Prologue

January 4, 2018

1 Probabilistic Programming and

2 Bayesian Methods for Hackers

2.0.1 Probabilistic Programming & Bayesian Methods for Hackers

Using Python and PyMC The Bayesian method is the natural approach to inference, yet it is hidden from readers behind chapters of slow, mathematical analysis. The typical text on Bayesian inference involves two to three chapters on probability theory, then enters what Bayesian inference is. Unfortunately, due to mathematical intractability of most Bayesian models, the reader is only shown simple, artificial examples. This can leave the user with a *so-what* feeling about Bayesian inference. In fact, this was the author's own prior opinion.

After some recent success of Bayesian methods in machine-learning competitions, I decided to investigate the subject again. Even with my mathematical background, it took me three straight-days of reading examples and trying to put the pieces together to understand the methods. There was simply not enough literature bridging theory to practice. The problem with my misunderstanding was the disconnect between Bayesian mathematics and probabilistic programming. That being said, I suffered then so the reader would not have to now. This book attempts to bridge the gap.

If Bayesian inference is the destination, then mathematical analysis is a particular path towards it. On the other hand, computing power is cheap enough that we can afford to take an alternate route via probabilistic programming. The latter path is much more useful, as it denies the necessity of mathematical intervention at each step, that is, we remove often-intractable mathematical analysis as a prerequisite to Bayesian inference. Simply put, this latter computational path proceeds via small intermediate jumps from beginning to end, where as the first path proceeds by enormous leaps, often landing far away from our target. Furthermore, without a strong mathematical background, the analysis required by the first path cannot even take place.

Bayesian Methods for Hackers is designed as a introduction to Bayesian inference from a computational/understanding-first, and mathematics-second, point of view. Of course as an introductory book, we can only leave it at that: an introductory book. For the mathematically trained, they may cure the curiosity this text generates with other texts designed with mathematical analysis in mind. For the enthusiast with less mathematical-background, or one who is not interested in the mathematics but simply the practice of Bayesian methods, this text should be sufficient and entertaining.

The choice of PyMC as the probabilistic programming language is two-fold. As of this writing, there is currently no central resource for examples and explanations in the PyMC universe. The

official documentation assumes prior knowledge of Bayesian inference and probabilistic programming. We hope this book encourages users at every level to look at PyMC. Secondly, with recent core developments and popularity of the scientific stack in Python, PyMC is likely to become a core component soon enough.

PyMC does have dependencies to run, namely NumPy and (optionally) SciPy. To not limit the user, the examples in this book will rely only on PyMC, NumPy, SciPy and Matplotlib.

2.1 Contents

(The below chapters are rendered via the *nbviewer* at nbviewer.ipython.org/, and are read-only and rendered in real-time. Interactive notebooks + examples can be downloaded by cloning!)

- **Prologue:** Why we do it.
- **Chapter 1: Introduction to Bayesian Methods** Introduction to the philosophy and practice of Bayesian methods and answering the question "What is probabilistic programming?" Examples include:
  - Inferring human behaviour changes from text message rates.
- **Chapter 2: A little more on PyMC** We explore modeling Bayesian problems using Python's PyMC library through examples. How do we create Bayesian models? Examples include:
  - Detecting the frequency of cheating students, while avoiding liars.
  - Calculating probabilities of the Challenger space-shuttle disaster.
- **Chapter 3: Opening the Black Box of MCMC** We discuss how MCMC operates and diagnostic tools. Examples include:
  - Bayesian clustering with mixture models
- **Chapter 4: The Greatest Theorem Never Told** We explore an incredibly useful, and dangerous, theorem: The Law of Large Numbers. Examples include:
  - Exploring a Kaggle dataset and the pitfalls of naive analysis
  - How to sort Reddit comments from best to worst (not as easy as you think)
- **Chapter 5: Would you rather loss an arm or a leg?** The introduction of Loss functions and their (awesome) use in Bayesian methods. Examples include:
  - Solving the Price is Right's Showdown
  - Optimizing financial predictions
  - Winning solution to the Kaggle Dark World's competition.
- **Chapter 6: Getting our *prior*-ities straight** Probably the most important chapter. We draw on expert opinions to answer questions. Examples include:
  - Multi-Armed Bandits and the Bayesian Bandit solution.
  - what is the relationship between data sample size and prior?
  - estimating financial unknowns using expert priors.

We explore useful tips to be objective in analysis, and common pitfalls of priors.

More questions about PyMC? Please post your modeling, convergence, or any other PyMC question on cross-validated, the statistics stack-exchange.

2.2 Using the book

The book can be read in three different ways, starting from most recommended to least recommended:

1. The most recommended option is to clone the repository to download the .ipynb files to your local machine. If you have IPython installed, you can view the chapters in your browser *plus* edit and run the code provided (and try some practice questions). This is the preferred option to read this book, though it comes with some dependencies.

    - IPython 0.13 is a requirement to view the ipynb files. It can be downloaded here
    - For Linux users, you should not have a problem installing Numpy, Scipy, Matplotlib and PyMC. For Windows users, check out pre-compiled versions if you have difficulty.
    - In the styles/ directory are a number of files (.matplotlirc) that used to make things pretty. These are not only designed for the book, but they offer many improvements over the default settings of matplotlib and the IPython notebook.

2. The second, preferred, option is to use the nbviewer.ipython.org site, which display IPython notebooks in the browser (example). The contents are updated synchronously as commits are made to the book. You can use the Contents section above to link to the chapters.

3. **PDF versions are coming.** PDFs are the least-preferred method to read the book, as pdf's are static and non-interactive. If PDFs are desired, they can be created dynamically using Chrome's builtin print-to-pdf feature.

2.3 Installation and configuration

If you would like to run the IPython notebooks locally, (option 1. above), you'll need to install the following:

- IPython 0.13 is a requirement to view the ipynb files. It can be downloaded here
- For Linux users, you should not have a problem installing Numpy, Scipy and PyMC. For Windows users, check out pre-compiled versions if you have difficulty.
- also recommended, for data-mining exercises, are PRAW and requests.
- In the styles/ directory are a number of files that are customized for the notebook. These are not only designed for the book, but they offer many improvements over the default settings of matplotlib and the IPython notebook. The in notebook style has not been finalized yet.

2.4 Contributions and Thanks

Thanks to all our contributing authors, including (in chronological order):

- Cameron Davidson-Pilon
- Stef Gibson
- Vincent Ohprecio

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:

$$P(X) = P(X \text{ and } A) + P(X \text{ and } \sim A) \tag{4}$$

$$\tag{5}$$

$$= P(X|A)P(A) + P(X| \sim A)P(\sim A) \tag{6}$$

$$\tag{7}$$

$$= P(X|A)p + P(X| \sim A)(1-p) \tag{8}$$

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

$$P(A|X) = \frac{1 \cdot p}{1 \cdot p + 0.5(1-p)} \tag{9}$$

$$\tag{10}$$

$$= \frac{2p}{1+p} \tag{11}$$

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 [ ]:
```

Ch2_MorePyMC_PyMC2

January 4, 2018

1 Chapter 2

This chapter introduces more PyMC syntax and design patterns, and ways to think about how to model a system from a Bayesian perspective. It also contains tips and data visualization techniques for assessing goodness-of-fit for your Bayesian model.

1.1 A little more on PyMC

1.1.1 Parent and Child relationships

To assist with describing Bayesian relationships, and to be consistent with PyMC's documentation, we introduce *parent and child* variables.

- *parent variables* are variables that influence another variable.
- *child variable* are variables that are affected by other variables, i.e. are the subject of parent variables.

A variable can be both a parent and child. For example, consider the PyMC code below.

```
In [1]: import pymc as pm


        parameter = pm.Exponential("poisson_param", 1)
        data_generator = pm.Poisson("data_generator", parameter)
        data_plus_one = data_generator + 1
```

`parameter` controls the parameter of `data_generator`, hence influences its values. The former is a parent of the latter. By symmetry, `data_generator` is a child of `parameter`.

Likewise, `data_generator` is a parent to the variable `data_plus_one` (hence making `data_generator` both a parent and child variable). Although it does not look like one, `data_plus_one` should be treated as a PyMC variable as it is a *function* of another PyMC variable, hence is a child variable to `data_generator`.

This nomenclature is introduced to help us describe relationships in PyMC modeling. You can access a variable's children and parent variables using the `children` and `parents` attributes attached to variables.

```
In [2]: print("Children of `parameter`: ")
        print(parameter.children)
        print("\nParents of `data_generator`: ")
        print(data_generator.parents)
        print("\nChildren of `data_generator`: ")
        print(data_generator.children)
```

```
Children of `parameter`:
{<pymc.distributions.new_dist_class.<locals>.new_class 'data_generator' at 0x7f812404a8d0>}

Parents of `data_generator`:
{'mu': <pymc.distributions.new_dist_class.<locals>.new_class 'poisson_param' at 0x7f812404a898

Children of `data_generator`:
{<pymc.PyMCObjects.Deterministic '(data_generator_add_1)' at 0x7f812404a908>}
```

Of course a child can have more than one parent, and a parent can have many children.

1.1.2 PyMC Variables

All PyMC variables also expose a `value` attribute. This method produces the *current* (possibly random) internal value of the variable. If the variable is a child variable, its value changes given the variable's parents' values. Using the same variables from before:

```
In [3]: print("parameter.value =", parameter.value)
        print("data_generator.value =", data_generator.value)
        print("data_plus_one.value =", data_plus_one.value)
```

```
parameter.value = 0.032177775515776275
data_generator.value = 0
data_plus_one.value = 1
```

PyMC is concerned with two types of programming variables: `stochastic` and `deterministic`.

- *stochastic variables* are variables that are not deterministic, i.e., even if you knew all the values of the variables' parents (if it even has any parents), it would still be random. Included in this category are instances of classes `Poisson`, `DiscreteUniform`, and `Exponential`.
- *deterministic variables* are variables that are not random if the variables' parents were known. This might be confusing at first: a quick mental check is *if I knew all of variable `foo`'s parent variables, I could determine what `foo`'s value is.*

We will detail each below.

Initializing Stochastic variables Initializing a stochastic variable requires a `name` argument, plus additional parameters that are class specific. For example:

`some_variable = pm.DiscreteUniform("discrete_uni_var", 0, 4)`

where 0, 4 are the `DiscreteUniform`-specific lower and upper bound on the random variable. The PyMC docs contain the specific parameters for stochastic variables. (Or use `object??`, for example `pm.DiscreteUniform??` if you are using IPython!)

The `name` attribute is used to retrieve the posterior distribution later in the analysis, so it is best to use a descriptive name. Typically, I use the Python variable's name as the `name`.

For multivariable problems, rather than creating a Python array of stochastic variables, addressing the `size` keyword in the call to a `Stochastic` variable creates multivariate array of (independent) stochastic variables. The array behaves like a Numpy array when used like one, and references to its `value` attribute return Numpy arrays.

The `size` argument also solves the annoying case where you may have many variables $\beta_i,\ i = 1, ..., N$ you wish to model. Instead of creating arbitrary names and variables for each one, like:

```
beta_1 = pm.Uniform("beta_1", 0, 1)
beta_2 = pm.Uniform("beta_2", 0, 1)
...
```

we can instead wrap them into a single variable:

```
betas = pm.Uniform("betas", 0, 1, size=N)
```

Calling `random()` We can also call on a stochastic variable's `random()` method, which (given the parent values) will generate a new, random value. Below we demonstrate this using the texting example from the previous chapter.

```
In [4]: lambda_1 = pm.Exponential("lambda_1", 1)  # prior on first behaviour
        lambda_2 = pm.Exponential("lambda_2", 1)  # prior on second behaviour
        tau = pm.DiscreteUniform("tau", lower=0, upper=10)  # prior on behaviour change

        print("lambda_1.value = %.3f" % lambda_1.value)
        print("lambda_2.value = %.3f" % lambda_2.value)
        print("tau.value = %.3f" % tau.value, "\n")

        lambda_1.random(), lambda_2.random(), tau.random()

        print("After calling random() on the variables...")
        print("lambda_1.value = %.3f" % lambda_1.value)
        print("lambda_2.value = %.3f" % lambda_2.value)
        print("tau.value = %.3f" % tau.value)
```

```
lambda_1.value = 4.970
lambda_2.value = 1.039
tau.value = 5.000

After calling random() on the variables...
lambda_1.value = 1.411
```

```
lambda_2.value = 1.806
tau.value = 3.000
```

The call to `random` stores a new value into the variable's `value` attribute. In fact, this new value is stored in the computer's cache for faster recall and efficiency.

1.1.3 Warning: *Don't update stochastic variables' values in-place.*

Straight from the PyMC docs, we quote [4]:

> `Stochastic` objects' values should not be updated in-place. This confuses PyMC's caching scheme... The only way a stochastic variable's value should be updated is using statements of the following form:
>
> ```
> A.value = new_value
> ```
>
> The following are in-place updates and should **never** be used:
>
> ```
> A.value += 3
> A.value[2,1] = 5
> A.value.attribute = new_attribute_value
> ```

Deterministic variables Since most variables you will be modeling are stochastic, we distinguish deterministic variables with a `pymc.deterministic` wrapper. (If you are unfamiliar with Python wrappers (also called decorators), that's no problem. Just prepend the `pymc.deterministic` decorator before the variable declaration and you're good to go. No need to know more.) The declaration of a deterministic variable uses a Python function:

```
@pm.deterministic
def some_deterministic_var(v1=v1,):
     #jelly goes here.
```

For all purposes, we can treat the object `some_deterministic_var` as a variable and not a Python function.

Prepending with the wrapper is the easiest way, but not the only way, to create deterministic variables: elementary operations, like addition, exponentials etc. implicitly create deterministic variables. For example, the following returns a deterministic variable:

```
In [5]: type(lambda_1 + lambda_2)

Out[5]: pymc.PyMCObjects.Deterministic
```

The use of the `deterministic` wrapper was seen in the previous chapter's text-message example. Recall the model for λ looked like:

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

And in PyMC code:

```
In [6]: import numpy as np
        n_data_points = 5  # in CH1 we had ~70 data points


        @pm.deterministic
        def lambda_(tau=tau, lambda_1=lambda_1, lambda_2=lambda_2):
            out = np.zeros(n_data_points)
            out[:tau] = lambda_1  # lambda before tau is lambda1
            out[tau:] = lambda_2  # lambda after tau is lambda2
            return out
```

Clearly, if τ, λ_1 and λ_2 are known, then λ is known completely, hence it is a deterministic variable.

Inside the deterministic decorator, the `Stochastic` variables passed in behave like scalars or Numpy arrays (if multivariable), and *not* like `Stochastic` variables. For example, running the following:

```
@pm.deterministic
def some_deterministic(stoch=some_stochastic_var):
    return stoch.value**2
```

will return an `AttributeError` detailing that `stoch` does not have a `value` attribute. It simply needs to be `stoch**2`. During the learning phase, it's the variable's `value` that is repeatedly passed in, not the actual variable.

Notice in the creation of the deterministic function we added defaults to each variable used in the function. This is a necessary step, and all variables *must* have default values.

1.1.4 Including observations in the Model

At this point, it may not look like it, but we have fully specified our priors. For example, we can ask and answer questions like "What does my prior distribution of λ_1 look like?"

```
In [7]: %matplotlib inline
        from IPython.core.pylabtools import figsize
        from matplotlib import pyplot as plt
        figsize(12.5, 4)


        samples = [lambda_1.random() for i in range(20000)]
        plt.hist(samples, bins=70, normed=True, histtype="stepfilled")
        plt.title("Prior distribution for $\lambda_1$")
        plt.xlim(0, 8);
```

To frame this in the notation of the first chapter, though this is a slight abuse of notation, we have specified $P(A)$. Our next goal is to include data/evidence/observations X into our model.

PyMC stochastic variables have a keyword argument `observed` which accepts a boolean (`False` by default). The keyword `observed` has a very simple role: fix the variable's current value, i.e. make `value` immutable. We have to specify an initial `value` in the variable's creation, equal to the observations we wish to include, typically an array (and it should be an Numpy array for speed). For example:

```
In [8]: data = np.array([10, 5])
        fixed_variable = pm.Poisson("fxd", 1, value=data, observed=True)
        print("value: ", fixed_variable.value)
        print("calling .random()")
        fixed_variable.random()
        print("value: ", fixed_variable.value)
```

```
value:  [10  5]
calling .random()
value:  [10  5]
```

This is how we include data into our models: initializing a stochastic variable to have a *fixed value*.

To complete our text message example, we fix the PyMC variable `observations` to the observed dataset.

```
In [9]: # We're using some fake data here
        data = np.array([10, 25, 15, 20, 35])
        obs = pm.Poisson("obs", lambda_, value=data, observed=True)
        print(obs.value)
```

```
[10 25 15 20 35]
```

1.1.5 Finally...

We wrap all the created variables into a `pm.Model` class. With this `Model` class, we can analyze the variables as a single unit. This is an optional step, as the fitting algorithms can be sent an array of the variables rather than a `Model` class. I may or may not use this class in future examples ;)

```
In [10]: model = pm.Model([obs, lambda_, lambda_1, lambda_2, tau])
```

1.2 Modeling approaches

A good starting point in Bayesian modeling is to think about *how your data might have been generated*. Put yourself in an omniscient position, and try to imagine how *you* would recreate the dataset.

In the last chapter we investigated text message data. We begin by asking how our observations may have been generated:

1. We started by thinking "what is the best random variable to describe this count data?" A Poisson random variable is a good candidate because it can represent count data. So we model the number of sms's received as sampled from a Poisson distribution.

2. Next, we think, "Ok, assuming sms's are Poisson-distributed, what do I need for the Poisson distribution?" Well, the Poisson distribution has a parameter λ.

3. Do we know λ? No. In fact, we have a suspicion that there are *two* λ values, one for the earlier behaviour and one for the later behaviour. We don't know when the behaviour switches though, but call the switchpoint τ.

4. What is a good distribution for the two λs? The exponential is good, as it assigns probabilities to positive real numbers. Well the exponential distribution has a parameter too, call it α.

5. Do we know what the parameter α might be? No. At this point, we could continue and assign a distribution to α, but it's better to stop once we reach a set level of ignorance: whereas we have a prior belief about λ, ("it probably changes over time", "it's likely between 10 and 30", etc.), we don't really have any strong beliefs about α. So it's best to stop here.

   What is a good value for α then? We think that the λs are between 10-30, so if we set α really low (which corresponds to larger probability on high values) we are not reflecting our prior well. Similar, a too-high alpha misses our prior belief as well. A good idea for α as to reflect our belief is to set the value so that the mean of λ, given α, is equal to our observed mean. This was shown in the last chapter.

6. We have no expert opinion of when τ might have occurred. So we will suppose τ is from a discrete uniform distribution over the entire timespan.

Below we give a graphical visualization of this, where arrows denote `parent-child` relationships. (provided by the Daft Python library)

PyMC, and other probabilistic programming languages, have been designed to tell these data-generation *stories*. More generally, B. Cronin writes [5]:

Probabilistic programming will unlock narrative explanations of data, one of the holy grails of business analytics and the unsung hero of scientific persuasion. People think in terms of stories - thus the unreasonable power of the anecdote to drive decision-making, well-founded or not. But existing analytics largely fails to provide this kind of story; instead, numbers seemingly appear out of thin air, with little of the causal context that humans prefer when weighing their options.

1.2.1 Same story; different ending.

Interestingly, we can create *new datasets* by retelling the story. For example, if we reverse the above steps, we can simulate a possible realization of the dataset.

1. Specify when the user's behaviour switches by sampling from DiscreteUniform$(0, 80)$:

```
In [11]: tau = pm.rdiscrete_uniform(0, 80)
         print(tau)
```

```
21
```

2. Draw λ_1 and λ_2 from an Exp(α) distribution:

```
In [12]: alpha = 1. / 20.
         lambda_1, lambda_2 = pm.rexponential(alpha, 2)
         print(lambda_1, lambda_2)
```

```
20.7789591495 62.1938883352
```

3. For days before τ, represent the user's received SMS count by sampling from Poi(λ_1), and sample from Poi(λ_2) for days after τ. For example:

```
In [13]: data = np.r_[pm.rpoisson(lambda_1, tau), pm.rpoisson(lambda_2, 80 - tau)]
```

4. Plot the artificial dataset:

```
In [14]: plt.bar(np.arange(80), data, color="#348ABD")
         plt.bar(tau - 1, data[tau - 1], color="r", label="user behaviour changed")
         plt.xlabel("Time (days)")
         plt.ylabel("count of text-msgs received")
         plt.title("Artificial dataset")
         plt.xlim(0, 80)
         plt.legend();
```

It is okay that our fictional dataset does not look like our observed dataset: the probability is incredibly small it indeed would. PyMC's engine is designed to find good parameters, λ_i, τ, that maximize this probability.

The ability to generate artificial datasets is an interesting side effect of our modeling, and we will see that this ability is a very important method of Bayesian inference. We produce a few more datasets below:

```
In [15]: def plot_artificial_sms_dataset():
             tau = pm.rdiscrete_uniform(0, 80)
             alpha = 1. / 20.
             lambda_1, lambda_2 = pm.rexponential(alpha, 2)
             data = np.r_[pm.rpoisson(lambda_1, tau), pm.rpoisson(lambda_2, 80 - tau)]
             plt.bar(np.arange(80), data, color="#348ABD")
             plt.bar(tau - 1, data[tau - 1], color="r", label="user behaviour changed")
             plt.xlim(0, 80)

         figsize(12.5, 5)
         plt.suptitle("More examples of artificial datasets", fontsize=14)
         for i in range(1, 5):
             plt.subplot(4, 1, i)
             plot_artificial_sms_dataset()
```

Later we will see how we use this to make predictions and test the appropriateness of our models.

Example: Bayesian A/B testing A/B testing is a statistical design pattern for determining the difference of effectiveness between two different treatments. For example, a pharmaceutical company is interested in the effectiveness of drug A vs drug B. The company will test drug A on some fraction of their trials, and drug B on the other fraction (this fraction is often 1/2, but we will relax this assumption). After performing enough trials, the in-house statisticians sift through the data to determine which drug yielded better results.

Similarly, front-end web developers are interested in which design of their website yields more sales or some other metric of interest. They will route some fraction of visitors to site A, and the other fraction to site B, and record if the visit yielded a sale or not. The data is recorded (in real-time), and analyzed afterwards.

Often, the post-experiment analysis is done using something called a hypothesis test like *difference of means test* or *difference of proportions test*. This involves often misunderstood quantities like a "Z-score" and even more confusing "p-values" (please don't ask). If you have taken a statistics course, you have probably been taught this technique (though not necessarily *learned* this technique). And if you were like me, you may have felt uncomfortable with their derivation -- good: the Bayesian approach to this problem is much more natural.

1.2.2 A Simple Case

As this is a hacker book, we'll continue with the web-dev example. For the moment, we will focus on the analysis of site A only. Assume that there is some true $0 < p_A < 1$ probability that users who, upon shown site A, eventually purchase from the site. This is the true effectiveness of site A. Currently, this quantity is unknown to us.

Suppose site A was shown to N people, and n people purchased from the site. One might conclude hastily that $p_A = \frac{n}{N}$. Unfortunately, the *observed frequency* $\frac{n}{N}$ does not necessarily equal p_A -- there is a difference between the *observed frequency* and the *true frequency* of an event. The true frequency can be interpreted as the probability of an event occurring. For example, the true frequency of rolling a 1 on a 6-sided die is $\frac{1}{6}$. Knowing the true frequency of events like:

- fraction of users who make purchases,
- frequency of social attributes,
- percent of internet users with cats etc.

are common requests we ask of Nature. Unfortunately, often Nature hides the true frequency from us and we must *infer* it from observed data.

The *observed frequency* is then the frequency we observe: say rolling the die 100 times you may observe 20 rolls of 1. The observed frequency, 0.2, differs from the true frequency, $\frac{1}{6}$. We can use Bayesian statistics to infer probable values of the true frequency using an appropriate prior and observed data.

With respect to our A/B example, we are interested in using what we know, N (the total trials administered) and n (the number of conversions), to estimate what p_A, the true frequency of buyers, might be.

To set up a Bayesian model, we need to assign prior distributions to our unknown quantities. *A priori*, what do we think p_A might be? For this example, we have no strong conviction about p_A, so for now, let's assume p_A is uniform over [0,1]:

```
In [16]: import pymc as pm

         # The parameters are the bounds of the Uniform.
         p = pm.Uniform('p', lower=0, upper=1)
```

Had we had stronger beliefs, we could have expressed them in the prior above.

For this example, consider $p_A = 0.05$, and $N = 1500$ users shown site A, and we will simulate whether the user made a purchase or not. To simulate this from N trials, we will use a *Bernoulli* distribution: if $X \sim \text{Ber}(p)$, then X is 1 with probability p and 0 with probability $1 - p$. Of course, in practice we do not know p_A, but we will use it here to simulate the data.

```
In [17]: # set constants
         p_true = 0.05  # remember, this is unknown.
         N = 1500

         # sample N Bernoulli random variables from Ber(0.05).
         # each random variable has a 0.05 chance of being a 1.
         # this is the data-generation step
         occurrences = pm.rbernoulli(p_true, N)

         print(occurrences)  # Remember: Python treats True == 1, and False == 0
         print(occurrences.sum())
```

```
[False False False ..., False False False]
77
```

The observed frequency is:

```
In [18]: # Occurrences.mean is equal to n/N.
         print("What is the observed frequency in Group A? %.4f" % occurrences.mean())
         print("Does this equal the true frequency? %s" % (occurrences.mean() == p_true))
```

```
What is the observed frequency in Group A? 0.0513
Does this equal the true frequency? False
```

We combine the observations into the PyMC `observed` variable, and run our inference algorithm:

```
In [19]: # include the observations, which are Bernoulli
         obs = pm.Bernoulli("obs", p, value=occurrences, observed=True)

         # To be explained in chapter 3
         mcmc = pm.MCMC([p, obs])
         mcmc.sample(18000, 1000)
```

```
 [-----------------100%-----------------] 18000 of 18000 complete in 1.5 sec
```

We plot the posterior distribution of the unknown p_A below:

```
In [20]: figsize(12.5, 4)
         plt.title("Posterior distribution of $p_A$, the true effectiveness of site A")
         plt.vlines(p_true, 0, 90, linestyle="--", label="true $p_A$ (unknown)")
         plt.hist(mcmc.trace("p")[:], bins=25, histtype="stepfilled", normed=True)
         plt.legend();
```

Our posterior distribution puts most weight near the true value of p_A, but also some weights in the tails. This is a measure of how uncertain we should be, given our observations. Try changing the number of observations, `N`, and observe how the posterior distribution changes.

1.2.3 *A* and *B* Together

A similar analysis can be done for site B's response data to determine the analogous p_B. But what we are really interested in is the *difference* between p_A and p_B. Let's infer p_A, p_B, *and* delta $= p_A - p_B$, all at once. We can do this using PyMC's deterministic variables. (We'll assume for this exercise that $p_B = 0.04$, so delta $= 0.01$, $N_B = 750$ (significantly less than N_A) and we will simulate site B's data like we did for site A's data)

```
In [21]: import pymc as pm
         figsize(12, 4)

         # these two quantities are unknown to us.
         true_p_A = 0.05
         true_p_B = 0.04

         # notice the unequal sample sizes -- no problem in Bayesian analysis.
         N_A = 1500
         N_B = 750

         # generate some observations
         observations_A = pm.rbernoulli(true_p_A, N_A)
         observations_B = pm.rbernoulli(true_p_B, N_B)
         print("Obs from Site A: ", observations_A[:30].astype(int), "...")
         print("Obs from Site B: ", observations_B[:30].astype(int), "...")
```

```
Obs from Site A:  [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0] ...
Obs from Site B:  [0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0] ...
```

```
In [22]: print(observations_A.mean())
         print(observations_B.mean())
```

```
0.054
0.0333333333333
```

```
In [23]: # Set up the pymc model. Again assume Uniform priors for p_A and p_B.
         p_A = pm.Uniform("p_A", 0, 1)
         p_B = pm.Uniform("p_B", 0, 1)


         # Define the deterministic delta function. This is our unknown of interest.
         @pm.deterministic
         def delta(p_A=p_A, p_B=p_B):
             return p_A - p_B

         # Set of observations, in this case we have two observation datasets.
         obs_A = pm.Bernoulli("obs_A", p_A, value=observations_A, observed=True)
         obs_B = pm.Bernoulli("obs_B", p_B, value=observations_B, observed=True)

         # To be explained in chapter 3.
         mcmc = pm.MCMC([p_A, p_B, delta, obs_A, obs_B])
         mcmc.sample(20000, 1000)
```

```
 [-----------------100%-----------------] 20000 of 20000 complete in 2.6 sec
```

Below we plot the posterior distributions for the three unknowns:

In [24]:
```
p_A_samples = mcmc.trace("p_A")[:]
p_B_samples = mcmc.trace("p_B")[:]
delta_samples = mcmc.trace("delta")[:]
```

In [25]:
```
figsize(12.5, 10)

# histogram of posteriors

ax = plt.subplot(311)

plt.xlim(0, .1)
plt.hist(p_A_samples, histtype='stepfilled', bins=25, alpha=0.85,
         label="posterior of $p_A$", color="#A60628", normed=True)
plt.vlines(true_p_A, 0, 80, linestyle="--", label="true $p_A$ (unknown)")
plt.legend(loc="upper right")
plt.title("Posterior distributions of $p_A$, $p_B$, and delta unknowns")

ax = plt.subplot(312)

plt.xlim(0, .1)
plt.hist(p_B_samples, histtype='stepfilled', bins=25, alpha=0.85,
         label="posterior of $p_B$", color="#467821", normed=True)
plt.vlines(true_p_B, 0, 80, linestyle="--", label="true $p_B$ (unknown)")
plt.legend(loc="upper right")

ax = plt.subplot(313)
plt.hist(delta_samples, histtype='stepfilled', bins=30, alpha=0.85,
         label="posterior of delta", color="#7A68A6", normed=True)
plt.vlines(true_p_A - true_p_B, 0, 60, linestyle="--",
           label="true delta (unknown)")
plt.vlines(0, 0, 60, color="black", alpha=0.2)
plt.legend(loc="upper right");
```

Notice that as a result of `N_B < N_A`, i.e. we have less data from site B, our posterior distribution of p_B is fatter, implying we are less certain about the true value of p_B than we are of p_A.

With respect to the posterior distribution of delta, we can see that the majority of the distribution is above delta $= 0$, implying there site A's response is likely better than site B's response. The probability this inference is incorrect is easily computable:

```
In [26]: # Count the number of samples less than 0, i.e. the area under the curve
         # before 0, represent the probability that site A is worse than site B.
         print("Probability site A is WORSE than site B: %.3f" % \
             (delta_samples < 0).mean())

         print("Probability site A is BETTER than site B: %.3f" % \
             (delta_samples > 0).mean())
```

```
Probability site A is WORSE than site B: 0.017
Probability site A is BETTER than site B: 0.983
```

If this probability is too high for comfortable decision-making, we can perform more trials on site B (as site B has less samples to begin with, each additional data point for site B contributes more inferential "power" than each additional data point for site A).

Try playing with the parameters `true_p_A`, `true_p_B`, `N_A`, and `N_B`, to see what the posterior of delta looks like. Notice in all this, the difference in sample sizes between site A and site B was never mentioned: it naturally fits into Bayesian analysis.

I hope the readers feel this style of A/B testing is more natural than hypothesis testing, which has probably confused more than helped practitioners. Later in this book, we will see two extensions of this model: the first to help dynamically adjust for bad sites, and the second will improve the speed of this computation by reducing the analysis to a single equation.

1.3 An algorithm for human deceit

Social data has an additional layer of interest as people are not always honest with responses, which adds a further complication into inference. For example, simply asking individuals "Have you ever cheated on a test?" will surely contain some rate of dishonesty. What you can say for certain is that the true rate is less than your observed rate (assuming individuals lie *only* about *not cheating*; I cannot imagine one who would admit "Yes" to cheating when in fact they hadn't cheated).

To present an elegant solution to circumventing this dishonesty problem, and to demonstrate Bayesian modeling, we first need to introduce the binomial distribution.

1.3.1 The Binomial Distribution

The binomial distribution is one of the most popular distributions, mostly because of its simplicity and usefulness. Unlike the other distributions we have encountered thus far in the book, the binomial distribution has 2 parameters: N, a positive integer representing N trials or number of instances of potential events, and p, the probability of an event occurring in a single trial. Like the Poisson distribution, it is a discrete distribution, but unlike the Poisson distribution, it only weighs integers from 0 to N. The mass distribution looks like:

$$P(X = k) = \binom{N}{k} p^k (1-p)^{N-k}$$

If X is a binomial random variable with parameters p and N, denoted $X \sim \text{Bin}(N, p)$, then X is the number of events that occurred in the N trials (obviously $0 \leq X \leq N$), and p is the probability of a single event. The larger p is (while still remaining between 0 and 1), the more events are likely to occur. The expected value of a binomial is equal to Np. Below we plot the mass probability distribution for varying parameters.

```
In [27]: figsize(12.5, 4)

         import scipy.stats as stats
         binomial = stats.binom

         parameters = [(10, .4), (10, .9)]
         colors = ["#348ABD", "#A60628"]

         for i in range(2):
             N, p = parameters[i]
             _x = np.arange(N + 1)
             plt.bar(_x - 0.5, binomial.pmf(_x, N, p), color=colors[i],
```

```
                edgecolor=colors[i],
                alpha=0.6,
                label="$N$: %d, $p$: %.1f" % (N, p),
                linewidth=3)

    plt.legend(loc="upper left")
    plt.xlim(0, 10.5)
    plt.xlabel("$k$")
    plt.ylabel("$P(X = k)$")
    plt.title("Probability mass distributions of binomial random variables");
```

The special case when $N = 1$ corresponds to the Bernoulli distribution. There is another connection between Bernoulli and Binomial random variables. If we have $X_1, X_2, ..., X_N$ Bernoulli random variables with the same p, then $Z = X_1 + X_2 + ... + X_N \sim \text{Binomial}(N, p)$.

The expected value of a Bernoulli random variable is p. This can be seen by noting the more general Binomial random variable has expected value Np and setting $N = 1$.

Example: Cheating among students We will use the binomial distribution to determine the frequency of students cheating during an exam. If we let N be the total number of students who took the exam, and assuming each student is interviewed post-exam (answering without consequence), we will receive integer X "Yes I did cheat" answers. We then find the posterior distribution of p, given N, some specified prior on p, and observed data X.

This is a completely absurd model. No student, even with a free-pass against punishment, would admit to cheating. What we need is a better *algorithm* to ask students if they had cheated. Ideally the algorithm should encourage individuals to be honest while preserving privacy. The following proposed algorithm is a solution I greatly admire for its ingenuity and effectiveness:

> In the interview process for each student, the student flips a coin, hidden from the interviewer. The student agrees to answer honestly if the coin comes up heads. Otherwise, if the coin comes up tails, the student (secretly) flips the coin again, and answers "Yes, I did cheat" if the coin flip lands heads, and "No, I did not cheat", if the coin flip lands tails. This way, the interviewer does not know if a "Yes" was the result of a guilty plea, or a Heads on a second coin toss. Thus privacy is preserved and the researchers receive honest answers.

I call this the Privacy Algorithm. One could of course argue that the interviewers are still receiving false data since some *Yes*'s are not confessions but instead randomness, but an alternative perspective is that the researchers are discarding approximately half of their original dataset since half of the responses will be noise. But they have gained a systematic data generation process that can be modeled. Furthermore, they do not have to incorporate (perhaps somewhat naively) the possibility of deceitful answers. We can use PyMC to dig through this noisy model, and find a posterior distribution for the true frequency of liars.

Suppose 100 students are being surveyed for cheating, and we wish to find p, the proportion of cheaters. There are a few ways we can model this in PyMC. I'll demonstrate the most explicit way, and later show a simplified version. Both versions arrive at the same inference. In our data-generation model, we sample p, the true proportion of cheaters, from a prior. Since we are quite ignorant about p, we will assign it a Uniform$(0, 1)$ prior.

```
In [28]: import pymc as pm

         N = 100
         p = pm.Uniform("freq_cheating", 0, 1)
```

Again, thinking of our data-generation model, we assign Bernoulli random variables to the 100 students: 1 implies they cheated and 0 implies they did not.

```
In [29]: true_answers = pm.Bernoulli("truths", p, size=N)
```

If we carry out the algorithm, the next step that occurs is the first coin-flip each student makes. This can be modeled again by sampling 100 Bernoulli random variables with $p = 1/2$: denote a 1 as a *Heads* and 0 a *Tails*.

```
In [30]: first_coin_flips = pm.Bernoulli("first_flips", 0.5, size=N)
         print(first_coin_flips.value)
```

```
[ True False  True  True  True False False  True False  True  True False
  True  True False  True False  True  True False  True False  True  True
  True  True False  True  True False  True False  True False  True False
 False  True  True False  True  True False False  True False False False
  True False  True  True False False False  True  True False False False
  True  True  True False False False False False  True False False  True
  True False False  True False False False False  True False  True False
 False False  True  True False False  True False  True  True  True False
 False False False  True]
```

Although *not everyone* flips a second time, we can still model the possible realization of second coin-flips:

```
In [31]: second_coin_flips = pm.Bernoulli("second_flips", 0.5, size=N)
```

Using these variables, we can return a possible realization of the *observed proportion* of "Yes" responses. We do this using a PyMC `deterministic` variable:

```
In [32]: @pm.deterministic
         def observed_proportion(t_a=true_answers,
                                 fc=first_coin_flips,
                                 sc=second_coin_flips):

             observed = fc * t_a + (1 - fc) * sc
             return observed.sum() / float(N)
```

The line `fc*t_a + (1-fc)*sc` contains the heart of the Privacy algorithm. Elements in this array are 1 *if and only if* i) the first toss is heads and the student cheated or ii) the first toss is tails, and the second is heads, and are 0 else. Finally, the last line sums this vector and divides by `float(N)`, produces a proportion.

```
In [33]: observed_proportion.value
```

```
Out[33]: 0.58999999999999997
```

Next we need a dataset. After performing our coin-flipped interviews the researchers received 35 "Yes" responses. To put this into a relative perspective, if there truly were no cheaters, we should expect to see on average 1/4 of all responses being a "Yes" (half chance of having first coin land Tails, and another half chance of having second coin land Heads), so about 25 responses in a cheat-free world. On the other hand, if *all students cheated*, we should expect to see approximately 3/4 of all responses be "Yes".

The researchers observe a Binomial random variable, with `N = 100` and `p = observed_proportion` with `value = 35`:

```
In [34]: X = 35

         observations = pm.Binomial("obs", N, observed_proportion, observed=True,
                                    value=X)
```

Below we add all the variables of interest to a `Model` container and run our black-box algorithm over the model.

```
In [35]: model = pm.Model([p, true_answers, first_coin_flips,
                           second_coin_flips, observed_proportion, observations])

         # To be explained in Chapter 3!
         mcmc = pm.MCMC(model)
         mcmc.sample(40000, 15000)
```

```
 [-----------------100%-----------------] 40000 of 40000 complete in 10.4 sec
```

```
In [36]: figsize(12.5, 3)
         p_trace = mcmc.trace("freq_cheating")[:]
         plt.hist(p_trace, histtype="stepfilled", normed=True, alpha=0.85, bins=30,
                  label="posterior distribution", color="#348ABD")
         plt.vlines([.05, .35], [0, 0], [5, 5], alpha=0.3)
         plt.xlim(0, 1)
         plt.legend();
```

With regards to the above plot, we are still pretty uncertain about what the true frequency of cheaters might be, but we have narrowed it down to a range between 0.05 to 0.35 (marked by the solid lines). This is pretty good, as *a priori* we had no idea how many students might have cheated (hence the uniform distribution for our prior). On the other hand, it is also pretty bad since there is a .3 length window the true value most likely lives in. Have we even gained anything, or are we still too uncertain about the true frequency?

I would argue, yes, we have discovered something. It is implausible, according to our posterior, that there are *no cheaters*, i.e. the posterior assigns low probability to $p = 0$. Since we started with a uniform prior, treating all values of p as equally plausible, but the data ruled out $p = 0$ as a possibility, we can be confident that there were cheaters.

This kind of algorithm can be used to gather private information from users and be *reasonably* confident that the data, though noisy, is truthful.

1.3.2 Alternative PyMC Model

Given a value for p (which from our god-like position we know), we can find the probability the student will answer yes:

$$P(\text{"Yes"}) = P(\text{Heads on first coin})P(\text{cheater}) + P(\text{Tails on first coin})P(\text{Heads on second coin}) \tag{1}$$

$$\tag{2}$$

$$= \frac{1}{2}p + \frac{1}{2}\frac{1}{2} \tag{3}$$

$$\tag{4}$$

$$= \frac{p}{2} + \frac{1}{4} \tag{5}$$

Thus, knowing p we know the probability a student will respond "Yes". In PyMC, we can create a deterministic function to evaluate the probability of responding "Yes", given p:

```
In [37]: p = pm.Uniform("freq_cheating", 0, 1)


         @pm.deterministic
         def p_skewed(p=p):
             return 0.5 * p + 0.25
```

I could have typed `p_skewed = 0.5*p + 0.25` instead for a one-liner, as the elementary operations of addition and scalar multiplication will implicitly create a `deterministic` variable, but I wanted to make the deterministic boilerplate explicit for clarity's sake.

If we know the probability of respondents saying "Yes", which is `p_skewed`, and we have $N = 100$ students, the number of "Yes" responses is a binomial random variable with parameters `N` and `p_skewed`.

This is where we include our observed 35 "Yes" responses. In the declaration of the `pm.Binomial`, we include `value = 35` and `observed = True`.

```
In [38]: yes_responses = pm.Binomial("number_cheaters", 100, p_skewed,
                                     value=35, observed=True)
```

Below we add all the variables of interest to a `Model` container and run our black-box algorithm over the model.

```
In [39]: model = pm.Model([yes_responses, p_skewed, p])

         # To Be Explained in Chapter 3!
         mcmc = pm.MCMC(model)
         mcmc.sample(25000, 2500)
```

```
 [-----------------100%-----------------] 25000 of 25000 complete in 1.2 sec
```

```
In [40]: figsize(12.5, 3)
         p_trace = mcmc.trace("freq_cheating")[:]
         plt.hist(p_trace, histtype="stepfilled", normed=True, alpha=0.85, bins=30,
                  label="posterior distribution", color="#348ABD")
         plt.vlines([.05, .35], [0, 0], [5, 5], alpha=0.2)
         plt.xlim(0, 1)
         plt.legend();
```

1.3.3 More PyMC Tricks

Protip: *Lighter* deterministic variables with `Lambda` class Sometimes writing a deterministic function using the `@pm.deterministic` decorator can seem like a chore, especially for a small function. I have already mentioned that elementary math operations *can* produce deterministic variables implicitly, but what about operations like indexing or slicing? Built-in `Lambda` functions can handle this with the elegance and simplicity required. For example,

```
beta = pm.Normal("coefficients", 0, size=(N, 1))
x = np.random.randn((N, 1))
linear_combination = pm.Lambda(lambda x=x, beta=beta: np.dot(x.T, beta))
```

Protip: Arrays of PyMC variables There is no reason why we cannot store multiple heterogeneous PyMC variables in a Numpy array. Just remember to set the `dtype` of the array to `object` upon initialization. For example:

```
In [41]: N = 10
         x = np.empty(N, dtype=object)
         for i in range(0, N):
             x[i] = pm.Exponential('x_%i' % i, (i + 1) ** 2)
```

The remainder of this chapter examines some practical examples of PyMC and PyMC modeling:

Example: Challenger Space Shuttle Disaster On January 28, 1986, the twenty-fifth flight of the U.S. space shuttle program ended in disaster when one of the rocket boosters of the Shuttle Challenger exploded shortly after lift-off, killing all seven crew members. The presidential commission on the accident concluded that it was caused by the failure of an O-ring in a field joint on the rocket booster, and that this failure was due to a faulty design that made the O-ring unacceptably sensitive to a number of factors including outside temperature. Of the previous 24 flights, data were available on failures of O-rings on 23, (one was lost at sea), and these data were discussed on the evening preceding the Challenger launch, but unfortunately only the data corresponding to the 7 flights on which there was a damage incident were considered important and these were thought to show no obvious trend. The data are shown below (see [1]):

```
In [42]: figsize(12.5, 3.5)
         np.set_printoptions(precision=3, suppress=True)
         challenger_data = np.genfromtxt("data/challenger_data.csv", skip_header=1,
                                         usecols=[1, 2], missing_values="NA",
                                         delimiter=",")
         # drop the NA values
         challenger_data = challenger_data[~np.isnan(challenger_data[:, 1])]

         # plot it, as a function of temperature (the first column)
         print("Temp (F), O-Ring failure?")
         print(challenger_data)

         plt.scatter(challenger_data[:, 0], challenger_data[:, 1], s=75, color="k",
                     alpha=0.5)
         plt.yticks([0, 1])
         plt.ylabel("Damage Incident?")
         plt.xlabel("Outside temperature (Fahrenheit)")
         plt.title("Defects of the Space Shuttle O-Rings vs temperature");
```

```
Temp (F), O-Ring failure?
[[ 66.   0.]
```

```
[ 70.   1.]
[ 69.   0.]
[ 68.   0.]
[ 67.   0.]
[ 72.   0.]
[ 73.   0.]
[ 70.   0.]
[ 57.   1.]
[ 63.   1.]
[ 70.   1.]
[ 78.   0.]
[ 67.   0.]
[ 53.   1.]
[ 67.   0.]
[ 75.   0.]
[ 70.   0.]
[ 81.   0.]
[ 76.   0.]
[ 79.   0.]
[ 75.   1.]
[ 76.   0.]
[ 58.   1.]]
```

It looks clear that *the probability* of damage incidents occurring increases as the outside temperature decreases. We are interested in modeling the probability here because it does not look like there is a strict cutoff point between temperature and a damage incident occurring. The best we can do is ask "At temperature t, what is the probability of a damage incident?". The goal of this example is to answer that question.

We need a function of temperature, call it $p(t)$, that is bounded between 0 and 1 (so as to model a probability) and changes from 1 to 0 as we increase temperature. There are actually many such functions, but the most popular choice is the *logistic function.*

$$p(t) = \frac{1}{1 + e^{\beta t}}$$

In this model, β is the variable we are uncertain about. Below is the function plotted for $\beta = 1, 3, -5$.

```
In [43]: figsize(12, 3)


         def logistic(x, beta):
             return 1.0 / (1.0 + np.exp(beta * x))

         x = np.linspace(-4, 4, 100)
         plt.plot(x, logistic(x, 1), label=r"$\beta = 1$")
         plt.plot(x, logistic(x, 3), label=r"$\beta = 3$")
         plt.plot(x, logistic(x, -5), label=r"$\beta = -5$")
         plt.title("Logistic functon plotted for several value of $\\beta$ parameter", fontsize
         plt.legend();
```

But something is missing. In the plot of the logistic function, the probability changes only near zero, but in our data above the probability changes around 65 to 70. We need to add a *bias* term to our logistic function:

$$p(t) = \frac{1}{1 + e^{\beta t + \alpha}}$$

Some plots are below, with differing α.

```
In [44]: def logistic(x, beta, alpha=0):
             return 1.0 / (1.0 + np.exp(np.dot(beta, x) + alpha))

         x = np.linspace(-4, 4, 100)

         plt.plot(x, logistic(x, 1), label=r"$\beta = 1$", ls="--", lw=1)
         plt.plot(x, logistic(x, 3), label=r"$\beta = 3$", ls="--", lw=1)
         plt.plot(x, logistic(x, -5), label=r"$\beta = -5$", ls="--", lw=1)

         plt.plot(x, logistic(x, 1, 1), label=r"$\beta = 1, \alpha = 1$",
                  color="#348ABD")
         plt.plot(x, logistic(x, 3, -2), label=r"$\beta = 3, \alpha = -2$",
```

```
            color="#A60628")
plt.plot(x, logistic(x, -5, 7), label=r"$\beta = -5, \alpha = 7$",
            color="#7A68A6")

plt.title("Logistic functon with bias, plotted for several value of $\\alpha$ bias pa
plt.legend(loc="lower left");
```

Adding a constant term α amounts to shifting the curve left or right (hence why it is called a *bias*).

Let's start modeling this in PyMC. The β, α parameters have no reason to be positive, bounded or relatively large, so they are best modeled by a *Normal random variable*, introduced next.

1.3.4 Normal distributions

A Normal random variable, denoted $X \sim N(\mu, 1/\tau)$, has a distribution with two parameters: the mean, μ, and the *precision*, τ. Those familiar with the Normal distribution already have probably seen σ^2 instead of τ^{-1}. They are in fact reciprocals of each other. The change was motivated by simpler mathematical analysis and is an artifact of older Bayesian methods. Just remember: the smaller τ, the larger the spread of the distribution (i.e. we are more uncertain); the larger τ, the tighter the distribution (i.e. we are more certain). Regardless, τ is always positive.

The probability density function of a $N(\mu, 1/\tau)$ random variable is:

$$f(x|\mu, \tau) = \sqrt{\frac{\tau}{2\pi}} \exp\left(-\frac{\tau}{2}(x-\mu)^2\right)$$

We plot some different density functions below.

```
In [45]: import scipy.stats as stats

         nor = stats.norm
         x = np.linspace(-8, 7, 150)
         mu = (-2, 0, 3)
         tau = (.7, 1, 2.8)
         colors = ["#348ABD", "#A60628", "#7A68A6"]
         parameters = zip(mu, tau, colors)

         for _mu, _tau, _color in parameters:
```

```
        plt.plot(x, nor.pdf(x, _mu, scale=1. / np.sqrt(_tau)),
                 label="$\mu = %d,\;\\tau = %.1f$" % (_mu, _tau), color=_color)
        plt.fill_between(x, nor.pdf(x, _mu, scale=1. / np.sqrt(_tau)), color=_color,
                         alpha=.33)

    plt.legend(loc="upper right")
    plt.xlabel("$x$")
    plt.ylabel("density function at $x$")
    plt.title("Probability distribution of three different Normal random \
    variables");
```

A Normal random variable can be take on any real number, but the variable is very likely to be relatively close to μ. In fact, the expected value of a Normal is equal to its μ parameter:

$$E[X|\mu, \tau] = \mu$$

and its variance is equal to the inverse of τ:

$$Var(X|\mu, \tau) = \frac{1}{\tau}$$

Below we continue our modeling of the Challenger space craft:

```
In [46]: import pymc as pm

         temperature = challenger_data[:, 0]
         D = challenger_data[:, 1]  # defect or not?

         # notice the`value` here. We explain why below.
         beta = pm.Normal("beta", 0, 0.001, value=0)
         alpha = pm.Normal("alpha", 0, 0.001, value=0)


         @pm.deterministic
         def p(t=temperature, alpha=alpha, beta=beta):
             return 1.0 / (1. + np.exp(beta * t + alpha))
```

We have our probabilities, but how do we connect them to our observed data? A *Bernoulli* random variable with parameter p, denoted $\text{Ber}(p)$, is a random variable that takes value 1 with probability p, and 0 else. Thus, our model can look like:

$$\text{Defect Incident, } D_i \sim \text{Ber}(\ p(t_i)\ ),\ \ i = 1..N$$

where $p(t)$ is our logistic function and t_i are the temperatures we have observations about. Notice in the above code we had to set the values of `beta` and `alpha` to 0. The reason for this is that if `beta` and `alpha` are very large, they make `p` equal to 1 or 0. Unfortunately, `pm.Bernoulli` does not like probabilities of exactly 0 or 1, though they are mathematically well-defined probabilities. So by setting the coefficient values to `0`, we set the variable `p` to be a reasonable starting value. This has no effect on our results, nor does it mean we are including any additional information in our prior. It is simply a computational caveat in PyMC.

```
In [47]: p.value
```

```
Out[47]: array([ 0.5,  0.5,  0.5,  0.5,  0.5,  0.5,  0.5,  0.5,  0.5,  0.5,  0.5,
                 0.5,  0.5,  0.5,  0.5,  0.5,  0.5,  0.5,  0.5,  0.5,  0.5,  0.5,
                 0.5])
```

```
In [48]: # connect the probabilities in `p` with our observations through a
         # Bernoulli random variable.
         observed = pm.Bernoulli("bernoulli_obs", p, value=D, observed=True)

         model = pm.Model([observed, beta, alpha])

         # Mysterious code to be explained in Chapter 3
         map_ = pm.MAP(model)
         map_.fit()
         mcmc = pm.MCMC(model)
         mcmc.sample(120000, 100000, 2)
```

```
 [-----------------100%-----------------] 120000 of 120000 complete in 9.6 sec
```

We have trained our model on the observed data, now we can sample values from the posterior. Let's look at the posterior distributions for α and β:

```
In [49]: alpha_samples = mcmc.trace('alpha')[:, None]  # best to make them 1d
         beta_samples = mcmc.trace('beta')[:, None]

         figsize(12.5, 6)

         # histogram of the samples:
         plt.subplot(211)
         plt.title(r"Posterior distributions of the variables $\alpha, \beta$")
         plt.hist(beta_samples, histtype='stepfilled', bins=35, alpha=0.85,
                  label=r"posterior of $\beta$", color="#7A68A6", normed=True)
         plt.legend()
```

```
plt.subplot(212)
plt.hist(alpha_samples, histtype='stepfilled', bins=35, alpha=0.85,
         label=r"posterior of $\alpha$", color="#A60628", normed=True)
plt.legend();
```

All samples of β are greater than 0. If instead the posterior was centered around 0, we may suspect that $\beta = 0$, implying that temperature has no effect on the probability of defect.

Similarly, all α posterior values are negative and far away from 0, implying that it is correct to believe that α is significantly less than 0.

Regarding the spread of the data, we are very uncertain about what the true parameters might be (though considering the low sample size and the large overlap of defects-to-nondefects this behaviour is perhaps expected).

Next, let's look at the *expected probability* for a specific value of the temperature. That is, we average over all samples from the posterior to get a likely value for $p(t_i)$.

```
In [50]: t = np.linspace(temperature.min() - 5, temperature.max() + 5, 50)[:, None]
         p_t = logistic(t.T, beta_samples, alpha_samples)

         mean_prob_t = p_t.mean(axis=0)
```

```
In [51]: figsize(12.5, 4)

         plt.plot(t, mean_prob_t, lw=3, label="average posterior \nprobability \
         of defect")
         plt.plot(t, p_t[0, :], ls="--", label="realization from posterior")
         plt.plot(t, p_t[-2, :], ls="--", label="realization from posterior")
         plt.scatter(temperature, D, color="k", s=50, alpha=0.5)
         plt.title("Posterior expected value of probability of defect; \
         plus realizations")
```

```
plt.legend(loc="lower left")
plt.ylim(-0.1, 1.1)
plt.xlim(t.min(), t.max())
plt.ylabel("probability")
plt.xlabel("temperature");
```

Above we also plotted two possible realizations of what the actual underlying system might be. Both are equally likely as any other draw. The blue line is what occurs when we average all the 20000 possible dotted lines together.

An interesting question to ask is for what temperatures are we most uncertain about the defect-probability? Below we plot the expected value line **and** the associated 95% intervals for each temperature.

```
In [52]: from scipy.stats.mstats import mquantiles

         # vectorized bottom and top 2.5% quantiles for "confidence interval"
         qs = mquantiles(p_t, [0.025, 0.975], axis=0)
         plt.fill_between(t[:, 0], *qs, alpha=0.7,
                          color="#7A68A6")

         plt.plot(t[:, 0], qs[0], label="95% CI", color="#7A68A6", alpha=0.7)

         plt.plot(t, mean_prob_t, lw=1, ls="--", color="k",
                  label="average posterior \nprobability of defect")

         plt.xlim(t.min(), t.max())
         plt.ylim(-0.02, 1.02)
         plt.legend(loc="lower left")
         plt.scatter(temperature, D, color="k", s=50, alpha=0.5)
         plt.xlabel("temp, $t$")

         plt.ylabel("probability estimate")
         plt.title("Posterior probability estimates given temp. $t$");
```

The *95% credible interval*, or 95% CI, painted in purple, represents the interval, for each temperature, that contains 95% of the distribution. For example, at 65 degrees, we can be 95% sure that the probability of defect lies between 0.25 and 0.75.

More generally, we can see that as the temperature nears 60 degrees, the CI's spread out over [0,1] quickly. As we pass 70 degrees, the CI's tighten again. This can give us insight about how to proceed next: we should probably test more O-rings around 60-65 temperature to get a better estimate of probabilities in that range. Similarly, when reporting to scientists your estimates, you should be very cautious about simply telling them the expected probability, as we can see this does not reflect how *wide* the posterior distribution is.

1.3.5 What about the day of the Challenger disaster?

On the day of the Challenger disaster, the outside temperature was 31 degrees Fahrenheit. What is the posterior distribution of a defect occurring, given this temperature? The distribution is plotted below. It looks almost guaranteed that the Challenger was going to be subject to defective O-rings.

```
In [53]: figsize(12.5, 2.5)

         prob_31 = logistic(31, beta_samples, alpha_samples)

         plt.xlim(0.995, 1)
         plt.hist(prob_31, bins=1000, normed=True, histtype='stepfilled')
         plt.title("Posterior distribution of probability of defect, given $t = 31$")
         plt.xlabel("probability of defect occurring in O-ring");
```

1.3.6 Is our model appropriate?

The skeptical reader will say "You deliberately chose the logistic function for $p(t)$ and the specific priors. Perhaps other functions or priors will give different results. How do I know I have chosen a good model?" This is absolutely true. To consider an extreme situation, what if I had chosen the function $p(t) = 1, \forall t$, which guarantees a defect always occurring: I would have again predicted disaster on January 28th. Yet this is clearly a poorly chosen model. On the other hand, if I did choose the logistic function for $p(t)$, but specified all my priors to be very tight around 0, likely we would have very different posterior distributions. How do we know our model is an expression of the data? This encourages us to measure the model's **goodness of fit**.

We can think: *how can we test whether our model is a bad fit?* An idea is to compare observed data (which if we recall is a *fixed* stochastic variable) with an artificial dataset which we can simulate. The rationale is that if the simulated dataset does not appear similar, statistically, to the observed dataset, then likely our model is not accurately represented the observed data.

Previously in this Chapter, we simulated artificial datasets for the SMS example. To do this, we sampled values from the priors. We saw how varied the resulting datasets looked like, and rarely did they mimic our observed dataset. In the current example, we should sample from the *posterior* distributions to create *very plausible datasets*. Luckily, our Bayesian framework makes this very easy. We only need to create a new `Stochastic` variable, that is exactly the same as our variable that stored the observations, but minus the observations themselves. If you recall, our `Stochastic` variable that stored our observed data was:

```
observed = pm.Bernoulli( "bernoulli_obs", p, value=D, observed=True)
```

Hence we create:

```
simulated_data = pm.Bernoulli("simulation_data", p)
```

Let's simulate 10 000:

```
In [54]: simulated = pm.Bernoulli("bernoulli_sim", p)
         N = 10000

         mcmc = pm.MCMC([simulated, alpha, beta, observed])
         mcmc.sample(N)
```

```
 [-----------------100%-----------------] 10000 of 10000 complete in 1.5 sec
```

```
In [55]: figsize(12.5, 5)

         simulations = mcmc.trace("bernoulli_sim")[:]
         print(simulations.shape)

         plt.title("Simulated dataset using posterior parameters")
         figsize(12.5, 6)
         for i in range(4):
```

```
        ax = plt.subplot(4, 1, i + 1)
        plt.scatter(temperature, simulations[1000 * i, :], color="k",
                    s=50, alpha=0.6)
```

```
(10000, 23)
```

Note that the above plots are different (if you can think of a cleaner way to present this, please send a pull request and answer here!).

We wish to assess how good our model is. "Good" is a subjective term of course, so results must be relative to other models.

We will be doing this graphically as well, which may seem like an even less objective method. The alternative is to use *Bayesian p-values*. These are still subjective, as the proper cutoff between good and bad is arbitrary. Gelman emphasises that the graphical tests are more illuminating [7] than p-value tests. We agree.

The following graphical test is a novel data-viz approach to logistic regression. The plots are called *separation plots*[8]. For a suite of models we wish to compare, each model is plotted on an individual separation plot. I leave most of the technical details about separation plots to the very accessible original paper, but I'll summarize their use here.

For each model, we calculate the proportion of times the posterior simulation proposed a value of 1 for a particular temperature, i.e. compute $P(\text{ Defect } = 1|t, \alpha, \beta)$ by averaging. This gives us the posterior probability of a defect at each data point in our dataset. For example, for the model we used above:

```
In [56]: posterior_probability = simulations.mean(axis=0)
         print("posterior prob of defect | realized defect ")
         for i in range(len(D)):
             print("%.2f                     |   %d" % (posterior_probability[i], D[i]))
```

```
posterior prob of defect | realized defect
0.41                     |   0
0.24                     |   1
0.28                     |   0
```

```
0.32                    |   0
0.36                    |   0
0.18                    |   0
0.16                    |   0
0.25                    |   0
0.77                    |   1
0.54                    |   1
0.24                    |   1
0.07                    |   0
0.37                    |   0
0.86                    |   1
0.36                    |   0
0.11                    |   0
0.24                    |   0
0.05                    |   0
0.10                    |   0
0.06                    |   0
0.12                    |   1
0.10                    |   0
0.75                    |   1
```

Next we sort each column by the posterior probabilities:

```
In [57]: ix = np.argsort(posterior_probability)
         print("probb | defect ")
         for i in range(len(D)):
             print("%.2f  |   %d" % (posterior_probability[ix[i]], D[ix[i]]))
```

```
probb | defect
0.05  |   0
0.06  |   0
0.07  |   0
0.10  |   0
0.10  |   0
0.11  |   0
0.12  |   1
0.16  |   0
0.18  |   0
0.24  |   1
0.24  |   1
0.24  |   0
0.25  |   0
0.28  |   0
0.32  |   0
0.36  |   0
0.36  |   0
0.37  |   0
```

```
0.41  |   0
0.54  |   1
0.75  |   1
0.77  |   1
0.86  |   1
```

We can present the above data better in a figure: I've wrapped this up into a `separation_plot` function.

```
In [58]: from separation_plot import separation_plot
```

```
figsize(11., 1.5)
separation_plot(posterior_probability, D)
```

The snaking-line is the sorted probabilities, blue bars denote defects, and empty space (or grey bars for the optimistic readers) denote non-defects. As the probability rises, we see more and more defects occur. On the right hand side, the plot suggests that as the posterior probability is large (line close to 1), then more defects are realized. This is good behaviour. Ideally, all the blue bars *should* be close to the right-hand side, and deviations from this reflect missed predictions.

The black vertical line is the expected number of defects we should observe, given this model. This allows the user to see how the total number of events predicted by the model compares to the actual number of events in the data.

It is much more informative to compare this to separation plots for other models. Below we compare our model (top) versus three others:

1. the perfect model, which predicts the posterior probability to be equal to 1 if a defect did occur.
2. a completely random model, which predicts random probabilities regardless of temperature.
3. a constant model: where $P(D = 1 \mid t) = c, \quad \forall t$. The best choice for c is the observed frequency of defects, in this case $7/23$.

```
In [59]: figsize(11., 1.25)

         # Our temperature-dependent model
         separation_plot(posterior_probability, D)
         plt.title("Temperature-dependent model")

         # Perfect model
         # i.e. the probability of defect is equal to if a defect occurred or not.
         p = D
```

```
separation_plot(p, D)
plt.title("Perfect model")

# random predictions
p = np.random.rand(23)
separation_plot(p, D)
plt.title("Random model")

# constant model
constant_prob = 7. / 23 * np.ones(23)
separation_plot(constant_prob, D)
plt.title("Constant-prediction model");
```

In the random model, we can see that as the probability increases there is no clustering of defects to the right-hand side. Similarly for the constant model.

The perfect model, the probability line is not well shown, as it is stuck to the bottom and top of the figure. Of course the perfect model is only for demonstration, and we cannot infer any scientific inference from it.

Exercises 1. Try putting in extreme values for our observations in the cheating example. What happens if we observe 25 affirmative responses? 10? 50?

2. Try plotting α samples versus β samples. Why might the resulting plot look like this?

```
In [60]: # type your code here.
         figsize(12.5, 4)

         plt.scatter(alpha_samples, beta_samples, alpha=0.1)
         plt.title("Why does the plot look like this?")
         plt.xlabel(r"$\alpha$")
         plt.ylabel(r"$\beta$");
```

1.3.7 References

- [1] Dalal, Fowlkes and Hoadley (1989),JASA, 84, 945-957.
- [2] German Rodriguez. Datasets. In WWS509. Retrieved 30/01/2013, from http://data.princeton.edu/wws509/datasets/#smoking.
- [3] McLeish, Don, and Cyntha Struthers. STATISTICS 450/850 Estimation and Hypothesis Testing. Winter 2012. Waterloo, Ontario: 2012. Print.
- [4] Fonnesbeck, Christopher. "Building Models." PyMC-Devs. N.p., n.d. Web. 26 Feb 2013. http://pymc-devs.github.com/pymc/modelbuilding.html.
- [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.
- [6] S.P. Brooks, E.A. Catchpole, and B.J.T. Morgan. Bayesian animal survival estimation. Statistical Science, 15: 357–376, 2000
- [7] Gelman, Andrew. "Philosophy and the practice of Bayesian statistics." British Journal of Mathematical and Statistical Psychology. (2012): n. page. Web. 2 Apr. 2013.
- [8] Greenhill, Brian, Michael D. Ward, and Audrey Sacks. "The Separation Plot: A New Visual Method for Evaluating the Fit of Binary Models." American Journal of Political Science. 55.No.4 (2011): n. page. Web. 2 Apr. 2013.

```
In [ ]: from IPython.core.display import HTML


        def css_styling():
            styles = open("../styles/custom.css", "r").read()
            return HTML(styles)
        css_styling()
```

```
In [ ]:
```

Ch2_MorePyMC_PyMC3

January 4, 2018

1 Chapter 2

```
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)
```

This chapter introduces more PyMC3 syntax and variables and ways to think about how to model a system from a Bayesian perspective. It also contains tips and data visualization techniques for assessing goodness-of-fit for your Bayesian model.

1.1 A little more on PyMC3

1.1.1 Model Context

In PyMC3, we typically handle all the variables we want in our model within the context of the `Model` object.

```
In [1]: import pymc3 as pm

        with pm.Model() as model:
            parameter = pm.Exponential("poisson_param", 1)
            data_generator = pm.Poisson("data_generator", parameter)
```

```
Applied log-transform to poisson_param and added transformed poisson_param_log_ to model.
```

This is an extra layer of convenience compared to PyMC. Any variables created within a given `Model`'s context will be automatically assigned to that model. If you try to define a variable outside of the context of a model, you will get an error.

We can continue to work within the context of the same model by using `with` with the name of the model object that we have already created.

```
In [2]: with model:
            data_plus_one = data_generator + 1
```

We can examine the same variables outside of the model context once they have been defined, but to define more variables that the model will recognize they have to be within the context.

```
In [3]: parameter.tag.test_value
```

```
Out[3]: array(0.693147177890573)
```

Each variable assigned to a model will be defined with its own name, the first string parameter (we will cover this further in the variables section). To create a different model object with the same name as one we have used previously, we need only run the first block of code again.

```
In [4]: with pm.Model() as model:
            theta = pm.Exponential("theta", 2)
            data_generator = pm.Poisson("data_generator", theta)
```

```
Applied log-transform to theta and added transformed theta_log_ to model.
```

We can also define an entirely separate model. Note that we are free to name our models whatever we like, so if we do not want to overwrite an old model we need only make another.

```
In [5]: with pm.Model() as ab_testing:
            p_A = pm.Uniform("P(A)", 0, 1)
            p_B = pm.Uniform("P(B)", 0, 1)
```

```
Applied interval-transform to P(A) and added transformed P(A)_interval_ to model.
Applied interval-transform to P(B) and added transformed P(B)_interval_ to model.
```

You probably noticed that PyMC3 will often give you notifications about transformations when you add variables to your model. These transformations are done internally by PyMC3 to modify the space that the variable is sampled in (when we get to actually sampling the model). This is an internal feature which helps with the convergence of our samples to the posterior distribution and serves to improve the results.

1.1.2 PyMC3 Variables

All PyMC3 variables have an initial value (i.e. test value). Using the same variables from before:

```
In [6]: print("parameter.tag.test_value =", parameter.tag.test_value)
        print("data_generator.tag.test_value =", data_generator.tag.test_value)
        print("data_plus_one.tag.test_value =", data_plus_one.tag.test_value)
```

```
parameter.tag.test_value = 0.693147177890573
data_generator.tag.test_value = 0
data_plus_one.tag.test_value = 1
```

The `test_value` is used only for the model, as the starting point for sampling if no other start is specified. It will not change as a result of sampling. This initial state can be changed at variable creation by specifying a value for the `testval` parameter.

```
In [7]: with pm.Model() as model:
            parameter = pm.Exponential("poisson_param", 1, testval=0.5)

        print("\nparameter.tag.test_value =", parameter.tag.test_value)
```

```
Applied log-transform to poisson_param and added transformed poisson_param_log_ to model.

parameter.tag.test_value = 0.49999999904767284
```

This can be helpful if you are using a more unstable prior that may require a better starting point.

PyMC3 is concerned with two types of programming variables: stochastic and deterministic.

- *stochastic variables* are variables that are not deterministic, i.e., even if you knew all the values of the variables' parameters and components, it would still be random. Included in this category are instances of classes `Poisson`, `DiscreteUniform`, and `Exponential`.
- *deterministic variables* are variables that are not random if the variables' parameters and components were known. This might be confusing at first: a quick mental check is *if I knew all of variable `foo`'s component variables, I could determine what `foo`'s value is.*

We will detail each below.

Initializing Stochastic variables Initializing a stochastic, or random, variable requires a `name` argument, plus additional parameters that are class specific. For example:

`some_variable = pm.DiscreteUniform("discrete_uni_var", 0, 4)`

where 0, 4 are the `DiscreteUniform`-specific lower and upper bound on the random variable. The PyMC3 docs contain the specific parameters for stochastic variables. (Or use `??` if you are using IPython!)

The `name` attribute is used to retrieve the posterior distribution later in the analysis, so it is best to use a descriptive name. Typically, I use the Python variable's name as the `name`.

For multivariable problems, rather than creating a Python array of stochastic variables, addressing the `shape` keyword in the call to a stochastic variable creates multivariate array of (independent) stochastic variables. The array behaves like a NumPy array when used like one, and references to its `tag.test_value` attribute return NumPy arrays.

The `shape` argument also solves the annoying case where you may have many variables β_i, $i = 1, ..., N$ you wish to model. Instead of creating arbitrary names and variables for each one, like:

```
beta_1 = pm.Uniform("beta_1", 0, 1)
beta_2 = pm.Uniform("beta_2", 0, 1)
...
```

we can instead wrap them into a single variable:

```
betas = pm.Uniform("betas", 0, 1, shape=N)
```

Deterministic variables We can create a deterministic variable similarly to how we create a stochastic variable. We simply call up the `Deterministic` class in PyMC3 and pass in the function that we desire

```
deterministic_variable = pm.Deterministic("deterministic variable", some_function_of_variables)
```

For all purposes, we can treat the object `some_deterministic_var` as a variable and not a Python function.

Calling `pymc3.Deterministic` is the most obvious way, but not the only way, to create deterministic variables. Elementary operations, like addition, exponentials etc. implicitly create deterministic variables. For example, the following returns a deterministic variable:

```
In [8]: with pm.Model() as model:
            lambda_1 = pm.Exponential("lambda_1", 1)
            lambda_2 = pm.Exponential("lambda_2", 1)
            tau = pm.DiscreteUniform("tau", lower=0, upper=10)

        new_deterministic_variable = lambda_1 + lambda_2
```

```
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.
```

If we want a `deterministic` variable to actually be tracked by our sampling, however, we need to define it explicitly as a named `deterministic` variable with the constructor.

The use of the `deterministic` variable was seen in the previous chapter's text-message example. Recall the model for λ looked like:

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

And in PyMC3 code:

```
In [9]: import numpy as np

        n_data_points = 5  # in CH1 we had ~70 data points
        idx = np.arange(n_data_points)
        with model:
            lambda_ = pm.math.switch(tau >= idx, lambda_1, lambda_2)
```

Clearly, if τ, λ_1 and λ_2 are known, then λ is known completely, hence it is a deterministic variable. We use the `switch` function here to change from λ_1 to λ_2 at the appropriate time. This function is directly from the `theano` package, which we will discuss in the next section.

Inside a `deterministic` variable, the stochastic variables passed in behave like scalars or NumPy arrays (if multivariable). We can do whatever we want with them as long as the dimensions match up in our calculations.

For example, running the following:

```
def subtract(x, y):
    return x - y

stochastic_1 = pm.Uniform("U_1", 0, 1)
stochastic_2 = pm.Uniform("U_2", 0, 1)

det_1 = pm.Deterministic("Delta", subtract(stochastic_1, stochastic_2))
```

Is perfectly valid PyMC3 code. Saying that our expressions behave like NumPy arrays is not exactly honest here, however. The main catch is that the expression that we are making *must* be compatible with `theano` tensors, which we will cover in the next section. Feel free to define whatever functions that you need in order to compose your model. However, if you need to do any array-like calculations that would require NumPy functions, make sure you use their equivalents in `theano`.

1.1.3 Theano

The majority of the heavy lifting done by PyMC3 is taken care of with the `theano` package. The notation in `theano` is remarkably similar to NumPy. It also supports many of the familiar computational elements of NumPy. However, while NumPy directly executes computations, e.g. when you run `a + b`, `theano` instead builds up a "compute graph" that tracks that you want to perform the `+` operation on the elements `a` and `b`. Only when you `eval()` a `theano` expression does the computation take place (i.e. `theano` is lazy evaluated). Once the compute graph is built, we can perform all kinds of mathematical optimizations (e.g. simplifications), compute gradients via autodiff, compile the entire graph to C to run at machine speed, and also compile it to run on the GPU. PyMC3 is basically a collection of `theano` symbolic expressions for various probability distributions that are combined to one big compute graph making up the whole model log probability, and a collection of inference algorithms that use that graph to compute probabilities and gradients. For practical purposes, what this means is that in order to build certain models we sometimes have to use `theano`.

Let's write some PyMC3 code that involves `theano` calculations.

```
In [10]: import theano.tensor as tt

         with pm.Model() as theano_test:
             p1 = pm.Uniform("p", 0, 1)
             p2 = 1 - p1
             p = tt.stack([p1, p2])

             assignment = pm.Categorical("assignment", p)
```

```
Applied interval-transform to p and added transformed p_interval_ to model.
```

Here we use `theano`'s `stack()` function in the same way we would use one of NumPy's stacking functions: to combine our two separate variables, `p1` and `p2`, into a vector with 2 elements. The stochastic `categorical` variable does not understand what we mean if we pass a NumPy array of `p1` and `p2` to it because they are both `theano` variables. Stacking them like this combines them

into one `theano` variable that we can use as the complementary pair of probabilities for our two categories.

Throughout the course of this book we use several `theano` functions to help construct our models. If you have more interest in looking at `theano` itself, be sure to check out the documentation.

After these technical considerations, we can get back to defining our model!

1.1.4 Including observations in the Model

At this point, it may not look like it, but we have fully specified our priors. For example, we can ask and answer questions like "What does my prior distribution of λ_1 look like?"

```
In [11]: %matplotlib inline
         from IPython.core.pylabtools import figsize
         import matplotlib.pyplot as plt
         import scipy.stats as stats
         figsize(12.5, 4)


         samples = [lambda_1.random()[0] for i in range(20000)]
         plt.hist(samples, bins=70, normed=True, histtype="stepfilled")
         plt.title("Prior distribution for $\lambda_1$")
         plt.xlim(0, 8);
```

To frame this in the notation of the first chapter, though this is a slight abuse of notation, we have specified $P(A)$. Our next goal is to include data/evidence/observations X into our model.

PyMC3 stochastic variables have a keyword argument `observed`. The keyword `observed` has a very simple role: fix the variable's current value to be the given data, typically a NumPy `array` or pandas `DataFrame`. For example:

```
In [12]: data = np.array([10, 5])
         with model:
             fixed_variable = pm.Poisson("fxd", 1, observed=data)
         print("value: ", fixed_variable.tag.test_value)
```

```
value:  [10  5]
```

This is how we include data into our models: initializing a stochastic variable to have a *fixed value*.

To complete our text message example, we fix the PyMC3 variable `observations` to the observed dataset.

```
In [13]: # We're using some fake data here
         data = np.array([10, 25, 15, 20, 35])
         with model:
             obs = pm.Poisson("obs", lambda_, observed=data)
         print(obs.tag.test_value)
```

```
[10 25 15 20 35]
```

1.2 Modeling approaches

A good starting thought to Bayesian modeling is to think about *how your data might have been generated*. Position yourself in an omniscient position, and try to imagine how *you* would recreate the dataset.

In the last chapter we investigated text message data. We begin by asking how our observations may have been generated:

1. We started by thinking "what is the best random variable to describe this count data?" A Poisson random variable is a good candidate because it can represent count data. So we model the number of sms's received as sampled from a Poisson distribution.

2. Next, we think, "Ok, assuming sms's are Poisson-distributed, what do I need for the Poisson distribution?" Well, the Poisson distribution has a parameter λ.

3. Do we know λ? No. In fact, we have a suspicion that there are *two* λ values, one for the earlier behaviour and one for the later behaviour. We don't know when the behaviour switches though, but call the switchpoint τ.

4. What is a good distribution for the two λs? The exponential is good, as it assigns probabilities to positive real numbers. Well the exponential distribution has a parameter too, call it α.

5. Do we know what the parameter α might be? No. At this point, we could continue and assign a distribution to α, but it's better to stop once we reach a set level of ignorance: whereas we have a prior belief about λ, ("it probably changes over time", "it's likely between 10 and 30", etc.), we don't really have any strong beliefs about α. So it's best to stop here.

   What is a good value for α then? We think that the λs are between 10-30, so if we set α really low (which corresponds to larger probability on high values) we are not reflecting our prior well. Similar, a too-high alpha misses our prior belief as well. A good idea for α as to reflect our belief is to set the value so that the mean of λ, given α, is equal to our observed mean. This was shown in the last chapter.

6. We have no expert opinion of when τ might have occurred. So we will suppose τ is from a discrete uniform distribution over the entire timespan.

Below we give a graphical visualization of this, where arrows denote `parent-child` relationships. (provided by the Daft Python library)

PyMC3, and other probabilistic programming languages, have been designed to tell these data-generation *stories*. More generally, B. Cronin writes [5]:

> Probabilistic programming will unlock narrative explanations of data, one of the holy grails of business analytics and the unsung hero of scientific persuasion. People think in terms of stories - thus the unreasonable power of the anecdote to drive decision-making, well-founded or not. But existing analytics largely fails to provide this kind of story; instead, numbers seemingly appear out of thin air, with little of the causal context that humans prefer when weighing their options.

1.2.1 Same story; different ending.

Interestingly, we can create *new datasets* by retelling the story. For example, if we reverse the above steps, we can simulate a possible realization of the dataset.

1. Specify when the user's behaviour switches by sampling from DiscreteUniform$(0, 80)$:

```
In [14]: tau = np.random.randint(0, 80)
         print(tau)
```

```
59
```

2. Draw λ_1 and λ_2 from an Exp(α) distribution:

```
In [15]: alpha = 1./20.
         lambda_1, lambda_2 = np.random.exponential(scale=1/alpha, size=2)
         print(lambda_1, lambda_2)
```

```
49.7521280843 10.1226712418
```

3. For days before τ, represent the user's received SMS count by sampling from Poi(λ_1), and sample from Poi(λ_2) for days after τ. For example:

```
In [16]: data = np.r_[stats.poisson.rvs(mu=lambda_1, size=tau), stats.poisson.rvs(mu=lambda_2,
```

4. Plot the artificial dataset:

```
In [17]: plt.bar(np.arange(80), data, color="#348ABD")
         plt.bar(tau-1, data[tau - 1], color="r", label="user behaviour changed")
         plt.xlabel("Time (days)")
         plt.ylabel("count of text-msgs received")
         plt.title("Artificial dataset")
         plt.xlim(0, 80)
         plt.legend();
```

It is okay that our fictional dataset does not look like our observed dataset: the probability is incredibly small it indeed would. PyMC3's engine is designed to find good parameters, λ_i, τ, that maximize this probability.

The ability to generate artificial dataset is an interesting side effect of our modeling, and we will see that this ability is a very important method of Bayesian inference. We produce a few more datasets below:

```
In [18]: def plot_artificial_sms_dataset():
             tau = stats.randint.rvs(0, 80)
             alpha = 1./20.
             lambda_1, lambda_2 = stats.expon.rvs(scale=1/alpha, size=2)
             data = np.r_[stats.poisson.rvs(mu=lambda_1, size=tau), stats.poisson.rvs(mu=lambda
             plt.bar(np.arange(80), data, color="#348ABD")
             plt.bar(tau - 1, data[tau-1], color="r", label="user behaviour changed")
             plt.xlim(0, 80);

         figsize(12.5, 5)
         plt.title("More example of artificial datasets")
         for i in range(4):
             plt.subplot(4, 1, i+1)
             plot_artificial_sms_dataset()
```

Later we will see how we use this to make predictions and test the appropriateness of our models.

Example: Bayesian A/B testing A/B testing is a statistical design pattern for determining the difference of effectiveness between two different treatments. For example, a pharmaceutical company is interested in the effectiveness of drug A vs drug B. The company will test drug A on some fraction of their trials, and drug B on the other fraction (this fraction is often 1/2, but we will relax this assumption). After performing enough trials, the in-house statisticians sift through the data to determine which drug yielded better results.

Similarly, front-end web developers are interested in which design of their website yields more sales or some other metric of interest. They will route some fraction of visitors to site A, and the other fraction to site B, and record if the visit yielded a sale or not. The data is recorded (in real-time), and analyzed afterwards.

Often, the post-experiment analysis is done using something called a hypothesis test like *difference of means test* or *difference of proportions test*. This involves often misunderstood quantities like a "Z-score" and even more confusing "p-values" (please don't ask). If you have taken a statistics course, you have probably been taught this technique (though not necessarily *learned* this technique). And if you were like me, you may have felt uncomfortable with their derivation -- good: the Bayesian approach to this problem is much more natural.

1.2.2 A Simple Case

As this is a hacker book, we'll continue with the web-dev example. For the moment, we will focus on the analysis of site A only. Assume that there is some true $0 < p_A < 1$ probability that users who, upon shown site A, eventually purchase from the site. This is the true effectiveness of site A. Currently, this quantity is unknown to us.

Suppose site A was shown to N people, and n people purchased from the site. One might conclude hastily that $p_A = \frac{n}{N}$. Unfortunately, the *observed frequency* $\frac{n}{N}$ does not necessarily equal p_A -- there is a difference between the *observed frequency* and the *true frequency* of an event. The true frequency can be interpreted as the probability of an event occurring. For example, the true frequency of rolling a 1 on a 6-sided die is $\frac{1}{6}$. Knowing the true frequency of events like:

- fraction of users who make purchases,
- frequency of social attributes,
- percent of internet users with cats etc.

are common requests we ask of Nature. Unfortunately, often Nature hides the true frequency from us and we must *infer* it from observed data.

The *observed frequency* is then the frequency we observe: say rolling the die 100 times you may observe 20 rolls of 1. The observed frequency, 0.2, differs from the true frequency, $\frac{1}{6}$. We can use Bayesian statistics to infer probable values of the true frequency using an appropriate prior and observed data.

With respect to our A/B example, we are interested in using what we know, N (the total trials administered) and n (the number of conversions), to estimate what p_A, the true frequency of buyers, might be.

To setup a Bayesian model, we need to assign prior distributions to our unknown quantities. *A priori*, what do we think p_A might be? For this example, we have no strong conviction about p_A, so for now, let's assume p_A is uniform over [0,1]:

```
In [19]: import pymc3 as pm

         # The parameters are the bounds of the Uniform.
         with pm.Model() as model:
             p = pm.Uniform('p', lower=0, upper=1)
```

```
Applied interval-transform to p and added transformed p_interval_ to model.
```

Had we had stronger beliefs, we could have expressed them in the prior above.

For this example, consider $p_A = 0.05$, and $N = 1500$ users shown site A, and we will simulate whether the user made a purchase or not. To simulate this from N trials, we will use a *Bernoulli* distribution: if $X \sim \text{Ber}(p)$, then X is 1 with probability p and 0 with probability $1 - p$. Of course, in practice we do not know p_A, but we will use it here to simulate the data.

```
In [20]: #set constants
         p_true = 0.05  # remember, this is unknown.
         N = 1500

         # sample N Bernoulli random variables from Ber(0.05).
         # each random variable has a 0.05 chance of being a 1.
         # this is the data-generation step
         occurrences = stats.bernoulli.rvs(p_true, size=N)

         print(occurrences) # Remember: Python treats True == 1, and False == 0
         print(np.sum(occurrences))
```

```
[1 0 1 ..., 0 0 0]
77
```

The observed frequency is:

```
In [21]: # Occurrences.mean is equal to n/N.
         print("What is the observed frequency in Group A? %.4f" % np.mean(occurrences))
         print("Does this equal the true frequency? %s" % (np.mean(occurrences) == p_true))
```

```
What is the observed frequency in Group A? 0.0513
Does this equal the true frequency? False
```

We combine the observations into the PyMC3 `observed` variable, and run our inference algorithm:

```
In [22]: #include the observations, which are Bernoulli
         with model:
             obs = pm.Bernoulli("obs", p, observed=occurrences)
             # To be explained in chapter 3
             step = pm.Metropolis()
             trace = pm.sample(18000, step=step)
             burned_trace = trace[1000:]
```

```
[-------100%-------] 18000 of 18000 in 1.7 sec. | SPS: 10329.7 | ETA: 0.0
```

We plot the posterior distribution of the unknown p_A below:

```
In [23]: figsize(12.5, 4)
         plt.title("Posterior distribution of $p_A$, the true effectiveness of site A")
         plt.vlines(p_true, 0, 90, linestyle="--", label="true $p_A$ (unknown)")
         plt.hist(burned_trace["p"], bins=25, histtype="stepfilled", normed=True)
         plt.legend();
```

Our posterior distribution puts most weight near the true value of p_A, but also some weights in the tails. This is a measure of how uncertain we should be, given our observations. Try changing the number of observations, `N`, and observe how the posterior distribution changes.

1.2.3 *A* and *B* Together

A similar anaylsis can be done for site B's response data to determine the analogous p_B. But what we are really interested in is the *difference* between p_A and p_B. Let's infer p_A, p_B, *and* delta $= p_A - p_B$, all at once. We can do this using PyMC3's deterministic variables. (We'll assume for this exercise that $p_B = 0.04$, so delta $= 0.01$, $N_B = 750$ (signifcantly less than N_A) and we will simulate site B's data like we did for site A's data)

In [24]:
```
import pymc3 as pm
figsize(12, 4)

#these two quantities are unknown to us.
true_p_A = 0.05
true_p_B = 0.04

#notice the unequal sample sizes -- no problem in Bayesian analysis.
N_A = 1500
N_B = 750

#generate some observations
observations_A = stats.bernoulli.rvs(true_p_A, size=N_A)
observations_B = stats.bernoulli.rvs(true_p_B, size=N_B)
print("Obs from Site A: ", observations_A[:30], "...")
print("Obs from Site B: ", observations_B[:30], "...")
```

```
Obs from Site A:  [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0] ...
Obs from Site B:  [0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0] ...
```

In [25]:
```
print(np.mean(observations_A))
print(np.mean(observations_B))
```

```
0.042
0.0346666666667
```

In [26]:
```
# Set up the pymc3 model. Again assume Uniform priors for p_A and p_B.
with pm.Model() as model:
    p_A = pm.Uniform("p_A", 0, 1)
    p_B = pm.Uniform("p_B", 0, 1)

    # Define the deterministic delta function. This is our unknown of interest.
    delta = pm.Deterministic("delta", p_A - p_B)


    # Set of observations, in this case we have two observation datasets.
    obs_A = pm.Bernoulli("obs_A", p_A, observed=observations_A)
    obs_B = pm.Bernoulli("obs_B", p_B, observed=observations_B)
```

```
    # To be explained in chapter 3.
    step = pm.Metropolis()
    trace = pm.sample(20000, step=step)
    burned_trace=trace[1000:]
```

```
Applied interval-transform to p_A and added transformed p_A_interval_ to model.
Applied interval-transform to p_B and added transformed p_B_interval_ to model.
 [-------100%-------] 20000 of 20000 in 3.2 sec. | SPS: 6201.6 | ETA: 0.0
```

Below we plot the posterior distributions for the three unknowns:

```
In [27]: p_A_samples = burned_trace["p_A"]
         p_B_samples = burned_trace["p_B"]
         delta_samples = burned_trace["delta"]
```

```
In [28]: figsize(12.5, 10)

         #histogram of posteriors

         ax = plt.subplot(311)

         plt.xlim(0, .1)
         plt.hist(p_A_samples, histtype='stepfilled', bins=25, alpha=0.85,
                  label="posterior of $p_A$", color="#A60628", normed=True)
         plt.vlines(true_p_A, 0, 80, linestyle="--", label="true $p_A$ (unknown)")
         plt.legend(loc="upper right")
         plt.title("Posterior distributions of $p_A$, $p_B$, and delta unknowns")

         ax = plt.subplot(312)

         plt.xlim(0, .1)
         plt.hist(p_B_samples, histtype='stepfilled', bins=25, alpha=0.85,
                  label="posterior of $p_B$", color="#467821", normed=True)
         plt.vlines(true_p_B, 0, 80, linestyle="--", label="true $p_B$ (unknown)")
         plt.legend(loc="upper right")

         ax = plt.subplot(313)
         plt.hist(delta_samples, histtype='stepfilled', bins=30, alpha=0.85,
                  label="posterior of delta", color="#7A68A6", normed=True)
         plt.vlines(true_p_A - true_p_B, 0, 60, linestyle="--",
                    label="true delta (unknown)")
         plt.vlines(0, 0, 60, color="black", alpha=0.2)
         plt.legend(loc="upper right");
```

Notice that as a result of `N_B < N_A`, i.e. we have less data from site B, our posterior distribution of p_B is fatter, implying we are less certain about the true value of p_B than we are of p_A.

With respect to the posterior distribution of delta, we can see that the majority of the distribution is above delta $= 0$, implying there site A's response is likely better than site B's response. The probability this inference is incorrect is easily computable:

```
In [29]: # Count the number of samples less than 0, i.e. the area under the curve
         # before 0, represent the probability that site A is worse than site B.
         print("Probability site A is WORSE than site B: %.3f" % \
             np.mean(delta_samples < 0))

         print("Probability site A is BETTER than site B: %.3f" % \
             np.mean(delta_samples > 0))
```

```
Probability site A is WORSE than site B: 0.208
Probability site A is BETTER than site B: 0.792
```

If this probability is too high for comfortable decision-making, we can perform more trials on site B (as site B has less samples to begin with, each additional data point for site B contributes more inferential "power" than each additional data point for site A).

Try playing with the parameters `true_p_A`, `true_p_B`, `N_A`, and `N_B`, to see what the posterior of delta looks like. Notice in all this, the difference in sample sizes between site A and site B was never mentioned: it naturally fits into Bayesian analysis.

I hope the readers feel this style of A/B testing is more natural than hypothesis testing, which has probably confused more than helped practitioners. Later in this book, we will see two extensions of this model: the first to help dynamically adjust for bad sites, and the second will improve the speed of this computation by reducing the analysis to a single equation.

1.3 An algorithm for human deceit

Social data has an additional layer of interest as people are not always honest with responses, which adds a further complication into inference. For example, simply asking individuals "Have you ever cheated on a test?" will surely contain some rate of dishonesty. What you can say for certain is that the true rate is less than your observed rate (assuming individuals lie *only* about *not cheating*; I cannot imagine one who would admit "Yes" to cheating when in fact they hadn't cheated).

To present an elegant solution to circumventing this dishonesty problem, and to demonstrate Bayesian modeling, we first need to introduce the binomial distribution.

1.3.1 The Binomial Distribution

The binomial distribution is one of the most popular distributions, mostly because of its simplicity and usefulness. Unlike the other distributions we have encountered thus far in the book, the binomial distribution has 2 parameters: N, a positive integer representing N trials or number of instances of potential events, and p, the probability of an event occurring in a single trial. Like the Poisson distribution, it is a discrete distribution, but unlike the Poisson distribution, it only weighs integers from 0 to N. The mass distribution looks like:

$$P(X = k) = \binom{N}{k} p^k (1-p)^{N-k}$$

If X is a binomial random variable with parameters p and N, denoted $X \sim \text{Bin}(N, p)$, then X is the number of events that occurred in the N trials (obviously $0 \leq X \leq N$). The larger p is (while still remaining between 0 and 1), the more events are likely to occur. The expected value of a binomial is equal to Np. Below we plot the mass probability distribution for varying parameters.

```
In [30]: figsize(12.5, 4)

         import scipy.stats as stats
         binomial = stats.binom

         parameters = [(10, .4), (10, .9)]
         colors = ["#348ABD", "#A60628"]

         for i in range(2):
             N, p = parameters[i]
             _x = np.arange(N + 1)
             plt.bar(_x - 0.5, binomial.pmf(_x, N, p), color=colors[i],
                     edgecolor=colors[i],
```

```
                alpha=0.6,
                label="$N$: %d, $p$: %.1f" % (N, p),
                linewidth=3)

    plt.legend(loc="upper left")
    plt.xlim(0, 10.5)
    plt.xlabel("$k$")
    plt.ylabel("$P(X = k)$")
    plt.title("Probability mass distributions of binomial random variables");
```

The special case when $N = 1$ corresponds to the Bernoulli distribution. There is another connection between Bernoulli and Binomial random variables. If we have $X_1, X_2, ..., X_N$ Bernoulli random variables with the same p, then $Z = X_1 + X_2 + ... + X_N \sim \text{Binomial}(N, p)$.

The expected value of a Bernoulli random variable is p. This can be seen by noting the more general Binomial random variable has expected value Np and setting $N = 1$.

Example: Cheating among students We will use the binomial distribution to determine the frequency of students cheating during an exam. If we let N be the total number of students who took the exam, and assuming each student is interviewed post-exam (answering without consequence), we will receive integer X "Yes I did cheat" answers. We then find the posterior distribution of p, given N, some specified prior on p, and observed data X.

This is a completely absurd model. No student, even with a free-pass against punishment, would admit to cheating. What we need is a better *algorithm* to ask students if they had cheated. Ideally the algorithm should encourage individuals to be honest while preserving privacy. The following proposed algorithm is a solution I greatly admire for its ingenuity and effectiveness:

> In the interview process for each student, the student flips a coin, hidden from the interviewer. The student agrees to answer honestly if the coin comes up heads. Otherwise, if the coin comes up tails, the student (secretly) flips the coin again, and answers "Yes, I did cheat" if the coin flip lands heads, and "No, I did not cheat", if the coin flip lands tails. This way, the interviewer does not know if a "Yes" was the result of a guilty plea, or a Heads on a second coin toss. Thus privacy is preserved and the researchers receive honest answers.

I call this the Privacy Algorithm. One could of course argue that the interviewers are still receiving false data since some *Yes*'s are not confessions but instead randomness, but an alternative perspective is that the researchers are discarding approximately half of their original dataset since half of the responses will be noise. But they have gained a systematic data generation process that can be modeled. Furthermore, they do not have to incorporate (perhaps somewhat naively) the possibility of deceitful answers. We can use PyMC3 to dig through this noisy model, and find a posterior distribution for the true frequency of liars.

Suppose 100 students are being surveyed for cheating, and we wish to find p, the proportion of cheaters. There are a few ways we can model this in PyMC3. I'll demonstrate the most explicit way, and later show a simplified version. Both versions arrive at the same inference. In our data-generation model, we sample p, the true proportion of cheaters, from a prior. Since we are quite ignorant about p, we will assign it a Uniform$(0, 1)$ prior.

```
In [31]: import pymc3 as pm

         N = 100
         with pm.Model() as model:
             p = pm.Uniform("freq_cheating", 0, 1)
```

```
Applied interval-transform to freq_cheating and added transformed freq_cheating_interval_ to mo
```

Again, thinking of our data-generation model, we assign Bernoulli random variables to the 100 students: 1 implies they cheated and 0 implies they did not.

```
In [32]: with model:
             true_answers = pm.Bernoulli("truths", p, shape=N, testval=np.random.binomial(1, 0
```

If we carry out the algorithm, the next step that occurs is the first coin-flip each student makes. This can be modeled again by sampling 100 Bernoulli random variables with $p = 1/2$: denote a 1 as a *Heads* and 0 a *Tails*.

```
In [33]: with model:
             first_coin_flips = pm.Bernoulli("first_flips", 0.5, shape=N, testval=np.random.bin
         print(first_coin_flips.tag.test_value)
```

```
[0 0 1 0 1 1 0 1 0 1 1 1 0 0 0 1 1 1 0 0 0 0 0 1 0 1 0 1 1 1 0 0 0 1 0 1 1
 1 1 0 1 0 0 1 1 1 1 0 0 0 0 0 0 1 1 0 0 1 1 0 1 0 0 1 0 1 1 0 0 0 0 0 1 1
 1 0 1 0 1 1 1 0 1 0 0 0 1 0 0 0 0 1 0 1 0 1 0 0 1 0]
```

Although *not everyone* flips a second time, we can still model the possible realization of second coin-flips:

```
In [34]: with model:
             second_coin_flips = pm.Bernoulli("second_flips", 0.5, shape=N, testval=np.random.b
```

Using these variables, we can return a possible realization of the *observed proportion* of "Yes" responses. We do this using a PyMC3 `deterministic` variable:

```
In [35]: import theano.tensor as tt
         with model:
             val = first_coin_flips*true_answers + (1 - first_coin_flips)*second_coin_flips
             observed_proportion = pm.Deterministic("observed_proportion", tt.sum(val)/float(N
```

The line `fc*t_a + (1-fc)*sc` contains the heart of the Privacy algorithm. Elements in this array are 1 *if and only if* i) the first toss is heads and the student cheated or ii) the first toss is tails, and the second is heads, and are 0 else. Finally, the last line sums this vector and divides by `float(N)`, produces a proportion.

```
In [36]: observed_proportion.tag.test_value

Out[36]: array(0.5600000023841858)
```

Next we need a dataset. After performing our coin-flipped interviews the researchers received 35 "Yes" responses. To put this into a relative perspective, if there truly were no cheaters, we should expect to see on average 1/4 of all responses being a "Yes" (half chance of having first coin land Tails, and another half chance of having second coin land Heads), so about 25 responses in a cheat-free world. On the other hand, if *all students cheated*, we should expected to see approximately 3/4 of all responses be "Yes".

The researchers observe a Binomial random variable, with `N = 100` and `p = observed_proportion` with `value = 35`:

```
In [37]: X = 35

         with model:
             observations = pm.Binomial("obs", N, observed_proportion, observed=X)
```

Below we add all the variables of interest to a `Model` container and run our black-box algorithm over the model.

```
In [38]: # To be explained in Chapter 3!
         with model:
             step = pm.Metropolis(vars=[p])
             trace = pm.sample(40000, step=step)
             burned_trace = trace[15000:]

Assigned BinaryGibbsMetropolis to truths
Assigned BinaryGibbsMetropolis to first_flips
Assigned BinaryGibbsMetropolis to second_flips
 [-------100%-------] 40000 of 40000 in 1891.9 sec. | SPS: 21.1 | ETA: -0.0

In [39]: figsize(12.5, 3)
         p_trace = burned_trace["freq_cheating"][15000:]
         plt.hist(p_trace, histtype="stepfilled", normed=True, alpha=0.85, bins=30,
                  label="posterior distribution", color="#348ABD")
         plt.vlines([.05, .35], [0, 0], [5, 5], alpha=0.3)
         plt.xlim(0, 1)
         plt.legend();
```

With regards to the above plot, we are still pretty uncertain about what the true frequency of cheaters might be, but we have narrowed it down to a range between 0.05 to 0.35 (marked by the solid lines). This is pretty good, as *a priori* we had no idea how many students might have cheated (hence the uniform distribution for our prior). On the other hand, it is also pretty bad since there is a .3 length window the true value most likely lives in. Have we even gained anything, or are we still too uncertain about the true frequency?

I would argue, yes, we have discovered something. It is implausible, according to our posterior, that there are *no cheaters*, i.e. the posterior assigns low probability to $p = 0$. Since we started with an uniform prior, treating all values of p as equally plausible, but the data ruled out $p = 0$ as a possibility, we can be confident that there were cheaters.

This kind of algorithm can be used to gather private information from users and be *reasonably* confident that the data, though noisy, is truthful.

1.3.2 Alternative PyMC3 Model

Given a value for p (which from our god-like position we know), we can find the probability the student will answer yes:

$$P(\text{"Yes"}) = P(\text{Heads on first coin})P(\text{cheater}) + P(\text{Tails on first coin})P(\text{Heads on second coin}) \tag{1}$$

$$\tag{2}$$

$$= \frac{1}{2}p + \frac{1}{2}\frac{1}{2} \tag{3}$$

$$\tag{4}$$

$$= \frac{p}{2} + \frac{1}{4} \tag{5}$$

Thus, knowing p we know the probability a student will respond "Yes". In PyMC3, we can create a deterministic function to evaluate the probability of responding "Yes", given p:

```
In [40]: with pm.Model() as model:
             p = pm.Uniform("freq_cheating", 0, 1)
             p_skewed = pm.Deterministic("p_skewed", 0.5*p + 0.25)
```

```
Applied interval-transform to freq_cheating and added transformed freq_cheating_interval_ to mo
```

I could have typed `p_skewed = 0.5*p + 0.25` instead for a one-liner, as the elementary operations of addition and scalar multiplication will implicitly create a `deterministic` variable, but I wanted to make the deterministic boilerplate explicit for clarity's sake.

If we know the probability of respondents saying "Yes", which is `p_skewed`, and we have $N = 100$ students, the number of "Yes" responses is a binomial random variable with parameters `N` and `p_skewed`.

This is where we include our observed 35 "Yes" responses. In the declaration of the `pm.Binomial`, we include `value = 35` and `observed = True`.

```
In [41]: with model:
             yes_responses = pm.Binomial("number_cheaters", 100, p_skewed, observed=35)
```

Below we add all the variables of interest to a `Model` container and run our black-box algorithm over the model.

```
In [42]: with model:
             # To Be Explained in Chapter 3!
             step = pm.Metropolis()
             trace = pm.sample(25000, step=step)
             burned_trace = trace[2500:]
```

```
 [-------100%-------] 25000 of 25000 in 2.1 sec. | SPS: 12171.2 | ETA: 0.0
```

```
In [43]: figsize(12.5, 3)
         p_trace = burned_trace["freq_cheating"]
         plt.hist(p_trace, histtype="stepfilled", normed=True, alpha=0.85, bins=30,
                  label="posterior distribution", color="#348ABD")
         plt.vlines([.05, .35], [0, 0], [5, 5], alpha=0.2)
         plt.xlim(0, 1)
         plt.legend();
```

1.3.3 More PyMC3 Tricks

Protip: Arrays of PyMC3 variables There is no reason why we cannot store multiple heterogeneous PyMC3 variables in a Numpy array. Just remember to set the `dtype` of the array to `object` upon initialization. For example:

```
In [44]: N = 10
         x = np.ones(N, dtype=object)
         with pm.Model() as model:
             for i in range(0, N):
                 x[i] = pm.Exponential('x_%i' % i, (i+1)**2)
```

```
Applied log-transform to x_0 and added transformed x_0_log_ to model.
Applied log-transform to x_1 and added transformed x_1_log_ to model.
Applied log-transform to x_2 and added transformed x_2_log_ to model.
Applied log-transform to x_3 and added transformed x_3_log_ to model.
Applied log-transform to x_4 and added transformed x_4_log_ to model.
Applied log-transform to x_5 and added transformed x_5_log_ to model.
Applied log-transform to x_6 and added transformed x_6_log_ to model.
Applied log-transform to x_7 and added transformed x_7_log_ to model.
Applied log-transform to x_8 and added transformed x_8_log_ to model.
Applied log-transform to x_9 and added transformed x_9_log_ to model.
```

The remainder of this chapter examines some practical examples of PyMC3 and PyMC3 modeling:

Example: Challenger Space Shuttle Disaster On January 28, 1986, the twenty-fifth flight of the U.S. space shuttle program ended in disaster when one of the rocket boosters of the Shuttle Challenger exploded shortly after lift-off, killing all seven crew members. The presidential commission on the accident concluded that it was caused by the failure of an O-ring in a field joint on the rocket booster, and that this failure was due to a faulty design that made the O-ring unacceptably sensitive to a number of factors including outside temperature. Of the previous 24 flights, data were available on failures of O-rings on 23, (one was lost at sea), and these data were discussed on the evening preceding the Challenger launch, but unfortunately only the data corresponding to the 7 flights on which there was a damage incident were considered important and these were thought to show no obvious trend. The data are shown below (see [1]):

```
In [45]: figsize(12.5, 3.5)
         np.set_printoptions(precision=3, suppress=True)
         challenger_data = np.genfromtxt("data/challenger_data.csv", skip_header=1,
                                         usecols=[1, 2], missing_values="NA",
                                         delimiter=",")
         #drop the NA values
         challenger_data = challenger_data[~np.isnan(challenger_data[:, 1])]

         #plot it, as a function of tempature (the first column)
         print("Temp (F), O-Ring failure?")
         print(challenger_data)

         plt.scatter(challenger_data[:, 0], challenger_data[:, 1], s=75, color="k",
                     alpha=0.5)
         plt.yticks([0, 1])
         plt.ylabel("Damage Incident?")
```

```
plt.xlabel("Outside temperature (Fahrenheit)")
plt.title("Defects of the Space Shuttle O-Rings vs temperature");
```

```
Temp (F), O-Ring failure?
[[ 66.   0.]
 [ 70.   1.]
 [ 69.   0.]
 [ 68.   0.]
 [ 67.   0.]
 [ 72.   0.]
 [ 73.   0.]
 [ 70.   0.]
 [ 57.   1.]
 [ 63.   1.]
 [ 70.   1.]
 [ 78.   0.]
 [ 67.   0.]
 [ 53.   1.]
 [ 67.   0.]
 [ 75.   0.]
 [ 70.   0.]
 [ 81.   0.]
 [ 76.   0.]
 [ 79.   0.]
 [ 75.   1.]
 [ 76.   0.]
 [ 58.   1.]]
```

It looks clear that *the probability* of damage incidents occurring increases as the outside temperature decreases. We are interested in modeling the probability here because it does not look like there is a strict cutoff point between temperature and a damage incident occurring. The best we can do is ask "At temperature t, what is the probability of a damage incident?". The goal of this example is to answer that question.

We need a function of temperature, call it $p(t)$, that is bounded between 0 and 1 (so as to model a probability) and changes from 1 to 0 as we increase temperature. There are actually many such functions, but the most popular choice is the *logistic function.*

$$p(t) = \frac{1}{1 + e^{\beta t}}$$

In this model, β is the variable we are uncertain about. Below is the function plotted for $\beta = 1, 3, -5$.

```
In [46]: figsize(12, 3)

         def logistic(x, beta):
             return 1.0 / (1.0 + np.exp(beta * x))

         x = np.linspace(-4, 4, 100)
         plt.plot(x, logistic(x, 1), label=r"$\beta = 1$")
         plt.plot(x, logistic(x, 3), label=r"$\beta = 3$")
         plt.plot(x, logistic(x, -5), label=r"$\beta = -5$")
         plt.legend();
```

But something is missing. In the plot of the logistic function, the probability changes only near zero, but in our data above the probability changes around 65 to 70. We need to add a *bias* term to our logistic function:

$$p(t) = \frac{1}{1 + e^{\beta t + \alpha}}$$

Some plots are below, with differing α.

```
In [47]: def logistic(x, beta, alpha=0):
             return 1.0 / (1.0 + np.exp(np.dot(beta, x) + alpha))

         x = np.linspace(-4, 4, 100)

         plt.plot(x, logistic(x, 1), label=r"$\beta = 1$", ls="--", lw=1)
         plt.plot(x, logistic(x, 3), label=r"$\beta = 3$", ls="--", lw=1)
         plt.plot(x, logistic(x, -5), label=r"$\beta = -5$", ls="--", lw=1)
```

```
plt.plot(x, logistic(x, 1, 1), label=r"$\beta = 1, \alpha = 1$",
         color="#348ABD")
plt.plot(x, logistic(x, 3, -2), label=r"$\beta = 3, \alpha = -2$",
         color="#A60628")
plt.plot(x, logistic(x, -5, 7), label=r"$\beta = -5, \alpha = 7$",
         color="#7A68A6")

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

Adding a constant term α amounts to shifting the curve left or right (hence why it is called a *bias*).

Let's start modeling this in PyMC3. The β, α parameters have no reason to be positive, bounded or relatively large, so they are best modeled by a *Normal random variable*, introduced next.

1.3.4 Normal distributions

A Normal random variable, denoted $X \sim N(\mu, 1/\tau)$, has a distribution with two parameters: the mean, μ, and the *precision*, τ. Those familiar with the Normal distribution already have probably seen σ^2 instead of τ^{-1}. They are in fact reciprocals of each other. The change was motivated by simpler mathematical analysis and is an artifact of older Bayesian methods. Just remember: the smaller τ, the larger the spread of the distribution (i.e. we are more uncertain); the larger τ, the tighter the distribution (i.e. we are more certain). Regardless, τ is always positive.

The probability density function of a $N(\mu, 1/\tau)$ random variable is:

$$f(x|\mu, \tau) = \sqrt{\frac{\tau}{2\pi}} \exp\left(-\frac{\tau}{2}(x-\mu)^2\right)$$

We plot some different density functions below.

```
In [48]: import scipy.stats as stats

         nor = stats.norm
         x = np.linspace(-8, 7, 150)
         mu = (-2, 0, 3)
         tau = (.7, 1, 2.8)
         colors = ["#348ABD", "#A60628", "#7A68A6"]
```

```
parameters = zip(mu, tau, colors)

for _mu, _tau, _color in parameters:
    plt.plot(x, nor.pdf(x, _mu, scale=1./_tau),
             label="$\mu = %d,\;\\tau = %.1f$" % (_mu, _tau), color=_color)
    plt.fill_between(x, nor.pdf(x, _mu, scale=1./_tau), color=_color,
                     alpha=.33)

plt.legend(loc="upper right")
plt.xlabel("$x$")
plt.ylabel("density function at $x$")
plt.title("Probability distribution of three different Normal random \
variables");
```

A Normal random variable can be take on any real number, but the variable is very likely to be relatively close to μ. In fact, the expected value of a Normal is equal to its μ parameter:

$$E[X|\mu,\tau] = \mu$$

and its variance is equal to the inverse of τ:

$$Var(X|\mu,\tau) = \frac{1}{\tau}$$

Below we continue our modeling of the Challenger space craft:

```
In [49]: import pymc3 as pm

         temperature = challenger_data[:, 0]
         D = challenger_data[:, 1]  # defect or not?

         #notice the`value` here. We explain why below.
         with pm.Model() as model:
             beta = pm.Normal("beta", mu=0, tau=0.001, testval=0)
             alpha = pm.Normal("alpha", mu=0, tau=0.001, testval=0)
             p = pm.Deterministic("p", 1.0/(1. + tt.exp(beta*temperature + alpha)))
```

We have our probabilities, but how do we connect them to our observed data? A *Bernoulli* random variable with parameter p, denoted $\text{Ber}(p)$, is a random variable that takes value 1 with probability p, and 0 else. Thus, our model can look like:

$$\text{Defect Incident, } D_i \sim \text{Ber}(\ p(t_i)\ ),\ \ i = 1..N$$

where $p(t)$ is our logistic function and t_i are the temperatures we have observations about. Notice in the above code we had to set the values of `beta` and `alpha` to 0. The reason for this is that if `beta` and `alpha` are very large, they make `p` equal to 1 or 0. Unfortunately, `pm.Bernoulli` does not like probabilities of exactly 0 or 1, though they are mathematically well-defined probabilities. So by setting the coefficient values to `0`, we set the variable `p` to be a reasonable starting value. This has no effect on our results, nor does it mean we are including any additional information in our prior. It is simply a computational caveat in PyMC3.

```
In [50]: # connect the probabilities in `p` with our observations through a
         # Bernoulli random variable.
         with model:
             observed = pm.Bernoulli("bernoulli_obs", p, observed=D)

             # Mysterious code to be explained in Chapter 3
             start = pm.find_MAP()
             step = pm.Metropolis()
             trace = pm.sample(120000, step=step, start=start)
             burned_trace = trace[100000::2]
```

```
 [-------100%-------] 120000 of 120000 in 16.5 sec. | SPS: 7283.1 | ETA: 0.0
```

We have trained our model on the observed data, now we can sample values from the posterior. Let's look at the posterior distributions for α and β:

```
In [51]: alpha_samples = burned_trace["alpha"][:, None]  # best to make them 1d
         beta_samples = burned_trace["beta"][:, None]

         figsize(12.5, 6)

         #histogram of the samples:
         plt.subplot(211)
         plt.title(r"Posterior distributions of the variables $\alpha, \beta$")
         plt.hist(beta_samples, histtype='stepfilled', bins=35, alpha=0.85,
                  label=r"posterior of $\beta$", color="#7A68A6", normed=True)
         plt.legend()

         plt.subplot(212)
         plt.hist(alpha_samples, histtype='stepfilled', bins=35, alpha=0.85,
                  label=r"posterior of $\alpha$", color="#A60628", normed=True)
         plt.legend();
```

All samples of β are greater than 0. If instead the posterior was centered around 0, we may suspect that $\beta = 0$, implying that temperature has no effect on the probability of defect.

Similarly, all α posterior values are negative and far away from 0, implying that it is correct to believe that α is significantly less than 0.

Regarding the spread of the data, we are very uncertain about what the true parameters might be (though considering the low sample size and the large overlap of defects-to-nondefects this behaviour is perhaps expected).

Next, let's look at the *expected probability* for a specific value of the temperature. That is, we average over all samples from the posterior to get a likely value for $p(t_i)$.

```
In [52]: t = np.linspace(temperature.min() - 5, temperature.max()+5, 50)[:, None]
         p_t = logistic(t.T, beta_samples, alpha_samples)

         mean_prob_t = p_t.mean(axis=0)
```

```
In [53]: figsize(12.5, 4)

         plt.plot(t, mean_prob_t, lw=3, label="average posterior \nprobability \
         of defect")
         plt.plot(t, p_t[0, :], ls="--", label="realization from posterior")
         plt.plot(t, p_t[-2, :], ls="--", label="realization from posterior")
         plt.scatter(temperature, D, color="k", s=50, alpha=0.5)
         plt.title("Posterior expected value of probability of defect; \
         plus realizations")
         plt.legend(loc="lower left")
         plt.ylim(-0.1, 1.1)
         plt.xlim(t.min(), t.max())
         plt.ylabel("probability")
         plt.xlabel("temperature");
```

Above we also plotted two possible realizations of what the actual underlying system might be. Both are equally likely as any other draw. The blue line is what occurs when we average all the 20000 possible dotted lines together.

An interesting question to ask is for what temperatures are we most uncertain about the defect-probability? Below we plot the expected value line **and** the associated 95% intervals for each temperature.

```
In [54]: from scipy.stats.mstats import mquantiles

         # vectorized bottom and top 2.5% quantiles for "confidence interval"
         qs = mquantiles(p_t, [0.025, 0.975], axis=0)
         plt.fill_between(t[:, 0], *qs, alpha=0.7,
                          color="#7A68A6")

         plt.plot(t[:, 0], qs[0], label="95% CI", color="#7A68A6", alpha=0.7)

         plt.plot(t, mean_prob_t, lw=1, ls="--", color="k",
                  label="average posterior \nprobability of defect")

         plt.xlim(t.min(), t.max())
         plt.ylim(-0.02, 1.02)
         plt.legend(loc="lower left")
         plt.scatter(temperature, D, color="k", s=50, alpha=0.5)
         plt.xlabel("temp, $t$")

         plt.ylabel("probability estimate")
         plt.title("Posterior probability estimates given temp. $t$");
```

The *95% credible interval*, or 95% CI, painted in purple, represents the interval, for each temperature, that contains 95% of the distribution. For example, at 65 degrees, we can be 95% sure that the probability of defect lies between 0.25 and 0.75.

More generally, we can see that as the temperature nears 60 degrees, the CI's spread out over [0,1] quickly. As we pass 70 degrees, the CI's tighten again. This can give us insight about how to proceed next: we should probably test more O-rings around 60-65 temperature to get a better estimate of probabilities in that range. Similarly, when reporting to scientists your estimates, you should be very cautious about simply telling them the expected probability, as we can see this does not reflect how *wide* the posterior distribution is.

1.3.5 What about the day of the Challenger disaster?

On the day of the Challenger disaster, the outside temperature was 31 degrees Fahrenheit. What is the posterior distribution of a defect occurring, given this temperature? The distribution is plotted below. It looks almost guaranteed that the Challenger was going to be subject to defective O-rings.

```
In [55]: figsize(12.5, 2.5)

         prob_31 = logistic(31, beta_samples, alpha_samples)

         plt.xlim(0.995, 1)
         plt.hist(prob_31, bins=1000, normed=True, histtype='stepfilled')
         plt.title("Posterior distribution of probability of defect, given $t = 31$")
         plt.xlabel("probability of defect occurring in O-ring");
```

1.3.6 Is our model appropriate?

The skeptical reader will say "You deliberately chose the logistic function for $p(t)$ and the specific priors. Perhaps other functions or priors will give different results. How do I know I have chosen a good model?" This is absolutely true. To consider an extreme situation, what if I had chosen the function $p(t) = 1, \forall t$, which guarantees a defect always occurring: I would have again predicted disaster on January 28th. Yet this is clearly a poorly chosen model. On the other hand, if I did choose the logistic function for $p(t)$, but specified all my priors to be very tight around 0, likely we would have very different posterior distributions. How do we know our model is an expression of the data? This encourages us to measure the model's **goodness of fit**.

We can think: *how can we test whether our model is a bad fit?* An idea is to compare observed data (which if we recall is a *fixed* stochastic variable) with artificial dataset which we can simulate. The rationale is that if the simulated dataset does not appear similar, statistically, to the observed dataset, then likely our model is not accurately represented the observed data.

Previously in this Chapter, we simulated artificial dataset for the SMS example. To do this, we sampled values from the priors. We saw how varied the resulting datasets looked like, and rarely did they mimic our observed dataset. In the current example, we should sample from the *posterior* distributions to create *very plausible datasets*. Luckily, our Bayesian framework makes this very easy. We only need to create a new `Stochastic` variable, that is exactly the same as our variable that stored the observations, but minus the observations themselves. If you recall, our `Stochastic` variable that stored our observed data was:

```
observed = pm.Bernoulli("bernoulli_obs", p, observed=D)
```

Hence we create:

```
simulated_data = pm.Bernoulli("simulation_data", p)
```

Let's simulate 10 000:

```
In [56]: N = 10000
         with pm.Model() as model:
             beta = pm.Normal("beta", mu=0, tau=0.001, testval=0)
             alpha = pm.Normal("alpha", mu=0, tau=0.001, testval=0)
             p = pm.Deterministic("p", 1.0/(1. + tt.exp(beta*temperature + alpha)))
             observed = pm.Bernoulli("bernoulli_obs", p, observed=D)

             simulated = pm.Bernoulli("bernoulli_sim", p, shape=p.tag.test_value.shape)
             step = pm.Metropolis(vars=[p])
             trace = pm.sample(N, step=step)
```

```
Assigned BinaryGibbsMetropolis to bernoulli_sim
 [-------100%-------] 10000 of 10000 in 27.8 sec. | SPS: 359.1 | ETA: 0.0
```

```
In [57]: figsize(12.5, 5)
```

```
simulations = trace["bernoulli_sim"]
print(simulations.shape)

plt.title("Simulated dataset using posterior parameters")
figsize(12.5, 6)
for i in range(4):
    ax = plt.subplot(4, 1, i+1)
    plt.scatter(temperature, simulations[1000*i, :], color="k",
                s=50, alpha=0.6)
```

```
(10000, 23)
```

Note that the above plots are different (if you can think of a cleaner way to present this, please send a pull request and answer here!).

We wish to assess how good our model is. "Good" is a subjective term of course, so results must be relative to other models.

We will be doing this graphically as well, which may seem like an even less objective method. The alternative is to use *Bayesian p-values*. These are still subjective, as the proper cutoff between good and bad is arbitrary. Gelman emphasises that the graphical tests are more illuminating [7] than p-value tests. We agree.

The following graphical test is a novel data-viz approach to logistic regression. The plots are called *separation plots*[8]. For a suite of models we wish to compare, each model is plotted on an individual separation plot. I leave most of the technical details about separation plots to the very accessible original paper, but I'll summarize their use here.

For each model, we calculate the proportion of times the posterior simulation proposed a value of 1 for a particular temperature, i.e. compute $P(\text{ Defect } = 1|t, \alpha, \beta)$ by averaging. This gives us the posterior probability of a defect at each data point in our dataset. For example, for the model we used above:

```
In [58]: posterior_probability = simulations.mean(axis=0)
         print("posterior prob of defect | realized defect ")
```

```
for i in range(len(D)):
    print("%.2f                    |   %d" % (posterior_probability[i], D[i]))
```

```
posterior prob of defect | realized defect
0.40                     |   0
0.25                     |   1
0.28                     |   0
0.32                     |   0
0.36                     |   0
0.19                     |   0
0.17                     |   0
0.25                     |   0
0.73                     |   1
0.53                     |   1
0.25                     |   1
0.10                     |   0
0.36                     |   0
0.80                     |   1
0.36                     |   0
0.13                     |   0
0.25                     |   0
0.07                     |   0
0.12                     |   0
0.09                     |   0
0.13                     |   1
0.12                     |   0
0.71                     |   1
```

Next we sort each column by the posterior probabilities:

```
In [59]: ix = np.argsort(posterior_probability)
         print("probb | defect ")
         for i in range(len(D)):
             print("%.2f  |   %d" % (posterior_probability[ix[i]], D[ix[i]]))
```

```
probb | defect
0.07  |   0
0.09  |   0
0.10  |   0
0.12  |   0
0.12  |   0
0.13  |   1
0.13  |   0
0.17  |   0
0.19  |   0
0.25  |   1
0.25  |   0
0.25  |   1
```

```
0.25  |   0
0.28  |   0
0.32  |   0
0.36  |   0
0.36  |   0
0.36  |   0
0.40  |   0
0.53  |   1
0.71  |   1
0.73  |   1
0.80  |   1
```

We can present the above data better in a figure: I've wrapped this up into a `separation_plot` function.

```
In [60]: from separation_plot import separation_plot


         figsize(11., 1.5)
         separation_plot(posterior_probability, D)
```

The snaking-line is the sorted probabilities, blue bars denote defects, and empty space (or grey bars for the optimistic readers) denote non-defects. As the probability rises, we see more and more defects occur. On the right hand side, the plot suggests that as the posterior probability is large (line close to 1), then more defects are realized. This is good behaviour. Ideally, all the blue bars *should* be close to the right-hand side, and deviations from this reflect missed predictions.

The black vertical line is the expected number of defects we should observe, given this model. This allows the user to see how the total number of events predicted by the model compares to the actual number of events in the data.

It is much more informative to compare this to separation plots for other models. Below we compare our model (top) versus three others:

1. the perfect model, which predicts the posterior probability to be equal 1 if a defect did occur.
2. a completely random model, which predicts random probabilities regardless of temperature.
3. a constant model: where $P(D = 1 \mid t) = c, \quad \forall t$. The best choice for c is the observed frequency of defects, in this case $7/23$.

```
In [61]: figsize(11., 1.25)

         # Our temperature-dependent model
         separation_plot(posterior_probability, D)
```

```
plt.title("Temperature-dependent model")

# Perfect model
# i.e. the probability of defect is equal to if a defect occurred or not.
p = D
separation_plot(p, D)
plt.title("Perfect model")

# random predictions
p = np.random.rand(23)
separation_plot(p, D)
plt.title("Random model")

# constant model
constant_prob = 7./23*np.ones(23)
separation_plot(constant_prob, D)
plt.title("Constant-prediction model");
```

In the random model, we can see that as the probability increases there is no clustering of defects to the right-hand side. Similarly for the constant model.

The perfect model, the probability line is not well shown, as it is stuck to the bottom and top of the figure. Of course the perfect model is only for demonstration, and we cannot infer any scientific inference from it.

Exercises 1. Try putting in extreme values for our observations in the cheating example. What happens if we observe 25 affirmative responses? 10? 50?

2. Try plotting α samples versus β samples. Why might the resulting plot look like this?

```
In [62]: #type your code here.
         figsize(12.5, 4)

         plt.scatter(alpha_samples, beta_samples, alpha=0.1)
         plt.title("Why does the plot look like this?")
         plt.xlabel(r"$\alpha$")
         plt.ylabel(r"$\beta$");
```

1.3.7 References

- [1] Dalal, Fowlkes and Hoadley (1989),JASA, 84, 945-957.
- [2] German Rodriguez. Datasets. In WWS509. Retrieved 30/01/2013, from http://data.princeton.edu/wws509/datasets/#smoking.
- [3] McLeish, Don, and Cyntha Struthers. STATISTICS 450/850 Estimation and Hypothesis Testing. Winter 2012. Waterloo, Ontario: 2012. Print.
- [4] Fonnesbeck, Christopher. "Building Models." PyMC-Devs. N.p., n.d. Web. 26 Feb 2013. http://pymc-devs.github.com/pymc/modelbuilding.html.
- [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.
- [6] S.P. Brooks, E.A. Catchpole, and B.J.T. Morgan. Bayesian animal survival estimation. Statistical Science, 15: 357–376, 2000

- [7] Gelman, Andrew. "Philosophy and the practice of Bayesian statistics." British Journal of Mathematical and Statistical Psychology. (2012): n. page. Web. 2 Apr. 2013.
- [8] Greenhill, Brian, Michael D. Ward, and Audrey Sacks. "The Separation Plot: A New Visual Method for Evaluating the Fit of Binary Models." American Journal of Political Science. 55.No.4 (2011): n. page. Web. 2 Apr. 2013.

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 object>
```

In [ ]:

Ch3_IntroMCMC_PyMC2

January 4, 2018

1 Chapter 3

1.1 Opening the black box of MCMC

The previous two chapters hid the inner-mechanics of PyMC, and more generally Markov Chain Monte Carlo (MCMC), from the reader. The reason for including this chapter is three-fold. The first is that any book on Bayesian inference must discuss MCMC. I cannot fight this. Blame the statisticians. Secondly, knowing the process of MCMC gives you insight into whether your algorithm has converged. (Converged to what? We will get to that) Thirdly, we'll understand *why* we are returned thousands of samples from the posterior as a solution, which at first thought can be odd.

1.1.1 The Bayesian landscape

When we setup a Bayesian inference problem with N unknowns, we are implicitly creating an N dimensional space for the prior distributions to exist in. Associated with the space is an additional dimension, which we can describe as the *surface*, or *curve*, that sits on top of the space, that reflects the *prior probability* of a particular point. The surface on the space is defined by our prior distributions. For example, if we have two unknowns p_1 and p_2, and priors for both are Uniform$(0, 5)$, the space created is a square of length 5 and the surface is a flat plane that sits on top of the square (representing that every point is equally likely).

```
In [1]: %matplotlib inline
        import scipy.stats as stats
        from IPython.core.pylabtools import figsize
        import numpy as np
        figsize(12.5, 4)

        import matplotlib.pyplot as plt
        from mpl_toolkits.mplot3d import Axes3D

        jet = plt.cm.jet
        fig = plt.figure()
        x = y = np.linspace(0, 5, 100)
        X, Y = np.meshgrid(x, y)
```

```
plt.subplot(121)
uni_x = stats.uniform.pdf(x, loc=0, scale=5)
uni_y = stats.uniform.pdf(y, loc=0, scale=5)
M = np.dot(uni_y[:, None], uni_x[None, :])
im = plt.imshow(M, interpolation='none', origin='lower',
                cmap=jet, vmax=1, vmin=-.15, extent=(0, 5, 0, 5))

plt.xlim(0, 5)
plt.ylim(0, 5)
plt.title("Landscape formed by Uniform priors.")

ax = fig.add_subplot(122, projection='3d')
ax.plot_surface(X, Y, M, cmap=plt.cm.jet, vmax=1, vmin=-.15)
ax.view_init(azim=390)
plt.title("Uniform prior landscape; alternate view");
```

Alternatively, if the two priors are Exp(3) and Exp(10), then the space is all positive numbers on the 2-D plane, and the surface induced by the priors looks like a water fall that starts at the point (0,0) and flows over the positive numbers.

The plots below visualize this. The more dark red the color, the more prior probability is assigned to that location. Conversely, areas with darker blue represent that our priors assign very low probability to that location.

```
In [2]: figsize(12.5, 5)
        fig = plt.figure()
        plt.subplot(121)

        exp_x = stats.expon.pdf(x, scale=3)
        exp_y = stats.expon.pdf(x, scale=10)
        M = np.dot(exp_y[:, None], exp_x[None, :])
        CS = plt.contour(X, Y, M)
        im = plt.imshow(M, interpolation='none', origin='lower',
```

```
                    cmap=jet, extent=(0, 5, 0, 5))
#plt.xlabel("prior on $p_1$")
#plt.ylabel("prior on $p_2$")
plt.title("$Exp(3), Exp(10)$ prior landscape")

ax = fig.add_subplot(122, projection='3d')
ax.plot_surface(X, Y, M, cmap=jet)
ax.view_init(azim=390)
plt.title("$Exp(3), Exp(10)$ prior landscape; \nalternate view");
```

These are simple examples in 2D space, where our brains can understand surfaces well. In practice, spaces and surfaces generated by our priors can be much higher dimensional.

If these surfaces describe our *prior distributions* on the unknowns, what happens to our space after we incorporate our observed data X? The data X does not change the space, but it changes the surface of the space by *pulling and stretching the fabric of the prior surface* to reflect where the true parameters likely live. More data means more pulling and stretching, and our original shape becomes mangled or insignificant compared to the newly formed shape. Less data, and our original shape is more present. Regardless, the resulting surface describes the *posterior distribution*.

Again I must stress that it is, unfortunately, impossible to visualize this in large dimensions. For two dimensions, the data essentially *pushes up* the original surface to make *tall mountains*. The tendency of the observed data to *push up* the posterior probability in certain areas is checked by the prior probability distribution, so that lower prior probability means more resistance. Thus in the double-exponential prior case above, a mountain (or multiple mountains) that might erupt near the (0,0) corner would be much higher than mountains that erupt closer to (5,5), since there is more resistance (low prior probability) near (5,5). The peak reflects the posterior probability of where the true parameters are likely to be found. Importantly, if the prior has assigned a probability of 0, then no posterior probability will be assigned there.

Suppose the priors mentioned above represent different parameters λ of two Poisson distributions. We observe a few data points and visualize the new landscape:

```
In [3]: # create the observed data
```

```
# sample size of data we observe, trying varying this (keep it less than 100 ;)
N = 1

# the true parameters, but of course we do not see these values...
lambda_1_true = 1
lambda_2_true = 3

#...we see the data generated, dependent on the above two values.
data = np.concatenate([
    stats.poisson.rvs(lambda_1_true, size=(N, 1)),
    stats.poisson.rvs(lambda_2_true, size=(N, 1))
], axis=1)
print("observed (2-dimensional,sample size = %d):" % N, data)

# plotting details.
x = y = np.linspace(.01, 5, 100)
likelihood_x = np.array([stats.poisson.pmf(data[:, 0], _x)
                         for _x in x]).prod(axis=1)
likelihood_y = np.array([stats.poisson.pmf(data[:, 1], _y)
                         for _y in y]).prod(axis=1)
L = np.dot(likelihood_x[:, None], likelihood_y[None, :])
```

```
observed (2-dimensional,sample size = 1): [[3 4]]
```

```
In [4]: figsize(12.5, 12)
        # matplotlib heavy lifting below, beware!
        plt.subplot(221)
        uni_x = stats.uniform.pdf(x, loc=0, scale=5)
        uni_y = stats.uniform.pdf(x, loc=0, scale=5)
        M = np.dot(uni_y[:, None], uni_x[None, :])
        im = plt.imshow(M, interpolation='none', origin='lower',
                        cmap=jet, vmax=1, vmin=-.15, extent=(0, 5, 0, 5))
        plt.scatter(lambda_2_true, lambda_1_true, c="k", s=50, edgecolor="none")
        plt.xlim(0, 5)
        plt.ylim(0, 5)
        plt.title("Landscape formed by Uniform priors on $p_1, p_2$.")

        plt.subplot(223)
        plt.contour(x, y, M * L)
        im = plt.imshow(M * L, interpolation='none', origin='lower',
                        cmap=jet, extent=(0, 5, 0, 5))
        plt.title("Landscape warped by %d data observation;\n Uniform priors on $p_1, p_2$." %
        plt.scatter(lambda_2_true, lambda_1_true, c="k", s=50, edgecolor="none")
        plt.xlim(0, 5)
        plt.ylim(0, 5)

        plt.subplot(222)
```

```
exp_x = stats.expon.pdf(x, loc=0, scale=3)
exp_y = stats.expon.pdf(x, loc=0, scale=10)
M = np.dot(exp_y[:, None], exp_x[None, :])

plt.contour(x, y, M)
im = plt.imshow(M, interpolation='none', origin='lower',
                cmap=jet, extent=(0, 5, 0, 5))
plt.scatter(lambda_2_true, lambda_1_true, c="k", s=50, edgecolor="none")
plt.xlim(0, 5)
plt.ylim(0, 5)
plt.title("Landscape formed by Exponential priors on $p_1, p_2$.")

plt.subplot(224)
# This is the likelihood times prior, that results in the posterior.
plt.contour(x, y, M * L)
im = plt.imshow(M * L, interpolation='none', origin='lower',
                cmap=jet, extent=(0, 5, 0, 5))

plt.scatter(lambda_2_true, lambda_1_true, c="k", s=50, edgecolor="none")
plt.title("Landscape warped by %d data observation;\n Exponential priors on \
$p_1, p_2$." % N)
plt.xlim(0, 5)
plt.ylim(0, 5)
```

Out[4]: (0, 5)

The plot on the left is the deformed landscape with the Uniform$(0, 5)$ priors, and the plot on the right is the deformed landscape with the exponential priors. Notice that the posterior landscapes look different from one another, though the data observed is identical in both cases. The reason is as follows. Notice the exponential-prior landscape, bottom right figure, puts very little *posterior* weight on values in the upper right corner of the figure: this is because *the prior does not put much weight there*. On the other hand, the uniform-prior landscape is happy to put posterior weight in the upper-right corner, as the prior puts more weight there.

Notice also the highest-point, corresponding to the darkest red, is biased towards (0,0) in the exponential case, which is the result from the exponential prior putting more prior weight in the (0,0) corner.

The black dot represents the true parameters. Even with 1 sample point, the mountains attempts to contain the true parameter. Of course, inference with a sample size of 1 is incredibly naive, and choosing such a small sample size was only illustrative.

It's a great exercise to try changing the sample size to other values (try 2,5,10,100?...) and observing how our "mountain" posterior changes.

1.1.2 Exploring the landscape using the MCMC

We should explore the deformed posterior space generated by our prior surface and observed data to find the posterior mountain. However, we cannot naively search the space: any computer scientist will tell you that traversing N-dimensional space is exponentially difficult in N: the size of the space quickly blows-up as we increase N (see the curse of dimensionality). What hope do we have to find these hidden mountains? The idea behind MCMC is to perform an intelligent search of the space. To say "search" implies we are looking for a particular point, which is perhaps not an accurate as we are really looking for a broad mountain.

Recall that MCMC returns *samples* from the posterior distribution, not the distribution itself. Stretching our mountainous analogy to its limit, MCMC performs a task similar to repeatedly asking "How likely is this pebble I found to be from the mountain I am searching for?", and completes its task by returning thousands of accepted pebbles in hopes of reconstructing the original mountain. In MCMC and PyMC lingo, the returned sequence of "pebbles" are the samples, cumulatively called the *traces*.

When I say MCMC intelligently searches, I really am saying MCMC will *hopefully* converge towards the areas of high posterior probability. MCMC does this by exploring nearby positions and moving into areas with higher probability. Again, perhaps "converge" is not an accurate term to describe MCMC's progression. Converging usually implies moving towards a point in space, but MCMC moves towards a *broader area* in the space and randomly walks in that area, picking up samples from that area.

Why Thousands of Samples? At first, returning thousands of samples to the user might sound like being an inefficient way to describe the posterior distributions. I would argue that this is extremely efficient. Consider the alternative possibilities:

1. Returning a mathematical formula for the "mountain ranges" would involve describing a N-dimensional surface with arbitrary peaks and valleys.
2. Returning the "peak" of the landscape, while mathematically possible and a sensible thing to do as the highest point corresponds to most probable estimate of the unknowns, ignores the shape of the landscape, which we have previously argued is very important in determining posterior confidence in unknowns.

Besides computational reasons, likely the strongest reason for returning samples is that we can easily use *The Law of Large Numbers* to solve otherwise intractable problems. I postpone this discussion for the next chapter. With the thousands of samples, we can reconstruct the posterior surface by organizing them in a histogram.

1.1.3 Algorithms to perform MCMC

There is a large family of algorithms that perform MCMC. Most of these algorithms can be expressed at a high level as follows: (Mathematical details can be found in the appendix.)

1. Start at current position.
2. Propose moving to a new position (investigate a pebble near you).
3. Accept/Reject the new position based on the position's adherence to the data and prior distributions (ask if the pebble likely came from the mountain).

4.
   1. If you accept: Move to the new position. Return to Step 1.
   2. Else: Do not move to new position. Return to Step 1.
5. After a large number of iterations, return all accepted positions.

This way we move in the general direction towards the regions where the posterior distributions exist, and collect samples sparingly on the journey. Once we reach the posterior distribution, we can easily collect samples as they likely all belong to the posterior distribution.

If the current position of the MCMC algorithm is in an area of extremely low probability, which is often the case when the algorithm begins (typically at a random location in the space), the algorithm will move in positions *that are likely not from the posterior* but better than everything else nearby. Thus the first moves of the algorithm are not reflective of the posterior.

In the above algorithm's pseudocode, notice that only the current position matters (new positions are investigated only near the current position). We can describe this property as *memorylessness*, i.e. the algorithm does not care *how* it arrived at its current position, only that it is there.

1.1.4 Other approximation solutions to the posterior

Besides MCMC, there are other procedures available for determining the posterior distributions. A Laplace approximation is an approximation of the posterior using simple functions. A more advanced method is Variational Bayes. All three methods, Laplace Approximations, Variational Bayes, and classical MCMC have their pros and cons. We will only focus on MCMC in this book. That being said, my friend Imri Sofar likes to classify MCMC algorithms as either "they suck", or "they really suck". He classifies the particular flavour of MCMC used by PyMC as just *sucks* ;)

Example: Unsupervised Clustering using a Mixture Model Suppose we are given the following dataset:

```
In [5]: figsize(12.5, 4)
        data = np.loadtxt("data/mixture_data.csv", delimiter=",")

        plt.hist(data, bins=20, color="k", histtype="stepfilled", alpha=0.8)
        plt.title("Histogram of the dataset")
        plt.ylim([0, None])
        print(data[:10], "...")
```

```
[ 115.85679142  152.26153716  178.87449059  162.93500815  107.02820697
  105.19141146  118.38288501  125.3769803   102.88054011  206.71326136] ...
```

What does the data suggest? It appears the data has a bimodal form, that is, it appears to have two peaks, one near 120 and the other near 200. Perhaps there are *two clusters* within this dataset.

This dataset is a good example of the data-generation modeling technique from last chapter. We can propose *how* the data might have been created. I suggest the following data generation algorithm:

1. For each data point, choose cluster 1 with probability p, else choose cluster 2.
2. Draw a random variate from a Normal distribution with parameters μ_i and σ_i where i was chosen in step 1.
3. Repeat.

This algorithm would create a similar effect as the observed dataset, so we choose this as our model. Of course, we do not know p or the parameters of the Normal distributions. Hence we must infer, or *learn*, these unknowns.

Denote the Normal distributions Nor_0 and Nor_1 (having variables' index start at 0 is just Pythonic). Both currently have unknown mean and standard deviation, denoted μ_i and σ_i, $i = 0, 1$ respectively. A specific data point can be from either Nor_0 or Nor_1, and we assume that the data point is assigned to Nor_0 with probability p.

An appropriate way to assign data points to clusters is to use a PyMC `Categorical` stochastic variable. Its parameter is a k-length array of probabilities that must sum to one and its `value` attribute is a integer between 0 and $k-1$ randomly chosen according to the crafted array of probabilities. (In our case $k = 2$) *A priori*, we do not know what the probability of assignment to cluster 1 is, so we create a uniform variable over 0,1 to model this. Call this `p`. Thus the probability array we enter into the `Categorical` variable is `[p, 1-p]`.

```
In [6]: import pymc as pm

        p = pm.Uniform("p", 0, 1)

        assignment = pm.Categorical("assignment", [p, 1 - p], size=data.shape[0])
        print("prior assignment, with p = %.2f:" % p.value)
        print(assignment.value[:10], "...")
```

```
prior assignment, with p = 0.32:
[0 1 1 0 1 1 1 1 0 1] ...
```

Looking at the above dataset, I would guess that the standard deviations of the two Normals are different. To maintain ignorance of what the standard deviations might be, we will initially model them as uniform on 0 to 100. Really we are talking about τ, the *precision* of the Normal distribution, but it is easier to think in terms of standard deviation. Our PyMC code will need to transform our standard deviation into precision by the relation:

$$\tau = \frac{1}{\sigma^2}$$

In PyMC, we can do this in one step by writing:

```
taus = 1.0/pm.Uniform( "stds", 0, 100, size= 2)**2
```

Notice that we specified `size=2`: we are modeling both τs as a single PyMC variable. Note that this does not induce a necessary relationship between the two τs, it is simply for succinctness.

We also need to specify priors on the centers of the clusters. The centers are really the μ parameters in this Normal distributions. Their priors can be modeled by a Normal distribution. Looking at the data, I have an idea where the two centers might be — I would guess somewhere around 120 and 190 respectively, though I am not very confident in these eyeballed estimates. Hence I will set $\mu_0 = 120, \mu_1 = 190$ and $\sigma_{0,1} = 10$ (recall we enter the τ parameter, so enter $1/\sigma^2 = 0.01$ in the PyMC variable.)

```
In [7]: stds = pm.Uniform("stds", 0, 100, size=2)
        taus = 1.0 / stds ** 2
        centers = pm.Normal("centers", [120, 190], [0.01, 0.01], size=2)

        """
        The below deterministic functions map an assignment, in this case 0 or 1,
        to a set of parameters, located in the (1,2) arrays `taus` and `centers`.
        """

        @pm.deterministic
        def center_i(assignment=assignment, centers=centers):
            return centers[assignment]

        @pm.deterministic
        def tau_i(assignment=assignment, taus=taus):
            return taus[assignment]

        print("Random assignments: ", assignment.value[:4], "...")
        print("Assigned center: ", center_i.value[:4], "...")
        print("Assigned precision: ", tau_i.value[:4], "...")
```

```
Random assignments:  [0 1 1 0] ...
Assigned center:  [ 107.08313959  186.02196393  186.02196393  107.08313959] ...
Assigned precision:  [ 0.02216706  0.0013977   0.0013977   0.02216706] ...
```

```
In [8]: # and to combine it with the observations:
        observations = pm.Normal("obs", center_i, tau_i, value=data, observed=True)
```

```
# below we create a model class
model = pm.Model([p, assignment, observations, taus, centers])
```

PyMC has an MCMC class, `MCMC` in the main namespace of PyMC, that implements the MCMC exploring algorithm. We initialize it by passing in a `Model` instance:

```
mcmc = pm.MCMC( model )
```

The method for asking the `MCMC` to explore the space is `sample( iterations )`, where `iterations` is the number of steps you wish the algorithm to perform. We try 50000 steps below:

```
In [9]: mcmc = pm.MCMC(model)
        mcmc.sample(50000)
```

```
 [-----------------100%-----------------] 50000 of 50000 complete in 23.6 sec
```

Below I plot the paths, or "traces", the unknown parameters (centers, precisions, and p) have taken thus far. The traces can be retrieved using the `trace` method in the `MCMC` object created, which accepts the assigned PyMC variable `name`. For example, `mcmc.trace("centers")` will retrieve a `Trace` object that can be indexed (using `[:]` or `.gettrace()` to retrieve all traces, or fancy-indexing like `[1000:]`).

```
In [10]: figsize(12.5, 9)
         plt.subplot(311)
         lw = 1
         center_trace = mcmc.trace("centers")[:]

         # for pretty colors later in the book.
         colors = ["#348ABD", "#A60628"] \
         if center_trace[-1, 0] > center_trace[-1, 1] \
             else ["#A60628", "#348ABD"]

         plt.plot(center_trace[:, 0], label="trace of center 0", c=colors[0], lw=lw)
         plt.plot(center_trace[:, 1], label="trace of center 1", c=colors[1], lw=lw)
         plt.title("Traces of unknown parameters")
         leg = plt.legend(loc="upper right")
         leg.get_frame().set_alpha(0.7)

         plt.subplot(312)
         std_trace = mcmc.trace("stds")[:]
         plt.plot(std_trace[:, 0], label="trace of standard deviation of cluster 0",
              c=colors[0], lw=lw)
         plt.plot(std_trace[:, 1], label="trace of standard deviation of cluster 1",
              c=colors[1], lw=lw)
         plt.legend(loc="upper left")

         plt.subplot(313)
```

```
p_trace = mcmc.trace("p")[:]
plt.plot(p_trace, label="$p$: frequency of assignment to cluster 0",
     color="#467821", lw=lw)
plt.xlabel("Steps")
plt.ylim(0, 1)
plt.legend();
```

Notice the following characteristics:

1. The traces converges, not to a single point, but to a *distribution* of possible points. This is *convergence* in an MCMC algorithm.
2. Inference using the first few thousand points is a bad idea, as they are unrelated to the final distribution we are interested in. Thus is it a good idea to discard those samples before using the samples for inference. We call this period before converge the *burn-in period*.
3. The traces appear as a random "walk" around the space, that is, the paths exhibit correlation with previous positions. This is both good and bad. We will always have correlation between current positions and the previous positions, but too much of it means we are not exploring the space well. This will be detailed in the Diagnostics section later in this chapter.

To achieve further convergence, we will perform more MCMC steps. Starting the MCMC again after it has already been called does not mean starting the entire algorithm over. In the pseudo-code algorithm of MCMC above, the only position that matters is the current position

(new positions are investigated near the current position), implicitly stored in PyMC variables' `value` attribute. Thus it is fine to halt an MCMC algorithm and inspect its progress, with the intention of starting it up again later. The `value` attributes are not overwritten.

We will sample the MCMC one hundred thousand more times and visualize the progress below:

```
In [11]: mcmc.sample(100000)
```

```
 [-----------------100%-----------------] 100000 of 100000 complete in 66.4 sec
```

```
In [12]: figsize(12.5, 4)
         center_trace = mcmc.trace("centers", chain=1)[:]
         prev_center_trace = mcmc.trace("centers", chain=0)[:]

         x = np.arange(50000)
         plt.plot(x, prev_center_trace[:, 0], label="previous trace of center 0",
              lw=lw, alpha=0.4, c=colors[1])
         plt.plot(x, prev_center_trace[:, 1], label="previous trace of center 1",
              lw=lw, alpha=0.4, c=colors[0])

         x = np.arange(50000, 150000)
         plt.plot(x, center_trace[:, 0], label="new trace of center 0", lw=lw, c="#348ABD")
         plt.plot(x, center_trace[:, 1], label="new trace of center 1", lw=lw, c="#A60628")

         plt.title("Traces of unknown center parameters")
         leg = plt.legend(loc="upper right")
         leg.get_frame().set_alpha(0.8)
         plt.xlabel("Steps");
```

The `trace` method in the `MCMC` instance has a keyword argument `chain`, that indexes which call to `sample` you would like to be returned. (Often we need to call `sample` multiple times, and the ability to retrieve past samples is a useful procedure). The default for `chain` is -1, which will return the samples from the lastest call to `sample`.

Cluster Investigation We have not forgotten our main challenge: identify the clusters. We have determined posterior distributions for our unknowns. We plot the posterior distributions of the center and standard deviation variables below:

```
In [13]: figsize(11.0, 4)
         std_trace = mcmc.trace("stds")[:]

         _i = [1, 2, 3, 4]
         for i in range(2):
             plt.subplot(2, 2, _i[2 * i])
             plt.title("Posterior of center of cluster %d" % i)
             plt.hist(center_trace[:, i], color=colors[i], bins=30,
                      histtype="stepfilled")

             plt.subplot(2, 2, _i[2 * i + 1])
             plt.title("Posterior of standard deviation of cluster %d" % i)
             plt.hist(std_trace[:, i], color=colors[i], bins=30,
                      histtype="stepfilled")
             # plt.autoscale(tight=True)

         plt.tight_layout()
```

The MCMC algorithm has proposed that the most likely centers of the two clusters are near 120 and 200 respectively. Similar inference can be applied to the standard deviation.

We are also given the posterior distributions for the labels of the data point, which is present in `mcmc.trace("assignment")`. Below is a visualization of this. The y-axis represents a subsample of the posterior labels for each data point. The x-axis are the sorted values of the data points. A red square is an assignment to cluster 1, and a blue square is an assignment to cluster 0.

```
In [14]: import matplotlib as mpl
         figsize(12.5, 4.5)
         plt.cmap = mpl.colors.ListedColormap(colors)
         plt.imshow(mcmc.trace("assignment")[::400, np.argsort(data)],
                cmap=plt.cmap, aspect=.4, alpha=.9)
```

```
plt.xticks(np.arange(0, data.shape[0], 40),
       ["%.2f" % s for s in np.sort(data)[::40]])
plt.ylabel("posterior sample")
plt.xlabel("value of $i$th data point")
plt.title("Posterior labels of data points");
```

Looking at the above plot, it appears that the most uncertainty is between 150 and 170. The above plot slightly misrepresents things, as the x-axis is not a true scale (it displays the value of the ith sorted data point.) A more clear diagram is below, where we have estimated the *frequency* of each data point belonging to the labels 0 and 1.

```
In [15]: cmap = mpl.colors.LinearSegmentedColormap.from_list("BMH", colors)
         assign_trace = mcmc.trace("assignment")[:]
         plt.scatter(data, 1 - assign_trace.mean(axis=0), cmap=cmap,
                 c=assign_trace.mean(axis=0), s=50)
         plt.ylim(-0.05, 1.05)
         plt.xlim(35, 300)
         plt.title("Probability of data point belonging to cluster 0")
         plt.ylabel("probability")
         plt.xlabel("value of data point");
```

Even though we modeled the clusters using Normal distributions, we didn't get just a single Normal distribution that *best* fits the data (whatever our definition of best is), but a distribution of values for the Normal's parameters. How can we choose just a single pair of values for the mean and variance and determine a *sorta-best-fit* gaussian?

One quick and dirty way (which has nice theoretical properties we will see in Chapter 5), is to use the *mean* of the posterior distributions. Below we overlay the Normal density functions, using the mean of the posterior distributions as the chosen parameters, with our observed data:

```
In [16]: norm = stats.norm
         x = np.linspace(20, 300, 500)
         posterior_center_means = center_trace.mean(axis=0)
         posterior_std_means = std_trace.mean(axis=0)
         posterior_p_mean = mcmc.trace("p")[:].mean()

         plt.hist(data, bins=20, histtype="step", normed=True, color="k",
              lw=2, label="histogram of data")
         y = posterior_p_mean * norm.pdf(x, loc=posterior_center_means[0],
                                         scale=posterior_std_means[0])
         plt.plot(x, y, label="Cluster 0 (using posterior-mean parameters)", lw=3)
         plt.fill_between(x, y, color=colors[1], alpha=0.3)

         y = (1 - posterior_p_mean) * norm.pdf(x, loc=posterior_center_means[1],
                                               scale=posterior_std_means[1])
         plt.plot(x, y, label="Cluster 1 (using posterior-mean parameters)", lw=3)
         plt.fill_between(x, y, color=colors[0], alpha=0.3)

         plt.legend(loc="upper left")
         plt.title("Visualizing Clusters using posterior-mean parameters");
```

1.1.5 Important: Don't mix posterior samples

In the above example, a possible (though less likely) scenario is that cluster 0 has a very large standard deviation, and cluster 1 has a small standard deviation. This would still satisfy the evidence, albeit less so than our original inference. Alternatively, it would be incredibly unlikely for *both* distributions to have a small standard deviation, as the data does not support this hypothesis at all. Thus the two standard deviations are *dependent* on each other: if one is small, the other must be large. In fact, *all* the unknowns are related in a similar manner. For example, if a standard deviation is large, the mean has a wider possible space of realizations. Conversely, a small standard deviation restricts the mean to a small area.

During MCMC, we are returned vectors representing samples from the unknown posteriors. Elements of different vectors cannot be used together, as this would break the above logic: perhaps a sample has returned that cluster 1 has a small standard deviation, hence all the other variables in that sample would incorporate that and be adjusted accordingly. It is easy to avoid this problem though, just make sure you are indexing traces correctly.

Another small example to illustrate the point. Suppose two variables, x and y, are related by $x + y = 10$. We model x as a Normal random variable with mean 4 and explore 500 samples.

```
In [17]: import pymc as pm

         x = pm.Normal("x", 4, 10)
         y = pm.Lambda("y", lambda x=x: 10 - x, trace=True)

         ex_mcmc = pm.MCMC(pm.Model([x, y]))
         ex_mcmc.sample(500)

         plt.plot(ex_mcmc.trace("x")[:])
         plt.plot(ex_mcmc.trace("y")[:])
         plt.title("Displaying (extreme) case of dependence between unknowns");
```

```
[-----------------100%-----------------] 500 of 500 complete in 0.0 sec
```

As you can see, the two variables are not unrelated, and it would be wrong to add the ith sample of x to the jth sample of y, unless $i = j$.

Returning to Clustering: Prediction The above clustering can be generalized to k clusters. Choosing $k = 2$ allowed us to visualize the MCMC better, and examine some very interesting plots.

What about prediction? Suppose we observe a new data point, say $x = 175$, and we wish to label it to a cluster. It is foolish to simply assign it to the *closer* cluster center, as this ignores the standard deviation of the clusters, and we have seen from the plots above that this consideration is very important. More formally: we are interested in the *probability* (as we cannot be certain about labels) of assigning $x = 175$ to cluster 1. Denote the assignment of x as L_x, which is equal to 0 or 1, and we are interested in $P(L_x = 1 \mid x = 175)$.

A naive method to compute this is to re-run the above MCMC with the additional data point appended. The disadvantage with this method is that it will be slow to infer for each novel data point. Alternatively, we can try a *less precise*, but much quicker method.

We will use Bayes Theorem for this. If you recall, Bayes Theorem looks like:

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

In our case, A represents $L_x = 1$ and X is the evidence we have: we observe that $x = 175$. For a particular sample set of parameters for our posterior distribution, $(\mu_0, \sigma_0, \mu_1, \sigma_1, p)$, we are interested in asking "Is the probability that x is in cluster 1 *greater* than the probability it is in cluster 0?", where the probability is dependent on the chosen parameters.

$$P(L_x = 1|x = 175) > P(L_x = 0|x = 175) \tag{1}$$

$$\tag{2}$$

$$\frac{P(x = 175|L_x = 1)P(L_x = 1)}{P(x = 175)} > \frac{P(x = 175|L_x = 0)P(L_x = 0)}{P(x = 175)} \tag{3}$$

As the denominators are equal, they can be ignored (and good riddance, because computing the quantity $P(x = 175)$ can be difficult).

$$P(x = 175|L_x = 1)P(L_x = 1) > P(x = 175|L_x = 0)P(L_x = 0)$$

```
In [18]: norm_pdf = stats.norm.pdf
         p_trace = mcmc.trace("p")[:]
         x = 175

         v = p_trace * norm_pdf(x, loc=center_trace[:, 0], scale=std_trace[:, 0]) > \
             (1 - p_trace) * norm_pdf(x, loc=center_trace[:, 1], scale=std_trace[:, 1])

         print("Probability of belonging to cluster 1:", v.mean())
```

```
Probability of belonging to cluster 1: 0.00968
```

Giving us a probability instead of a label is a very useful thing. Instead of the naive

```
L = 1 if prob > 0.5 else 0
```

we can optimize our guesses using a *loss function*, which the entire fifth chapter is devoted to.

1.1.6 Using `MAP` to improve convergence

If you ran the above example yourself, you may have noticed that our results were not consistent: perhaps your cluster division was more scattered, or perhaps less scattered. The problem is that our traces are a function of the *starting values* of the MCMC algorithm.

It can be mathematically shown that letting the MCMC run long enough, by performing many steps, the algorithm *should forget its initial position*. In fact, this is what it means to say the MCMC converged (in practice though we can never achieve total convergence). Hence if we observe different posterior analysis, it is likely because our MCMC has not fully converged yet, and we should not use samples from it yet (we should use a larger burn-in period).

In fact, poor starting values can prevent any convergence, or significantly slow it down. Ideally, we would like to have the chain start at the *peak* of our landscape, as this is exactly where the posterior distributions exist. Hence, if we started at the "peak", we could avoid a lengthy burn-in period and incorrect inference. Generally, we call this "peak" the *maximum a posterior* or, more simply, the *MAP*.

Of course, we do not know where the MAP is. PyMC provides an object that will approximate, if not find, the MAP location. In the PyMC main namespace is the `MAP` object that accepts a PyMC `Model` instance. Calling `.fit()` from the `MAP` instance sets the variables in the model to their MAP values.

```
map_ = pm.MAP( model )
map_.fit()
```

The `MAP.fit()` methods has the flexibility of allowing the user to choose which optimization algorithm to use (after all, this is a optimization problem: we are looking for the values that maximize our landscape), as not all optimization algorithms are created equal. The default optimization algorithm in the call to `fit` is scipy's `fmin` algorithm (which attempts to minimize the *negative of the landscape*). An alternative algorithm that is available is Powell's Method, a favourite of PyMC blogger Abraham Flaxman [1], by calling `fit(method='fmin_powell')`. From my experience, I use the default, but if my convergence is slow or not guaranteed, I experiment with Powell's method.

The MAP can also be used as a solution to the inference problem, as mathematically it is the *most likely* value for the unknowns. But as mentioned earlier in this chapter, this location ignores the uncertainty and doesn't return a distribution.

Most often it is a good idea, and rarely a bad idea, to prepend your call to `mcmc` with a call to `MAP(model).fit()`. The intermediate call to `fit` is hardly computationally intensive, and will save you time later due to a shorter burn-in period.

Speaking of the burn-in period It is still a good idea to provide a burn-in period, even if we are using `MAP` prior to calling `MCMC.sample`, just to be safe. We can have PyMC automatically discard the first n samples by specifying the `burn` parameter in the call to `sample`. As one does not know when the chain has fully converged, I like to assign the first *half* of my samples to be discarded, sometimes up to 90% of my samples for longer runs. To continue the clustering example from above, my new code would look something like:

```
model = pm.Model( [p, assignment, taus, centers ] )

map_ = pm.MAP( model )
```

```
map_.fit() #stores the fitted variables' values in foo.value

mcmc = pm.MCMC( model )
mcmc.sample( 100000, 50000 )
```

1.2 Diagnosing Convergence

1.2.1 Autocorrelation

Autocorrelation is a measure of how related a series of numbers is with itself. A measurement of 1.0 is perfect positive autocorrelation, 0 no autocorrelation, and -1 is perfect negative correlation. If you are familiar with standard *correlation*, then autocorrelation is just how correlated a series, x_τ, at time t is with the series at time $t-k$:

$$R(k) = Corr(x_t, x_{t-k})$$

For example, consider the two series:

$$x_t \sim \text{Normal}(0,1), \quad x_0 = 0$$
$$y_t \sim \text{Normal}(y_{t-1},1), \quad y_0 = 0$$

which have example paths like:

```
In [19]: figsize(12.5, 4)

         import pymc as pm
         x_t = pm.rnormal(0, 1, 200)
         x_t[0] = 0
         y_t = np.zeros(200)
         for i in range(1, 200):
             y_t[i] = pm.rnormal(y_t[i - 1], 1)

         plt.plot(y_t, label="$y_t$", lw=3)
         plt.plot(x_t, label="$x_t$", lw=3)
         plt.xlabel("time, $t$")
         plt.legend();
```

One way to think of autocorrelation is "If I know the position of the series at time s, can it help me know where I am at time t?" In the series x_t, the answer is No. By construction, x_t are random variables. If I told you that $x_2 = 0.5$, could you give me a better guess about x_3? No.

On the other hand, y_t is autocorrelated. By construction, if I knew that $y_2 = 10$, I can be very confident that y_3 will not be very far from 10. Similarly, I can even make a (less confident guess) about y_4: it will probably not be near 0 or 20, but a value of 5 is not too unlikely. I can make a similar argument about y_5, but again, I am less confident. Taking this to its logical conclusion, we must concede that as k, the lag between time points, increases the autocorrelation decreases. We can visualize this:

```
In [20]: def autocorr(x):
             # from http://tinyurl.com/afz57c4
             result = np.correlate(x, x, mode='full')
             result = result / np.max(result)
             return result[result.size // 2:]

         colors = ["#348ABD", "#A60628", "#7A68A6"]

         x = np.arange(1, 200)
         plt.bar(x, autocorr(y_t)[1:], width=1, label="$y_t$",
                 edgecolor=colors[0], color=colors[0])
         plt.bar(x, autocorr(x_t)[1:], width=1, label="$x_t$",
                 color=colors[1], edgecolor=colors[1])

         plt.legend(title="Autocorrelation")
         plt.ylabel("measured correlation \nbetween $y_t$ and $y_{t-k}$.")
         plt.xlabel("k (lag)")
         plt.title("Autocorrelation plot of $y_t$ and $x_t$ for differing $k$ lags.");
```

Notice that as k increases, the autocorrelation of y_t decreases from a very high point. Compare with the autocorrelation of x_t which looks like noise (which it really is), hence we can conclude no autocorrelation exists in this series.

How does this relate to MCMC convergence? By the nature of the MCMC algorithm, we will always be returned samples that exhibit autocorrelation (this is because of the step `from your current position, move to a position near you`).

A chain that is [Isn't meandering exploring?] exploring the space well will exhibit very high autocorrelation. Visually, if the trace seems to meander like a river, and not settle down, the chain will have high autocorrelation.

This does not imply that a converged MCMC has low autocorrelation. Hence low autocorrelation is not necessary for convergence, but it is sufficient. PyMC has a built-in autocorrelation plotting function in the `Matplot` module.

1.2.2 Thinning

Another issue can arise if there is high-autocorrelation between posterior samples. Many post-processing algorithms require samples to be *independent* of each other. This can be solved, or at least reduced, by only returning to the user every nth sample, thus removing some autocorrelation. Below we perform an autocorrelation plot for y_t with differing levels of thinning:

```
In [21]: max_x = 200 // 3 + 1
         x = np.arange(1, max_x)

         plt.bar(x, autocorr(y_t)[1:max_x], edgecolor=colors[0],
                 label="no thinning", color=colors[0], width=1)
         plt.bar(x, autocorr(y_t[::2])[1:max_x], edgecolor=colors[1],
                 label="keeping every 2nd sample", color=colors[1], width=1)
         plt.bar(x, autocorr(y_t[::3])[1:max_x], width=1, edgecolor=colors[2],
                 label="keeping every 3rd sample", color=colors[2])

         plt.autoscale(tight=True)
         plt.legend(title="Autocorrelation plot for $y_t$", loc="lower left")
         plt.ylabel("measured correlation \nbetween $y_t$ and $y_{t-k}$.")
         plt.xlabel("k (lag)")
         plt.title("Autocorrelation of $y_t$ (no thinning vs. thinning) \
         at differing $k$ lags.");
```

With more thinning, the autocorrelation drops quicker. There is a tradeoff though: higher thinning requires more MCMC iterations to achieve the same number of returned samples. For example, 10 000 samples unthinned is 100 000 with a thinning of 10 (though the latter has less autocorrelation).

What is a good amount of thinning? The returned samples will always exhibit some autocorrelation, regardless of how much thinning is done. So long as the autocorrelation tends to zero, you are probably ok. Typically thinning of more than 10 is not necessary.

PyMC exposes a `thinning` parameter in the call to `sample`, for example: `sample( 10000, burn = 5000, thinning = 5)`.

1.2.3 `pymc.Matplot.plot()`

It seems silly to have to manually create histograms, autocorrelation plots and trace plots each time we perform MCMC. The authors of PyMC have included a visualization tool for just this purpose.

As the title suggests, the `pymc.Matplot` module contains a poorly named function `plot`, which I prefer to import as `mcplot` so there is no conflict with other namespaces. `plot`, or `mcplot` as I suggest, accepts an `MCMC` object and will return posterior distributions, traces and auto-correlations for each variable (up to 10 variables).

Below we use the tool to plot the centers of the clusters, after sampling 25 000 more times and `thinning = 10`.

```
In [22]: from pymc.Matplot import plot as mcplot

         mcmc.sample(25000, 0, 10)
         mcplot(mcmc.trace("centers", 2), common_scale=False);
```

```
 [-----------------100%-----------------] 25000 of 25000 complete in 11.7 secPlotting centers_
Plotting centers_1
```

There are really two figures here, one for each unknown in the `centers` variable. In each figure, the subfigure in the top left corner is the trace of the variable. This is useful for inspecting that possible "meandering" property that is a result of non-convergence.

The largest plot on the right-hand side is the histograms of the samples, plus a few extra features. The thickest vertical line represents the posterior mean, which is a good summary of posterior distribution. The interval between the two dashed vertical lines in each the posterior distributions represent the *95% credible interval*, not to be confused with a *95% confidence interval*. I won't get into the latter, but the former can be interpreted as "there is a 95% chance the parameter of interest lies in this interval". (Changing default parameters in the call to `mcplot` provides alternatives to 95%.) When communicating your results to others, it is incredibly important to state this interval. One of our purposes for studying Bayesian methods is to have a clear understanding of our uncertainty in unknowns. Combined with the posterior mean, the 95% credible interval provides a reliable interval to communicate the likely location of the unknown (provided by the mean) *and* the uncertainty (represented by the width of the interval).

The plots titled `center_0_acorr` and `center_1_acorr` are the generated autocorrelation plots. They look different than the ones I have displayed above, but the only difference is that 0-lag is centered in the middle of the figure, whereas I have 0 centered to the left.

1.3 Useful tips for MCMC

Bayesian inference would be the *de facto* method if it weren't for MCMC's computational difficulties. In fact, MCMC is what turns most users off practical Bayesian inference. Below I present some good heuristics to help convergence and speed up the MCMC engine:

1.3.1 Intelligent starting values

It would be great to start the MCMC algorithm off near the posterior distribution, so that it will take little time to start sampling correctly. We can aid the algorithm by telling where we *think* the posterior distribution will be by specifying the `value` parameter in the `Stochastic` variable creation. In many cases we can produce a reasonable guess for the parameter. For example, if we have data from a Normal distribution, and we wish to estimate the μ parameter, then a good starting value would be the *mean* of the data.

```
mu = pm.Uniform( "mu", 0, 100, value = data.mean() )
```

For most parameters in models, there is a frequentist estimate of it. These estimates are a good starting value for our MCMC algorithms. Of course, this is not always possible for some variables, but including as many appropriate initial values is always a good idea. Even if your guesses are wrong, the MCMC will still converge to the proper distribution, so there is little to lose.

This is what using `MAP` tries to do, by giving good initial values to the MCMC. So why bother specifying user-defined values? Well, even giving `MAP` good values will help it find the maximum a-posterior.

Also important, *bad initial values* are a source of major bugs in PyMC and can hurt convergence.

Priors If the priors are poorly chosen, the MCMC algorithm may not converge, or at least have difficulty converging. Consider what may happen if the prior chosen does not even contain the true parameter: the prior assigns 0 probability to the unknown, hence the posterior will assign 0 probability as well. This can cause pathological results.

For this reason, it is best to carefully choose the priors. Often, lack of convergence or evidence of samples crowding to boundaries implies something is wrong with the chosen priors (see *Folk Theorem of Statistical Computing* below).

Covariance matrices and eliminating parameters

1.3.2 The Folk Theorem of Statistical Computing

> *If you are having computational problems, probably your model is wrong.*

1.4 Conclusion

PyMC provides a very strong backend to performing Bayesian inference, mostly because it has abstracted the inner mechanics of MCMC from the user. Despite this, some care must be applied to ensure your inference is not being biased by the iterative nature of MCMC.

1.4.1 References

1. Flaxman, Abraham. "Powell's Methods for Maximization in PyMC." Healthy Algorithms. N.p., 9 02 2012. Web. 28 Feb 2013. http://healthyalgorithms.com/2012/02/09/powells-method-for-maximization-in-pymc/.


```
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 0x102453550>

In [ ]:

Ch3_IntroMCMC_PyMC3

January 4, 2018

1 Chapter 3

```
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) ____
```

1.1 Opening the black box of MCMC

The previous two chapters hid the inner-mechanics of PyMC3, and more generally Markov Chain Monte Carlo (MCMC), from the reader. The reason for including this chapter is three-fold. The first is that any book on Bayesian inference must discuss MCMC. I cannot fight this. Blame the statisticians. Secondly, knowing the process of MCMC gives you insight into whether your algorithm has converged. (Converged to what? We will get to that) Thirdly, we'll understand *why* we are returned thousands of samples from the posterior as a solution, which at first thought can be odd.

1.1.1 The Bayesian landscape

When we setup a Bayesian inference problem with N unknowns, we are implicitly creating an N dimensional space for the prior distributions to exist in. Associated with the space is an additional dimension, which we can describe as the *surface*, or *curve*, that sits on top of the space, that reflects the *prior probability* of a particular point. The surface on the space is defined by our prior distributions. For example, if we have two unknowns p_1 and p_2, and priors for both are Uniform$(0, 5)$, the space created is a square of length 5 and the surface is a flat plane that sits on top of the square (representing that every point is equally likely).

```
In [1]: %matplotlib inline
        import scipy.stats as stats
        from IPython.core.pylabtools import figsize
        import numpy as np
        figsize(12.5, 4)

        import matplotlib.pyplot as plt
        from mpl_toolkits.mplot3d import Axes3D

        jet = plt.cm.jet
        fig = plt.figure()
```

```
x = y = np.linspace(0, 5, 100)
X, Y = np.meshgrid(x, y)

plt.subplot(121)
uni_x = stats.uniform.pdf(x, loc=0, scale=5)
uni_y = stats.uniform.pdf(y, loc=0, scale=5)
M = np.dot(uni_x[:, None], uni_y[None, :])
im = plt.imshow(M, interpolation='none', origin='lower',
                cmap=jet, vmax=1, vmin=-.15, extent=(0, 5, 0, 5))

plt.xlim(0, 5)
plt.ylim(0, 5)
plt.title("Landscape formed by Uniform priors.")

ax = fig.add_subplot(122, projection='3d')
ax.plot_surface(X, Y, M, cmap=plt.cm.jet, vmax=1, vmin=-.15)
ax.view_init(azim=390)
plt.title("Uniform prior landscape; alternate view");
```

Alternatively, if the two priors are Exp(3) and Exp(10), then the space is all positive numbers on the 2-D plane, and the surface induced by the priors looks like a water fall that starts at the point (0,0) and flows over the positive numbers.

The plots below visualize this. The more dark red the color, the more prior probability is assigned to that location. Conversely, areas with darker blue represent that our priors assign very low probability to that location.

```
In [2]: figsize(12.5, 5)
        fig = plt.figure()
        plt.subplot(121)

        exp_x = stats.expon.pdf(x, scale=3)
        exp_y = stats.expon.pdf(x, scale=10)
        M = np.dot(exp_x[:, None], exp_y[None, :])
```

```
CS = plt.contour(X, Y, M)
im = plt.imshow(M, interpolation='none', origin='lower',
                cmap=jet, extent=(0, 5, 0, 5))
#plt.xlabel("prior on $p_1$")
#plt.ylabel("prior on $p_2$")
plt.title("$Exp(3), Exp(10)$ prior landscape")

ax = fig.add_subplot(122, projection='3d')
ax.plot_surface(X, Y, M, cmap=jet)
ax.view_init(azim=390)
plt.title("$Exp(3), Exp(10)$ prior landscape; \nalternate view");
```

These are simple examples in 2D space, where our brains can understand surfaces well. In practice, spaces and surfaces generated by our priors can be much higher dimensional.

If these surfaces describe our *prior distributions* on the unknowns, what happens to our space after we incorporate our observed data X? The data X does not change the space, but it changes the surface of the space by *pulling and stretching the fabric of the prior surface* to reflect where the true parameters likely live. More data means more pulling and stretching, and our original shape becomes mangled or insignificant compared to the newly formed shape. Less data, and our original shape is more present. Regardless, the resulting surface describes the *posterior distribution*.

Again I must stress that it is, unfortunately, impossible to visualize this in large dimensions. For two dimensions, the data essentially *pushes up* the original surface to make *tall mountains*. The tendency of the observed data to *push up* the posterior probability in certain areas is checked by the prior probability distribution, so that less prior probability means more resistance. Thus in the double-exponential prior case above, a mountain (or multiple mountains) that might erupt near the (0,0) corner would be much higher than mountains that erupt closer to (5,5), since there is more resistance (low prior probability) near (5,5). The peak reflects the posterior probability of where the true parameters are likely to be found. Importantly, if the prior has assigned a probability of 0, then no posterior probability will be assigned there.

Suppose the priors mentioned above represent different parameters λ of two Poisson distributions. We observe a few data points and visualize the new landscape:

In [3]:
```
# create the observed data

# sample size of data we observe, trying varying this (keep it less than 100 ;)
N = 1

# the true parameters, but of course we do not see these values...
lambda_1_true = 1
lambda_2_true = 3

#...we see the data generated, dependent on the above two values.
data = np.concatenate([
    stats.poisson.rvs(lambda_1_true, size=(N, 1)),
    stats.poisson.rvs(lambda_2_true, size=(N, 1))
], axis=1)
print("observed (2-dimensional,sample size = %d):" % N, data)

# plotting details.
x = y = np.linspace(.01, 5, 100)
likelihood_x = np.array([stats.poisson.pmf(data[:, 0], _x)
                         for _x in x]).prod(axis=1)
likelihood_y = np.array([stats.poisson.pmf(data[:, 1], _y)
                         for _y in y]).prod(axis=1)
L = np.dot(likelihood_x[:, None], likelihood_y[None, :])
```

```
observed (2-dimensional,sample size = 1): [[0 2]]
```

In [4]:
```
figsize(12.5, 12)
# matplotlib heavy lifting below, beware!
plt.subplot(221)
uni_x = stats.uniform.pdf(x, loc=0, scale=5)
uni_y = stats.uniform.pdf(x, loc=0, scale=5)
M = np.dot(uni_x[:, None], uni_y[None, :])
im = plt.imshow(M, interpolation='none', origin='lower',
                cmap=jet, vmax=1, vmin=-.15, extent=(0, 5, 0, 5))
plt.scatter(lambda_2_true, lambda_1_true, c="k", s=50, edgecolor="none")
plt.xlim(0, 5)
plt.ylim(0, 5)
plt.title("Landscape formed by Uniform priors on $p_1, p_2$.")

plt.subplot(223)
plt.contour(x, y, M * L)
im = plt.imshow(M * L, interpolation='none', origin='lower',
                cmap=jet, extent=(0, 5, 0, 5))
plt.title("Landscape warped by %d data observation;\n Uniform priors on $p_1, p_2$." %
plt.scatter(lambda_2_true, lambda_1_true, c="k", s=50, edgecolor="none")
plt.xlim(0, 5)
plt.ylim(0, 5)
```

```
plt.subplot(222)
exp_x = stats.expon.pdf(x, loc=0, scale=3)
exp_y = stats.expon.pdf(x, loc=0, scale=10)
M = np.dot(exp_x[:, None], exp_y[None, :])

plt.contour(x, y, M)
im = plt.imshow(M, interpolation='none', origin='lower',
                cmap=jet, extent=(0, 5, 0, 5))
plt.scatter(lambda_2_true, lambda_1_true, c="k", s=50, edgecolor="none")
plt.xlim(0, 5)
plt.ylim(0, 5)
plt.title("Landscape formed by Exponential priors on $p_1, p_2$.")

plt.subplot(224)
# This is the likelihood times prior, that results in the posterior.
plt.contour(x, y, M * L)
im = plt.imshow(M * L, interpolation='none', origin='lower',
                cmap=jet, extent=(0, 5, 0, 5))

plt.scatter(lambda_2_true, lambda_1_true, c="k", s=50, edgecolor="none")
plt.title("Landscape warped by %d data observation;\n Exponential priors on \
$p_1, p_2$." % N)
plt.xlim(0, 5)
plt.ylim(0, 5);
```

The plot on the left is the deformed landscape with the Uniform$(0, 5)$ priors, and the plot on the right is the deformed landscape with the exponential priors. Notice that the posterior landscapes look different from one another, though the data observed is identical in both cases. The reason is as follows. Notice the exponential-prior landscape, bottom right figure, puts very little *posterior* weight on values in the upper right corner of the figure: this is because *the prior does not put much weight there*. On the other hand, the uniform-prior landscape is happy to put posterior weight in the upper-right corner, as the prior puts more weight there.

Notice also the highest-point, corresponding the the darkest red, is biased towards (0,0) in the exponential case, which is the result from the exponential prior putting more prior weight in the (0,0) corner.

The black dot represents the true parameters. Even with 1 sample point, the mountains attempts to contain the true parameter. Of course, inference with a sample size of 1 is incredibly naive, and choosing such a small sample size was only illustrative.

It's a great exercise to try changing the sample size to other values (try 2,5,10,100?...) and observing how our "mountain" posterior changes.

1.1.2 Exploring the landscape using the MCMC

We should explore the deformed posterior space generated by our prior surface and observed data to find the posterior mountain. However, we cannot naively search the space: any computer scientist will tell you that traversing N-dimensional space is exponentially difficult in N: the size of the space quickly blows-up as we increase N (see the curse of dimensionality). What hope do we have to find these hidden mountains? The idea behind MCMC is to perform an intelligent search of the space. To say "search" implies we are looking for a particular point, which is perhaps not an accurate as we are really looking for a broad mountain.

Recall that MCMC returns *samples* from the posterior distribution, not the distribution itself. Stretching our mountainous analogy to its limit, MCMC performs a task similar to repeatedly asking "How likely is this pebble I found to be from the mountain I am searching for?", and completes its task by returning thousands of accepted pebbles in hopes of reconstructing the original mountain. In MCMC and PyMC3 lingo, the returned sequence of "pebbles" are the samples, cumulatively called the *traces*.

When I say MCMC intelligently searches, I really am saying MCMC will *hopefully* converge towards the areas of high posterior probability. MCMC does this by exploring nearby positions and moving into areas with higher probability. Again, perhaps "converge" is not an accurate term to describe MCMC's progression. Converging usually implies moving towards a point in space, but MCMC moves towards a *broader area* in the space and randomly walks in that area, picking up samples from that area.

Why Thousands of Samples? At first, returning thousands of samples to the user might sound like being an inefficient way to describe the posterior distributions. I would argue that this is extremely efficient. Consider the alternative possibilities:

1. Returning a mathematical formula for the "mountain ranges" would involve describing a N-dimensional surface with arbitrary peaks and valleys.
2. Returning the "peak" of the landscape, while mathematically possible and a sensible thing to do as the highest point corresponds to most probable estimate of the unknowns, ignores the shape of the landscape, which we have previously argued is very important in determining posterior confidence in unknowns.

Besides computational reasons, likely the strongest reason for returning samples is that we can easily use *The Law of Large Numbers* to solve otherwise intractable problems. I postpone this discussion for the next chapter. With the thousands of samples, we can reconstruct the posterior surface by organizing them in a histogram.

1.1.3 Algorithms to perform MCMC

There is a large family of algorithms that perform MCMC. Most of these algorithms can be expressed at a high level as follows: (Mathematical details can be found in the appendix.)

1. Start at current position.

2. Propose moving to a new position (investigate a pebble near you).

3. Accept/Reject the new position based on the position's adherence to the data and prior distributions (ask if the pebble likely came from the mountain).

4.
   1. If you accept: Move to the new position. Return to Step 1.
   2. Else: Do not move to new position. Return to Step 1.
5. After a large number of iterations, return all accepted positions.

This way we move in the general direction towards the regions where the posterior distributions exist, and collect samples sparingly on the journey. Once we reach the posterior distribution, we can easily collect samples as they likely all belong to the posterior distribution.

If the current position of the MCMC algorithm is in an area of extremely low probability, which is often the case when the algorithm begins (typically at a random location in the space), the algorithm will move in positions *that are likely not from the posterior* but better than everything else nearby. Thus the first moves of the algorithm are not reflective of the posterior.

In the above algorithm's pseudocode, notice that only the current position matters (new positions are investigated only near the current position). We can describe this property as *memorylessness*, i.e. the algorithm does not care *how* it arrived at its current position, only that it is there.

1.1.4 Other approximation solutions to the posterior

Besides MCMC, there are other procedures available for determining the posterior distributions. A Laplace approximation is an approximation of the posterior using simple functions. A more advanced method is Variational Bayes. All three methods, Laplace Approximations, Variational Bayes, and classical MCMC have their pros and cons. We will only focus on MCMC in this book. That being said, my friend Imri Sofar likes to classify MCMC algorithms as either "they suck", or "they really suck". He classifies the particular flavour of MCMC used by PyMC3 as just *sucks* ;)

Example: Unsupervised Clustering using a Mixture Model Suppose we are given the following dataset:

```
In [5]: figsize(12.5, 4)
        data = np.loadtxt("data/mixture_data.csv", delimiter=",")

        plt.hist(data, bins=20, color="k", histtype="stepfilled", alpha=0.8)
        plt.title("Histogram of the dataset")
        plt.ylim([0, None]);
        print(data[:10], "...")
```

```
[ 115.85679142  152.26153716  178.87449059  162.93500815  107.02820697
  105.19141146  118.38288501  125.3769803   102.88054011  206.71326136] ...
```

What does the data suggest? It appears the data has a bimodal form, that is, it appears to have two peaks, one near 120 and the other near 200. Perhaps there are *two clusters* within this dataset.

This dataset is a good example of the data-generation modeling technique from last chapter. We can propose *how* the data might have been created. I suggest the following data generation algorithm:

1. For each data point, choose cluster 1 with probability p, else choose cluster 2.
2. Draw a random variate from a Normal distribution with parameters μ_i and σ_i where i was chosen in step 1.
3. Repeat.

This algorithm would create a similar effect as the observed dataset, so we choose this as our model. Of course, we do not know p or the parameters of the Normal distributions. Hence we must infer, or *learn*, these unknowns.

Denote the Normal distributions N_0 and N_1 (having variables' index start at 0 is just Pythonic). Both currently have unknown mean and standard deviation, denoted μ_i and σ_i, $i = 0, 1$ respectively. A specific data point can be from either N_0 or N_1, and we assume that the data point is assigned to N_0 with probability p.

An appropriate way to assign data points to clusters is to use a PyMC3 `Categorical` stochastic variable. Its parameter is a k-length array of probabilities that must sum to one and its `value` attribute is a integer between 0 and $k-1$ randomly chosen according to the crafted array of probabilities (In our case $k = 2$). *A priori*, we do not know what the probability of assignment to cluster 1 is, so we form a uniform variable on $(0, 1)$. We call call this p_1, so the probability of belonging to cluster 2 is therefore $p_2 = 1 - p_1$.

Unfortunately, we can't we just give `[p1, p2]` to our `Categorical` variable. PyMC3 uses Theano under the hood to construct the models so we need to use `theano.tensor.stack()` to combine p_1 and p_2 into a vector that it can understand. We pass this vector into the `Categorical` variable as well as the `testval` parameter to give our variable an idea of where to start from.

```
In [6]: import pymc3 as pm
        import theano.tensor as T

        with pm.Model() as model:
            p1 = pm.Uniform('p', 0, 1)
```

```
        p2 = 1 - p1
        p = T.stack([p1, p2])
        assignment = pm.Categorical("assignment", p,
                                    shape=data.shape[0],
                                    testval=np.random.randint(0, 2, data.shape[0]))

    print("prior assignment, with p = %.2f:" % p1.tag.test_value)
    print(assignment.tag.test_value[:10])
```

```
Applied interval-transform to p and added transformed p_interval_ to model.
prior assignment, with p = 0.50:
[0 0 0 0 1 1 1 0 0 1]
```

Looking at the above dataset, I would guess that the standard deviations of the two Normals are different. To maintain ignorance of what the standard deviations might be, we will initially model them as uniform on 0 to 100. We will include both standard deviations in our model using a single line of PyMC3 code:

```
sds = pm.Uniform("sds", 0, 100, shape=2)
```

Notice that we specified `shape=2`: we are modeling both σs as a single PyMC3 variable. Note that this does not induce a necessary relationship between the two σs, it is simply for succinctness.

We also need to specify priors on the centers of the clusters. The centers are really the μ parameters in these Normal distributions. Their priors can be modeled by a Normal distribution. Looking at the data, I have an idea where the two centers might be — I would guess somewhere around 120 and 190 respectively, though I am not very confident in these eyeballed estimates. Hence I will set $\mu_0 = 120, \mu_1 = 190$ and $\sigma_0 = \sigma_1 = 10$.

```
In [7]: with model:
            sds = pm.Uniform("sds", 0, 100, shape=2)
            centers = pm.Normal("centers",
                                mu=np.array([120, 190]),
                                sd=np.array([10, 10]),
                                shape=2)

            center_i = pm.Deterministic('center_i', centers[assignment])
            sd_i = pm.Deterministic('sd_i', sds[assignment])

            # and to combine it with the observations:
            observations = pm.Normal("obs", mu=center_i, sd=sd_i, observed=data)

        print("Random assignments: ", assignment.tag.test_value[:4], "...")
        print("Assigned center: ", center_i.tag.test_value[:4], "...")
        print("Assigned standard deviation: ", sd_i.tag.test_value[:4])
```

```
Applied interval-transform to sds and added transformed sds_interval_ to model.
Random assignments:  [0 0 0 0] ...
Assigned center:  [ 120.  120.  120.  120.] ...
```

```
Assigned standard deviation:  [ 50.  50.  50.  50.]
```

Notice how we continue to build the model within the context of `Model()`. This automatically adds the variables that we create to our model. As long as we work within this context we will be working with the same variables that we have already defined.

Similarly, any sampling that we do within the context of `Model()` will be done only on the model whose context in which we are working. We will tell our model to explore the space that we have so far defined by defining the sampling methods, in this case `Metropolis()` for our continuous variables and `ElemwiseCategorical()` for our categorical variable. We will use these sampling methods together to explore the space by using `sample( iterations, step )`, where `iterations` is the number of steps you wish the algorithm to perform and `step` is the way in which you want to handle those steps. We use our combination of `Metropolis()` and `ElemwiseCategorical()` for the `step` and sample 25000 `iterations` below.

```
In [8]: with model:
            step1 = pm.Metropolis(vars=[p, sds, centers])
            step2 = pm.ElemwiseCategorical(vars=[assignment])
            trace = pm.sample(25000, step=[step1, step2])
```

```
 [-------100%-------] 25000 of 25000 in 130.7 sec. | SPS: 191.3 | ETA: 0.0
```

We have stored the paths of all our variables, or "traces", in the `trace` variable. These paths are the routes the unknown parameters (centers, precisions, and p) have taken thus far. The individual path of each variable is indexed by the PyMC3 variable `name` that we gave that variable when defining it within our model. For example, `trace["sds"]` will return a `numpy array` object that we can then index and slice as we would any other `numpy array` object.

```
In [9]: figsize(12.5, 9)
        plt.subplot(311)
        lw = 1
        center_trace = trace["centers"]

        # for pretty colors later in the book.
        colors = ["#348ABD", "#A60628"] if center_trace[-1, 0] > center_trace[-1, 1] \
            else ["#A60628", "#348ABD"]

        plt.plot(center_trace[:, 0], label="trace of center 0", c=colors[0], lw=lw)
        plt.plot(center_trace[:, 1], label="trace of center 1", c=colors[1], lw=lw)
        plt.title("Traces of unknown parameters")
        leg = plt.legend(loc="upper right")
        leg.get_frame().set_alpha(0.7)

        plt.subplot(312)
        std_trace = trace["sds"]
        plt.plot(std_trace[:, 0], label="trace of standard deviation of cluster 0",
             c=colors[0], lw=lw)
        plt.plot(std_trace[:, 1], label="trace of standard deviation of cluster 1",
             c=colors[1], lw=lw)
```

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

plt.subplot(313)
p_trace = trace["p"]
plt.plot(p_trace, label="$p$: frequency of assignment to cluster 0",
     color=colors[0], lw=lw)
plt.xlabel("Steps")
plt.ylim(0, 1)
plt.legend();
```

Notice the following characteristics:

1. The traces converges, not to a single point, but to a *distribution* of possible points. This is *convergence* in an MCMC algorithm.
2. Inference using the first few thousand points is a bad idea, as they are unrelated to the final distribution we are interested in. Thus is it a good idea to discard those samples before using the samples for inference. We call this period before converge the *burn-in period*.
3. The traces appear as a random "walk" around the space, that is, the paths exhibit correlation with previous positions. This is both good and bad. We will always have correlation between current positions and the previous positions, but too much of it means we are not exploring the space well. This will be detailed in the Diagnostics section later in this chapter.

To achieve further convergence, we will perform more MCMC steps. In the pseudo-code algorithm of MCMC above, the only position that matters is the current position (new positions are investigated near the current position), implicitly stored as part of the `trace` object. To continue where we left off, we pass the `trace` that we have already stored into the `sample()` function with the same step value. The values that we have already calculated will not be overwritten. This ensures that our sampling continues where it left off in the same way that it left off.

We will sample the MCMC fifty thousand more times and visualize the progress below:

```
In [10]: with model:
             trace = pm.sample(50000, step=[step1, step2], trace=trace)
```

```
 [-------100%-------] 50000 of 50000 in 215.4 sec. | SPS: 232.2 | ETA: 0.0
```

```
In [11]: figsize(12.5, 4)
         center_trace = trace["centers"][25000:]
         prev_center_trace = trace["centers"][:25000]

         x = np.arange(25000)
         plt.plot(x, prev_center_trace[:, 0], label="previous trace of center 0",
              lw=lw, alpha=0.4, c=colors[1])
         plt.plot(x, prev_center_trace[:, 1], label="previous trace of center 1",
              lw=lw, alpha=0.4, c=colors[0])

         x = np.arange(25000, 75000)
         plt.plot(x, center_trace[:, 0], label="new trace of center 0", lw=lw, c="#348ABD")
         plt.plot(x, center_trace[:, 1], label="new trace of center 1", lw=lw, c="#A60628")

         plt.title("Traces of unknown center parameters")
         leg = plt.legend(loc="upper right")
         leg.get_frame().set_alpha(0.8)
         plt.xlabel("Steps");
```

Cluster Investigation We have not forgotten our main challenge: identify the clusters. We have determined posterior distributions for our unknowns. We plot the posterior distributions of the center and standard deviation variables below:

```
In [12]: figsize(11.0, 4)
         std_trace = trace["sds"][25000:]
         prev_std_trace = trace["sds"][:25000]

         _i = [1, 2, 3, 4]
         for i in range(2):
             plt.subplot(2, 2, _i[2 * i])
             plt.title("Posterior of center of cluster %d" % i)
             plt.hist(center_trace[:, i], color=colors[i], bins=30,
                      histtype="stepfilled")

             plt.subplot(2, 2, _i[2 * i + 1])
             plt.title("Posterior of standard deviation of cluster %d" % i)
             plt.hist(std_trace[:, i], color=colors[i], bins=30,
                      histtype="stepfilled")
             # plt.autoscale(tight=True)

         plt.tight_layout()
```

The MCMC algorithm has proposed that the most likely centers of the two clusters are near 120 and 200 respectively. Similar inference can be applied to the standard deviation.

We are also given the posterior distributions for the labels of the data point, which is present in `trace["assignment"]`. Below is a visualization of this. The y-axis represents a subsample of the posterior labels for each data point. The x-axis are the sorted values of the data points. A red square is an assignment to cluster 1, and a blue square is an assignment to cluster 0.

```
In [13]: import matplotlib as mpl
         figsize(12.5, 4.5)
         plt.cmap = mpl.colors.ListedColormap(colors)
         plt.imshow(trace["assignment"][::400, np.argsort(data)],
```

```
        cmap=plt.cmap, aspect=.4, alpha=.9)
plt.xticks(np.arange(0, data.shape[0], 40),
        ["%.2f" % s for s in np.sort(data)[::40]])
plt.ylabel("posterior sample")
plt.xlabel("value of $i$th data point")
plt.title("Posterior labels of data points");
```

Looking at the above plot, it appears that the most uncertainty is between 150 and 170. The above plot slightly misrepresents things, as the x-axis is not a true scale (it displays the value of the ith sorted data point.) A more clear diagram is below, where we have estimated the *frequency* of each data point belonging to the labels 0 and 1.

```
In [14]: cmap = mpl.colors.LinearSegmentedColormap.from_list("BMH", colors)
         assign_trace = trace["assignment"]
         plt.scatter(data, 1 - assign_trace.mean(axis=0), cmap=cmap,
                 c=assign_trace.mean(axis=0), s=50)
         plt.ylim(-0.05, 1.05)
         plt.xlim(35, 300)
         plt.title("Probability of data point belonging to cluster 0")
         plt.ylabel("probability")
         plt.xlabel("value of data point");
```

Even though we modeled the clusters using Normal distributions, we didn't get just a single Normal distribution that *best* fits the data (whatever our definition of best is), but a distribution of values for the Normal's parameters. How can we choose just a single pair of values for the mean and variance and determine a *sorta-best-fit* gaussian?

One quick and dirty way (which has nice theoretical properties we will see in Chapter 5), is to use the *mean* of the posterior distributions. Below we overlay the Normal density functions, using the mean of the posterior distributions as the chosen parameters, with our observed data:

```
In [15]: norm = stats.norm
         x = np.linspace(20, 300, 500)
         posterior_center_means = center_trace.mean(axis=0)
         posterior_std_means = std_trace.mean(axis=0)
         posterior_p_mean = trace["p"].mean()

         plt.hist(data, bins=20, histtype="step", normed=True, color="k",
              lw=2, label="histogram of data")
         y = posterior_p_mean * norm.pdf(x, loc=posterior_center_means[0],
                                         scale=posterior_std_means[0])
         plt.plot(x, y, label="Cluster 0 (using posterior-mean parameters)", lw=3)
         plt.fill_between(x, y, color=colors[1], alpha=0.3)

         y = (1 - posterior_p_mean) * norm.pdf(x, loc=posterior_center_means[1],
                                               scale=posterior_std_means[1])
         plt.plot(x, y, label="Cluster 1 (using posterior-mean parameters)", lw=3)
         plt.fill_between(x, y, color=colors[0], alpha=0.3)

         plt.legend(loc="upper left")
         plt.title("Visualizing Clusters using posterior-mean parameters");
```

1.1.5 Important: Don't mix posterior samples

In the above example, a possible (though less likely) scenario is that cluster 0 has a very large standard deviation, and cluster 1 has a small standard deviation. This would still satisfy the evidence, albeit less so than our original inference. Alternatively, it would be incredibly unlikely for *both* distributions to have a small standard deviation, as the data does not support this hypothesis at all. Thus the two standard deviations are *dependent* on each other: if one is small, the other must be large. In fact, *all* the unknowns are related in a similar manner. For example, if a standard deviation is large, the mean has a wider possible space of realizations. Conversely, a small standard deviation restricts the mean to a small area.

During MCMC, we are returned vectors representing samples from the unknown posteriors. Elements of different vectors cannot be used together, as this would break the above logic: perhaps a sample has returned that cluster 1 has a small standard deviation, hence all the other variables in that sample would incorporate that and be adjusted accordingly. It is easy to avoid this problem though, just make sure you are indexing traces correctly.

Another small example to illustrate the point. Suppose two variables, x and y, are related by $x + y = 10$. We model x as a Normal random variable with mean 4 and explore 500 samples.

```
In [16]: import pymc3 as pm

         with pm.Model() as model:
             x = pm.Normal("x", mu=4, tau=10)
             y = pm.Deterministic("y", 10 - x)

             trace_2 = pm.sample(10000, pm.Metropolis())

         plt.plot(trace_2["x"])
         plt.plot(trace_2["y"])
         plt.title("Displaying (extreme) case of dependence between unknowns");
```

```
[-------100%-------] 10000 of 10000 in 0.9 sec. | SPS: 11550.9 | ETA: 0.0
```

As you can see, the two variables are not unrelated, and it would be wrong to add the ith sample of x to the jth sample of y, unless $i = j$.

Returning to Clustering: Prediction The above clustering can be generalized to k clusters. Choosing $k = 2$ allowed us to visualize the MCMC better, and examine some very interesting plots.

What about prediction? Suppose we observe a new data point, say $x = 175$, and we wish to label it to a cluster. It is foolish to simply assign it to the *closer* cluster center, as this ignores the standard deviation of the clusters, and we have seen from the plots above that this consideration is very important. More formally: we are interested in the *probability* (as we cannot be certain about labels) of assigning $x = 175$ to cluster 1. Denote the assignment of x as L_x, which is equal to 0 or 1, and we are interested in $P(L_x = 1 \mid x = 175)$.

A naive method to compute this is to re-run the above MCMC with the additional data point appended. The disadvantage with this method is that it will be slow to infer for each novel data point. Alternatively, we can try a *less precise*, but much quicker method.

We will use Bayes Theorem for this. If you recall, Bayes Theorem looks like:

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

In our case, A represents $L_x = 1$ and X is the evidence we have: we observe that $x = 175$. For a particular sample set of parameters for our posterior distribution, $(\mu_0, \sigma_0, \mu_1, \sigma_1, p)$, we are interested in asking "Is the probability that x is in cluster 1 *greater* than the probability it is in cluster 0?", where the probability is dependent on the chosen parameters.

$$P(L_x = 1|x = 175) > P(L_x = 0|x = 175) \tag{1}$$

$$\tag{2}$$

$$\frac{P(x = 175|L_x = 1)P(L_x = 1)}{P(x = 175)} > \frac{P(x = 175|L_x = 0)P(L_x = 0)}{P(x = 175)} \tag{3}$$

As the denominators are equal, they can be ignored (and good riddance, because computing the quantity $P(x = 175)$ can be difficult).

$$P(x = 175|L_x = 1)P(L_x = 1) > P(x = 175|L_x = 0)P(L_x = 0)$$

```
In [17]: norm_pdf = stats.norm.pdf
         p_trace = trace["p"][25000:]
         prev_p_trace = trace["p"][:25000]
         x = 175

         v = p_trace * norm_pdf(x, loc=center_trace[:, 0], scale=std_trace[:, 0]) > \
             (1 - p_trace) * norm_pdf(x, loc=center_trace[:, 1], scale=std_trace[:, 1])

         print("Probability of belonging to cluster 1:", v.mean())
```

```
Probability of belonging to cluster 1: 0.01062
```

Giving us a probability instead of a label is a very useful thing. Instead of the naive

```
L = 1 if prob > 0.5 else 0
```

we can optimize our guesses using a *loss function*, which the entire fifth chapter is devoted to.

1.1.6 Using `MAP` to improve convergence

If you ran the above example yourself, you may have noticed that our results were not consistent: perhaps your cluster division was more scattered, or perhaps less scattered. The problem is that our traces are a function of the *starting values* of the MCMC algorithm.

It can be mathematically shown that letting the MCMC run long enough, by performing many steps, the algorithm *should forget its initial position*. In fact, this is what it means to say the MCMC converged (in practice though we can never achieve total convergence). Hence if we observe different posterior analysis, it is likely because our MCMC has not fully converged yet, and we should not use samples from it yet (we should use a larger burn-in period).

In fact, poor starting values can prevent any convergence, or significantly slow it down. Ideally, we would like to have the chain start at the *peak* of our landscape, as this is exactly where the posterior distributions exist. Hence, if we started at the "peak", we could avoid a lengthy burn-in period and incorrect inference. Generally, we call this "peak" the *maximum a posterior* or, more simply, the *MAP*.

Of course, we do not know where the MAP is. PyMC3 provides a function that will approximate, if not find, the MAP location. In the PyMC3 main namespace is the `find_MAP` function. If you call this function within the context of `Model()`, it will calculate the MAP which you can then pass to `pm.sample()` as a `start` parameter.

```
start = pm.find_MAP()
trace = pm.sample(2000, step=pm.Metropolis, start=start)
```

The `find_MAP()` function has the flexibility of allowing the user to choose which optimization algorithm to use (after all, this is a optimization problem: we are looking for the values that maximize our landscape), as not all optimization algorithms are created equal. The default optimization algorithm in function call is the Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm to find the maximum of the log-posterior. As an alternative, you can use other optimization algorithms from the `scipy.optimize` module. For example, you can use Powell's Method, a favourite of PyMC blogger Abraham Flaxman [1], by calling `find_MAP(fmin=scipy.optimize.fmin_powell)`. The default works well enough, but if convergence is slow or not guaranteed, feel free to experiment with Powell's method or the other algorithms available.

The MAP can also be used as a solution to the inference problem, as mathematically it is the *most likely* value for the unknowns. But as mentioned earlier in this chapter, this location ignores the uncertainty and doesn't return a distribution.

Speaking of the burn-in period It is still a good idea to decide on a burn-in period, even if we are using `find_MAP()` prior to sampling, just to be safe. We can no longer automatically discard sample with a `burn` parameter in the `sample()` function as we could in PyMC2, but it is easy enough to simply discard the beginning section of the trace just through array slicing. As one does not know when the chain has fully converged, a good rule of thumb is to discard the first *half* of your samples, sometimes up to 90% of the samples for longer runs. To continue the clustering example from above, the new code would look something like:

```
with pm.Model() as model:
    start = pm.find_MAP()

    step = pm.Metropolis()
```

```
    trace = pm.sample(100000, step=step, start=start)

burned_trace = trace[50000:]
```

1.2 Diagnosing Convergence

1.2.1 Autocorrelation

Autocorrelation is a measure of how related a series of numbers is with itself. A measurement of 1.0 is perfect positive autocorrelation, 0 no autocorrelation, and -1 is perfect negative correlation. If you are familiar with standard *correlation*, then autocorrelation is just how correlated a series, x_τ, at time t is with the series at time $t-k$:

$$R(k) = Corr(x_t, x_{t-k})$$

For example, consider the two series:

$$x_t \sim \text{Normal}(0,1), \quad x_0 = 0$$
$$y_t \sim \text{Normal}(y_{t-1},1), \quad y_0 = 0$$

which have example paths like:

```
In [18]: figsize(12.5, 4)

         import pymc3 as pm
         x_t = np.random.normal(0, 1, 200)
         x_t[0] = 0
         y_t = np.zeros(200)
         for i in range(1, 200):
             y_t[i] = np.random.normal(y_t[i - 1], 1)

         plt.plot(y_t, label="$y_t$", lw=3)
         plt.plot(x_t, label="$x_t$", lw=3)
         plt.xlabel("time, $t$")
         plt.legend();
```

One way to think of autocorrelation is "If I know the position of the series at time s, can it help me know where I am at time t?" In the series x_t, the answer is No. By construction, x_t are random variables. If I told you that $x_2 = 0.5$, could you give me a better guess about x_3? No.

On the other hand, y_t is autocorrelated. By construction, if I knew that $y_2 = 10$, I can be very confident that y_3 will not be very far from 10. Similarly, I can even make a (less confident guess) about y_4: it will probably not be near 0 or 20, but a value of 5 is not too unlikely. I can make a similar argument about y_5, but again, I am less confident. Taking this to it's logical conclusion, we must concede that as k, the lag between time points, increases the autocorrelation decreases. We can visualize this:

```
In [19]: def autocorr(x):
             # from http://tinyurl.com/afz57c4
             result = np.correlate(x, x, mode='full')
             result = result / np.max(result)
             return result[result.size // 2:]

         colors = ["#348ABD", "#A60628", "#7A68A6"]

         x = np.arange(1, 200)
         plt.bar(x, autocorr(y_t)[1:], width=1, label="$y_t$",
                 edgecolor=colors[0], color=colors[0])
         plt.bar(x, autocorr(x_t)[1:], width=1, label="$x_t$",
                 color=colors[1], edgecolor=colors[1])

         plt.legend(title="Autocorrelation")
         plt.ylabel("measured correlation \nbetween $y_t$ and $y_{t-k}$.")
         plt.xlabel("k (lag)")
         plt.title("Autocorrelation plot of $y_t$ and $x_t$ for differing $k$ lags.");
```

Notice that as k increases, the autocorrelation of y_t decreases from a very high point. Compare with the autocorrelation of x_t which looks like noise (which it really is), hence we can conclude no autocorrelation exists in this series.

How does this relate to MCMC convergence? By the nature of the MCMC algorithm, we will always be returned samples that exhibit autocorrelation (this is because of the step `from your current position, move to a position near you`).

A chain that is [Isn't meandering exploring?] exploring the space well will exhibit very high autocorrelation. Visually, if the trace seems to meander like a river, and not settle down, the chain will have high autocorrelation.

This does not imply that a converged MCMC has low autocorrelation. Hence low autocorrelation is not necessary for convergence, but it is sufficient. PyMC3 has a built-in autocorrelation plotting function in the `plots` module.

1.2.2 Thinning

Another issue can arise if there is high-autocorrelation between posterior samples. Many post-processing algorithms require samples to be *independent* of each other. This can be solved, or at least reduced, by only returning to the user every nth sample, thus removing some autocorrelation. Below we perform an autocorrelation plot for y_t with differing levels of thinning:

```
In [20]: max_x = 200 // 3 + 1
         x = np.arange(1, max_x)

         plt.bar(x, autocorr(y_t)[1:max_x], edgecolor=colors[0],
                 label="no thinning", color=colors[0], width=1)
         plt.bar(x, autocorr(y_t[::2])[1:max_x], edgecolor=colors[1],
                 label="keeping every 2nd sample", color=colors[1], width=1)
         plt.bar(x, autocorr(y_t[::3])[1:max_x], width=1, edgecolor=colors[2],
                 label="keeping every 3rd sample", color=colors[2])

         plt.autoscale(tight=True)
         plt.legend(title="Autocorrelation plot for $y_t$", loc="lower left")
         plt.ylabel("measured correlation \nbetween $y_t$ and $y_{t-k}$.")
         plt.xlabel("k (lag)")
         plt.title("Autocorrelation of $y_t$ (no thinning vs. thinning) \
         at differing $k$ lags.");
```

With more thinning, the autocorrelation drops quicker. There is a tradeoff though: higher thinning requires more MCMC iterations to achieve the same number of returned samples. For example, 10 000 samples unthinned is 100 000 with a thinning of 10 (though the latter has less autocorrelation).

What is a good amount of thinning? The returned samples will always exhibit some autocorrelation, regardless of how much thinning is done. So long as the autocorrelation tends to zero, you are probably ok. Typically thinning of more than 10 is not necessary.

1.2.3 `pymc3.plots`

It seems silly to have to manually create histograms, autocorrelation plots and trace plots each time we perform MCMC. The authors of PyMC3 have included a visualization tool for just this purpose.

The `pymc3.plots` module contains a few different plotting functions that you might find useful. For each different plotting function contained therein, you simply pass a `trace` returned from sampling as well as a list, `varnames`, of the variables that you are interested in. This module can provide you with plots of autocorrelation and the posterior distributions of each variable and their traces, among others.

Below we use the tool to plot the centers of the clusters.

```
In [21]: pm.plots.traceplot(trace=trace, varnames=["centers"])
         pm.plots.plot_posterior(trace=trace["centers"][:,0])
         pm.plots.plot_posterior(trace=trace["centers"][:,1])
         pm.plots.autocorrplot(trace=trace, varnames=["centers"]);
```

The first plotting function gives us the posterior density of each unknown in the `centers` variable as well as the `trace` of each. `trace` plot is useful for inspecting that possible "meandering" property that is a result of non-convergence. The density plot gives us an idea of the shape of the distribution of each unknown, but it is better to look at each of them individually.

The second plotting function(s) provides us with a histogram of the samples with a few added features. The text overlay in the center shows us the posterior mean, which is a good summary of posterior distribution. The interval marked by the horizontal black line overlay represents the *95% credible interval*, sometimes called the *highest posterior density interval* and not to be confused with a *95% confidence interval*. We won't get into the latter, but the former can be interpreted as "there is a 95% chance the parameter of interest lies in this interval". When communicating your results to others, it is incredibly important to state this interval. One of our purposes for studying Bayesian methods is to have a clear understanding of our uncertainty in unknowns. Combined with the posterior mean, the 95% credible interval provides a reliable interval to communicate the likely location of the unknown (provided by the mean) *and* the uncertainty (represented by the width of the interval).

The last plots, titled `center_0` and `center_1` are the generated autocorrelation plots, similar to the ones displayed above.

1.3 Useful tips for MCMC

Bayesian inference would be the *de facto* method if it weren't for MCMC's computational difficulties. In fact, MCMC is what turns most users off practical Bayesian inference. Below I present some good heuristics to help convergence and speed up the MCMC engine:

1.3.1 Intelligent starting values

It would be great to start the MCMC algorithm off near the posterior distribution, so that it will take little time to start sampling correctly. We can aid the algorithm by telling where we *think* the posterior distribution will be by specifying the `testval` parameter in the `Stochastic` variable creation. In many cases we can produce a reasonable guess for the parameter. For example, if we have data from a Normal distribution, and we wish to estimate the μ parameter, then a good starting value would be the *mean* of the data.

```
mu = pm.Uniform( "mu", 0, 100, testval = data.mean() )
```

For most parameters in models, there is a frequentist estimate of it. These estimates are a good starting value for our MCMC algorithms. Of course, this is not always possible for some variables, but including as many appropriate initial values is always a good idea. Even if your guesses are wrong, the MCMC will still converge to the proper distribution, so there is little to lose.

This is what using `MAP` tries to do, by giving good initial values to the MCMC. So why bother specifying user-defined values? Well, even giving `MAP` good values will help it find the maximum a-posterior.

Also important, *bad initial values* are a source of major bugs in PyMC3 and can hurt convergence.

Priors If the priors are poorly chosen, the MCMC algorithm may not converge, or atleast have difficulty converging. Consider what may happen if the prior chosen does not even contain the true parameter: the prior assigns 0 probability to the unknown, hence the posterior will assign 0 probability as well. This can cause pathological results.

For this reason, it is best to carefully choose the priors. Often, lack of covergence or evidence of samples crowding to boundaries implies something is wrong with the chosen priors (see *Folk Theorem of Statistical Computing* below).

Covariance matrices and eliminating parameters

1.3.2 The Folk Theorem of Statistical Computing

> *If you are having computational problems, probably your model is wrong.*

1.4 Conclusion

PyMC3 provides a very strong backend to performing Bayesian inference, mostly because it has abstracted the inner mechanics of MCMC from the user. Despite this, some care must be applied to ensure your inference is not being biased by the iterative nature of MCMC.

1.4.1 References

1. Flaxman, Abraham. "Powell's Methods for Maximization in PyMC." Healthy Algorithms. N.p., 9 02 2012. Web. 28 Feb 2013. http://healthyalgorithms.com/2012/02/09/powells-method-for-maximization-in-pymc/.

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 object>
```

In [ ]:

Ch4_LawOfLargeNumbers_PyMC2

January 4, 2018

1 Chapter 4

1.1 The greatest theorem never told

This chapter focuses on an idea that is always bouncing around our minds, but is rarely made explicit outside books devoted to statistics. In fact, we've been using this simple idea in every example thus far.

1.1.1 The Law of Large Numbers

Let Z_i be N independent samples from some probability distribution. According to *the Law of Large numbers*, so long as the expected value $E[Z]$ is finite, the following holds,

$$\frac{1}{N}\sum_{i=1}^{N} Z_i \to E[Z], \quad N \to \infty.$$

In words:

> The average of a sequence of random variables from the same distribution converges to the expected value of that distribution.

This may seem like a boring result, but it will be the most useful tool you use.

1.1.2 Intuition

If the above Law is somewhat surprising, it can be made clearer by examining a simple example.

Consider a random variable Z that can take only two values, c_1 and c_2. Suppose we have a large number of samples of Z, denoting a specific sample Z_i. The Law says that we can approximate the expected value of Z by averaging over all samples. Consider the average:

$$\frac{1}{N}\sum_{i=1}^{N} Z_i$$

By construction, Z_i can only take on c_1 or c_2, hence we can partition the sum over these two values:

$$\frac{1}{N}\sum_{i=1}^{N} Z_i = \frac{1}{N}\Big(\sum_{Z_i=c_1} c_1 + \sum_{Z_i=c_2} c_2\Big) \tag{1}$$

$$\tag{2}$$

$$= c_1 \sum_{Z_i=c_1} \frac{1}{N} + c_2 \sum_{Z_i=c_2} \frac{1}{N} \tag{3}$$

$$\tag{4}$$

$$= c_1 \times \text{ (approximate frequency of } c_1) \tag{5}$$

$$\tag{6}$$

$$+ c_2 \times \text{ (approximate frequency of } c_2) \tag{7}$$

$$\tag{8}$$

$$\approx c_1 \times P(Z = c_1) + c_2 \times P(Z = c_2) \tag{9}$$

$$\tag{10}$$

$$= E[Z] \tag{11}$$

Equality holds in the limit, but we can get closer and closer by using more and more samples in the average. This Law holds for almost *any distribution*, minus some important cases we will encounter later.

Example

Below is a diagram of the Law of Large numbers in action for three different sequences of Poisson random variables.

We sample `sample_size = 100000` Poisson random variables with parameter $\lambda = 4.5$. (Recall the expected value of a Poisson random variable is equal to its parameter.) We calculate the average for the first n samples, for $n = 1$ to `sample_size`.

```
In [6]: %matplotlib inline
        import numpy as np
        from IPython.core.pylabtools import figsize
        import matplotlib.pyplot as plt

        figsize(12.5, 5)
        import pymc as pm

        sample_size = 100000
        expected_value = lambda_ = 4.5
        poi = pm.rpoisson
        N_samples = range(1, sample_size, 100)

        for k in range(3):
```

```
    samples = poi(lambda_, size=sample_size)

    partial_average = [samples[:i].mean() for i in N_samples]

    plt.plot(N_samples, partial_average, lw=1.5, label="average \
of  $n$ samples; seq. %d" % k)


plt.plot(N_samples, expected_value * np.ones_like(partial_average),
         ls="--", label="true expected value", c="k")

plt.ylim(4.35, 4.65)
plt.title("Convergence of the average of \n random variables to its \
expected value")
plt.ylabel("average of $n$ samples")
plt.xlabel("# of samples, $n$")
plt.legend();
```

Looking at the above plot, it is clear that when the sample size is small, there is greater variation in the average (compare how *jagged and jumpy* the average is initially, then *smooths* out). All three paths *approach* the value 4.5, but just flirt with it as N gets large. Mathematicians and statistician have another name for *flirting*: convergence.

Another very relevant question we can ask is *how quickly am I converging to the expected value?* Let's plot something new. For a specific N, let's do the above trials thousands of times and compute how far away we are from the true expected value, on average. But wait — *compute on average*? This is simply the law of large numbers again! For example, we are interested in, for a specific N, the quantity:

$$D(N) = \sqrt{E\left[\left(\frac{1}{N}\sum_{i=1}^{N} Z_i - 4.5\right)^2\right]}$$

The above formulae is interpretable as a distance away from the true value (on average), for some N. (We take the square root so the dimensions of the above quantity and our random variables are the same). As the above is an expected value, it can be approximated using the law of large numbers: instead of averaging Z_i, we calculate the following multiple times and average them:

$$Y_k = \left(\frac{1}{N} \sum_{i=1}^{N} Z_i - 4.5 \right)^2$$

By computing the above many, N_y, times (remember, it is random), and averaging them:

$$\frac{1}{N_Y} \sum_{k=1}^{N_Y} Y_k \to E[Y_k] = E\left[\left(\frac{1}{N} \sum_{i=1}^{N} Z_i - 4.5 \right)^2 \right]$$

Finally, taking the square root:

$$\sqrt{\frac{1}{N_Y} \sum_{k=1}^{N_Y} Y_k} \approx D(N)$$

In [2]:
```
figsize(12.5, 4)

N_Y = 250  # use this many to approximate D(N)
N_array = np.arange(1000, 50000, 2500)  # use this many samples in the approx. to the
D_N_results = np.zeros(len(N_array))

lambda_ = 4.5
expected_value = lambda_  # for X ~ Poi(lambda) , E[ X ] = lambda


def D_N(n):
    """
    This function approx. D_n, the average variance of using n samples.
    """
    Z = poi(lambda_, size=(n, N_Y))
    average_Z = Z.mean(axis=0)
    return np.sqrt(((average_Z - expected_value) ** 2).mean())


for i, n in enumerate(N_array):
    D_N_results[i] = D_N(n)


plt.xlabel("$N$")
plt.ylabel("expected squared-distance from true value")
plt.plot(N_array, D_N_results, lw=3,
         label="expected distance between\n\
expected value and \naverage of $N$ random variables.")
```

```
plt.plot(N_array, np.sqrt(expected_value) / np.sqrt(N_array), lw=2, ls="--",
         label=r"$\frac{\sqrt{\lambda}}{\sqrt{N}}$")
plt.legend()
plt.title("How 'fast' is the sample average converging? ");
```

As expected, the expected distance between our sample average and the actual expected value shrinks as N grows large. But also notice that the *rate* of convergence decreases, that is, we need only 10 000 additional samples to move from 0.020 to 0.015, a difference of 0.005, but *20 000* more samples to again decrease from 0.015 to 0.010, again only a 0.005 decrease.

It turns out we can measure this rate of convergence. Above I have plotted a second line, the function $\sqrt{\lambda}/\sqrt{N}$. This was not chosen arbitrarily. In most cases, given a sequence of random variable distributed like Z, the rate of converge to $E[Z]$ of the Law of Large Numbers is

$$\frac{\sqrt{Var(Z)}}{\sqrt{N}}$$

This is useful to know: for a given large N, we know (on average) how far away we are from the estimate. On the other hand, in a Bayesian setting, this can seem like a useless result: Bayesian analysis is OK with uncertainty so what's the *statistical* point of adding extra precise digits? Though drawing samples can be so computationally cheap that having a *larger* N is fine too.

1.1.3 How do we compute $Var(Z)$ though?

The variance is simply another expected value that can be approximated! Consider the following, once we have the expected value (by using the Law of Large Numbers to estimate it, denote it μ), we can estimate the variance:

$$\frac{1}{N}\sum_{i=1}^{N}(Z_i - \mu)^2 \rightarrow E[\,(Z-\mu)^2\,] = Var(Z)$$

1.1.4 Expected values and probabilities

There is an even less explicit relationship between expected value and estimating probabilities. Define the *indicator function*

$$\mathbb{1}_A(x) = \begin{cases} 1 & x \in A \\ \\ 0 & else \end{cases}$$

Then, by the law of large numbers, if we have many samples X_i, we can estimate the probability of an event A, denoted $P(A)$, by:

$$\frac{1}{N}\sum_{i=1}^{N} \mathbb{1}_A(X_i) \rightarrow E[\mathbb{1}_A(X)] = P(A)$$

Again, this is fairly obvious after a moments thought: the indicator function is only 1 if the event occurs, so we are summing only the times the event occurs and dividing by the total number of trials (consider how we usually approximate probabilities using frequencies). For example, suppose we wish to estimate the probability that a $Z \sim Exp(.5)$ is greater than 10, and we have many samples from a $Exp(.5)$ distribution.

$$P(Z > 10) = \frac{1}{N}\sum_{i=1}^{N} \mathbb{1}_{z>10}(Z_i)$$

```
In [3]: import pymc as pm
        N = 10000
        print(np.mean([pm.rexponential(0.5) > 10 for i in range(N)]))
```

```
0.0061
```

1.1.5 What does this all have to do with Bayesian statistics?

Point estimates, to be introduced in the next chapter, in Bayesian inference are computed using expected values. In more analytical Bayesian inference, we would have been required to evaluate complicated expected values represented as multi-dimensional integrals. No longer. If we can sample from the posterior distribution directly, we simply need to evaluate averages. Much easier. If accuracy is a priority, plots like the ones above show how fast you are converging. And if further accuracy is desired, just take more samples from the posterior.

When is enough enough? When can you stop drawing samples from the posterior? That is the practitioners decision, and also dependent on the variance of the samples (recall from above a high variance means the average will converge slower).

We also should understand when the Law of Large Numbers fails. As the name implies, and comparing the graphs above for small N, the Law is only true for large sample sizes. Without this, the asymptotic result is not reliable. Knowing in what situations the Law fails can give us *confidence in how unconfident we should be*. The next section deals with this issue.

1.2 The Disorder of Small Numbers

The Law of Large Numbers is only valid as N gets *infinitely* large: never truly attainable. While the law is a powerful tool, it is foolhardy to apply it liberally. Our next example illustrates this.

Example: Aggregated geographic data Often data comes in aggregated form. For instance, data may be grouped by state, county, or city level. Of course, the population numbers vary per geographic area. If the data is an average of some characteristic of each the geographic areas, we must be conscious of the Law of Large Numbers and how it can *fail* for areas with small populations.

We will observe this on a toy dataset. Suppose there are five thousand counties in our dataset. Furthermore, population number in each state are uniformly distributed between 100 and 1500. The way the population numbers are generated is irrelevant to the discussion, so we do not justify this. We are interested in measuring the average height of individuals per county. Unbeknownst to us, height does **not** vary across county, and each individual, regardless of the county he or she is currently living in, has the same distribution of what their height may be:

$$\text{height} \sim \text{Normal}(150, 15)$$

We aggregate the individuals at the county level, so we only have data for the *average in the county*. What might our dataset look like?

```
In [4]: figsize(12.5, 4)
        std_height = 15
        mean_height = 150

        n_counties = 5000
        pop_generator = pm.rdiscrete_uniform
        norm = pm.rnormal

        # generate some artificial population numbers
        population = pop_generator(100, 1500, size=n_counties)

        average_across_county = np.zeros(n_counties)
        for i in range(n_counties):
            # generate some individuals and take the mean
            average_across_county[i] = norm(mean_height, 1. / std_height ** 2,
                                            size=population[i]).mean()

        # located the counties with the apparently most extreme average heights.
        i_min = np.argmin(average_across_county)
        i_max = np.argmax(average_across_county)

        # plot population size vs. recorded average
        plt.scatter(population, average_across_county, alpha=0.5, c="#7A68A6")
        plt.scatter([population[i_min], population[i_max]],
                    [average_across_county[i_min], average_across_county[i_max]],
                    s=60, marker="o", facecolors="none",
                    edgecolors="#A60628", linewidths=1.5,
                    label="extreme heights")

        plt.xlim(100, 1500)
        plt.title("Average height vs. County Population")
        plt.xlabel("County Population")
```

```
plt.ylabel("Average height in county")
plt.plot([100, 1500], [150, 150], color="k", label="true expected \
height", ls="--")
plt.legend(scatterpoints=1);
```

What do we observe? *Without accounting for population sizes* we run the risk of making an enormous inference error: if we ignored population size, we would say that the county with the shortest and tallest individuals have been correctly circled. But this inference is wrong for the following reason. These two counties do *not* necessarily have the most extreme heights. The error results from the calculated average of smaller populations not being a good reflection of the true expected value of the population (which in truth should be $\mu = 150$). The sample size/population size/N, whatever you wish to call it, is simply too small to invoke the Law of Large Numbers effectively.

We provide more damning evidence against this inference. Recall the population numbers were uniformly distributed over 100 to 1500. Our intuition should tell us that the counties with the most extreme population heights should also be uniformly spread over 100 to 1500, and certainly independent of the county's population. Not so. Below are the population sizes of the counties with the most extreme heights.

```
In [5]: print("Population sizes of 10 'shortest' counties: ")
        print(population[np.argsort(average_across_county)[:10]])
        print("\nPopulation sizes of 10 'tallest' counties: ")
        print(population[np.argsort(-average_across_county)[:10]])
```

```
Population sizes of 10 'shortest' counties:
[100 103 138 182 194 100 118 161 156 186]

Population sizes of 10 'tallest' counties:
[100 147 132 193 270 130 414 101 150 109]
```

Not at all uniform over 100 to 1500. This is an absolute failure of the Law of Large Numbers.

Example: Kaggle's *U.S. Census Return Rate Challenge* Below is data from the 2010 US census, which partitions populations beyond counties to the level of block groups (which are aggregates of city blocks or equivalents). The dataset is from a Kaggle machine learning competition some colleagues and I participated in. The objective was to predict the census letter mail-back rate of a group block, measured between 0 and 100, using census variables (median income, number of females in the block-group, number of trailer parks, average number of children etc.). Below we plot the census mail-back rate versus block group population:

```
In [6]: figsize(12.5, 6.5)
        data = np.genfromtxt("./data/census_data.csv", skip_header=1,
                             delimiter=",")
        plt.scatter(data[:, 1], data[:, 0], alpha=0.5, c="#7A68A6")
        plt.title("Census mail-back rate vs Population")
        plt.ylabel("Mail-back rate")
        plt.xlabel("population of block-group")
        plt.xlim(-100, 15e3)
        plt.ylim(-5, 105)

        i_min = np.argmin(data[:, 0])
        i_max = np.argmax(data[:, 0])

        plt.scatter([data[i_min, 1], data[i_max, 1]],
                    [data[i_min, 0], data[i_max, 0]],
                    s=60, marker="o", facecolors="none",
                    edgecolors="#A60628", linewidths=1.5,
                    label="most extreme points")

        plt.legend(scatterpoints=1);
```

The above is a classic phenomenon in statistics. I say *classic* referring to the "shape" of the scatter plot above. It follows a classic triangular form, that tightens as we increase the sample size (as the Law of Large Numbers becomes more exact).

I am perhaps overstressing the point and maybe I should have titled the book *"You don't have big data problems!"*, but here again is an example of the trouble with *small datasets*, not big ones. Simply, small datasets cannot be processed using the Law of Large Numbers. Compare with applying the Law without hassle to big datasets (ex. big data). I mentioned earlier that paradoxically big data prediction problems are solved by relatively simple algorithms. The paradox is partially resolved by understanding that the Law of Large Numbers creates solutions that are *stable*, i.e. adding or subtracting a few data points will not affect the solution much. On the other hand, adding or removing data points to a small dataset can create very different results.

For further reading on the hidden dangers of the Law of Large Numbers, I would highly recommend the excellent manuscript The Most Dangerous Equation.

Example: How to order Reddit submissions You may have disagreed with the original statement that the Law of Large numbers is known to everyone, but only implicitly in our subconscious decision making. Consider ratings on online products: how often do you trust an average 5-star rating if there is only 1 reviewer? 2 reviewers? 3 reviewers? We implicitly understand that with such few reviewers that the average rating is **not** a good reflection of the true value of the product.

This has created flaws in how we sort items, and more generally, how we compare items. Many people have realized that sorting online search results by their rating, whether the objects be books, videos, or online comments, return poor results. Often the seemingly top videos or comments have perfect ratings only from a few enthusiastic fans, and truly more quality videos or comments are hidden in later pages with *falsely-substandard* ratings of around 4.8. How can we correct this?

Consider the popular site Reddit (I purposefully did not link to the website as you would never come back). The site hosts links to stories or images, and a very popular part of the site are the comments associated with each link. Redditors can vote up or down on each submission (called upvotes and downvotes). Reddit, by default, will sort submissions to a given subreddit by Hot, that is, the submissions that have the most upvotes recently.

How would you determine which submissions are the best? There are a number of ways to achieve this:

1. *Popularity*: A submission is considered good if it has many upvotes. A problem with this model is that a submission with hundreds of upvotes, but thousands of downvotes. While being very popular, the submission is likely more controversial than best.
2. *Difference*: Using the *difference* of upvotes and downvotes. This solves the above problem, but fails when we consider the temporal nature of submission. Depending on when a submission is posted, the website may be experiencing high or low traffic. The difference method will bias the Top submissions to be the those made during high traffic periods, which have accumulated more upvotes than submissions that were not so graced, but are not necessarily the best.
3. *Time adjusted*: Consider using Difference divided by the age of the submission. This creates a *rate*, something like *difference per second*, or *per minute*. An immediate counter-example is, if we use per second, a 1 second old submission with 1 upvote would be better than a 100 second old submission with 99 upvotes. One can avoid this by only considering at least t second old submission. But what is a good t value? Does this mean no submission younger

than t is good? We end up comparing unstable quantities with stable quantities (young vs. old submissions).

4. *Ratio*: Rank submissions by the ratio of upvotes to total number of votes (upvotes plus downvotes). This solves the temporal issue, such that new submissions who score well can be considered Top just as likely as older submissions, provided they have many upvotes to total votes. The problem here is that a submission with a single upvote (ratio = 1.0) will beat a submission with 999 upvotes and 1 downvote (ratio = 0.999), but clearly the latter submission is *more likely* to be better.

I used the phrase *more likely* for good reason. It is possible that the former submission, with a single upvote, is in fact a better submission than the latter with 999 upvotes. The hesitation to agree with this is because we have not seen the other 999 potential votes the former submission might get. Perhaps it will achieve an additional 999 upvotes and 0 downvotes and be considered better than the latter, though not likely.

What we really want is an estimate of the *true upvote ratio*. Note that the true upvote ratio is not the same as the observed upvote ratio: the true upvote ratio is hidden, and we only observe upvotes vs. downvotes (one can think of the true upvote ratio as "what is the underlying probability someone gives this submission a upvote, versus a downvote"). So the 999 upvote/1 downvote submission probably has a true upvote ratio close to 1, which we can assert with confidence thanks to the Law of Large Numbers, but on the other hand we are much less certain about the true upvote ratio of the submission with only a single upvote. Sounds like a Bayesian problem to me.

One way to determine a prior on the upvote ratio is to look at the historical distribution of upvote ratios. This can be accomplished by scraping Reddit's submissions and determining a distribution. There are a few problems with this technique though:

1. Skewed data: The vast majority of submissions have very few votes, hence there will be many submissions with ratios near the extremes (see the "triangular plot" in the above Kaggle dataset), effectively skewing our distribution to the extremes. One could try to only use submissions with votes greater than some threshold. Again, problems are encountered. There is a tradeoff between number of submissions available to use and a higher threshold with associated ratio precision.
2. Biased data: Reddit is composed of different subpages, called subreddits. Two examples are *r/aww*, which posts pics of cute animals, and *r/politics*. It is very likely that the user behaviour towards submissions of these two subreddits are very different: visitors are likely to be more friendly and affectionate in the former, and would therefore upvote submissions more, compared to the latter, where submissions are likely to be controversial and disagreed upon. Therefore not all submissions are the same.

In light of these, I think it is better to use a `Uniform` prior.

With our prior in place, we can find the posterior of the true upvote ratio. The Python script `top_showerthoughts_submissions.py` will scrape the best posts from the `showerthoughts` community on Reddit. This is a text-only community so the title of each post *is* the post. Below is the top post as well as some other sample posts:

```
In [1]: # adding a number to the end of the %run call will get the ith top post.
        %run top_showerthoughts_submissions.py 2
```

```
print("Post contents: \n")
print(top_post)
```

```
Post contents:

Toilet paper should be free and have advertising printed on it.
```

```
In [2]: """
        contents: an array of the text from the last 100 top submissions to a subreddit
        votes: a 2d numpy array of upvotes, downvotes for each submission.
        """
        n_submissions = len(votes)
        submissions = np.random.randint( n_submissions, size=4)
        print("Some Submissions (out of %d total) \n-----------"%n_submissions)
        for i in submissions:
            print('"' + contents[i] + '"')
            print("upvotes/downvotes: ",votes[i,:], "\n")
```

```
Some Submissions (out of 98 total)
-----------
"You will never feel how long time is until you have allergies and snot slowly dripping out of
upvotes/downvotes:  [71  6]

"What if porn ads weren't fake and all these years I've been missing out on these local mums i
upvotes/downvotes:  [43 11]

"You'll be real lucky to find a Penny in Canada."
upvotes/downvotes:  [28 11]

""Smells Like Teen Spirit" is as old to listeners of today as "Yellow Submarine" was to listene
upvotes/downvotes:  [92 10]
```

For a given true upvote ratio p and N votes, the number of upvotes will look like a Binomial random variable with parameters p and N. (This is because of the equivalence between upvote ratio and probability of upvoting versus downvoting, out of N possible votes/trials). We create a function that performs Bayesian inference on p, for a particular comment's upvote/downvote pair.

```
In [4]: import pymc as pm


        def posterior_upvote_ratio(upvotes, downvotes, samples=20000):
            """
            This function accepts the number of upvotes and downvotes a particular submission
            and the number of posterior samples to return to the user. Assumes a uniform prior
            """
```

```
    N = upvotes + downvotes
    upvote_ratio = pm.Uniform("upvote_ratio", 0, 1)
    observations = pm.Binomial("obs", N, upvote_ratio, value=upvotes, observed=True)
    # do the fitting; first do a MAP as it is cheap and useful.
    map_ = pm.MAP([upvote_ratio, observations]).fit()
    mcmc = pm.MCMC([upvote_ratio, observations])
    mcmc.sample(samples, samples / 4)
    return mcmc.trace("upvote_ratio")[:]
```

Below are the resulting posterior distributions.

```
In [5]: figsize(11., 8)
        posteriors = []
        colours = ["#348ABD", "#A60628", "#7A68A6", "#467821", "#CF4457"]
        for i in range(len(submissions)):
            j = submissions[i]
            posteriors.append(posterior_upvote_ratio(votes[j, 0], votes[j, 1]))
            plt.hist(posteriors[i], bins=18, normed=True, alpha=.9,
                     histtype="step", color=colours[i % 5], lw=3,
                     label='(%d up:%d down)\n%s...' % (votes[j, 0], votes[j, 1], contents[j][:5
            plt.hist(posteriors[i], bins=18, normed=True, alpha=.2,
                     histtype="stepfilled", color=colours[i], lw=3, )

        plt.legend(loc="upper left")
        plt.xlim(0, 1)
        plt.title("Posterior distributions of upvote ratios on different submissions");
```

```
---------------------------------------------------------------------------

NameError                                 Traceback (most recent call last)

<ipython-input-5-53899899b283> in <module>()
----> 1 figsize(11., 8)
      2 posteriors = []
      3 colours = ["#348ABD", "#A60628", "#7A68A6", "#467821", "#CF4457"]
      4 for i in range(len(submissions)):
      5     j = submissions[i]


NameError: name 'figsize' is not defined
```

Some distributions are very tight, others have very long tails (relatively speaking), expressing our uncertainty with what the true upvote ratio might be.

1.2.1 Sorting!

We have been ignoring the goal of this exercise: how do we sort the submissions from *best to worst*? Of course, we cannot sort distributions, we must sort scalar numbers. There are many

ways to distill a distribution down to a scalar: expressing the distribution through its expected value, or mean, is one way. Choosing the mean is a bad choice though. This is because the mean does not take into account the uncertainty of distributions.

I suggest using the *95% least plausible value*, defined as the value such that there is only a 5% chance the true parameter is lower (think of the lower bound on the 95% credible region). Below are the posterior distributions with the 95% least-plausible value plotted:

```
In [40]: N = posteriors[0].shape[0]
         lower_limits = []

         for i in range(len(submissions)):
             j = submissions[i]
             plt.hist(posteriors[i], bins=20, normed=True, alpha=.9,
                      histtype="step", color=colours[i], lw=3,
                      label='(%d up:%d down)\n%s...' % (votes[j, 0], votes[j, 1], contents[j][
             plt.hist(posteriors[i], bins=20, normed=True, alpha=.2,
                      histtype="stepfilled", color=colours[i], lw=3, )
             v = np.sort(posteriors[i])[int(0.05 * N)]
             # plt.vlines( v, 0, 15 , color = "k", alpha = 1, linewidths=3 )
             plt.vlines(v, 0, 10, color=colours[i], linestyles="--", linewidths=3)
             lower_limits.append(v)
             plt.legend(loc="upper left")

         plt.legend(loc="upper left")
         plt.title("Posterior distributions of upvote ratios on different submissions");
         order = np.argsort(-np.array(lower_limits))
         print(order, lower_limits)
```

```
[0 1 3 2] [0.95553912986585299, 0.94130501756135543, 0.80681345969724116, 0.88775207639838272]
```

The best submissions, according to our procedure, are the submissions that are *most-likely* to score a high percentage of upvotes. Visually those are the submissions with the 95% least plausible value close to 1.

Why is sorting based on this quantity a good idea? By ordering by the 95% least plausible value, we are being the most conservative with what we think is best. That is, even in the worst case scenario, when we have severely overestimated the upvote ratio, we can be sure the best comments are still on top. Under this ordering, we impose the following very natural properties:

1. given two submissions with the same observed upvote ratio, we will assign the submission with more votes as better (since we are more confident it has a higher ratio).
2. given two submissions with the same number of votes, we still assign the submission with more upvotes as *better*.

1.2.2 But this is too slow for real-time!

I agree, computing the posterior of every submission takes a long time, and by the time you have computed it, likely the data has changed. I delay the mathematics to the appendix, but I suggest using the following formula to compute the lower bound very fast.

$$\frac{a}{a+b} - 1.65\sqrt{\frac{ab}{(a+b)^2(a+b+1)}}$$

where

$$a = 1 + u \tag{12}$$

$$\tag{13}$$

$$b = 1 + d \tag{14}$$

$$\tag{15}$$

$$\tag{16}$$

u is the number of upvotes, and d is the number of downvotes. The formula is a shortcut in Bayesian inference, which will be further explained in Chapter 6 when we discuss priors in more detail.

```
In [41]: def intervals(u, d):
             a = 1. + u
             b = 1. + d
             mu = a / (a + b)
             std_err = 1.65 * np.sqrt((a * b) / ((a + b) ** 2 * (a + b + 1.)))
             return (mu, std_err)

         print("Approximate lower bounds:")
         posterior_mean, std_err = intervals(votes[:, 0], votes[:, 1])
         lb = posterior_mean - std_err
         print(lb)
         print("\n")
         print("Top 40 Sorted according to approximate lower bounds:")
         print("\n")
         order = np.argsort(-lb)
         ordered_contents = []
         for i in order[:40]:
             ordered_contents.append(contents[i])
             print(votes[i, 0], votes[i, 1], contents[i])
             print("-------------")
```

```
Approximate lower bounds:
[ 0.9335036   0.95310536  0.94166971  0.90854227  0.88683909  0.85564276
  0.85607414  0.93758888  0.95697574  0.91015237  0.9112593   0.91305389
  0.91341024  0.83335231  0.87543995  0.87081169  0.92748782  0.90747915
  0.89063214  0.89804044  0.91295322  0.78329196  0.91901344  0.79950031
  0.84776174  0.83540757  0.77406294  0.81391583  0.7296015   0.79338766
  0.82895671  0.85331368  0.81849519  0.72362912  0.83662174  0.81019924
  0.78564811  0.84570434  0.8400282   0.76944053  0.85827725  0.74417233
  0.8189683   0.8027221   0.79190256  0.9033107   0.81639188  0.76627386
  0.8010596   0.63657302  0.62988646  0.75041771  0.85355829  0.84522753
  0.75627191  0.8458571   0.80877728  0.66764706  0.69623887  0.71480224
  0.72921035  0.86797314  0.73955911  0.90742546  0.80364062  0.72331349
  0.79249393  0.72708753  0.81109538  0.66235556  0.80480879  0.72039455
  0.73945971  0.83846154  0.69        0.70597731  0.68175931  0.59412132
  0.6011942   0.73158407  0.69121436  0.68134548  0.87746603  0.79809005
```

```
  0.6296728   0.87152685  0.81814153  0.86498277  0.81018384  0.54207776
  0.6296728   0.74107856  0.53025484  0.71034959  0.80149882  0.85773646
  0.58343356  0.62971097]


Top 40 Sorted according to approximate lower bounds:


586 18 Someone should develop an AI specifically for reading Terms & Conditions and flagging du
-------------
2354 98 Porn is the only industry where it is not only acceptable but standard to separate peop
-------------
1924 101 All polls are biased towards people who are willing to take polls
-------------
949 50 They should charge less for drinks in the drive-thru because you can't refill them.
-------------
3726 238 When I was in elementary school and going through the DARE program, I was positive a g
-------------
164 7 "Noted" is the professional way of saying "K".
-------------
100 4 The best answer to the interview question "What is your greatest weakness?" is "intervie
-------------
267 17 At some point every parent has stopped wiping their child's butt and hoped for the best
-------------
291 19 You've been doing weird cameos in your friends' dreams since kindergarten.
-------------
121 6 Is it really fair to say a person over 85 has heart failure? Technically, that heart has
-------------
523 39 I wonder if America's internet is censored in a similar way that North Korea's is, but w
-------------
539 41 It's surreal to think that the sun and moon and stars we gaze up at are the same objects
-------------
1509 131 Kenny's family is poor because they're always paying for his funeral.
-------------
164 10 Black hair ties are probably the most popular bracelets in the world.
-------------
26 0 Now that I am a parent of multiple children I have realized that my parents were lying thi
-------------
41 1 If I was as careful with my whole paycheck as I am with my last $20 I'd be a whole lot bet
-------------
125 8 Surfing the internet without ads feels like a summer evening without mosquitoes
-------------
157 12 I wonder if Superman ever put a pair of glasses on Lois Lane's dog, and she was like "wh
-------------
1411 157 My life is really like Rihanna's song, "just work work work work work" and the rest of
-------------
19 0 Binoculars are like walkie talkies for the deaf.
-------------
```

```
221 22 I'm honestly slightly concerned how often Reddit commenters make me laugh compared to my
-------------
18 0 Living on the coast is having the window seat of the land you live on.
-------------
188 19 I have not been thankful enough in the last few years that the Black Eyed Peas are no lo
-------------
29 1 Rewatching Mr. Bean, I've realised that the character is an eccentric genius and not a bli
-------------
17 0 Sitting on a cold toilet seat or a warm toilet seat both suck for different reasons.
-------------
54 4 You will never feel how long time is until you have allergies and snot slowly dripping out
-------------
16 0 I sneer at people who read tabloids, but every time I look someone up on Wikipedia the fir
-------------
1485 222 Kid's menus at restaurants should be smaller portions of the same adult dishes at lowe
-------------
1417 212 Eventually once all phones are waterproof we'll be able to push people into pools agai
-------------
35 2 Childhood and adolescence are thinking that no one has ever felt the way you do and that n
-------------
60 5 Myspace is so outdated that jokes about it being outdated has become outdated
-------------
87 9 Yahoo!ő is the RadioShackő of the Internet.
-------------
33 2 People who "tell it like it is" rarely do so to say something nice
-------------
49 4 The world must have been a spookier place altogether when candles and gas lamps were the o
-------------
41 3 Closing your eyes after turning off your alarm is a very dangerous game.
-------------
47 4 As a kid, seeing someone step on a banana peel and not slip was a disappointment.
-------------
23 1 The phonebook was the biggest invasion of privacy that everyone was oddly ok with.
-------------
53 5 I'm actually the most productive when I procrastinate because I'm doing everything I possi
-------------
86 10 "Smells Like Teen Spirit" is as old to listeners of today as "Yellow Submarine" was to li
-------------
240 36 if an ocean didnt stop immigrants from coming to America what makes us think a wall will
-------------
```

We can view the ordering visually by plotting the posterior mean and bounds, and sorting by the lower bound. In the plot below, notice that the left error-bar is sorted (as we suggested this is the best way to determine an ordering), so the means, indicated by dots, do not follow any strong pattern.

```
In [42]: r_order = order[::-1][-40:]
         plt.errorbar(posterior_mean[r_order], np.arange(len(r_order)),
```

```
                xerr=std_err[r_order], capsize=0, fmt="o",
                color="#7A68A6")
plt.xlim(0.3, 1)
plt.yticks(np.arange(len(r_order) - 1, -1, -1), map(lambda x: x[:30].replace("\n", ""
```

In the graphic above, you can see why sorting by mean would be sub-optimal.

1.2.3 Extension to Starred rating systems

The above procedure works well for upvote-downvotes schemes, but what about systems that use star ratings, e.g. 5 star rating systems. Similar problems apply with simply taking the average: an item with two perfect ratings would beat an item with thousands of perfect ratings, but a single sub-perfect rating.

We can consider the upvote-downvote problem above as binary: 0 is a downvote, 1 if an upvote. A N-star rating system can be seen as a more continuous version of above, and we can set n stars rewarded is equivalent to rewarding $\frac{n}{N}$. For example, in a 5-star system, a 2 star rating corresponds to 0.4. A perfect rating is a 1. We can use the same formula as before, but with a, b defined differently:

$$\frac{a}{a+b} - 1.65\sqrt{\frac{ab}{(a+b)^2(a+b+1)}}$$

where

$$a = 1 + S \tag{17}$$

$$\tag{18}$$

$$b = 1 + N - S \tag{19}$$

$$\tag{20}$$

$$\tag{21}$$

where N is the number of users who rated, and S is the sum of all the ratings, under the equivalence scheme mentioned above.

Example: Counting Github stars What is the average number of stars a Github repository has? How would you calculate this? There are over 6 million repositories, so there is more than enough data to invoke the Law of Large numbers. Let's start pulling some data. TODO

1.2.4 Conclusion

While the Law of Large Numbers is cool, it is only true so much as its name implies: with large sample sizes only. We have seen how our inference can be affected by not considering *how the data is shaped*.

1. By (cheaply) drawing many samples from the posterior distributions, we can ensure that the Law of Large Number applies as we approximate expected values (which we will do in the next chapter).

2. Bayesian inference understands that with small sample sizes, we can observe wild randomness. Our posterior distribution will reflect this by being more spread rather than tightly concentrated. Thus, our inference should be correctable.

3. There are major implications of not considering the sample size, and trying to sort objects that are unstable leads to pathological orderings. The method provided above solves this problem.

1.2.5 Appendix

Derivation of sorting comments formula Basically what we are doing is using a Beta prior (with parameters $a = 1, b = 1$, which is a uniform distribution), and using a Binomial likelihood with observations $u, N = u + d$. This means our posterior is a Beta distribution with parameters $a' = 1 + u, b' = 1 + (N - u) = 1 + d$. We then need to find the value, x, such that 0.05 probability is less than x. This is usually done by inverting the CDF (Cumulative Distribution Function), but the CDF of the beta, for integer parameters, is known but is a large sum [3].

We instead use a Normal approximation. The mean of the Beta is $\mu = a'/(a' + b')$ and the variance is

$$\sigma^2 = \frac{a'b'}{(a' + b')^2(a' + b' + 1)}$$

Hence we solve the following equation for x and have an approximate lower bound.

$$0.05 = \Phi\left(\frac{(x-\mu)}{\sigma}\right)$$

Φ being the cumulative distribution for the normal distribution

Exercises 1. How would you estimate the quantity $E[\cos X]$, where $X \sim \text{Exp}(4)$? What about $E[\cos X | X < 1]$, i.e. the expected value *given* we know X is less than 1? Would you need more samples than the original samples size to be equally accurate?

```
In [43]: # Enter code here
         import scipy.stats as stats
         exp = stats.expon(scale=4)
         N = int(1e5)
         X = exp.rvs(N)
         # ...
```

2. The following table was located in the paper "Going for Three: Predicting the Likelihood of Field Goal Success with Logistic Regression" [2]. The table ranks football field-goal kickers by their percent of non-misses. What mistake have the researchers made?

Kicker Careers Ranked by Make Percentage Rank
Kicker
Make %
Number of Kicks
1
Garrett Hartley
87.7
57
2
Matt Stover
86.8
335
3
Robbie Gould
86.2
224
4
Rob Bironas
86.1
223
5
Shayne Graham
85.4
254
...
...

...
51
Dave Rayner
72.2
90
52
Nick Novak
71.9
64
53
Tim Seder
71.0
62
54
Jose Cortez
70.7
75
55
Wade Richey
66.1
56

In August 2013, a popular post on the average income per programmer of different languages was trending. Here's the summary chart: (reproduced without permission, cause when you lie with stats, you gunna get the hammer). What do you notice about the extremes?

Average household income by programming language Language
Average Household Income ($)
Data Points
Puppet
87,589.29
112
Haskell
89,973.82
191
PHP
94,031.19
978
CoffeeScript
94,890.80
435
VimL
94,967.11
532
Shell
96,930.54
979

Lua
96,930.69
101
Erlang
97,306.55
168
Clojure
97,500.00
269
Python
97,578.87
2314
JavaScript
97,598.75
3443
Emacs Lisp
97,774.65
355
C#
97,823.31
665
Ruby
98,238.74
3242
C++
99,147.93
845
CSS
99,881.40
527
Perl
100,295.45
990
C
100,766.51
2120
Go
101,158.01
231
Scala
101,460.91
243
ColdFusion
101,536.70
109
Objective-C
101,801.60
562

Groovy
102,650.86
116
Java
103,179.39
1402
XSLT
106,199.19
123
ActionScript
108,119.47
113

1.2.6 References

1. Wainer, Howard. *The Most Dangerous Equation*. American Scientist, Volume 95.
2. Clarck, Torin K., Aaron W. Johnson, and Alexander J. Stimpson. "Going for Three: Predicting the Likelihood of Field Goal Success with Logistic Regression." (2013): n. page. Web. 20 Feb. 2013.
3. http://en.wikipedia.org/wiki/Beta_function#Incomplete_beta_function


```
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 0x10d46b550>
```

Ch4_LawOfLargeNumbers_PyMC3

January 4, 2018

1 Chapter 4

```
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)
```

1.1 The greatest theorem never told

This chapter focuses on an idea that is always bouncing around our minds, but is rarely made explicit outside books devoted to statistics. In fact, we've been using this simple idea in every example thus far.

1.1.1 The Law of Large Numbers

Let Z_i be N independent samples from some probability distribution. According to *the Law of Large numbers*, so long as the expected value $E[Z]$ is finite, the following holds,

$$\frac{1}{N}\sum_{i=1}^{N} Z_i \rightarrow E[Z], \quad N \rightarrow \infty.$$

In words:

> The average of a sequence of random variables from the same distribution converges to the expected value of that distribution.

This may seem like a boring result, but it will be the most useful tool you use.

1.1.2 Intuition

If the above Law is somewhat surprising, it can be made more clear by examining a simple example.

Consider a random variable Z that can take only two values, c_1 and c_2. Suppose we have a large number of samples of Z, denoting a specific sample Z_i. The Law says that we can approximate the expected value of Z by averaging over all samples. Consider the average:

$$\frac{1}{N}\sum_{i=1}^{N} \; Z_i$$

By construction, Z_i can only take on c_1 or c_2, hence we can partition the sum over these two values:

$$\frac{1}{N}\sum_{i=1}^{N} Z_i = \frac{1}{N}\Big(\sum_{Z_i=c_1} c_1 + \sum_{Z_i=c_2} c_2\Big) \tag{1}$$

$$\tag{2}$$

$$= c_1 \sum_{Z_i=c_1} \frac{1}{N} + c_2 \sum_{Z_i=c_2} \frac{1}{N} \tag{3}$$

$$\tag{4}$$

$$= c_1 \times \text{ (approximate frequency of } c_1) \tag{5}$$

$$\tag{6}$$

$$+ c_2 \times \text{ (approximate frequency of } c_2) \tag{7}$$

$$\tag{8}$$

$$\approx c_1 \times P(Z = c_1) + c_2 \times P(Z = c_2) \tag{9}$$

$$\tag{10}$$

$$= E[Z] \tag{11}$$

Equality holds in the limit, but we can get closer and closer by using more and more samples in the average. This Law holds for almost *any distribution*, minus some important cases we will encounter later.

Example

Below is a diagram of the Law of Large numbers in action for three different sequences of Poisson random variables.

We sample `sample_size = 100000` Poisson random variables with parameter $\lambda = 4.5$. (Recall the expected value of a Poisson random variable is equal to its parameter.) We calculate the average for the first n samples, for $n = 1$ to `sample_size`.

```
In [1]: %matplotlib inline
        import numpy as np
        from IPython.core.pylabtools import figsize
        import matplotlib.pyplot as plt

        figsize( 12.5, 5 )

        sample_size = 100000
        expected_value = lambda_ = 4.5
        poi = np.random.poisson
        N_samples = range(1,sample_size,100)

        for k in range(3):
```

```
    samples = poi( lambda_, sample_size )

    partial_average = [ samples[:i].mean() for i in N_samples ]

    plt.plot( N_samples, partial_average, lw=1.5,label="average \
of  $n$ samples; seq. %d"%k)


plt.plot( N_samples, expected_value*np.ones_like( partial_average),
    ls = "--", label = "true expected value", c = "k" )

plt.ylim( 4.35, 4.65)
plt.title( "Convergence of the average of \n random variables to its \
expected value" )
plt.ylabel( "average of $n$ samples" )
plt.xlabel( "# of samples, $n$")
plt.legend();
```

Looking at the above plot, it is clear that when the sample size is small, there is greater variation in the average (compare how *jagged and jumpy* the average is initially, then *smooths* out). All three paths *approach* the value 4.5, but just flirt with it as N gets large. Mathematicians and statistician have another name for *flirting*: convergence.

Another very relevant question we can ask is *how quickly am I converging to the expected value?* Let's plot something new. For a specific N, let's do the above trials thousands of times and compute how far away we are from the true expected value, on average. But wait — *compute on average*? This is simply the law of large numbers again! For example, we are interested in, for a specific N, the quantity:

$$D(N) = \sqrt{E\left[\left(\frac{1}{N}\sum_{i=1}^{N} Z_i - 4.5 \right)^2 \right]}$$

The above formulae is interpretable as a distance away from the true value (on average), for some N. (We take the square root so the dimensions of the above quantity and our random variables are the same). As the above is an expected value, it can be approximated using the law of large numbers: instead of averaging Z_i, we calculate the following multiple times and average them:

$$Y_k = \left(\frac{1}{N}\sum_{i=1}^{N} Z_i - 4.5 \right)^2$$

By computing the above many, N_y, times (remember, it is random), and averaging them:

$$\frac{1}{N_Y}\sum_{k=1}^{N_Y} Y_k \rightarrow E[Y_k] = E\left[\left(\frac{1}{N}\sum_{i=1}^{N} Z_i - 4.5 \right)^2 \right]$$

Finally, taking the square root:

$$\sqrt{\frac{1}{N_Y}\sum_{k=1}^{N_Y} Y_k} \approx D(N)$$

In [2]:
```
figsize( 12.5, 4)

N_Y = 250 #use this many to approximate D(N)
N_array = np.arange( 1000, 50000, 2500 ) #use this many samples in the approx. to the
D_N_results = np.zeros( len( N_array ) )

lambda_ = 4.5
expected_value = lambda_ #for X ~ Poi(lambda) , E[ X ] = lambda

def D_N( n ):
    """
    This function approx. D_n, the average variance of using n samples.
    """
    Z = poi( lambda_, (n, N_Y) )
    average_Z = Z.mean(axis=0)
    return np.sqrt( (  (average_Z - expected_value)**2  ).mean() )


for i,n in enumerate(N_array):
    D_N_results[i] =  D_N(n)


plt.xlabel( "$N$" )
plt.ylabel( "expected squared-distance from true value" )
```

```
plt.plot(N_array, D_N_results, lw = 3,
            label="expected distance between\n\
expected value and \naverage of $N$ random variables.")
plt.plot( N_array, np.sqrt(expected_value)/np.sqrt(N_array), lw = 2, ls = "--",
        label = r"$\frac{\sqrt{\lambda}}{\sqrt{N}}$" )
plt.legend()
plt.title( "How 'fast' is the sample average converging? " );
```

As expected, the expected distance between our sample average and the actual expected value shrinks as N grows large. But also notice that the *rate* of convergence decreases, that is, we need only 10 000 additional samples to move from 0.020 to 0.015, a difference of 0.005, but *20 000* more samples to again decrease from 0.015 to 0.010, again only a 0.005 decrease.

It turns out we can measure this rate of convergence. Above I have plotted a second line, the function $\sqrt{\lambda}/\sqrt{N}$. This was not chosen arbitrarily. In most cases, given a sequence of random variable distributed like Z, the rate of convergence to $E[Z]$ of the Law of Large Numbers is

$$\frac{\sqrt{\ Var(Z)}}{\sqrt{N}}$$

This is useful to know: for a given large N, we know (on average) how far away we are from the estimate. On the other hand, in a Bayesian setting, this can seem like a useless result: Bayesian analysis is OK with uncertainty so what's the *statistical* point of adding extra precise digits? Though drawing samples can be so computationally cheap that having a *larger* N is fine too.

1.1.3 How do we compute $Var(Z)$ though?

The variance is simply another expected value that can be approximated! Consider the following, once we have the expected value (by using the Law of Large Numbers to estimate it, denote it μ), we can estimate the variance:

$$\frac{1}{N}\sum_{i=1}^{N}(Z_i-\mu)^2 \rightarrow E[\ (Z-\mu)^2\ ] = Var(Z)$$

1.1.4 Expected values and probabilities

There is an even less explicit relationship between expected value and estimating probabilities. Define the *indicator function*

$$\mathbb{1}_A(x) = \begin{cases} 1 & x \in A \\ \\ 0 & else \end{cases}$$

Then, by the law of large numbers, if we have many samples X_i, we can estimate the probability of an event A, denoted $P(A)$, by:

$$\frac{1}{N}\sum_{i=1}^{N} \mathbb{1}_A(X_i) \to E[\mathbb{1}_A(X)] = P(A)$$

Again, this is fairly obvious after a moments thought: the indicator function is only 1 if the event occurs, so we are summing only the times the event occurs and dividing by the total number of trials (consider how we usually approximate probabilities using frequencies). For example, suppose we wish to estimate the probability that a $Z \sim Exp(.5)$ is greater than 5, and we have many samples from a $Exp(.5)$ distribution.

$$P(Z > 5) = \frac{1}{N}\sum_{i=1}^{N} \mathbb{1}_{z>5}(Z_i)$$

```
In [3]: N = 10000
        print( np.mean( [ np.random.exponential( 0.5 ) > 5 for i in range(N) ] ) )
```

```
0.0001
```

1.1.5 What does this all have to do with Bayesian statistics?

Point estimates, to be introduced in the next chapter, in Bayesian inference are computed using expected values. In more analytical Bayesian inference, we would have been required to evaluate complicated expected values represented as multi-dimensional integrals. No longer. If we can sample from the posterior distribution directly, we simply need to evaluate averages. Much easier. If accuracy is a priority, plots like the ones above show how fast you are converging. And if further accuracy is desired, just take more samples from the posterior.

When is enough enough? When can you stop drawing samples from the posterior? That is the practitioners decision, and also dependent on the variance of the samples (recall from above a high variance means the average will converge slower).

We also should understand when the Law of Large Numbers fails. As the name implies, and comparing the graphs above for small N, the Law is only true for large sample sizes. Without this, the asymptotic result is not reliable. Knowing in what situations the Law fails can give us *confidence in how unconfident we should be*. The next section deals with this issue.

1.2 The Disorder of Small Numbers

The Law of Large Numbers is only valid as N gets *infinitely* large: never truly attainable. While the law is a powerful tool, it is foolhardy to apply it liberally. Our next example illustrates this.

Example: Aggregated geographic data Often data comes in aggregated form. For instance, data may be grouped by state, county, or city level. Of course, the population numbers vary per geographic area. If the data is an average of some characteristic of each the geographic areas, we must be conscious of the Law of Large Numbers and how it can *fail* for areas with small populations.

We will observe this on a toy dataset. Suppose there are five thousand counties in our dataset. Furthermore, population number in each state are uniformly distributed between 100 and 1500. The way the population numbers are generated is irrelevant to the discussion, so we do not justify this. We are interested in measuring the average height of individuals per county. Unbeknownst to us, height does **not** vary across county, and each individual, regardless of the county he or she is currently living in, has the same distribution of what their height may be:

$$\text{height} \sim \text{Normal}(150, 15)$$

We aggregate the individuals at the county level, so we only have data for the *average in the county*. What might our dataset look like?

```
In [4]: figsize( 12.5, 4)
        std_height = 15
        mean_height = 150

        n_counties = 5000
        pop_generator = np.random.randint
        norm = np.random.normal

        #generate some artificial population numbers
        population = pop_generator(100, 1500, n_counties )

        average_across_county = np.zeros( n_counties )
        for i in range( n_counties ):
            #generate some individuals and take the mean
            average_across_county[i] = norm(mean_height, 1./std_height,
                                            population[i] ).mean()

        #located the counties with the apparently most extreme average heights.
        i_min = np.argmin( average_across_county )
        i_max = np.argmax( average_across_county )

        #plot population size vs. recorded average
        plt.scatter( population, average_across_county, alpha = 0.5, c="#7A68A6")
        plt.scatter( [ population[i_min], population[i_max] ],
                   [average_across_county[i_min], average_across_county[i_max] ],
                   s = 60, marker = "o", facecolors = "none",
                   edgecolors = "#A60628", linewidths = 1.5,
                    label="extreme heights")

        plt.xlim( 100, 1500 )
        plt.title( "Average height vs. County Population")
        plt.xlabel("County Population")
```

```
plt.ylabel("Average height in county")
plt.plot( [100, 1500], [150, 150], color = "k", label = "true expected \
height", ls="--" )
plt.legend(scatterpoints = 1);
```

What do we observe? *Without accounting for population sizes* we run the risk of making an enormous inference error: if we ignored population size, we would say that the county with the shortest and tallest individuals have been correctly circled. But this inference is wrong for the following reason. These two counties do *not* necessarily have the most extreme heights. The error results from the calculated average of smaller populations not being a good reflection of the true expected value of the population (which in truth should be $\mu = 150$). The sample size/population size/N, whatever you wish to call it, is simply too small to invoke the Law of Large Numbers effectively.

We provide more damning evidence against this inference. Recall the population numbers were uniformly distributed over 100 to 1500. Our intuition should tell us that the counties with the most extreme population heights should also be uniformly spread over 100 to 1500, and certainly independent of the county's population. Not so. Below are the population sizes of the counties with the most extreme heights.

```
In [5]: print("Population sizes of 10 'shortest' counties: ")
        print(population[ np.argsort( average_across_county )[:10] ], '\n')
        print("Population sizes of 10 'tallest' counties: ")
        print(population[ np.argsort( -average_across_county )[:10] ])
```

```
Population sizes of 10 'shortest' counties:
[109 135 135 133 109 157 175 120 105 131]

Population sizes of 10 'tallest' counties:
[122 133 313 109 124 280 106 198 326 216]
```

Not at all uniform over 100 to 1500. This is an absolute failure of the Law of Large Numbers.

Example: Kaggle's *U.S. Census Return Rate Challenge* Below is data from the 2010 US census, which partitions populations beyond counties to the level of block groups (which are aggregates of city blocks or equivalents). The dataset is from a Kaggle machine learning competition some colleagues and I participated in. The objective was to predict the census letter mail-back rate of a group block, measured between 0 and 100, using census variables (median income, number of females in the block-group, number of trailer parks, average number of children etc.). Below we plot the census mail-back rate versus block group population:

```
In [6]: figsize( 12.5, 6.5 )
        data = np.genfromtxt( "./data/census_data.csv", skip_header=1,
                              delimiter= ",")
        plt.scatter( data[:,1], data[:,0], alpha = 0.5, c="#7A68A6")
        plt.title("Census mail-back rate vs Population")
        plt.ylabel("Mail-back rate")
        plt.xlabel("population of block-group")
        plt.xlim(-100, 15e3 )
        plt.ylim( -5, 105)

        i_min = np.argmin(  data[:,0] )
        i_max = np.argmax(  data[:,0] )

        plt.scatter( [ data[i_min,1], data[i_max, 1] ],
                     [ data[i_min,0],  data[i_max,0] ],
                     s = 60, marker = "o", facecolors = "none",
                     edgecolors = "#A60628", linewidths = 1.5,
                     label="most extreme points")

        plt.legend(scatterpoints = 1);
```

The above is a classic phenomenon in statistics. I say *classic* referring to the "shape" of the scatter plot above. It follows a classic triangular form, that tightens as we increase the sample size (as the Law of Large Numbers becomes more exact).

I am perhaps overstressing the point and maybe I should have titled the book *"You don't have big data problems!"*, but here again is an example of the trouble with *small datasets*, not big ones. Simply, small datasets cannot be processed using the Law of Large Numbers. Compare with applying the Law without hassle to big datasets (ex. big data). I mentioned earlier that paradoxically big data prediction problems are solved by relatively simple algorithms. The paradox is partially resolved by understanding that the Law of Large Numbers creates solutions that are *stable*, i.e. adding or subtracting a few data points will not affect the solution much. On the other hand, adding or removing data points to a small dataset can create very different results.

For further reading on the hidden dangers of the Law of Large Numbers, I would highly recommend the excellent manuscript The Most Dangerous Equation.

Example: How to order Reddit submissions You may have disagreed with the original statement that the Law of Large numbers is known to everyone, but only implicitly in our subconscious decision making. Consider ratings on online products: how often do you trust an average 5-star rating if there is only 1 reviewer? 2 reviewers? 3 reviewers? We implicitly understand that with such few reviewers that the average rating is **not** a good reflection of the true value of the product.

This has created flaws in how we sort items, and more generally, how we compare items. Many people have realized that sorting online search results by their rating, whether the objects be books, videos, or online comments, return poor results. Often the seemingly top videos or comments have perfect ratings only from a few enthusiastic fans, and truly more quality videos or comments are hidden in later pages with *falsely-substandard* ratings of around 4.8. How can we correct this?

Consider the popular site Reddit (I purposefully did not link to the website as you would never come back). The site hosts links to stories or images, called submissions, for people to comment on. Redditors can vote up or down on each submission (called upvotes and downvotes). Reddit, by default, will sort submissions to a given subreddit by Hot, that is, the submissions that have the most upvotes recently.

How would you determine which submissions are the best? There are a number of ways to achieve this:

1. *Popularity*: A submission is considered good if it has many upvotes. A problem with this model is that a submission with hundreds of upvotes, but thousands of downvotes. While being very *popular*, the submission is likely more controversial than best.
2. *Difference*: Using the *difference* of upvotes and downvotes. This solves the above problem, but fails when we consider the temporal nature of submission. Depending on when a submission is posted, the website may be experiencing high or low traffic. The difference method will bias the *Top* submissions to be the those made during high traffic periods, which have accumulated more upvotes than submissions that were not so graced, but are not necessarily the best.
3. *Time adjusted*: Consider using Difference divided by the age of the submission. This creates a *rate*, something like *difference per second*, or *per minute*. An immediate counter-example is, if we use per second, a 1 second old submission with 1 upvote would be better than a 100 second old submission with 99 upvotes. One can avoid this by only considering at least t second old submission. But what is a good t value? Does this mean no submission younger

than t is good? We end up comparing unstable quantities with stable quantities (young vs. old submissions).

4. *Ratio*: Rank submissions by the ratio of upvotes to total number of votes (upvotes plus downvotes). This solves the temporal issue, such that new submissions who score well can be considered Top just as likely as older submissions, provided they have many upvotes to total votes. The problem here is that a submission with a single upvote (ratio = 1.0) will beat a submission with 999 upvotes and 1 downvote (ratio = 0.999), but clearly the latter submission is *more likely* to be better.

I used the phrase *more likely* for good reason. It is possible that the former submission, with a single upvote, is in fact a better submission than the later with 999 upvotes. The hesitation to agree with this is because we have not seen the other 999 potential votes the former submission might get. Perhaps it will achieve an additional 999 upvotes and 0 downvotes and be considered better than the latter, though not likely.

What we really want is an estimate of the *true upvote ratio*. Note that the true upvote ratio is not the same as the observed upvote ratio: the true upvote ratio is hidden, and we only observe upvotes vs. downvotes (one can think of the true upvote ratio as "what is the underlying probability someone gives this submission a upvote, versus a downvote"). So the 999 upvote/1 downvote submission probably has a true upvote ratio close to 1, which we can assert with confidence thanks to the Law of Large Numbers, but on the other hand we are much less certain about the true upvote ratio of the submission with only a single upvote. Sounds like a Bayesian problem to me.

One way to determine a prior on the upvote ratio is to look at the historical distribution of upvote ratios. This can be accomplished by scraping Reddit's submissions and determining a distribution. There are a few problems with this technique though:

1. Skewed data: The vast majority of submissions have very few votes, hence there will be many submissions with ratios near the extremes (see the "triangular plot" in the above Kaggle dataset), effectively skewing our distribution to the extremes. One could try to only use submissions with votes greater than some threshold. Again, problems are encountered. There is a tradeoff between number of submissions available to use and a higher threshold with associated ratio precision.
2. Biased data: Reddit is composed of different subpages, called subreddits. Two examples are *r/aww*, which posts pics of cute animals, and *r/politics*. It is very likely that the user behaviour towards submissions of these two subreddits are very different: visitors are likely friendly and affectionate in the former, and would therefore upvote submissions more, compared to the latter, where submissions are likely to be controversial and disagreed upon. Therefore not all submissions are the same.

In light of these, I think it is better to use a `Uniform` prior.

