Chapter 1

The Philosophy of Bayesian Inference

Think of what you would do in the following situation:

Example

> You are a skilled programmer, but bugs still slip into your code. After a particularly difficult implementation of an algorithm, you decide to test your code on a trivial example. It passes. You test the code on a harder problem. It passes once again. And it passes the next, *even more difficult*, test too! You are starting to believe that there are may be no bugs present...

If you think this way, then congratulations, you're already a Bayesian practitioner! Bayesian inference is simply updating your beliefs after considering new evidence. A Bayesian can rarely be certain about a result, but he or she can be very confident. Just like in the example above, we can never be 100% sure that our code is bug-free unless we test it on every possible problem; something rarely possible in practice. Instead, we can test it on a large number of problems, and if it succeeds we can feel more *confident* about our code. Bayesian inference works identically: we can only update our beliefs about an outcome, rarely can we be absolutely sure unless we rule out all other alternatives. We will see that being uncertain can have its advantages.

The Bayesian state of mind

Bayesian inference differs from more traditional statistical analysis by preserving *uncertainty* about our beliefs. At first, this sounds like a bad statistical technique. Isn't statistics all about deriving *certainty* from randomness? The Bayesian method believes that probability is better seen as a measure of *believability in an event*. In fact, we will see in a moment that this is the natural interpretation of probability.

For this to be clearer, we consider an alternative interpretation: *Frequentist* methods assume that probability is the long-run frequency of events (hence the bestowed title). For example, the *probability of plane accidents* under a frequentist philosophy is the *long-term frequency of plane accidents*. This makes logical sense for many probabilities, but becomes more difficult to understand when events have no long-term frequency of occurances. Consider: we often assign probabilities to outcomes of presidential elections, but the election itself only happens once! Frequentists get around this by invoking alternative realities [cite].

Bayesians, on the other hand, have a more intuitive approach. Bayesians interpret a probability as measure of *belief*, or knowledge, of an event occurring. A belief of 0 is you have no confidence that the event will occur; conversely, a belief of 1 implies you are absolutely certain of an event occurring.

Beliefs between 0 and 1 allow for weightings of other outcomes. This definition agrees with the probability of a plane accident example, for having observed the frequency of plane accidents, your belief should be equal to that frequency. Similarly, under this definition of probability being equal to beliefs, it is clear how we can speak about probabilities (read: belief) of presidential election outcomes.

Think about how we can extend this definition of probability to events that are not *really* random. That is, think about how we can extend this to anything that is fixed, but we are unsure about:

- Your code either has a bug in it or not, but we do not know for certain which is true. Though we have a belief about the presence or absence of a bug.
- A medical patient is exhibiting symptoms x, y and z. There are a number of diseases that could be causing all of them, but only has a single disease is present. A doctor has beliefs about which disease.
- My boss probably makes a lot of money.

This philosophy of treating beliefs as probability is natural to humans. We employ it constantly as we interact with the world and only see partial evidence. Alternatively, you have to be *trained* to think like a frequentist.

To align ourselves with traditional probability notation, we denote our belief about event A as $P(A)$.

John Maynard Keynes, a great economist and thinker, said

> "When the facts change, I change my mind. What do you do, sir?"

This quote reflects the way a Bayesian updates his or her beliefs after seeing evidence. Even - especially- if the evidence is counter to what was initialed believed, it cannot be ignored. We denote our updated belief as $P(A|X)$, interpreted as the probability of A given the evidence X. We call it the *posterior probability* so as to contrast the pre-evidence *prior probability*. Consider the posterior probabilities (read: posterior belief) of the above examples, after observing evidence X.:

- I. $P(A)$: This big, complex code likely has a bug in it.
  II. $P(A|X)$: The code passed all X tests; there still might be a bug, but its presence is less likely now.

- I. $P(A)$: The patient could have any number of diseases.
  II. $P(A|X)$: Performing a urine test generated evidence X, ruling out some of the possible diseases from consideration.

- I. $P(A)$: My boss probably makes a lot of money.
  II. $P(A|X)$: I saw by boss *making it rain* at a bar with X dollar bills. He's probably rich and a fool too.

It's clear that in each example we did not completely discard the prior belief after seeing new evidence, but we *re-weighted the prior* to incorporate the new evidence (i.e. we put more weight, or confidence, on some beliefs versus others).

By introducing prior uncertainity about events, we are already admitting that any guess we make is

potentially very wrong. After observing data, evidence, or other information, and we update our beliefs, our guess becomes *less wrong*. This is the opposite side of the prediction coin, where typically we try to be *more right*.

Bayesian Inference in Practice

If frequentist and Bayesian inference were computer programming functions, with inputs being statistical problems, then the two would be different in what they return to the user. The frequentist inference function would return a number, whereas the Bayesian function would return a *distribution*.

For example, in our debugging problem above, calling the frequentist function with the argument "My code passed all X tests; is my code bug-free?" would return a *YES*. On the other hand, asking our Bayesian function "Often my code has bugs. My code passed all X tests; is my code bug-free?" would return something very different: a distribution over *YES* and *NO*. The function might return

> *YES*, with probability 0.8; *NO*, with probability 0.2

This is very different from the answer the frequentist function returned. Notice that the Bayesian function accepted an additional argument: *"Often my code has bugs"*. This parameter, the *prior*, is that intuition in your head that says "wait- something looks different with this situation", or conversely "yes, this is what I expected". In our example, the programmer often sees debugging tests fail, but this time we didn't, which signals an alert in our head. By including the prior parameter, we are telling the Bayesian function to include our personal intuition. Technically this parameter in the Bayesian function is optional, but we will see excluding it has its own consequences.

As we aquire more and more instances of evidence, our prior belief is *washed out* by the new evidence. This is to be expected. For example, if your prior belief is something ridiculous, like "I expect the sun to explode today", and each day you are proved wrong, you would hope that any inference would correct you, or at least align your beliefs.

