta[0] \* X1 + beta[1] \* X2))  
 Y\_obs = pm.Bernoulli("Y\_obs", p, observed=df['Y\_bin'])  
   
 **# Posterior**  
 trace = pm.sample(100, return\_inferencedata=False, chains=4)  
 az.plot\_trace(trace)**[OUT]:**



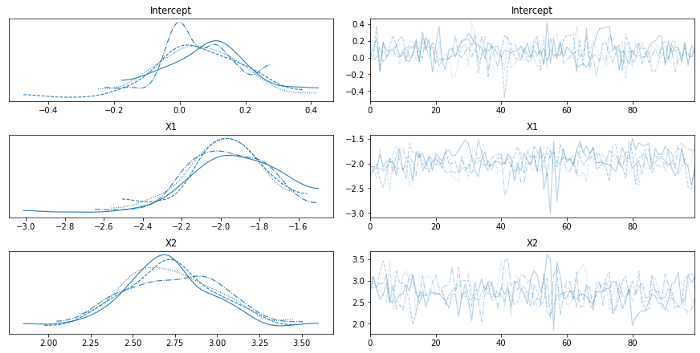
Output for our first logistic regression

First of all, keep in mind that the parameters we are estimating here are not the same we created for our linear regression, even though they have the same names. In the first case, we used them directly to create Y, whereas here we are estimating the parameters that lie within the logistic regression, so we shouldn’t use the initial values we set for alpha and beta as ground truth for these ones.

**Logistic regression using GLM**

Again, let’s try to solve the classification problem above, but using GLM:

**[IN]:**  
from pymc3.glm.families import Binomialwith pm.Model() as model\_glm\_logistic:  
 GLM.from\_formula('Y\_bin ~ X1 + X2', df, family=Binomial())  
 trace = pm.sample(100)  
 pm.traceplot(trace)  
 plt.show()



Output for logistic regression — GLM version

The difference between this version (logistic regression) to the first GLM (linear regression) remains mainly on the *family=Binomial()* parameter, at least in terms of constructing the code, obviously.

**Conclusion**

As you can see, the intuition behind Bayesian inference is simple: your posterior will be somewhere between your prior (subjective) beliefs and the data you observed, meaning different people will end up with different distributions. Yet, the methods that come from it are not that simple, since they require us to think in a different way: reasoning always in terms of distributions, and not of fixed parameters.

This was just an introduction to basic Bayesian statistics and inference, but the subject goes much deeper, and there are much more interesting methods that stem from it: Naive Bayesian Classifiers, Bayesian Neural Networks, Bayesian Hypothesis Testing, Bayesian Hierarchical Modeling and Bayesian Reinforcement Learning. Hopefully I’ll write about those topics in another article soon.

**Going further**

Here’s a list of some extra sources of information, if you want to dig deeper into Bayesian statistics: