Ch1_Introduction_PyMC3

January 4, 2018

1 Probabilistic Programming

2 and Bayesian Methods for Hackers

Version 0.1 `Original content created by Cam Davidson-Pilon`

`Ported to Python 3 and PyMC3 by Max Margenot (@clean_utensils) and Thomas Wiecki (@twiecki) at Quantopian (@quantopian)` ___

Welcome to *Bayesian Methods for Hackers*. The full Github repository is available at github/Probabilistic-Programming-and-Bayesian-Methods-for-Hackers. The other chapters can be found on the project's homepage. We hope you enjoy the book, and we encourage any contributions!

3 Chapter 1

3.1 The Philosophy of Bayesian Inference

> 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 may be no bugs in this code...

If you think this way, then congratulations, you already are thinking Bayesian! 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, but still not certain. Bayesian inference works identically: we update our beliefs about an outcome; rarely can we be absolutely sure unless we rule out all other alternatives.

3.1.1 The Bayesian state of mind

Bayesian inference differs from more traditional statistical inference by preserving *uncertainty*. At first, this sounds like a bad statistical technique. Isn't statistics all about deriving *certainty* from randomness? To reconcile this, we need to start thinking like Bayesians.

The Bayesian world-view interprets probability as measure of *believability in an event*, that is, how confident we are in an event occurring. 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 of probability: *Frequentist*, known as the more *classical* version of statistics, 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 interpreted as the *long-term frequency of plane accidents*. This makes logical sense for many probabilities of events, but becomes more difficult to understand when events have no long-term frequency of occurrences. 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 and saying across all these realities, the frequency of occurrences defines the probability.

Bayesians, on the other hand, have a more intuitive approach. Bayesians interpret a probability as measure of *belief*, or confidence, of an event occurring. Simply, a probability is a summary of an opinion. An individual who assigns a belief of 0 to an event has no confidence that the event will occur; conversely, assigning a belief of 1 implies that the individual is 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, an individual's belief should be equal to that frequency, excluding any outside information. Similarly, under this definition of probability being equal to beliefs, it is meaningful to speak about probabilities (beliefs) of presidential election outcomes: how confident are you candidate A will win?

Notice in the paragraph above, I assigned the belief (probability) measure to an *individual*, not to Nature. This is very interesting, as this definition leaves room for conflicting beliefs between individuals. Again, this is appropriate for what naturally occurs: different individuals have different beliefs of events occurring, because they possess different *information* about the world. The existence of different beliefs does not imply that anyone is wrong. Consider the following examples demonstrating the relationship between individual beliefs and probabilities:

- I flip a coin, and we both guess the result. We would both agree, assuming the coin is fair, that the probability of Heads is 1/2. Assume, then, that I peek at the coin. Now I know for certain what the result is: I assign probability 1.0 to either Heads or Tails (whichever it is). Now what is *your* belief that the coin is Heads? My knowledge of the outcome has not changed the coin's results. Thus we assign different probabilities to the result.

- 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 a single disease is present. A doctor has beliefs about which disease, but a second doctor may have slightly different beliefs.

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 truths, but gather evidence to form beliefs. 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 $P(A)$. We call this quantity the *prior probability*.

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 initially believed, the evidence cannot be ignored. We denote our updated belief as $P(A|X)$, interpreted as the probability of A given the evidence X. We call the updated belief the *posterior probability* so as to contrast it with the prior probability. For example, consider the posterior probabilities (read: posterior beliefs) of the above examples, after observing some evidence X:

1. $P(A)$: the coin has a 50 percent chance of being Heads. $P(A|X)$: You look at the coin, observe a Heads has landed, denote this information X, and trivially assign probability 1.0 to Heads and 0.0 to Tails.

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

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

It's clear that in each example we did not completely discard the prior belief after seeing new evidence X, 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 uncertainty about events, we are already admitting that any guess we make is potentially very wrong. After observing data, evidence, or other information, we update our beliefs, and our guess becomes *less wrong*. This is the alternative side of the prediction coin, where typically we try to be *more right*.

3.1.2 Bayesian Inference in Practice

If frequentist and Bayesian inference were 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, representing an estimate (typically a summary statistic like the sample average etc.), whereas the Bayesian function would return *probabilities*.

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: probabilities of *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 is the *prior*. By including the prior parameter, we are telling the Bayesian function to include our belief about the situation. Technically this parameter in the Bayesian function is optional, but we will see excluding it has its own consequences.

Incorporating evidence As we acquire 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 better. Bayesian inference will correct this belief.

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 (often) 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 probabilities (instead of a scalar estimate), we *preserve the uncertainty* that reflects the instability of statistical inference of a small N dataset.

One may think that for large N, one can be indifferent between the two techniques since they offer similar inference, and might lean towards the computationally-simpler, frequentist methods. An individual in this position 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.). N is never enough because if it were "enough" you'd already be on to the next problem for which you need more data.

3.1.3 Are frequentist methods incorrect then?

No.

Frequentist methods are still useful or state-of-the-art in many areas. Tools such as least squares linear regression, LASSO regression, and expectation-maximization algorithms are all powerful and fast. Bayesian methods complement these techniques by solving problems that these approaches cannot, or by illuminating the underlying system with more flexible modeling.

A note on *Big Data* Paradoxically, big data's predictive analytic problems are actually solved by relatively simple algorithms [2][4]. 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 quote from above and ask "Do I really have big data?")

The much more difficult analytic 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 readily solved, then we should be more interested in the *not-quite-big enough* datasets.

3.1.4 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, beliefs formed by previous information, e.g., our prior belief about bugs being in our code before performing tests.

Secondly, we observe our evidence. To continue our buggy-code 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 its discoverer Thomas Bayes:

$$P(A|X) = \frac{P(X|A)P(A)}{P(X)} \tag{1}$$

$$\tag{2}$$

$$\propto P(X|A)P(A) \quad (\propto \text{ is proportional to }) \tag{3}$$

The above formula is not unique to Bayesian inference: it is a mathematical fact with uses outside Bayesian inference. Bayesian inference merely uses it to connect prior probabilities $P(A)$ with an updated posterior probabilities $P(A|X)$.

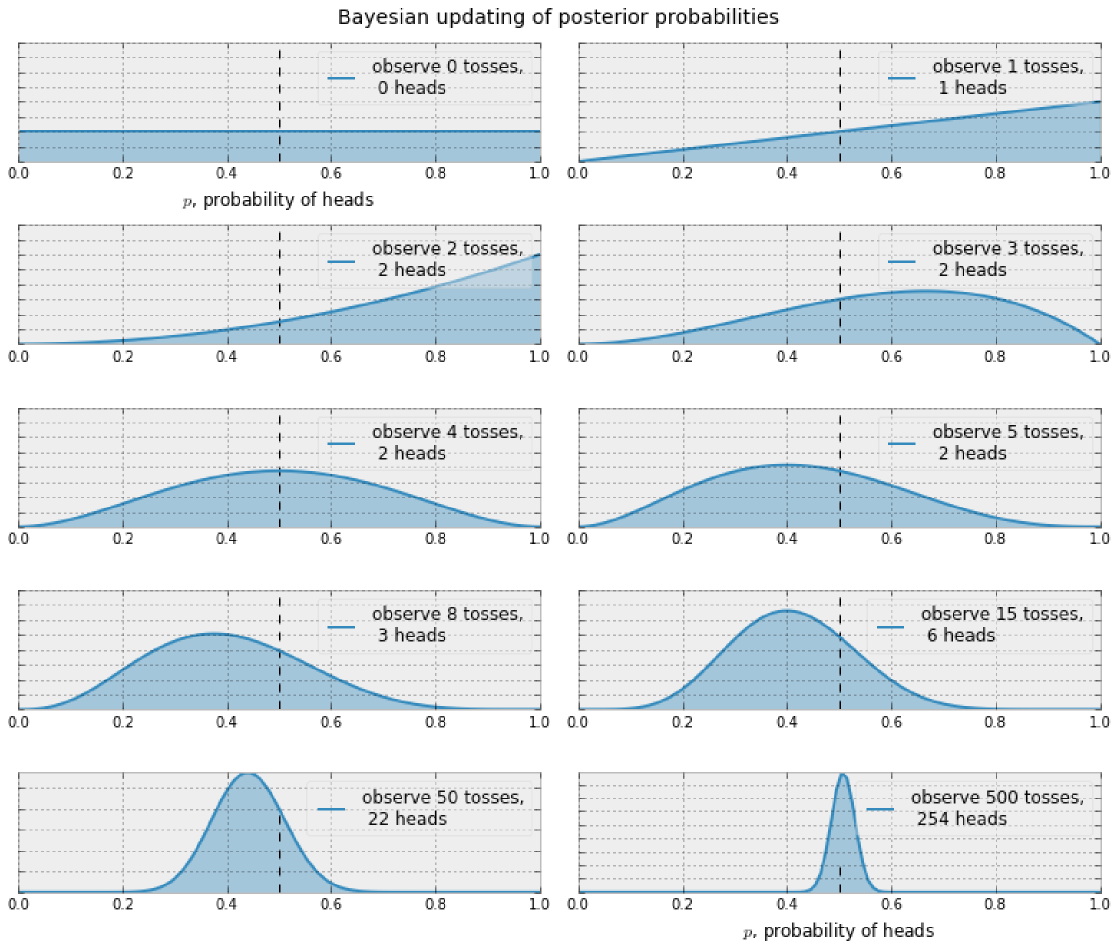
Example: Mandatory coin-flip example Every statistics text must contain a coin-flipping example, I'll use it here to get it out of the way. Suppose, naively, that you are unsure about the probability of heads in a coin flip (spoiler alert: it's 50%). You believe there is some true underlying ratio, call it p, but have no prior opinion on what p might be.

We begin to flip a coin, and record the observations: either H or T. This is our observed data. An interesting question to ask is how our inference changes as we observe more and more data? More specifically, what do our posterior probabilities look like when we have little data, versus when we have lots of data.

Below we plot a sequence of updating posterior probabilities as we observe increasing amounts of data (coin flips).

```
In [1]: """
        The book uses a custom matplotlibrc file, which provides the unique styles for
        matplotlib plots. If executing this book, and you wish to use the book's
        styling, provided are two options:
            1. Overwrite your own matplotlibrc file with the rc-file provided in the
               book's styles/ dir. See http://matplotlib.org/users/customizing.html
            2. Also in the styles is  bmh_matplotlibrc.json file. This can be used to
               update the styles in only this notebook. Try running the following code:

                import json
                s = json.load(open("../styles/bmh_matplotlibrc.json"))
                matplotlib.rcParams.update(s)

        """

        # The code below can be passed over, as it is currently not important, plus it
        # uses advanced topics we have not covered yet. LOOK AT PICTURE, MICHAEL!
        %matplotlib inline
        from IPython.core.pylabtools import figsize
        import numpy as np
        from matplotlib import pyplot as plt
        figsize(11, 9)

        import scipy.stats as stats

        dist = stats.beta
```

```
n_trials = [0, 1, 2, 3, 4, 5, 8, 15, 50, 500]
data = stats.bernoulli.rvs(0.5, size=n_trials[-1])
x = np.linspace(0, 1, 100)

# For the already prepared, I'm using Binomial's conj. prior.
for k, N in enumerate(n_trials):
    sx = plt.subplot(len(n_trials)/2, 2, k+1)
    plt.xlabel("$p$, probability of heads") \
        if k in [0, len(n_trials)-1] else None
    plt.setp(sx.get_yticklabels(), visible=False)
    heads = data[:N].sum()
    y = dist.pdf(x, 1 + heads, 1 + N - heads)
    plt.plot(x, y, label="observe %d tosses,\n %d heads" % (N, heads))
    plt.fill_between(x, 0, y, color="#348ABD", alpha=0.4)
    plt.vlines(0.5, 0, 4, color="k", linestyles="--", lw=1)

    leg = plt.legend()
    leg.get_frame().set_alpha(0.4)
    plt.autoscale(tight=True)


plt.suptitle("Bayesian updating of posterior probabilities",
             y=1.02,
             fontsize=14)

plt.tight_layout()
```



The posterior probabilities are represented by the curves, and our uncertainty is proportional to the width of the curve. As the plot above shows, as we start to observe data our posterior probabilities start to shift and move around. Eventually, as we observe more and more data (coin-flips), our probabilities will tighten closer and closer around the true value of $p = 0.5$ (marked by a dashed line).

Notice that the plots are not always *peaked* at 0.5. There is no reason it should be: recall we assumed we did not have a prior opinion of what p is. In fact, if we observe quite extreme data, say 8 flips and only 1 observed heads, our distribution would look very biased *away* from lumping around 0.5 (with no prior opinion, how confident would you feel betting on a fair coin after observing 8 tails and 1 head?). As more data accumulates, we would see more and more probability being assigned at $p = 0.5$, though never all of it.

The next example is a simple demonstration of the mathematics of Bayesian inference.

Example: Bug, or just sweet, unintended feature? 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 of 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 all tests.

$P(X)$ is a little bit trickier: The event X can be divided into two possibilities, event X occurring even though our code *indeed has* bugs (denoted $\sim A$, spoken *not* A), or event X without bugs (A). $P(X)$ can be represented as:

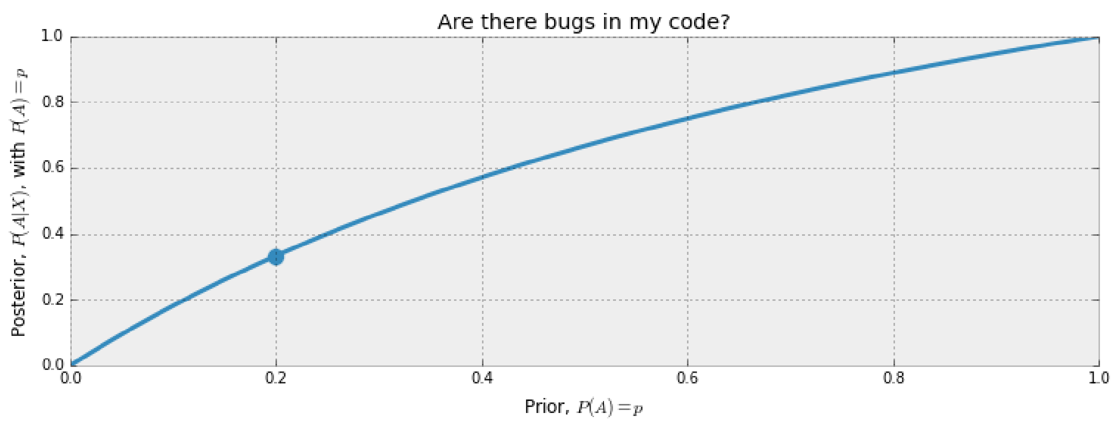
$$
\begin{aligned}
P(X) &= P(X \text{ and } A) + P(X \text{ and } \sim A) && (4)\\
& && (5)\\
&= P(X|A)P(A) + P(X|\sim A)P(\sim A) && (6)\\
& && (7)\\
&= P(X|A)p + P(X|\sim A)(1-p) && (8)
\end{aligned}
$$

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 reduced. 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

$$
\begin{aligned}
P(A|X) &= \frac{1 \cdot p}{1 \cdot p + 0.5(1-p)} && (9)\\
& && (10)\\
&= \frac{2p}{1+p} && (11)
\end{aligned}
$$

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

```
In [2]: figsize(12.5, 4)
        p = np.linspace(0, 1, 50)
        plt.plot(p, 2*p/(1+p), color="#348ABD", lw=3)
        #plt.fill_between(p, 2*p/(1+p), alpha=.5, facecolor=["#A60628"])
        plt.scatter(0.2, 2*(0.2)/1.2, s=140, c="#348ABD")
        plt.xlim(0, 1)
        plt.ylim(0, 1)
        plt.xlabel("Prior, $P(A) = p$")
        plt.ylabel("Posterior, $P(A|X)$, with $P(A) = p$")
        plt.title("Are there bugs in my code?");
```

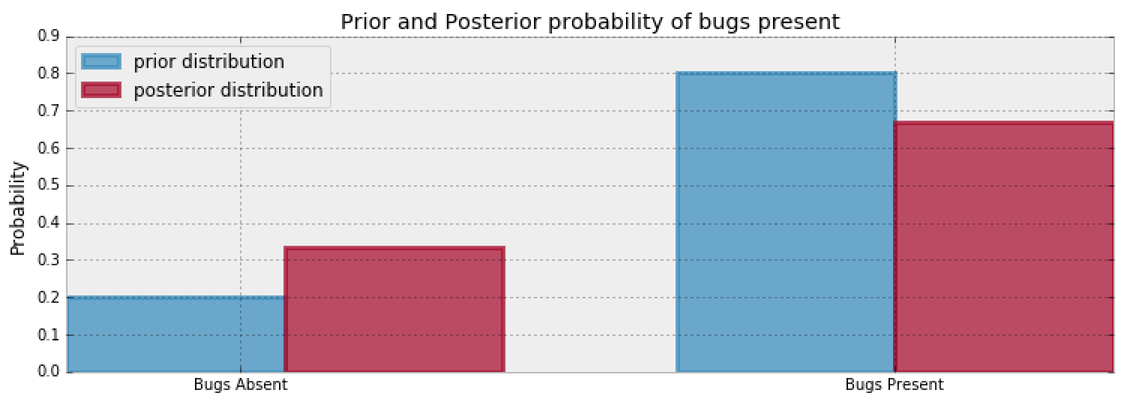


We can see the biggest gains if we observe the X tests passed when the prior probability, p, is low. Let's settle on a specific value for the prior. I'm a strong programmer (I think), so I'm going to give myself a realistic prior of 0.20, that is, there is a 20% chance that I write code bug-free. To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

Recall that the prior is a probability: p is the prior probability that there *are no bugs*, so $1 - p$ is the prior probability that there *are bugs*.

Similarly, our posterior is also a probability, 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 look like? Below is a chart of both the prior and the posterior probabilities.

```
In [3]: figsize(12.5, 4)
        colours = ["#348ABD", "#A60628"]

        prior = [0.20, 0.80]
        posterior = [1./3, 2./3]
        plt.bar([0, .7], prior, alpha=0.70, width=0.25,
                color=colours[0], label="prior distribution",
                lw="3", edgecolor=colours[0])

        plt.bar([0+0.25, .7+0.25], posterior, alpha=0.7,
                width=0.25, color=colours[1],
                label="posterior distribution",
                lw="3", edgecolor=colours[1])

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



Notice that after we observed X occur, the probability of bugs being absent increased. By increasing the number of tests, we can approach confidence (probability 1) that there are no bugs present.

This was a very simple example of Bayesian inference and Bayes rule. Unfortunately, the mathematics necessary to perform more complicated Bayesian inference only becomes more difficult, except for artificially constructed cases. We will later see that this type of mathematical analysis is actually unnecessary. First we must broaden our modeling tools. The next section deals with *probability distributions*. If you are already familiar, feel free to skip (or at least skim), but for the less familiar the next section is essential.

3.2 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 outcomes Z can take. Graphically, a probability distribution is a curve where the probability of an outcome is proportional to the height of the curve. You can see examples in the first figure of this chapter.

We can divide random variables into three classifications:

- 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. Discrete random variables become more clear when we contrast them 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 progressively make the values more and more precise.

- Z **is mixed**: Mixed random variables assign probabilities to both discrete and continuous random variables, i.e. it is a combination of the above two categories.

3.2.1 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:

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

λ is called a parameter of the distribution, and it controls the distribution's shape. For the Poisson distribution, 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. One can describe λ as the *intensity* of the Poisson distribution.

Unlike λ, which can be any positive number, the value k in the above formula 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

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

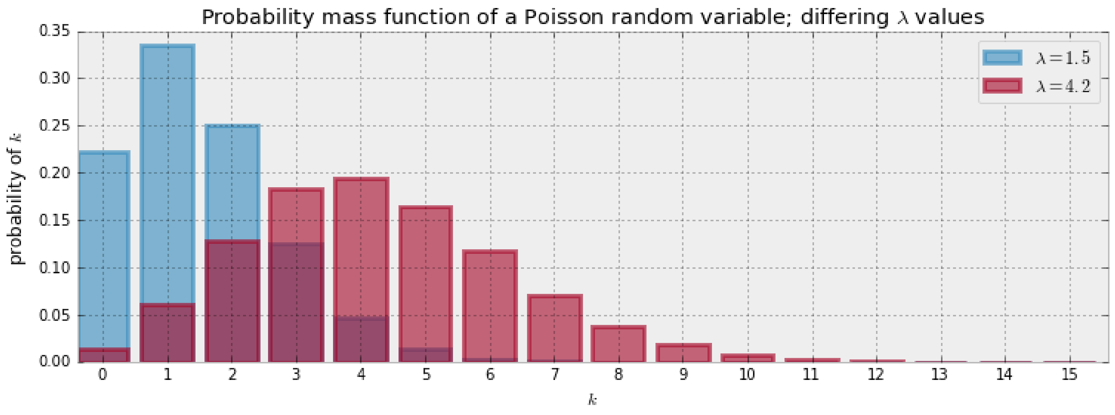
One useful property of the Poisson distribution is that its expected value is equal to its parameter, i.e.:

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

We will use this property often, so it's useful to remember. Below, we plot the probability mass distribution for different λ values. The first thing to notice is that by increasing λ, we add more probability of larger values occurring. Second, notice that although the graph ends at 15, the distributions do not. They assign positive probability to every non-negative integer.

```
In [4]: figsize(12.5, 4)

        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=0.60,
                edgecolor=colours[0], lw="3")

        plt.bar(a, poi.pmf(a, lambda_[1]), color=colours[1],
                label="$\lambda = %.1f$" % lambda_[1], alpha=0.60,
                edgecolor=colours[1], lw="3")
```

```
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");
```



3.2.2 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 *exponential density*. The density function for an exponential random variable looks like this:

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

Like a Poisson random variable, an exponential random variable can take on only non-negative values. But unlike a Poisson variable, the exponential can take on *any* non-negative values, including non-integral values such as 4.25 or 5.612401. This property makes it a poor choice for count data, which must be an integer, but a great choice for time data, temperature data (measured in Kelvins, of course), or any other precise *and positive* variable. The graph below shows two probability density functions with different λ values.

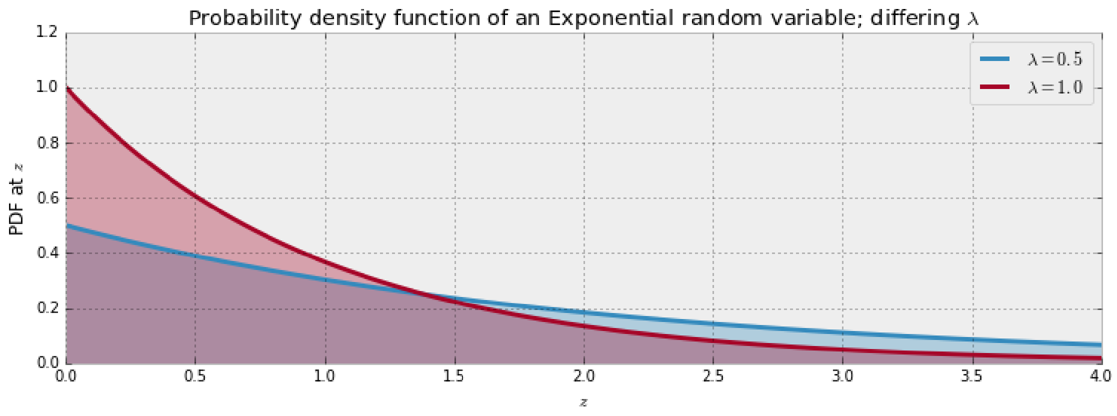
When a random variable Z has an exponential distribution with parameter λ, we say Z *is exponential* and write

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

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

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

```
In [5]: a = np.linspace(0, 4, 100)
        expo = stats.expon
        lambda_ = [0.5, 1]

        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.ylim(0,1.2)
        plt.title("Probability density function of an Exponential random variable;\
         differing $\lambda$");
```



3.2.3 But what is λ ?

This question is what motivates statistics. In the real world, λ is hidden from us. We see only Z, and must go backwards to try and determine λ. The problem is difficult because there is no 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 best!

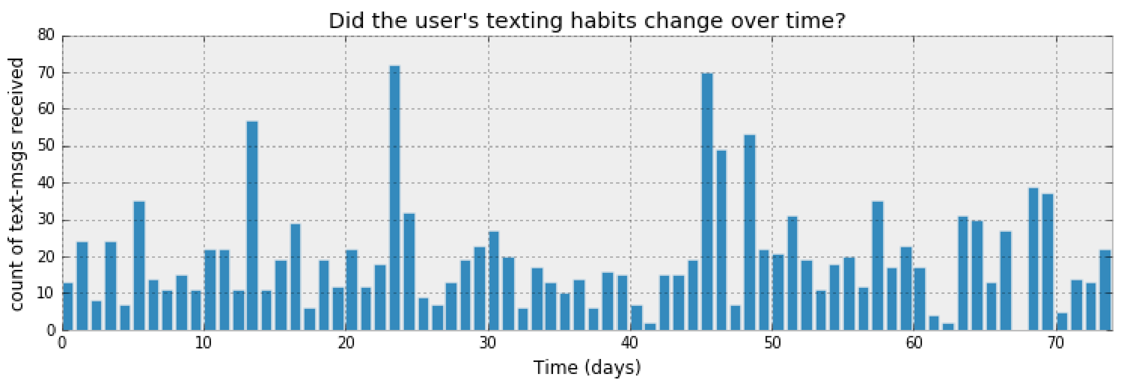
Bayesian inference is concerned with *beliefs* about what λ might be. Rather than try to guess λ exactly, we can only talk about what λ is likely to be by assigning a probability distribution to λ.

This might seem odd at first. After all, λ is fixed; it is not (necessarily) random! How can we assign probabilities to values of a non-random variable? Ah, we have fallen for our old, frequentist way of thinking. Recall that under Bayesian philosophy, we *can* assign probabilities if we interpret them as beliefs. And it is entirely acceptable to have *beliefs* about the parameter λ.

Example: Inferring behaviour from text-message data Let's try to model a more interesting example, one that concerns the rate at which a user sends and receives text messages:

You are given a series of daily text-message counts from a user of your system. The data, plotted over time, appears in the chart below. You are curious to know if the user's text-messaging habits have changed over time, either gradually or suddenly. How can you model this? (This is in fact my own text-message data. Judge my popularity as you wish.)

```
In [6]: figsize(12.5, 3.5)
        count_data = np.loadtxt("data/txtdata.csv")
        n_count_data = len(count_data)
        plt.bar(np.arange(n_count_data), count_data, color="#348ABD")
        plt.xlabel("Time (days)")
        plt.ylabel("count of text-msgs received")
        plt.title("Did the user's texting habits change over time?")
        plt.xlim(0, n_count_data);
```



Before we start modeling, see what you can figure out just by looking at the chart above. Would you say there was a change in behaviour during this time period?

How can we start to model this? Well, as we have conveniently already seen, a Poisson random variable is a very appropriate model for this type of *count* data. Denoting day i's text-message count by C_i,

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

We are not sure what the value of the λ parameter really is, however. Looking at the chart above, it appears that the rate might become higher late in the observation period, which is equivalent to saying that λ increases at some point during the observations. (Recall that a higher value of λ assigns more probability to larger outcomes. That is, there is a higher probability of many text messages having been sent on a given day.)

How can we represent this observation mathematically? Let's assume that on some day during the observation period (call it τ), the parameter λ suddenly jumps to a higher value. So we really have two λ parameters: one for the period before τ, and one for the rest of the observation period. In the 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 and indeed $\lambda_1 = \lambda_2$, then the λs posterior distributions should look about equal.

We are interested in inferring the unknown λs. To use Bayesian inference, we need to assign prior probabilities to the different possible values of λ. What would be good prior probability distributions for λ_1 and λ_2? Recall that λ can be any positive number. As we saw earlier, the *exponential* distribution provides a continuous density function for positive numbers, so it might be a good choice for modeling λ_i. But recall that the exponential distribution takes a parameter of its own, so we'll need to include that parameter in our model. Let's call that parameter α.

$$\lambda_1 \sim \text{Exp}(\alpha) \tag{12}$$

$$\lambda_2 \sim \text{Exp}(\alpha) \tag{13}$$

α is called a *hyper-parameter* or *parent variable*. In literal terms, it is a parameter that influences other parameters. Our initial guess at α does not influence the model too strongly, so we have some flexibility in our choice. A good rule of thumb is to set the exponential parameter equal to the inverse of the average of the count data. Since we're modeling λ using an exponential distribution, we can use the expected value identity shown earlier to get:

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

An alternative, and something I encourage the reader to try, would be to have two priors: one for each λ_i. Creating two exponential distributions with different α values reflects our prior belief that the rate changed at some point during the observations.

What about τ? Because of the noisiness of the data, it's difficult to pick out a priori when τ might have occurred. Instead, we can assign a *uniform prior belief* to every possible day. This is equivalent to saying

$$\tau \sim \text{DiscreteUniform(1,70)} \tag{14}$$

$$\tag{15}$$

$$\Rightarrow P(\tau = k) = \frac{1}{70} \tag{16}$$

So after all this, what does our overall prior distribution for the unknown variables look like? Frankly, *it doesn't matter*. What we should understand is that it's an ugly, complicated mess involving symbols only a mathematician could love. And things will only get uglier the more complicated our models become. Regardless, all we really care about is the posterior distribution.

We next turn to PyMC3, a Python library for performing Bayesian analysis that is undaunted by the mathematical monster we have created.

3.3 Introducing our first hammer: PyMC3

PyMC3 is a Python library for programming Bayesian analysis [3]. It is a fast, well-maintained library. The only unfortunate part is that its documentation is lacking in certain areas, especially those that bridge the gap between beginner and hacker. One of this book's main goals is to solve that problem, and also to demonstrate why PyMC3 is so cool.

We will model the problem above using PyMC3. 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 away from this field. The code is not random; it is probabilistic in the sense that we create probability models using programming variables as the model's components. Model components are first-class primitives within the PyMC3 framework.

B. Cronin [5] has a very motivating description of probabilistic programming:

> Another way of thinking about this: unlike a traditional program, which only runs in the forward directions, a probabilistic program is run in both the forward and backward direction. It runs forward to compute the consequences of the assumptions it contains about the world (i.e., the model space it represents), but it also runs backward from the data to constrain the possible explanations. In practice, many probabilistic programming systems will cleverly interleave these forward and backward operations to efficiently home in on the best explanations.

Because of the confusion engendered by the term *probabilistic programming*, I'll refrain from using it. Instead, I'll simply say *programming*, since that's what it really is.

PyMC3 code is easy to read. The only novel thing should be the syntax. Simply remember that we are representing the model's components ($\tau, \lambda_1, \lambda_2$) as variables.

```
In [7]: import pymc3 as pm
        import theano.tensor as tt

        with pm.Model() as model:
            alpha = 1.0/count_data.mean()  # Recall count_data is the
                                           # variable that holds our txt counts
            lambda_1 = pm.Exponential("lambda_1", alpha)
            lambda_2 = pm.Exponential("lambda_2", alpha)

            tau = pm.DiscreteUniform("tau", lower=0, upper=n_count_data - 1)
```

```
Applied log-transform to lambda_1 and added transformed lambda_1_log_ to model.
Applied log-transform to lambda_2 and added transformed lambda_2_log_ to model.
```

In the code above, we create the PyMC3 variables corresponding to λ_1 and λ_2. We assign them to PyMC3's *stochastic variables*, so-called because they are treated by the back end as random number generators.

```
In [8]: with model:
            idx = np.arange(n_count_data) # Index
            lambda_ = pm.math.switch(tau > idx, lambda_1, lambda_2)
```

This code creates a new function `lambda_`, but really we can think of it as a random variable: the random variable λ from above. The `switch()` function assigns `lambda_1` or `lambda_2` as the value of `lambda_`, depending on what side of `tau` we are on. The values of `lambda_` up until `tau` are `lambda_1` and the values afterwards are `lambda_2`.

Note that because `lambda_1`, `lambda_2` and `tau` are random, `lambda_` will be random. We are **not** fixing any variables yet.

```
In [9]: with model:
            observation = pm.Poisson("obs", lambda_, observed=count_data)
```

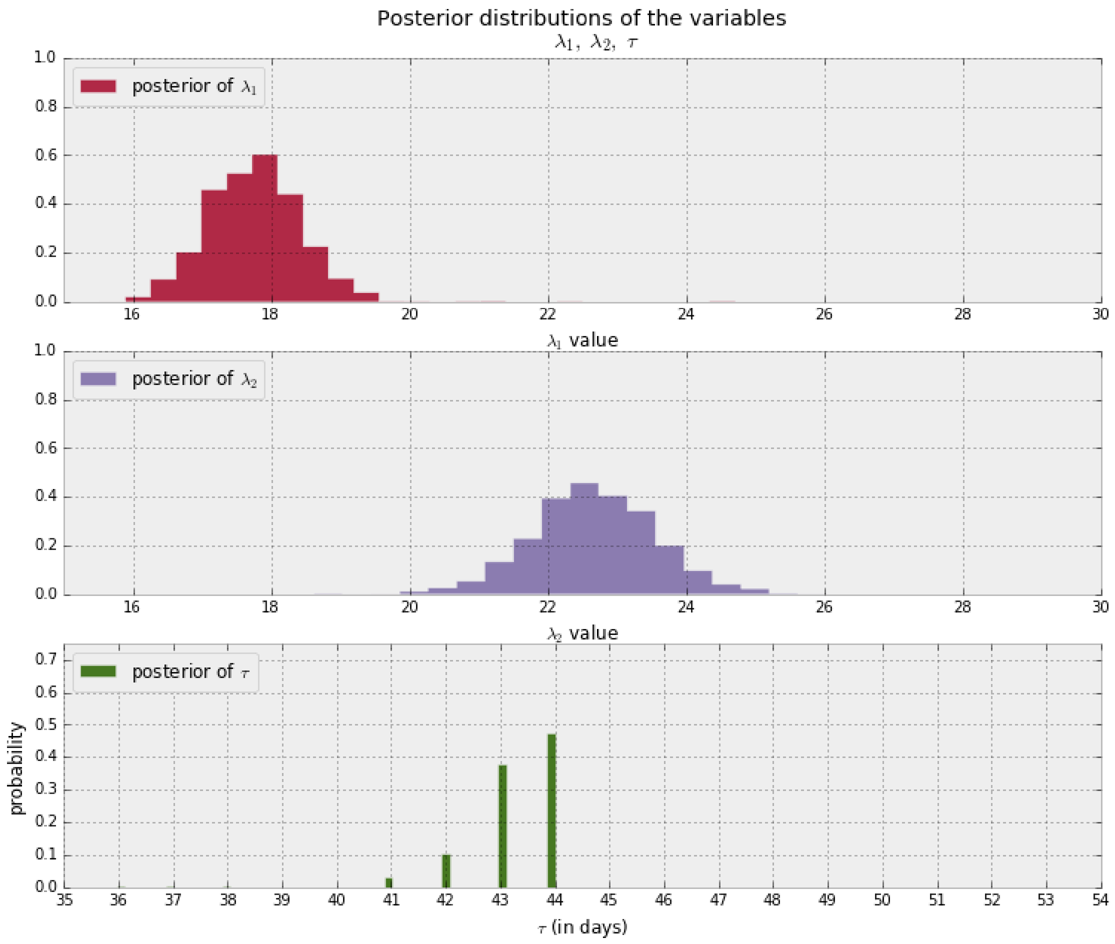
The variable `observation` combines our data, `count_data`, with our proposed data-generation scheme, given by the variable `lambda_`, through the `observed` keyword.

The code below will be explained in Chapter 3, but I show it here so you can see where our results come from. One can think of it as a *learning* step. The machinery being employed is called *Markov Chain Monte Carlo* (MCMC), which I also delay explaining until Chapter 3. This technique returns thousands of random variables from the posterior distributions of λ_1, λ_2 and τ. We can plot a histogram of the random variables to see what the posterior distributions look like. Below, we collect the samples (called *traces* in the MCMC literature) into histograms.

```
In [10]: ### Mysterious code to be explained in Chapter 3.
         with model:
             step = pm.Metropolis()
             trace = pm.sample(10000, tune=5000,step=step)

100%|| 10000/10000 [00:02<00:00, 4511.50it/s]
```

```
In [11]: lambda_1_samples = trace['lambda_1']
         lambda_2_samples = trace['lambda_2']
         tau_samples = trace['tau']
```

```
In [12]: figsize(12.5, 10)
         #histogram of the samples:

         ax = plt.subplot(311)
         ax.set_autoscaley_on(False)

         plt.hist(lambda_1_samples, histtype='stepfilled', bins=30, alpha=0.85,
                  label="posterior of $\lambda_1$", color="#A60628", normed=True)
         plt.legend(loc="upper left")
         plt.title(r"""Posterior distributions of the variables
             $\lambda_1,\;\lambda_2,\;\tau$""")
         plt.xlim([15, 30])
         plt.xlabel("$\lambda_1$ value")

         ax = plt.subplot(312)
         ax.set_autoscaley_on(False)
         plt.hist(lambda_2_samples, histtype='stepfilled', bins=30, alpha=0.85,
                  label="posterior of $\lambda_2$", color="#7A68A6", normed=True)
         plt.legend(loc="upper left")
         plt.xlim([15, 30])
         plt.xlabel("$\lambda_2$ value")

         plt.subplot(313)
         w = 1.0 / tau_samples.shape[0] * np.ones_like(tau_samples)
         plt.hist(tau_samples, bins=n_count_data, alpha=1,
                  label=r"posterior of $\tau$",
                  color="#467821", weights=w, rwidth=2.)
         plt.xticks(np.arange(n_count_data))

         plt.legend(loc="upper left")
         plt.ylim([0, .75])
         plt.xlim([35, len(count_data)-20])
         plt.xlabel(r"$\tau$ (in days)")
         plt.ylabel("probability");
```



3.3.1 Interpretation

Recall that Bayesian methodology returns a *distribution*. Hence we now have distributions to describe the unknown λs and τ. What have we gained? Immediately, we can see the uncertainty in our estimates: the wider the distribution, the less certain our posterior belief should be. We can also see what the plausible values for the parameters are: λ_1 is around 18 and λ_2 is around 23. The posterior distributions of the two λs are clearly distinct, indicating that it is indeed likely that there was a change in the user's text-message behaviour.

What other observations can you make? If you look at the original data again, do these results seem reasonable?

Notice also that the posterior distributions for the λs do not look like exponential distributions, even though our priors for these variables were exponential. In fact, the posterior distributions are not really of any form that we recognize from the original model. But that's OK! This is one of the benefits of taking a computational point of view. If we had instead done this analysis using mathematical approaches, we would have been stuck with an analytically intractable (and messy) distribution. Our use of a computational approach makes us indifferent to mathematical tractability.

Our analysis also returned a distribution for τ. Its posterior distribution looks a little different from the other two because it is a discrete random variable, so it doesn't assign probabilities to intervals. We can see that near day 45, there was a 50% chance that the user's behaviour changed. Had no change occurred, or had the change been gradual over time, the posterior distribution of τ would have been more spread out, reflecting that many days were plausible candidates for τ. By contrast, in the actual results we see that only three or four days make any sense as potential transition points.

3.3.2 Why would I want samples from the posterior, anyways?

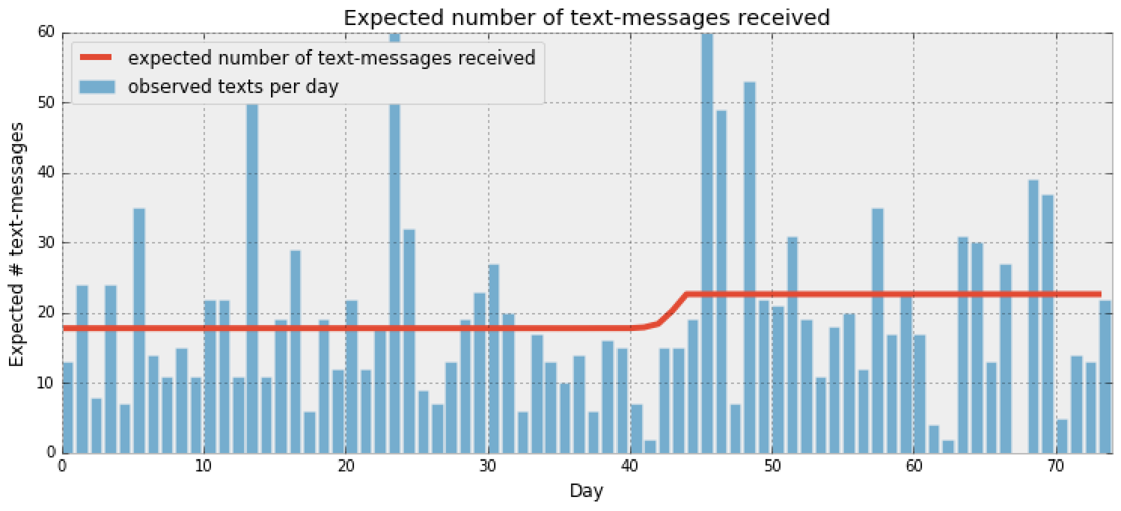
We will deal with this question for the remainder of the book, and it is an understatement to say that it will lead us to some amazing results. For now, let's end this chapter with one more example.

We'll use the posterior samples to answer the following question: what is the expected number of texts at day t, $0 \leq t \leq 70$? Recall that the expected value of a Poisson variable is equal to its parameter λ. Therefore, the question is equivalent to *what is the expected value of λ at time t*?

In the code below, let i index samples from the posterior distributions. Given a day t, we average over all possible λ_i for that day t, using $\lambda_i = \lambda_{1,i}$ if $t < \tau_i$ (that is, if the behaviour change has not yet occurred), else we use $\lambda_i = \lambda_{2,i}$.

```
In [13]: figsize(12.5, 5)
         # tau_samples, lambda_1_samples, lambda_2_samples contain
         # N samples from the corresponding posterior distribution
         N = tau_samples.shape[0]
         expected_texts_per_day = np.zeros(n_count_data)
         for day in range(0, n_count_data):
             # ix is a bool index of all tau samples corresponding to
             # the switchpoint occurring prior to value of 'day'
             ix = day < tau_samples
             # Each posterior sample corresponds to a value for tau.
             # for each day, that value of tau indicates whether we're "before"
             # (in the lambda1 "regime") or
             #  "after" (in the lambda2 "regime") the switchpoint.
             # by taking the posterior sample of lambda1/2 accordingly, we can average
             # over all samples to get an expected value for lambda on that day.
             # As explained, the "message count" random variable is Poisson distributed,
             # and therefore lambda (the poisson parameter) is the expected value of
             # "message count".
             expected_texts_per_day[day] = (lambda_1_samples[ix].sum()
                                               + lambda_2_samples[~ix].sum()) / N
```

```
plt.plot(range(n_count_data), expected_texts_per_day, lw=4, color="#E24A33",
         label="expected number of text-messages received")
plt.xlim(0, n_count_data)
plt.xlabel("Day")
plt.ylabel("Expected # text-messages")
plt.title("Expected number of text-messages received")
plt.ylim(0, 60)
plt.bar(np.arange(len(count_data)), count_data, color="#348ABD", alpha=0.65,
        label="observed texts per day")

plt.legend(loc="upper left");
```



Our analysis shows strong support for believing the user's behavior did change (λ_1 would have been close in value to λ_2 had this not been true), and that the change was sudden rather than gradual (as demonstrated by τ's strongly peaked posterior distribution). We can speculate what might have caused this: a cheaper text-message rate, a recent weather-to-text subscription, or perhaps a new relationship. (In fact, the 45th day corresponds to Christmas, and I moved away to Toronto the next month, leaving a girlfriend behind.)

Exercises 1. Using `lambda_1_samples` and `lambda_2_samples`, what is the mean of the posterior distributions of λ_1 and λ_2?

```
In [14]: #type your code here.
```

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

```
In [15]: #type your code here.
```

3. What is the mean of λ_1 **given** that we know τ is less than 45. That is, suppose we have been given new information that the change in behaviour occurred prior to day 45. What is the expected value of λ_1 now? (You do not need to redo the PyMC3 part. Just consider all instances where `tau_samples < 45`.)

```
In [16]: #type your code here.
```

3.3.3 References

- [1] Gelman, Andrew. N.p.. Web. 22 Jan 2013. N is never large enough.
- [2] Norvig, Peter. 2009. The Unreasonable Effectiveness of Data.
- [3] 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`
- [4] Jimmy Lin and Alek Kolcz. Large-Scale Machine Learning at Twitter. Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD 2012), pages 793-804, May 2012, Scottsdale, Arizona.
- [5] 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`.

```
In [1]: from IPython.core.display import HTML
        def css_styling():
            styles = open("../styles/custom.css", "r").read()
            return HTML(styles)
        css_styling()

Out[1]: <IPython.core.display.HTML at 0x10f034850>

In [ ]:
```