Denote N as the number of instances of evidence we possess. As we gather an *infinite* amount of evidence, say as $N \to \infty$, our Bayesian results align with frequentist results. Hence for large N, statistical inference is more or less objective. On the other hand, for small N, inference is much more *unstable*: frequentist estimates have more variance and larger confidence intervals. This is where Bayesian analysis excels. By introducing a prior, and returning a distribution (instead of an scalar estimate), we *preserve the uncertainity* to reflect the instability of stasticial inference of a small N.

One may think that for large N, one can be indifferent between the two techniques, and might lean towards the computational-simpler frequentist methods. An analysist should consider the following quote by Andrew Gelman (2005)[1], before making such a decision:

> Sample sizes are never large. If N is too small to get a sufficiently-precise estimate, you need to get more data (or make more assumptions). But once N is "large enough," you can start subdividing the data to learn more (for example, in a public opinion poll, once you have a good estimate for the entire country, you can estimate among men and women, northerners and southerners, different age groups, etc etc). N is never enough because if it were "enough" you'd already be on to the next problem for which you need

more data.

A note on *Big Data*

Paradoxically, big data's prediction problems are actually solved by relatively simple models [2]. Thus we can argue that big data's prediction difficulty does not lie in the algorithm used, but instead on the computational difficulties of storage and execution on big data. (One should also consider Gelman's qoute from above and ask "Do I really have a big data prediction problem?")

The much more difficult prediction problems involve *medium data* and, especially troublesome, *really small data*. Using a similar argument as Gelman's above, if big data problems are *big enough* to be easily solved, then we should be more interested in the smaller-sized data.

Our Bayesian framework

We are interested in beliefs, which can be interpreted as probabilities by thinking Bayesian. We have a *prior* belief in event A: it is what you believe before looking at any evidence, e.g., our prior belief about bugs being in our code before performing tests.

Secondly, we observe our evidence. To continue our example, if our code passes X tests, we want to update our belief to incorporate this. We call this new belief the *posterior* probability. Updating our belief is done via the following equation, known as Bayes' Theorem, after Thomas Bayes:

$$\begin{aligned} P(A|X) &= \frac{P(X|A)P(A)}{P(X)} \\ &\propto P(X|A)P(A) \ \ (\propto \text{is proportional to }) \end{aligned}$$

The above formula is not unique to Bayesian inference: it is a mathematical fact with uses outside Bayesian inference. Bayesian inference merely uses it.

Example

Let A denote the event that our code has no bugs in it. Let X denote the event that the code passes all debugging tests. For now, we will leave the prior probability as no bugs as a variable, i.e. $P(A) = p$. We are interested in $P(A|X)$, i.e. the probability of no bugs, given our debugging tests X. To use the formula above, we need to compute some quantities:

What is $P(X|A)$, i.e., the probability that the code passes X tests *given* there are no bugs? Well, it is equal to 1, for a code with no bugs will pass any tests.

$P(X)$ is a little bit trickier: The event X can be divided into two possibilities, event X with bugs (denoted $\sim A$, spoken *not* A), or event X without bugs (A). $P(X)$ can be represented as: [needs to be filled in better]

$$
\begin{aligned}
P(X) &= P(X \text{ and } A) + P(X \text{ and } \sim A) \\
&= P(X|A)P(A) + P(X|\sim A)P(\sim A) \\
&= P(X|A)p + P(X|\sim A)(1-p)
\end{aligned}
$$

We have already computed $P(X|A)$ above. On the other hand, $P(X|\sim A)$ is subjective: our code can pass tests but still have a bug in it, though the probability there is a bug present is less. Note this is dependent on the number of tests performed, the degree of complication in the tests, etc. Let's be conservative and assign $P(X|\sim A) = 0.5$. Then

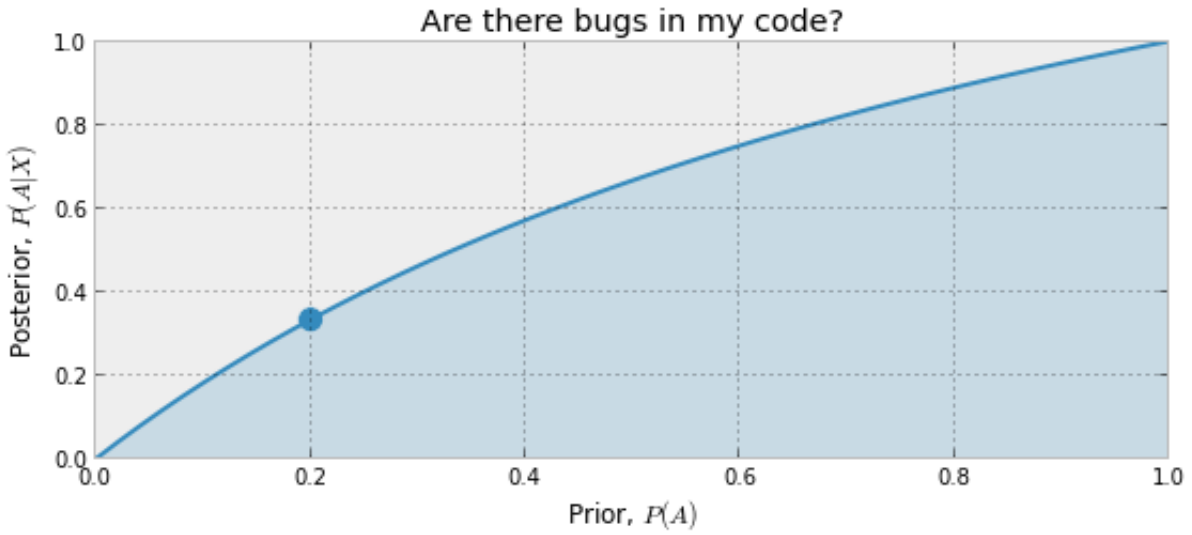
$$
\begin{aligned}
P(A|X) &= \frac{1 \cdot p}{1 \cdot p + 0.5(1-p)} \\
&= \frac{2p}{1+p}
\end{aligned}
$$

This is the posterior probability distribution. What does it look like as a function of our prior, $p \in [0,1]$?

```
figsize(9,3.5)

p = np.linspace( 0,1, 50)
plt.plot( p, 2*p/(1+p), color = "#348ABD" )
plt.fill_between( p, 2*p/(1+p), alpha = .2, facecolor = ["#348ABD"])
plt.scatter( 0.2, 2*(0.2)/1.2, s = 140, c ="#348ABD"  )
plt.xlim( 0, 1)
plt.ylim( 0, 1)
plt.xlabel( "Prior, $P(A)$")
plt.ylabel("Posterior, $P(A|X)$")
plt.title( "Are there bugs in my code?")
```

```
<matplotlib.text.Text at 0x8667eb8>
```



We can see the biggest gains if we observe the X tests passed are when the prior probability, p, is low. Let's settle on a specific value for the prior. I'm a (I think) strong programmer, so I'm going to give myself a realistic prior of 0.20, that is, there is a 20% chance that I write code bug-free. To be more realistic, this prior should be a function of how complicated is code is and large the code is, but let's pin it at 0.20. Then my updated belief that my code is bug-free is 0.33.

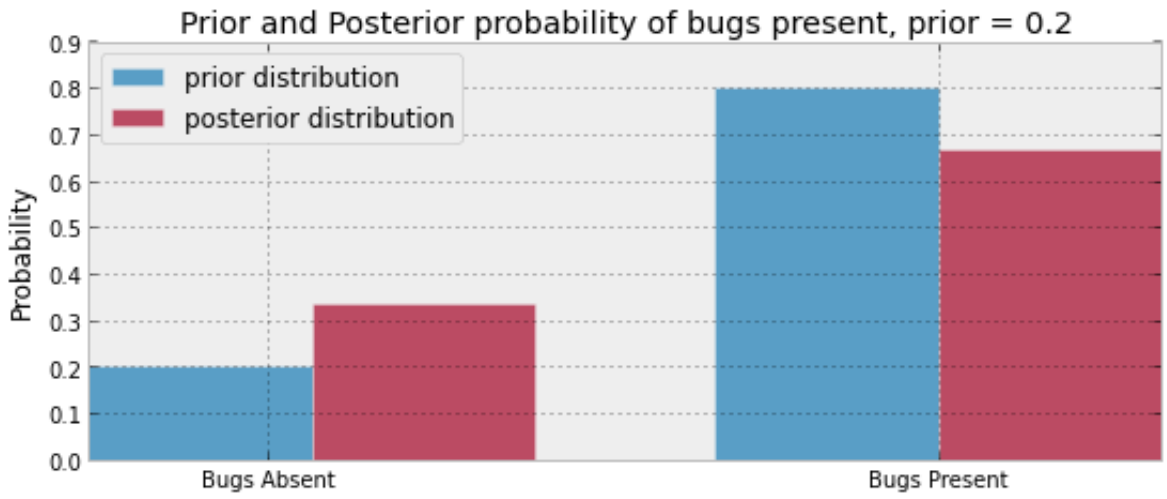
Let's not forget from the idea that the prior is a probability distribution: p is the prior probability that there *are no bugs*, so $1-p$ is the prior probability that there *are bugs*. What does our prior probability distribution look like?

Similarly, our posterior is also a probability distribution, with $P(A|X)$ the probability there is no bug *given we saw all tests pass*, hence $1-P(A|X)$ is the probability there is a bug *given all tests passed*. What does our posterior probability distribution look like? Below is a graph of both the prior and the posterior distributions.

```
prior = [0.20, 0.80]
postierior =  [1./3, 2./3]
plt.bar( [0,.7], prior ,alpha = 0.80, width = 0.25, \
            color = "#348ABD", label = "prior distribution" )
plt.bar( [0+0.25,.7+0.25], postierior ,alpha = 0.7, \
          width = 0.25, color =  "#A60628", label = "posterior distribution" )

plt.xticks( [0.20,.95], ["Bugs Absent", "Bugs Present"] )
plt.title("Prior and Posterior probability of bugs present, prior = 0.2")
plt.ylabel("Probability")
plt.legend(loc="upper left")
```

```
<matplotlib.legend.Legend at 0x8275d30>
```



Notice that after we observed X occur, the probability of no bugs present increased. By increasing the number of tests, we can approach confidence (probability 1) that there are not bugs.

This was a very simple example, but the mathematics from here only becomes difficult except for artifically constructed instances. We will see that this math is actually unnecessary. First we must broaden our modeling tools.

Probability Distributions

Let's quickly recall what a probability distribution is: Let Z be some random variable. Then associated with Z is a probability distribution function that assigns probabilities to the different values Z can take. There are three cases:

- Z is discrete: Discrete random variables may only assume values on a specified list. Things like populations, movie ratings, and number of votes are all discrete random variables. It's more clear when we contrast it with...
- Z is continuous: Continuous random variable can take on arbitrarily exact values. For example, temperature, speed, time, color are all modeled as continuous variables because you can constantly make the values more and more precise.
- Z is mixed: Mixed random variables assign probabilities to both discrete and continuous random variables, i.e. is is a combination of the above two categories.

Discrete Case

If Z is discrete, then its distribution is called a *probability mass function*, which measures the probability Z takes on the value k, denoted $P(Z = k)$. Note that the probability mass function completely describes the random variable Z, that is, if we know the mass function, we know how Z should behave. There are popular probability mass functions that consistently appear: we will introduce them as needed, but let's introduce the first very useful probability mass function. We say Z is *Poisson*-distributed if:

$$P(Z = k) = \frac{\lambda^k e^{-\lambda}}{k!}, \quad k = 0, 1, 2, \ldots$$

What is λ? It is called the parameter, and it describes the shape of the distribution. For the Poisson random variable, λ can be any positive number. By increasing λ, we add more probability to larger values, and conversely by decreasing λ we add more probability to smaller values. Unlike λ, which can be any positive number, k must be a non-negative integer, i.e., k must take on values 0,1,2, and so on. This is very important, because if you wanted to model a population you could not make sense of populations with 4.25 or 5.612 members.

If a random variable Z has a Poisson mass distribution, we denote this by writing

$$Z \sim \text{Poi}(\lambda)$$

One very useful property of the Poisson random variable, given we know λ, is that its expected value is equal to the parameter, ie.:

$$E[\, Z \mid \lambda \,] = \lambda$$

We will use this property often, so it's something useful to remember.

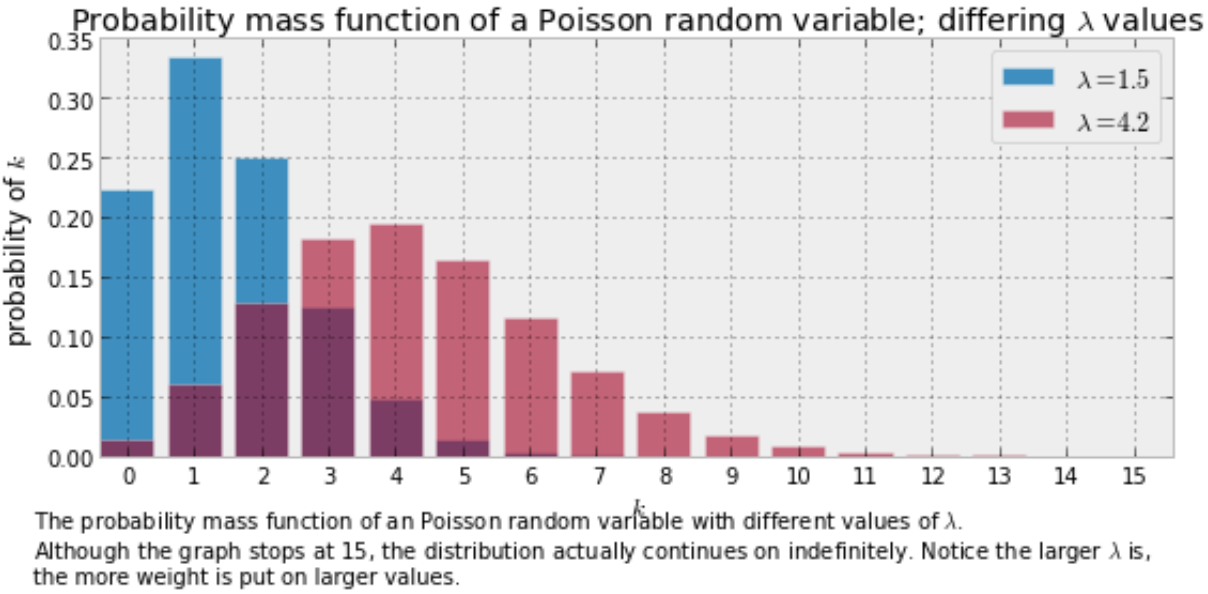
```
import scipy.stats as stats
a = np.arange( 16 )
poi = stats.poisson
lambda_ = [1.5, 4.25 ]
colours = ["#348ABD", "#A60628"]

plt.bar( a, poi.pmf( a, lambda_[0]), color=colours[0], \
    label = "$\lambda = %.1f$"%lambda_[0], alpha = 0.95)
plt.bar( a, poi.pmf( a, lambda_[1]), color=colours[1], \
    label = "$\lambda = %.1f$"%lambda_[1], alpha = 0.60)

plt.xticks( a + 0.4, a )
plt.legend()
```

```
plt.ylabel("probability of $k$")
plt.xlabel("$k$")
plt.title("Probability mass function of a Poisson random variable; differing \
$\lambda$ values")
plt.text(-1., -.106, """The probability mass function of an Poisson random \
variable with different values of $\lambda$.
Although the graph stops at 15, the \
distribution actually continues on indefinitely. Notice the larger $\lambda$ is,
the more weight is put on larger values.""")
```

```
<matplotlib.text.Text at 0x994af60>
```



The probability mass function of an Poisson random variable with different values of λ.
Although the graph stops at 15, the distribution actually continues on indefinitely. Notice the larger λ is, the more weight is put on larger values.

Continuous Case

Instead of a probability mass function, a continuous random variable has a *probability density function*. This might seem like unnecessary nomenclature, but the density function and the mass function are very different creatures. An example of continuous random variable is a random variable with a *exponential density*. The density function for an exponential random variable looks like:

$$f_Z(z|\lambda) = \lambda e^{-\lambda z}, \quad z \geq 0$$

Like the Poisson random variable, an exponential random variable can only take on non-negative values. But unlike a Poisson random variable, the exponential can take on *any* non-negative values, like 4.25 or 5.612401. This makes it a poor choice for count data, which must be integers, but a great choice for time data, or temperature data (measured in Kelvins, of course), or any other precise variable. Below are two probability density functions with different λ value.

When a random variable Z has an exponential distribution with parameter λ, we say Z *is exponential* and write

$$Z \sim \text{Exp}(\lambda)$$

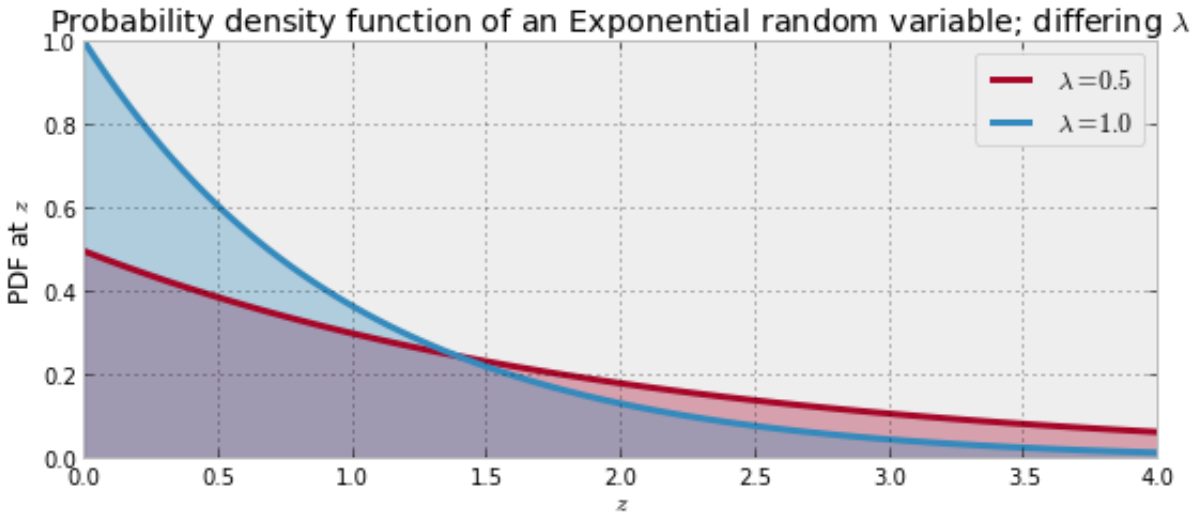
Given a specific λ, the expected value of an exponential random variable is equal to the inverse of λ, that is:

$$E[\ Z\ |\ \lambda\ ] = \frac{1}{\lambda}$$

```
a = np.linspace(0,4, 100)
expo = stats.expon
lambda_ = [0.5, 1]
colours = [ "#A60628", "#348ABD"]
for l,c in zip(lambda_,colours):
    plt.plot( a, expo.pdf( a, scale=1./l), lw=3, color=c, label = \
    "$\lambda = %.1f$"%l)
    plt.fill_between( a, expo.pdf( a, scale=1./l), color=c, alpha = .33)

plt.legend()
plt.ylabel("PDF at $z$")
plt.xlabel("$z$")
plt.title("Probability density function of an Exponential random variable;\
 differing $\lambda$")
plt.text(-.1, -.34, """The probability density function of a Exponential random \
variable with different values of $\lambda$.
Although the graph stops at 4.0, the \
distribution actually continues on indefinitely. Notice the larger $\lambda$ is,
the more weight is put on larger values.""")
```

```
<matplotlib.text.Text at 0x9bc39b0>
```



The probability density function of a Exponential random variable with different values of λ.
Although the graph stops at 4.0, the distribution actually continues on indefinitely. Notice the larger λ is, the more weight is put on larger values.

But what is λ ?

That question is what motivates statistics. In the real world, λ is hidden from us. We only see Z, and must go backwards to try and determine λ. The problem is so difficult because there is not a one-to-one mapping from Z to λ. Many different methods have been created to solve the problem of estimating λ, but since λ is never actually observed, no one can say for certain which method is better!

Bayesian inference is concerned with *beliefs* about what λ is. Rather than try to guess λ exactly, we can only talk about what λ is likely to be by assigning a probability distribution to λ.

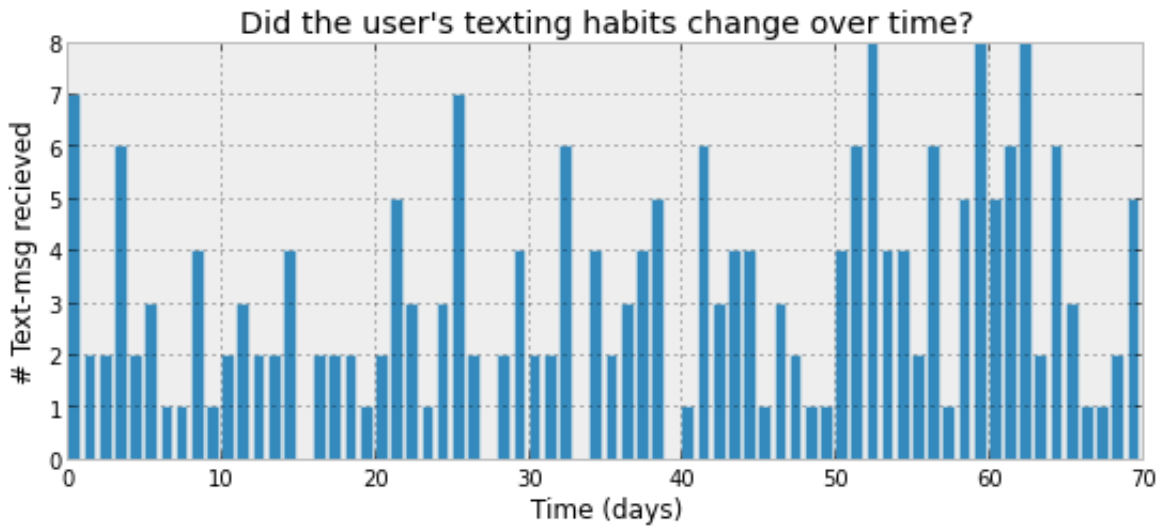
This might seem odd at first: after all, λ is fixed, it is not (necessarily) random! How can we assign probabilities to a non-random event. Ah, we have fallen for the frequentist interpretation. Recall, under our Bayesian philosophy, we *can* assign probabilties if we interpret them as beliefs. And it is entirely acceptable to have *beliefs* about the parameter λ.

Example

Let's try to model a more interesting example, concerning text-message rates:

> You are given a series of text-message counts from a user of your system. The data, plotted over time, appears in the graph below. You are curious if the user's text-messaging habits changed over time, either gradually or suddenly. How can you model this?

```
count_data = np.loadtxt("chp1data/textmsgdata.csv")
plt.bar( np.arange( len(count_data) ), count_data, color ="#348ABD"  )
plt.xlabel( "Time (days)")
plt.ylabel("# Text-msg recieved")
plt.title("Did the user's texting habits change over time?")
print
```



How can we start to model this? Well, as I conveniently already introduced, a Poisson random variable would be a very appropriate model for this *count* data. Denoting a day i's text-message count C_i,

$$C_i \sim \text{Poisson}(\lambda)$$

We are not sure about what the λ parameter is though. Looking at the chart above, it appears that the rate becomes higher at some later date, which is equivalently saying the parameter λ increases at some later date (recall a higher λ means more probability on larger values).

How can we mathematically represent this? We can think, that at some later date (call it τ), the

parameter λ suddenly jumps to a higher value. So we create two λ parameters, one for before the day τ, and one for after. In literature, a sudden transition like this would be called a *switchpoint*:

$$\lambda = \begin{cases} \lambda_1 & \text{if } t < \tau \\ \lambda_2 & \text{if } t \geq \tau \end{cases}$$

If, in reality, no sudden change occurred, the λ's should look about equal. What would be a good prior distribution on λ_1 and λ_2?

Recall that λ_i, $i = 1, 2$, can be any positive number. The *exponential* random variable has a density function for any positive number. But again, we need a parameter for this exponential distribution: call it α.

$$\begin{aligned} \lambda_1 &\sim \text{Exp}(\alpha) \\ \lambda_2 &\sim \text{Exp}(\alpha) \end{aligned}$$

α is called a *hyper-parameter*, literally a parameter that influence other parameters. The influence is not too strong, so we can choose α liberally. A good rule of thumb is to set the exponential parameter equal to the inverse of the average of the count data, since

$$\frac{1}{N}\sum_{i=0}^{N} C_i \approx E[\, \lambda \mid \alpha] = \frac{1}{\alpha}$$

Alternatively, and something I encourage the reader to try, is to have two priors: one for each λ; creating two exponential distributions with different α values reflects our belief was that the rate changed (increased) after some period.

What about τ? Well, due to the randomness, it is too difficult to pick out when τ might have occurred. Instead, we can assign an *uniform prior belief* to every possible day. This is equivalent to saying

$$\tau \sim \text{DiscreteUniform}(0,70)$$

So after all this, what does our prior distribution look like? Frankly, *it doesn't matter*. What we should understand is that it would be an ugly, complicated, mess involving symbols only a mathematician would love. And things would only get uglier the more complicated our model. Next, we turn to PyMC, which is agnostic to the mathematical monster we created.

Introducing our first hammer: PyMC

PyMC is a Python library for programming Bayesian analysis. It is a fast, well-maintained library. The only unfortunate part is that documentation can be lacking in application areas, especially the bridge between problem to solution. This book will aids this problem, and explains why PyMC is so cool.

We will model the above problem using the PyMC library. This type of programming is called *probabilistic programming*, an unfortunate misnomer that invokes ideas of randomly-generated code and has likely confused and frightened users from the field. The code is not random. The title is given because we create probability models using variables as the model's components. This will be

the last time I use the term *probablistic programming*. Instead, I'll simply use *programming*, as that is what it really is.

The PyMC code is easy to follow along: the only novel thing should be the syntax, and I will interrupt the code to explain sections. Simply remember we are representing the model's components $(\tau, \lambda_1, \lambda_2 \ )$ as variables:

```
import pymc as mc

n = count_data.shape[0]

alpha = 1.0/count_data.mean() #recall count_data is
                              #   the variable that holds our txt counts

lambda_1 = mc.Exponential( "lambda_1",  alpha )
lambda_2 = mc.Exponential( "lambda_2", alpha )

tau = mc.DiscreteUniform( "tau", lower = 0, upper = n )
```

In the above code, we create the PyMC variables corresponding to λ_1, λ_2 in lines `8,9`. We assign them to PyMC's *stochastic variables*, called stochastic variables because they are treated by the backend as random number generators. We can test this by calling their built-in `random()` method.

```
print "Random output:", tau.random(),tau.random(), tau.random()
```

```
Random output: 2 15 49
```

```
@mc.deterministic
def lambda_( tau = tau, lambda_1 = lambda_1, lambda_2 = lambda_2 ):
    out = np.zeros( n )
    out[:tau] = lambda_1 #lambda before tau is lambda1
    out[tau:] = lambda_2 #lambda after tau is lambda1
    return out
```

This code is creating a new function `lambda_`, but really we think of it as a random variable: the random variable λ from above. Note that because `lambda_1`, `lambda_2` and `alpha` are random, `lambda_` will be random. We are not fixing any variables yet. The `@mc.deterministic` is a decorator to tell PyMC that this is a deterministic function, i.e., if the arguments were deterministic (which they are not), the output would be deterministic as well.

```
observation = mc.Poisson( "obs", lambda_, value = count_data, observed = True)

model = mc.Model( {"obs":observation, "lamba1":lambda_1, \
                   "lambda2":lambda_2, "tau":tau} )
```

The variable `observations` combines our data, `count_data`, with our proposed data-generation scheme, given by the variable `lambda_`, through the `value` keyword. We also set `observed = True` to tell PyMC that this should stay fixed in our analysis. Finally, PyMC wants us to collect all the variables of interest and create a `Model` instance out of them. This makes our life easier when we try to retrieve the results.

```
mcmc = mc.MCMC(model)
mcmc.sample( 300000, 200000, 3, verbose = 0 )
```

```
[****************100%******************]  300000 of 300000 complete
```

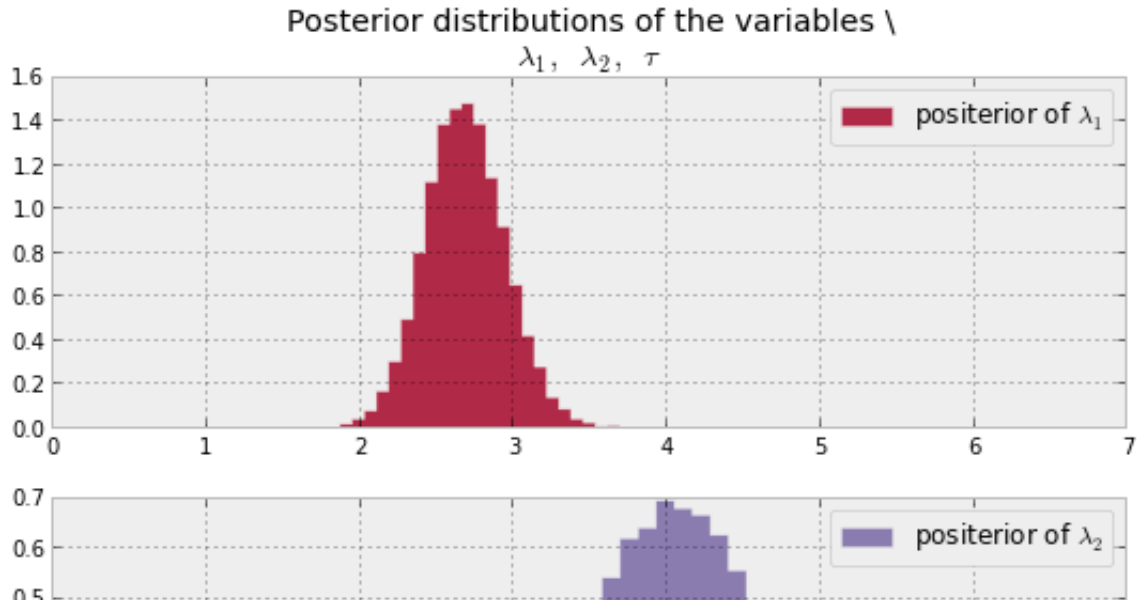
The above code will be explained in the Chapter 2, but this is where our results come from. The machinery being employed is called *Monte Carlo Markov Chains*. It returns random variables from the posterior distributions of λ_1, λ_2 and τ. Be can plot a histogram of the random variables to see what the posterior distribution looks like. Below, we collect the samples (called *traces* in MCMC literature).

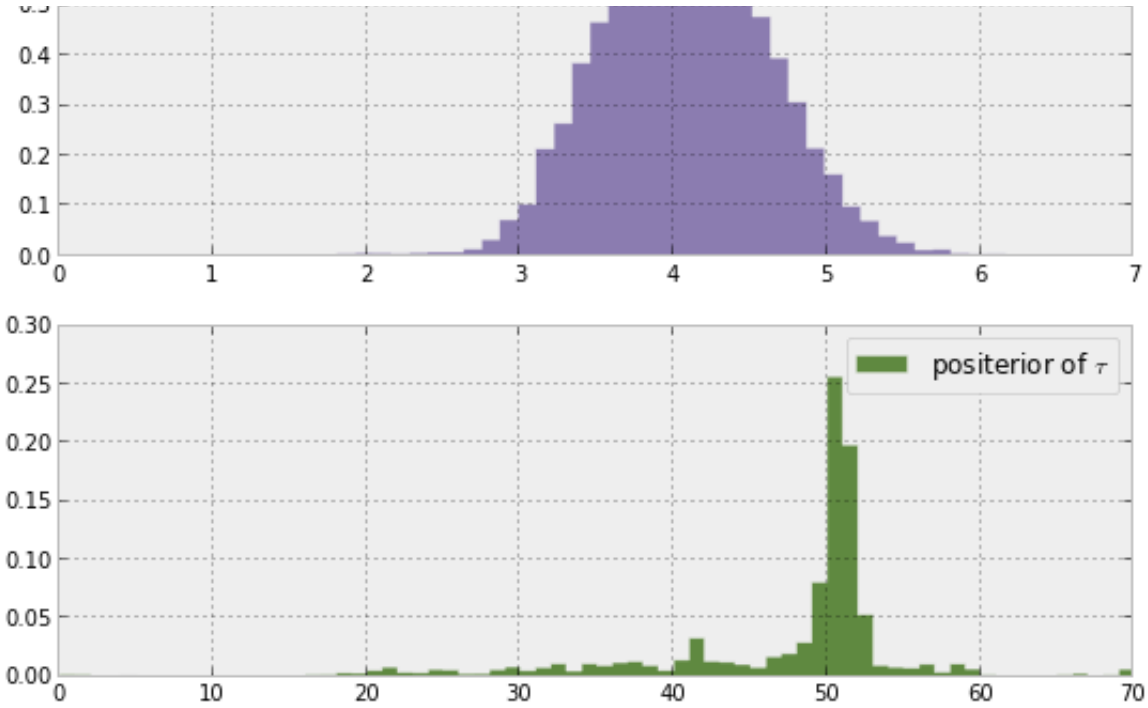
```
lambda_1_samples = mcmc.trace( 'lambda_1' )[:]
lambda_2_samples = mcmc.trace( 'lambda_2' )[:]
tau_samples = mcmc.trace( 'tau' )[:]
```

```
figsize(9, 10)
#histogram of the samples:
plt.subplot(311)
plt.xlim( 0, 7)
plt.hist( lambda_1_samples, histtype='stepfilled', bins = 120, alpha = 0.85, \
        label = "positerior of $\lambda_1$", color = "#A60628",normed = True )
plt.legend()
plt.title(r"Posterior distributions of the variables \
$\lambda_1, \;\lambda_2, \;\tau$")

plt.subplot(312)
plt.xlim( 0, 7)
plt.hist( lambda_2_samples,histtype='stepfilled', bins = 120, alpha = 0.85, \
         label = "positerior of $\lambda_2$",color="#7A68A6",normed = True )
plt.legend()

plt.subplot(313)
plt.hist( tau_samples, bins = 70, alpha = 0.85, label = r"positerior of $\tau$", \
         color="#467821", normed = True, histtype='stepfilled' )
plt.legend()
print
```





Interpretation

Recall that the Bayesian methodology returns a *distribution*, hence why we have distributions to describe the unknown λ's and τ. What have we gained? Immediately we can see the uncertainty in our estimates: the more variance in the distribution, the less certain our posterior belief should be. We can also say what a plausible value for the parameters might be: λ_1 probably falls between 2.25 and 2.75, and λ_2 probably falls between 3.5 and 4.5. The distributions of the two λs look very different, suggesting likely there was a change in the user's text-message behavior.

Also notice that posteriors' distributions do not look like any Poisson distributions. They are really not anything we recognize. But this is OK. This is one of the benefits of taking a computational point-of-view. If we had instead done this mathematically, we would have been stuck with a very intractable (and messy) distribution. Via computations, we are agnostic to the tractability.

Our analysis also returned a distribution for what τ might be. Had no change occurred, or the change been gradual, the posterior distribution of τ would have been more spread out. On the contrary, it is very peaked. It appears that near day 50, the individual's text-message behavior suddenly changed.

Why would I want samples from the posterior, anyways?

We will deal with this question for the remainder of the book, and it is an understatement to say we can perform amazingly useful things. For now, let's finishing with using posterior samples to answer the follow question: what is the expected number of texts at day t, $0 \le t \le 70$? Recall that the expected value of a Poisson is equal to its parameter λ, then the question is equivalent to *what is the expected value of λ at time t*?

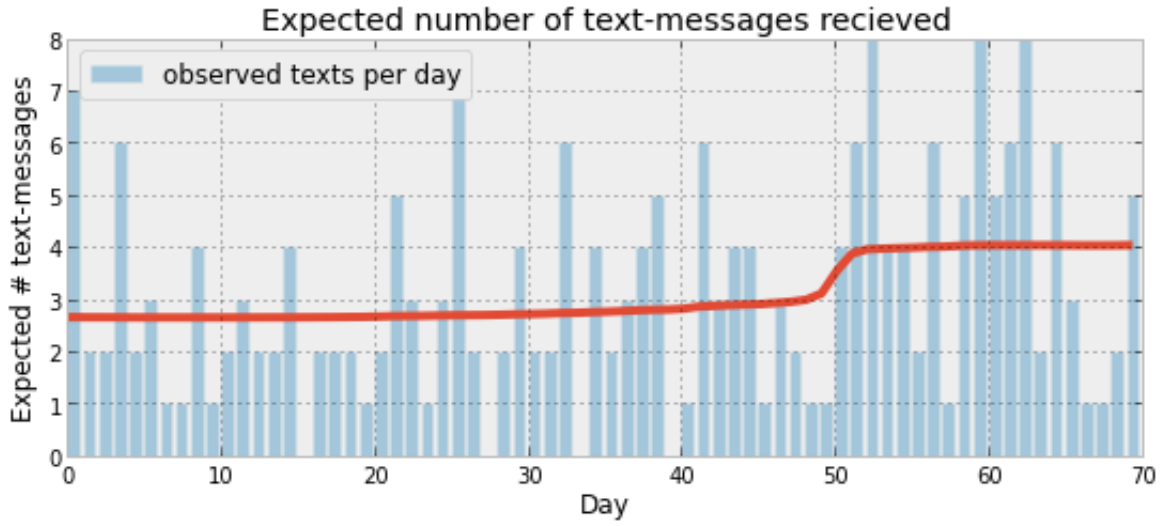
In the code below, we are calculating the following: Let i index a particular sample from the posterior distributions. Given a day t, we average over all λ_i on that day t, using $\lambda_{1,i}$ if $t < \tau_i$ else we use $\lambda_{2,i}$.

```
figsize( 9, 3.5)
N = tau_samples.shape[0]
expected_texts_per_day = np.zeros(70)
for  day in range(0, 70):
    ix = day < tau_samples
    expected_texts_per_day[day] = ( lambda_1_samples[ix].sum() + \
                                        lambda_2_samples[~ix].sum() ) / N

plt.plot( range(0,70), expected_texts_per_day, lw =4, color = "#E24A33" )
plt.ylim( 0, 8)
plt.xlabel( "Day" )
plt.ylabel( "Expected # text-messages" )
plt.title( "Expected number of text-messages recieved")

plt.bar( np.arange( len(count_data) ), count_data, color ="#348ABD", alpha = 0.4,\
label="observed texts per day")
plt.legend(loc="upper left")
```

```
<matplotlib.legend.Legend at 0x14def240>
```



Our analysis shows strong support that the user's behavior suddenly changed, versus no change (λ_1 would appear like λ_2 had this been true), versus a gradual change (more variation in the posterior of τ had this been true). We can speculate what might have caused this: a cheaper text-message rate, a recent weather-2-text subscription, or a new relationship.

The next Chapter will explore PyMC through examples.

Exercises

1. Using `lambda_1_samples` and `lambda_2_samples`, what is the mean of the positior

distributions of λ_1 and λ_2?

```
#type your code here.
```

2. What is the expected percent increase text-message rates? `hint:` compute the mean of `lambda_1_samples/lambda_2_samples`. Note that quanitity is very different from `lambda_1_samples.mean()/lambda_2_samples.mean()`.

```
#type your code here.
```

3. Looking at the posterior distribution graph of τ, why do you think there is a small number of posterior τ samples near 0? `hint:` Look at the data again.

4. What is the mean of λ_1 given τ is less than 45.

```
#type your code here.
```

References

- [1] Gelman, Andrew. N.p.. Web. 22 Jan 2013. http://andrewgelman.com/2005/07/n_is_never_larg/.
- [2] Norvig, Peter. 2009. *The Unreasonable Effectivness of Data*.