With our prior in place, we can find the posterior of the true upvote ratio. The Python script `top_showerthoughts_submissions.py` will scrape the best posts from the `showerthoughts` community on Reddit. This is a text-only community so the title of each post *is* the post. Below is the top post as well as some other sample posts:

```
In [7]: #adding a number to the end of the %run call will get the ith top post.
        %run top_showerthoughts_submissions.py 2
```

```
print("Post contents: \n")
print(top_post)
```

```
Post contents:

Toilet paper should be free and have advertising printed on it.
```

```
In [8]: """
        contents: an array of the text from the last 100 top submissions to a subreddit
        votes: a 2d numpy array of upvotes, downvotes for each submission.
        """
        n_submissions = len(votes)
        submissions = np.random.randint( n_submissions, size=4)
        print("Some Submissions (out of %d total) \n-----------"%n_submissions)
        for i in submissions:
            print('"' + contents[i] + '"')
            print("upvotes/downvotes: ",votes[i,:], "\n")
```

```
Some Submissions (out of 98 total)
-----------
"Rappers from the 90's used guns when they had beef rappers today use Twitter."
upvotes/downvotes:  [32  3]

"All polls are biased towards people who are willing to take polls"
upvotes/downvotes:  [1918  101]

"Taco Bell should give customers an extra tortilla so they can make a burrito out of all the s
upvotes/downvotes:  [79 17]

"There should be an /r/alanismorissette where it's just examples of people using "ironic" inco
upvotes/downvotes:  [33  6]
```

For a given true upvote ratio p and N votes, the number of upvotes will look like a Binomial random variable with parameters p and N. (This is because of the equivalence between upvote ratio and probability of upvoting versus downvoting, out of N possible votes/trials). We create a function that performs Bayesian inference on p, for a particular submission's upvote/downvote pair.

```
In [10]: import pymc3 as pm

         def posterior_upvote_ratio( upvotes, downvotes, samples = 20000):
             """
             This function accepts the number of upvotes and downvotes a particular submission
             and the number of posterior samples to return to the user. Assumes a uniform prio
             """
             N = upvotes + downvotes
```

```
    with pm.Model() as model:
        upvote_ratio = pm.Uniform("upvote_ratio", 0, 1)
        observations = pm.Binomial( "obs",  N, upvote_ratio, observed=upvotes)

        trace = pm.sample(samples, step=pm.Metropolis())

    burned_trace = trace[int(samples/4):]
    return burned_trace["upvote_ratio"]
```

Below are the resulting posterior distributions.

```
In [12]: figsize( 11., 8)
         posteriors = []
         colours = ["#348ABD", "#A60628", "#7A68A6", "#467821", "#CF4457"]
         for i in range(len(submissions)):
             j = submissions[i]
             posteriors.append( posterior_upvote_ratio( votes[j, 0], votes[j,1] ) )
             plt.hist( posteriors[i], bins = 10, normed = True, alpha = .9,
                     histtype="step",color = colours[i%5], lw = 3,
                     label = '(%d up:%d down)\n%s...'%(votes[j, 0], votes[j,1], contents[j][:50
             plt.hist( posteriors[i], bins = 10, normed = True, alpha = .2,
                     histtype="stepfilled",color = colours[i], lw = 3, )

         plt.legend(loc="upper left")
         plt.xlim( 0, 1)
         plt.title("Posterior distributions of upvote ratios on different submissions");
```

```
Applied interval-transform to upvote_ratio and added transformed upvote_ratio_interval_ to mode
 [-------100%-------] 20000 of 20000 in 1.4 sec. | SPS: 14595.5 | ETA: 0.0Applied interval-tran
 [-------100%-------] 20000 of 20000 in 1.3 sec. | SPS: 15189.5 | ETA: 0.0Applied interval-tran
 [-------100%-------] 20000 of 20000 in 1.3 sec. | SPS: 15429.0 | ETA: 0.0Applied interval-tran
 [-------100%-------] 20000 of 20000 in 1.3 sec. | SPS: 15146.5 | ETA: 0.0
```

Some distributions are very tight, others have very long tails (relatively speaking), expressing our uncertainty with what the true upvote ratio might be.

1.2.1 Sorting!

We have been ignoring the goal of this exercise: how do we sort the submissions from *best to worst*? Of course, we cannot sort distributions, we must sort scalar numbers. There are many ways to distill a distribution down to a scalar: expressing the distribution through its expected value, or mean, is one way. Choosing the mean is a bad choice though. This is because the mean does not take into account the uncertainty of distributions.

I suggest using the *95% least plausible value*, defined as the value such that there is only a 5% chance the true parameter is lower (think of the lower bound on the 95% credible region). Below are the posterior distributions with the 95% least-plausible value plotted:

```
In [13]: N = posteriors[0].shape[0]
         lower_limits = []

         for i in range(len(submissions)):
             j = submissions[i]
             plt.hist( posteriors[i], bins = 20, normed = True, alpha = .9,
                     histtype="step",color = colours[i], lw = 3,
                     label = '(%d up:%d down)\n%s...'%(votes[j, 0], votes[j,1], contents[j][:5
```

```
        plt.hist( posteriors[i], bins = 20, normed = True, alpha = .2,
                histtype="stepfilled",color = colours[i], lw = 3, )
        v = np.sort( posteriors[i] )[ int(0.05*N) ]
        #plt.vlines( v, 0, 15 , color = "k", alpha = 1, linewidths=3 )
        plt.vlines( v, 0, 10 , color = colours[i], linestyles = "--",  linewidths=3  )
        lower_limits.append(v)
        plt.legend(loc="upper left")

    plt.legend(loc="upper left")
    plt.title("Posterior distributions of upvote ratios on different submissions");
    order = np.argsort( -np.array( lower_limits ) )
    print(order, lower_limits)
```

```
[1 0 2 3] [0.80034320917496615, 0.94092009444598201, 0.74660503350561902, 0.72190353389632911]
```

The best submissions, according to our procedure, are the submissions that are *most-likely* to score a high percentage of upvotes. Visually those are the submissions with the 95% least plausible value close to 1.

Why is sorting based on this quantity a good idea? By ordering by the 95% least plausible value, we are being the most conservative with what we think is best. That is, even in the worst case scenario, when we have severely overestimated the upvote ratio, we can be sure the best submissions are still on top. Under this ordering, we impose the following very natural properties:

1. given two submissions with the same observed upvote ratio, we will assign the submission with more votes as better (since we are more confident it has a higher ratio).
2. given two submissions with the same number of votes, we still assign the submission with more upvotes as *better*.

1.2.2 But this is too slow for real-time!

I agree, computing the posterior of every submission takes a long time, and by the time you have computed it, likely the data has changed. I delay the mathematics to the appendix, but I suggest using the following formula to compute the lower bound very fast.

$$\frac{a}{a+b} - 1.65\sqrt{\frac{ab}{(a+b)^2(a+b+1)}}$$

where

$$a = 1 + u \tag{12}$$
$$\tag{13}$$
$$b = 1 + d \tag{14}$$
$$\tag{15}$$
$$\tag{16}$$

u is the number of upvotes, and d is the number of downvotes. The formula is a shortcut in Bayesian inference, which will be further explained in Chapter 6 when we discuss priors in more detail.

```
In [14]: def intervals(u,d):
             a = 1. + u
             b = 1. + d
             mu = a/(a+b)
             std_err = 1.65*np.sqrt( (a*b)/( (a+b)**2*(a+b+1.) ) )
             return ( mu, std_err )

         print("Approximate lower bounds:")
         posterior_mean, std_err  = intervals(votes[:,0],votes[:,1])
         lb = posterior_mean - std_err
         print(lb)
         print("\n")
         print("Top 40 Sorted according to approximate lower bounds:")
         print("\n")
         order = np.argsort( -lb )
         ordered_contents = []
         for i in order[:40]:
             ordered_contents.append( contents[i] )
             print(votes[i,0], votes[i,1], contents[i])
             print("-------------")
```

```
Approximate lower bounds:
[ 0.93349005  0.9532194   0.94149718  0.90859764  0.88705356  0.8558795
  0.85644927  0.93752679  0.95767101  0.91131012  0.910073    0.915999
  0.9140058   0.83276025  0.87593961  0.87436674  0.92830849  0.90642832
  0.89187973  0.89950891  0.91295322  0.78607629  0.90250203  0.79950031
  0.85219422  0.83703439  0.7619808   0.81301134  0.7313114   0.79137561
  0.82701445  0.85542404  0.82309334  0.75211374  0.82934814  0.82674958
  0.80933194  0.87448152  0.85350205  0.75460106  0.82934814  0.74417233
  0.79924258  0.8189683   0.75460106  0.90744016  0.83838023  0.78802791
  0.78400654  0.64638659  0.62047936  0.76137738  0.81365241  0.83838023
  0.78457533  0.84980627  0.79249393  0.69020315  0.69593922  0.70758151
  0.70268831  0.91620627  0.73346864  0.86382644  0.80877728  0.72708753
  0.79822085  0.68333632  0.81699014  0.65100453  0.79809005  0.74702492
  0.77318569  0.83221179  0.66500492  0.68134548  0.7249286   0.59412132
  0.58191312  0.73142963  0.73142963  0.66251028  0.87152685  0.74107856
  0.60935684  0.87152685  0.77484517  0.88783675  0.81814153  0.54569789
  0.6122496   0.75613569  0.53511973  0.74556767  0.81814153  0.85773646
  0.6122496   0.64814153]


Top 40 Sorted according to approximate lower bounds:


596 18 Someone should develop an AI specifically for reading Terms & Conditions and flagging du
-------------
2360 98 Porn is the only industry where it is not only acceptable but standard to separate peop
-------------
1918 101 All polls are biased towards people who are willing to take polls
-------------
948 50 They should charge less for drinks in the drive-thru because you can't refill them.
-------------
3740 239 When I was in elementary school and going through the DARE program, I was positive a g
-------------
166 7 "Noted" is the professional way of saying "K".
-------------
29 0 Rewatching Mr. Bean, I've realised that the character is an eccentric genius and not a bli
-------------
289 18 You've been doing weird cameos in your friends' dreams since kindergarten.
-------------
269 17 At some point every parent has stopped wiping their child's butt and hoped for the best
-------------
121 6 Is it really fair to say a person over 85 has heart failure? Technically, that heart has
-------------
535 40 It's surreal to think that the sun and moon and stars we gaze up at are the same objects
-------------
527 40 I wonder if America's internet is censored in a similar way that North Korea's is, but w
-------------
1510 131 Kenny's family is poor because they're always paying for his funeral.
```

```
-------------
43 1 If I was as careful with my whole paycheck as I am with my last $20 I'd be a whole lot be
-------------
162 10 Black hair ties are probably the most popular bracelets in the world.
-------------
107 6 The best answer to the interview question "What is your greatest weakness?" is "intervie
-------------
127 8 Surfing the internet without ads feels like a summer evening without mosquitoes
-------------
159 12 I wonder if Superman ever put a pair of glasses on Lois Lane's dog, and she was like "w
-------------
21 0 Sitting on a cold toilet seat or a warm toilet seat both suck for different reasons.
-------------
1414 157 My life is really like Rihanna's song, "just work work work work work" and the rest o
-------------
222 22 I'm honestly slightly concerned how often Reddit commenters make me laugh compared to m
-------------
52 3 The world must have been a spookier place altogether when candles and gas lamps were the
-------------
194 19 I have not been thankful enough in the last few years that the Black Eyed Peas are no l
-------------
18 0 Living on the coast is having the window seat of the land you live on.
-------------
18 0 Binoculars are like walkie talkies for the deaf.
-------------
28 1 Now that I am a parent of multiple children I have realized that my parents were lying th
-------------
16 0 I sneer at people who read tabloids, but every time I look someone up on Wikipedia the fi
-------------
1559 233 Kid's menus at restaurants should be smaller portions of the same adult dishes at low
-------------
1426 213 Eventually once all phones are waterproof we'll be able to push people into pools aga
-------------
61 5 Myspace is so outdated that jokes about it being outdated has become outdated
-------------
52 4 As a kid, seeing someone step on a banana peel and not slip was a disappointment.
-------------
90 9 Yahoo!ő is the RadioShackő of the Internet.
-------------
34 2 People who "tell it like it is" rarely do so to say something nice
-------------
39 3 Closing your eyes after turning off your alarm is a very dangerous game.
-------------
39 3 Your known 'first word' is the first word your parents heard you speak. In reality, it ma
-------------
87 10 "Smells Like Teen Spirit" is as old to listeners of today as "Yellow Submarine" was to l
-------------
239 36 if an ocean didnt stop immigrants from coming to America what makes us think a wall wil
```

```
-------------
22 1 The phonebook was the biggest invasion of privacy that everyone was oddly ok with.
-------------
57 6 I'm actually the most productive when I procrastinate because I'm doing everything I possi
-------------
57 6 You will never feel how long time is until you have allergies and snot slowly dripping ou
-------------
```

We can view the ordering visually by plotting the posterior mean and bounds, and sorting by the lower bound. In the plot below, notice that the left error-bar is sorted (as we suggested this is the best way to determine an ordering), so the means, indicated by dots, do not follow any strong pattern.

```
In [15]: r_order = order[::-1][-40:]
         plt.errorbar( posterior_mean[r_order], np.arange( len(r_order) ),
                       xerr=std_err[r_order], capsize=0, fmt="o",
                        color = "#7A68A6")
         plt.xlim( 0.3, 1)
         plt.yticks( np.arange( len(r_order)-1,-1,-1 ), map( lambda x: x[:30].replace("\n","")
```

In the graphic above, you can see why sorting by mean would be sub-optimal.

1.2.3 Extension to Starred rating systems

The above procedure works well for upvote-downvotes schemes, but what about systems that use star ratings, e.g. 5 star rating systems. Similar problems apply with simply taking the average: an

item with two perfect ratings would beat an item with thousands of perfect ratings, but a single sub-perfect rating.

We can consider the upvote-downvote problem above as binary: 0 is a downvote, 1 if an upvote. A N-star rating system can be seen as a more continuous version of above, and we can set n stars rewarded is equivalent to rewarding $\frac{n}{N}$. For example, in a 5-star system, a 2 star rating corresponds to 0.4. A perfect rating is a 1. We can use the same formula as before, but with a, b defined differently:

$$\frac{a}{a+b} - 1.65\sqrt{\frac{ab}{(a+b)^2(a+b+1)}}$$

where

$$a = 1 + S \tag{17}$$

$$\tag{18}$$

$$b = 1 + N - S \tag{19}$$

$$\tag{20}$$

$$\tag{21}$$

where N is the number of users who rated, and S is the sum of all the ratings, under the equivalence scheme mentioned above.

Example: Counting Github stars What is the average number of stars a Github repository has? How would you calculate this? There are over 6 million respositories, so there is more than enough data to invoke the Law of Large numbers. Let's start pulling some data. TODO

1.2.4 Conclusion

While the Law of Large Numbers is cool, it is only true so much as its name implies: with large sample sizes only. We have seen how our inference can be affected by not considering *how the data is shaped*.

1. By (cheaply) drawing many samples from the posterior distributions, we can ensure that the Law of Large Number applies as we approximate expected values (which we will do in the next chapter).

2. Bayesian inference understands that with small sample sizes, we can observe wild randomness. Our posterior distribution will reflect this by being more spread rather than tightly concentrated. Thus, our inference should be correctable.

3. There are major implications of not considering the sample size, and trying to sort objects that are unstable leads to pathological orderings. The method provided above solves this problem.

1.2.5 Appendix

Derivation of sorting submissions formula Basically what we are doing is using a Beta prior (with parameters $a = 1, b = 1$, which is a uniform distribution), and using a Binomial likelihood with observations $u, N = u + d$. This means our posterior is a Beta distribution with parameters $a' = 1 + u, b' = 1 + (N - u) = 1 + d$. We then need to find the value, x, such that 0.05 probability is less than x. This is usually done by inverting the CDF (Cumulative Distribution Function), but the CDF of the beta, for integer parameters, is known but is a large sum [3].

We instead use a Normal approximation. The mean of the Beta is $\mu = a'/(a' + b')$ and the variance is

$$\sigma^2 = \frac{a'b'}{(a' + b')^2(a' + b' + 1)}$$

Hence we solve the following equation for x and have an approximate lower bound.

$$0.05 = \Phi\left(\frac{(x - \mu)}{\sigma}\right)$$

Φ being the cumulative distribution for the normal distribution

Exercises 1. How would you estimate the quantity $E[\cos X]$, where $X \sim \text{Exp}(4)$? What about $E[\cos X | X < 1]$, i.e. the expected value *given* we know X is less than 1? Would you need more samples than the original samples size to be equally accurate?

```
In [16]: ## Enter code here
         import scipy.stats as stats
         exp = stats.expon( scale=4 )
         N = 1e5
         X = exp.rvs( int(N) )
         ## ...
```

2. The following table was located in the paper "Going for Three: Predicting the Likelihood of Field Goal Success with Logistic Regression" [2]. The table ranks football field-goal kickers by their percent of non-misses. What mistake have the researchers made?

Kicker Careers Ranked by Make Percentage Rank
Kicker
Make %
Number of Kicks
1
Garrett Hartley
87.7
57
2
Matt Stover
86.8
335

3
Robbie Gould
86.2
224
4
Rob Bironas
86.1
223
5
Shayne Graham
85.4
254
...
...
...
51
Dave Rayner
72.2
90
52
Nick Novak
71.9
64
53
Tim Seder
71.0
62
54
Jose Cortez
70.7
75
55
Wade Richey
66.1
56

In August 2013, a popular post on the average income per programmer of different languages was trending. Here's the summary chart: (reproduced without permission, cause when you lie with stats, you gunna get the hammer). What do you notice about the extremes?

Average household income by programming language Language
Average Household Income ($)
Data Points
Puppet
87,589.29
112
Haskell

89,973.82
191
PHP
94,031.19
978
CoffeeScript
94,890.80
435
VimL
94,967.11
532
Shell
96,930.54
979
Lua
96,930.69
101
Erlang
97,306.55
168
Clojure
97,500.00
269
Python
97,578.87
2314
JavaScript
97,598.75
3443
Emacs Lisp
97,774.65
355
C#
97,823.31
665
Ruby
98,238.74
3242
C++
99,147.93
845
CSS
99,881.40
527
Perl
100,295.45
990
C

100,766.51
2120
Go
101,158.01
231
Scala
101,460.91
243
ColdFusion
101,536.70
109
Objective-C
101,801.60
562
Groovy
102,650.86
116
Java
103,179.39
1402
XSLT
106,199.19
123
ActionScript
108,119.47
113

1.2.6 References

1. Wainer, Howard. *The Most Dangerous Equation*. American Scientist, Volume 95.
2. Clarck, Torin K., Aaron W. Johnson, and Alexander J. Stimpson. "Going for Three: Predicting the Likelihood of Field Goal Success with Logistic Regression." (2013): n. page. Web. 20 Feb. 2013.
3. http://en.wikipedia.org/wiki/Beta_function#Incomplete_beta_function


```
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 object>
```

Ch5_LossFunctions_PyMC2

January 4, 2018

1 Chapter 5

1.0.1 Would you rather lose an arm or a leg?

Statisticians can be a sour bunch. Instead of considering their winnings, they only measure how much they have lost. In fact, they consider their wins as *negative losses*. But what's interesting is *how they measure their losses.*

For example, consider the following example:

> A meteorologist is predicting the probability of a possible hurricane striking his city. He estimates, with 95% confidence, that the probability of it *not* striking is between 99% - 100%. He is very happy with his precision and advises the city that a major evacuation is unnecessary. Unfortunately, the hurricane does strike and the city is flooded.

This stylized example shows the flaw in using a pure accuracy metric to measure outcomes. Using a measure that emphasizes estimation accuracy, while an appealing and *objective* thing to do, misses the point of why you are even performing the statistical inference in the first place: results of inference. The author Nassim Taleb of *The Black Swan* and *Antifragility* stresses the importance of the *payoffs* of decisions, *not the accuracy*. Taleb distills this quite succinctly: "I would rather be vaguely right than very wrong."

1.1 Loss Functions

We introduce what statisticians and decision theorists call *loss functions*. A loss function is a function of the true parameter, and an estimate of that parameter

$$L(\theta, \hat{\theta}) = f(\theta, \hat{\theta})$$

The important point of loss functions is that it measures how *bad* our current estimate is: the larger the loss, the worse the estimate is according to the loss function. A simple, and very common, example of a loss function is the *squared-error loss*:

$$L(\theta, \hat{\theta}) = (\theta - \hat{\theta})^2$$

The squared-error loss function is used in estimators like linear regression, UMVUEs and many areas of machine learning. We can also consider an asymmetric squared-error loss function, something like:

$$L(\theta, \hat{\theta}) = \begin{cases} (\theta - \hat{\theta})^2 & \hat{\theta} < \theta \\ \\ c(\theta - \hat{\theta})^2 & \hat{\theta} \geq \theta, \;\; 0 < c < 1 \end{cases}$$

which represents that estimating a value larger than the true estimate is preferable to estimating a value below. A situation where this might be useful is in estimating web traffic for the next month, where an over-estimated outlook is preferred so as to avoid an underallocation of server resources.

A negative property about the squared-error loss is that it puts a disproportionate emphasis on large outliers. This is because the loss increases quadratically, and not linearly, as the estimate moves away. That is, the penalty of being three units away is much less than being five units away, but the penalty is not much greater than being one unit away, though in both cases the magnitude of difference is the same:

$$\frac{1^2}{3^2} < \frac{3^2}{5^2}, \;\; \text{although} \;\; 3 - 1 = 5 - 3$$

This loss function imposes that large errors are *very* bad. A more *robust* loss function that increases linearly with the difference is the *absolute-loss*

$$L(\theta, \hat{\theta}) = |\theta - \hat{\theta}|$$

Other popular loss functions include:

- $L(\theta, \hat{\theta}) = \mathbb{1}_{\hat{\theta} \neq \theta}$ is the zero-one loss often used in machine learning classification algorithms.
- $L(\theta, \hat{\theta}) = -\hat{\theta} \log(\theta) - (1 - \hat{\theta}) \log(1 - \theta), \;\; \hat{\theta} \in 0, 1, \; \theta \in [0, 1]$, called the *log-loss*, also used in machine learning.

Historically, loss functions have been motivated from 1) mathematical convenience, and 2) they are robust to application, i.e., they are objective measures of loss. The first reason has really held back the full breadth of loss functions. With computers being agnostic to mathematical convenience, we are free to design our own loss functions, which we take full advantage of later in this Chapter.

With respect to the second point, the above loss functions are indeed objective, in that they are most often a function of the difference between estimate and true parameter, independent of signage or payoff of choosing that estimate. This last point, its independence of payoff, causes quite pathological results though. Consider our hurricane example above: the statistician equivalently predicted that the probability of the hurricane striking was between 0% to 1%. But if he had ignored being precise and instead focused on outcomes (99% chance of no flood, 1% chance of flood), he might have advised differently.

By shifting our focus from trying to be incredibly precise about parameter estimation to focusing on the outcomes of our parameter estimation, we can customize our estimates to be optimized for our application. This requires us to design new loss functions that reflect our goals and outcomes. Some examples of more interesting loss functions:

- $L(\theta, \hat{\theta}) = \frac{|\theta - \hat{\theta}|}{\theta(1-\theta)}, \ \ \hat{\theta}, \theta \in [0, 1]$ emphasizes an estimate closer to 0 or 1 since if the true value θ is near 0 or 1, the loss will be *very* large unless $\hat{\theta}$ is similarly close to 0 or 1. This loss function might be used by a political pundit whose job requires him or her to give confident "Yes/No" answers. This loss reflects that if the true parameter is close to 1 (for example, if a political outcome is very likely to occur), he or she would want to strongly agree as to not look like a skeptic.

- $L(\theta, \hat{\theta}) = 1 - \exp\left(-(\theta - \hat{\theta})^2\right)$ is bounded between 0 and 1 and reflects that the user is indifferent to sufficiently-far-away estimates. It is similar to the zero-one loss above, but not quite as penalizing to estimates that are close to the true parameter.

- Complicated non-linear loss functions can programmed:

```
def loss(true_value, estimate):
    if estimate*true_value > 0:
        return abs(estimate - true_value)
    else:
       return abs(estimate)*(estimate - true_value)**2
```

- Another example is from the book *The Signal and The Noise*. Weather forecasters have a interesting loss function for their predictions. [2]

  > People notice one type of mistake — the failure to predict rain — more than other, false alarms. If it rains when it isn't supposed to, they curse the weatherman for ruining their picnic, whereas an unexpectedly sunny day is taken as a serendipitous bonus.
  >
  > [The Weather Channel's bias] is limited to slightly exaggerating the probability of rain when it is unlikely to occur — saying there is a 20 percent change when they know it is really a 5 or 10 percent chance — covering their butts in the case of an unexpected sprinkle.

As you can see, loss functions can be used for good and evil: with great power, comes great — well you know.

1.2 Loss functions in the real world

So far we have been under the unrealistic assumption that we know the true parameter. Of course if we knew the true parameter, bothering to guess an estimate is pointless. Hence a loss function is really only practical when the true parameter is unknown.

In Bayesian inference, we have a mindset that the unknown parameters are really random variables with prior and posterior distributions. Concerning the posterior distribution, a value drawn from it is a *possible* realization of what the true parameter could be. Given that realization, we can compute a loss associated with an estimate. As we have a whole distribution of what the unknown parameter could be (the posterior), we should be more interested in computing the *expected loss* given an estimate. This expected loss is a better estimate of the true loss than comparing the given loss from only a single sample from the posterior.

First it will be useful to explain a *Bayesian point estimate*. The systems and machinery present in the modern world are not built to accept posterior distributions as input. It is also rude to

hand someone over a distribution when all they asked for was an estimate. In the course of an individual's day, when faced with uncertainty we still act by distilling our uncertainty down to a single action. Similarly, we need to distill our posterior distribution down to a single value (or vector in the multivariate case). If the value is chosen intelligently, we can avoid the flaw of frequentist methodologies that mask the uncertainty and provide a more informative result.The value chosen, if from a Bayesian posterior, is a Bayesian point estimate.

Suppose $P(\theta|X)$ is the posterior distribution of θ after observing data X, then the following function is understandable as the *expected loss of choosing estimate $\hat{\theta}$ to estimate θ*:

$$l(\hat{\theta}) = E_\theta \left[\; L(\theta, \hat{\theta}) \; \right]$$

This is also known as the *risk* of estimate $\hat{\theta}$. The subscript θ under the expectation symbol is used to denote that θ is the unknown (random) variable in the expectation, something that at first can be difficult to consider.

We spent all of last chapter discussing how to approximate expected values. Given N samples $\theta_i, \; i = 1, ..., N$ from the posterior distribution, and a loss function L, we can approximate the expected loss of using estimate $\hat{\theta}$ by the Law of Large Numbers:

$$\frac{1}{N} \sum_{i=1}^{N} L(\theta_i, \hat{\theta}) \approx E_\theta \left[\; L(\theta, \hat{\theta}) \; \right] = l(\hat{\theta})$$

Notice that measuring your loss via an *expected value* uses more information from the distribution than the MAP estimate which, if you recall, will only find the maximum value of the distribution and ignore the shape of the distribution. Ignoring information can over-expose yourself to tail risks, like the unlikely hurricane, and leaves your estimate ignorant of how ignorant you really are about the parameter.

Similarly, compare this with frequentist methods, that traditionally only aim to minimize the error, and do not consider the *loss associated with the result of that error*. Compound this with the fact that frequentist methods are almost guaranteed to never be absolutely accurate. Bayesian point estimates fix this by planning ahead: your estimate is going to be wrong, you might as well err on the right side of wrong.

Example: Optimizing for the *Showcase* on *The Price is Right* Bless you if you are ever chosen as a contestant on the Price is Right, for here we will show you how to optimize your final price on the *Showcase*. For those who forget the rules:

1. Two contestants compete in *The Showcase*.
2. Each contestant is shown a unique suite of prizes.
3. After the viewing, the contestants are asked to bid on the price for their unique suite of prizes.
4. If a bid price is over the actual price, the bid's owner is disqualified from winning.
5. If a bid price is under the true price by less than $250, the winner is awarded both prizes.

The difficulty in the game is balancing your uncertainty in the prices, keeping your bid low enough so as to not bid over, and trying to bid close to the price.

Suppose we have recorded the *Showcases* from previous *The Price is Right* episodes and have *prior* beliefs about what distribution the true price follows. For simplicity, suppose it follows a Normal:

$$\text{True Price} \sim \text{Normal}(\mu_p, \sigma_p)$$

In a later chapter, we will actually use *real Price is Right Showcase data* to form the historical prior, but this requires some advanced PyMC use so we will not use it here. For now, we will assume $\mu_p = 35000$ and $\sigma_p = 7500$.

We need a model of how we should be playing the *Showcase*. For each prize in the prize suite, we have an idea of what it might cost, but this guess could differ significantly from the true price. (Couple this with increased pressure being onstage and you can see why some bids are so wildly off). Let's suppose your beliefs about the prices of prizes also follow Normal distributions:

$$\text{Prize}_i \sim \text{Normal}(\mu_i, \sigma_i), \quad i = 1, 2$$

This is really why Bayesian analysis is great: we can specify what we think a fair price is through the μ_i parameter, and express uncertainty of our guess in the σ_i parameter.

We'll assume two prizes per suite for brevity, but this can be extended to any number. The true price of the prize suite is then given by $\text{Prize}_1 + \text{Prize}_2 + \epsilon$, where ϵ is some error term.

We are interested in the updated True Price given we have observed both prizes and have belief distributions about them. We can perform this using PyMC.

Lets make some values concrete. Suppose there are two prizes in the observed prize suite:

1. A trip to wonderful Toronto, Canada!
2. A lovely new snowblower!

We have some guesses about the true prices of these objects, but we are also pretty uncertain about them. I can express this uncertainty through the parameters of the Normals:

$$\text{snowblower} \sim \text{Normal}(3000, 500) \tag{1}$$
$$\tag{2}$$
$$\text{Toronto} \sim \text{Normal}(12000, 3000) \tag{3}$$
$$\tag{4}$$
$$\tag{5}$$

For example, I believe that the true price of the trip to Toronto is 12 000 dollars, and that there is a 68.2% chance the price falls 1 standard deviation away from this, i.e. my confidence is that there is a 68.2% chance the trip is in [9 000, 15 000].

We can create some PyMC code to perform inference on the true price of the suite.

```
In [1]: %matplotlib inline
        import scipy.stats as stats
        from IPython.core.pylabtools import figsize
        import numpy as np
        import matplotlib.pyplot as plt

        figsize(12.5, 9)

        norm_pdf = stats.norm.pdf
```

```
plt.subplot(311)
x = np.linspace(0, 60000, 200)
sp1 = plt.fill_between(x, 0, norm_pdf(x, 35000, 7500),
                       color="#348ABD", lw=3, alpha=0.6,
                       label="historical total prices")
p1 = plt.Rectangle((0, 0), 1, 1, fc=sp1.get_facecolor()[0])
plt.legend([p1], [sp1.get_label()])

plt.subplot(312)
x = np.linspace(0, 10000, 200)
sp2 = plt.fill_between(x, 0, norm_pdf(x, 3000, 500),
                       color="#A60628", lw=3, alpha=0.6,
                       label="snowblower price guess")

p2 = plt.Rectangle((0, 0), 1, 1, fc=sp2.get_facecolor()[0])
plt.legend([p2], [sp2.get_label()])

plt.subplot(313)
x = np.linspace(0, 25000, 200)
sp3 = plt.fill_between(x, 0, norm_pdf(x, 12000, 3000),
                       color="#7A68A6", lw=3, alpha=0.6,
                       label="Trip price guess")
plt.autoscale(tight=True)
p3 = plt.Rectangle((0, 0), 1, 1, fc=sp3.get_facecolor()[0])
plt.legend([p3], [sp3.get_label()]);
```

```
In [2]: import pymc as pm

        data_mu = [3e3, 12e3]

        data_std = [5e2, 3e3]

        mu_prior = 35e3
        std_prior = 75e2

        true_price = pm.Normal("true_price", mu_prior, 1.0 / std_prior ** 2)


        prize_1 = pm.Normal("first_prize", data_mu[0], 1.0 / data_std[0] ** 2)
        prize_2 = pm.Normal("second_prize", data_mu[1], 1.0 / data_std[1] ** 2)
        price_estimate = prize_1 + prize_2


        @pm.potential
        def error(true_price=true_price, price_estimate=price_estimate):
                return pm.normal_like(true_price, price_estimate, 1 / (3e3) ** 2)


        mcmc = pm.MCMC([true_price, prize_1, prize_2, price_estimate, error])
        mcmc.sample(50000, 10000)

        price_trace = mcmc.trace("true_price")[:]

 [-----------------100%-----------------] 50000 of 50000 complete in 8.0 sec

In [3]: figsize(12.5, 4)

        import scipy.stats as stats

        x = np.linspace(5000, 40000)
        plt.plot(x, stats.norm.pdf(x, 35000, 7500), c="k", lw=2,
                 label="prior dist. of suite price")

        _hist = plt.hist(price_trace, bins=35, normed=True, histtype="stepfilled")
        plt.title("Posterior of the true price estimate")
        plt.vlines(mu_prior, 0, 1.1 * np.max(_hist[0]), label="prior's mean",
                   linestyles="--")
        plt.vlines(price_trace.mean(), 0, 1.1 * np.max(_hist[0]),
                   label="posterior's mean", linestyles="-.")
        plt.legend(loc="upper left");
```

Notice that because of our two observed prizes and subsequent guesses (including uncertainty about those guesses), we shifted our mean price estimate down about $15 000 dollars from the previous mean price.

A frequentist, seeing the two prizes and having the same beliefs about their prices, would bid $\mu_1 + \mu_2 = 35000$, regardless of any uncertainty. Meanwhile, the *naive Bayesian* would simply pick the mean of the posterior distribution. But we have more information about our eventual outcomes; we should incorporate this into our bid. We will use the loss function above to find the *best* bid (*best* according to our loss).

What might a contestant's loss function look like? I would think it would look something like:

```
def showcase_loss(guess, true_price, risk=80000):
    if true_price < guess:
        return risk
    elif abs(true_price - guess) <= 250:
        return -2*np.abs(true_price)
    else:
        return np.abs(true_price - guess - 250)
```

where `risk` is a parameter that defines of how bad it is if your guess is over the true price. A lower `risk` means that you are more comfortable with the idea of going over. If we do bid under and the difference is less than $250, we receive both prizes (modeled here as receiving twice the original prize). Otherwise, when we bid under the `true_price` we want to be as close as possible, hence the `else` loss is a increasing function of the distance between the guess and true price.

For every possible bid, we calculate the *expected loss* associated with that bid. We vary the `risk` parameter to see how it affects our loss:

```
In [4]: figsize(12.5, 7)
        # numpy friendly showdown_loss


        def showdown_loss(guess, true_price, risk=80000):
                loss = np.zeros_like(true_price)
                ix = true_price < guess
                loss[~ix] = np.abs(guess - true_price[~ix])
                close_mask = [abs(true_price - guess) <= 250]
```

```
        loss[close_mask] = -2 * true_price[close_mask]
        loss[ix] = risk
        return loss


guesses = np.linspace(5000, 50000, 70)
risks = np.linspace(30000, 150000, 6)
expected_loss = lambda guess, risk: \
    showdown_loss(guess, price_trace, risk).mean()

for _p in risks:
    results = [expected_loss(_g, _p) for _g in guesses]
    plt.plot(guesses, results, label="%d" % _p)

plt.title("Expected loss of different guesses, \nvarious risk-levels of \
overestimating")
plt.legend(loc="upper left", title="Risk parameter")
plt.xlabel("price bid")
plt.ylabel("expected loss")
plt.xlim(5000, 30000);
```

1.2.1 Minimizing our losses

It would be wise to choose the estimate that minimizes our expected loss. This corresponds to the minimum point on each of the curves above. More formally, we would like to minimize our expected loss by finding the solution to

$$\arg\min_{\hat{\theta}} \quad E_\theta \left[\, L(\theta, \hat{\theta}) \, \right]$$

The minimum of the expected loss is called the *Bayes action*. We can solve for the Bayes action using Scipy's optimization routines. The function in `fmin` in `scipy.optimize` module uses an intelligent search to find a minimum (not necessarily a *global* minimum) of any uni- or multivariate function. For most purposes, `fmin` will provide you with a good answer.

We'll compute the minimum loss for the *Showcase* example above:

```
In [5]: import scipy.optimize as sop

        ax = plt.subplot(111)


        for _p in risks:
            _color = next(ax._get_lines.prop_cycler)
            _min_results = sop.fmin(expected_loss, 15000, args=(_p,),disp = False)
            _results = [expected_loss(_g, _p) for _g in guesses]
            plt.plot(guesses, _results , color = _color['color'])
            plt.scatter(_min_results, 0, s = 60, \
                        color= _color['color'], label = "%d"%_p)
            plt.vlines(_min_results, 0, 120000, color = _color['color'], linestyles="--")
            print("minimum at risk %d: %.2f" % (_p, _min_results))

        plt.title("Expected loss & Bayes actions of different guesses, \n \
        various risk-levels of overestimating")
        plt.legend(loc="upper left", scatterpoints=1, title="Bayes action at risk:")
        plt.xlabel("price guess")
        plt.ylabel("expected loss")
        plt.xlim(7000, 30000)
        plt.ylim(-1000, 80000);
```

```
minimum at risk 30000: 14068.38
minimum at risk 54000: 12967.09
minimum at risk 78000: 12031.03
minimum at risk 102000: 11991.21
minimum at risk 126000: 11803.17
minimum at risk 150000: 11803.17
```

As intuition suggests, as we decrease the risk threshold (care about overbidding less), we increase our bid, willing to edge closer to the true price. It is interesting how far away our optimized loss is from the posterior mean, which was about 20 000.

Suffice to say, in higher dimensions being able to eyeball the minimum expected loss is impossible. Hence why we require use of Scipy's `fmin` function.

1.2.2 Shortcuts

For some loss functions, the Bayes action is known in closed form. We list some of them below:

- If using the mean-squared loss, the Bayes action is the mean of the posterior distribution, i.e. the value
$$E_\theta\left[\theta\right]$$
minimizes $E_\theta\left[\ (\theta - \hat{\theta})^2\ \right]$. Computationally this requires us to calculate the average of the posterior samples [See chapter 4 on The Law of Large Numbers]

- Whereas the *median* of the posterior distribution minimizes the expected absolute-loss. The sample median of the posterior samples is an appropriate and very accurate approximation to the true median.

- In fact, it is possible to show that the MAP estimate is the solution to using a loss function that shrinks to the zero-one loss.

Maybe it is clear now why the first-introduced loss functions are used most often in the mathematics of Bayesian inference: no complicated optimizations are necessary. Luckily, we have machines to do the complications for us.

1.3 Machine Learning via Bayesian Methods

Whereas frequentist methods strive to achieve the best precision about all possible parameters, machine learning cares to achieve the best *prediction* among all possible parameters. Of course, one way to achieve accurate predictions is to aim for accurate predictions, but often your prediction measure and what frequentist methods are optimizing for are very different.

For example, least-squares linear regression is the most simple active machine learning algorithm. I say active as it engages in some learning, whereas predicting the sample mean is technically *simpler*, but is learning very little if anything. The loss that determines the coefficients of the regressors is a squared-error loss. On the other hand, if your prediction loss function (or score function, which is the negative loss) is not a squared-error, like AUC, ROC, precision, etc., your least-squares line will not be optimal for the prediction loss function. This can lead to prediction results that are suboptimal.

Finding Bayes actions is equivalent to finding parameters that optimize *not parameter accuracy* but an arbitrary performance measure, however we wish to define performance (loss functions, AUC, ROC, precision/recall etc.).

The next two examples demonstrate these ideas. The first example is a linear model where we can choose to predict using the least-squares loss or a novel, outcome-sensitive loss.

The second example is adapted from a Kaggle data science project. The loss function associated with our predictions is incredibly complicated.

Example: Financial prediction Suppose the future return of a stock price is very small, say 0.01 (or 1%). We have a model that predicts the stock's future price, and our profit and loss is directly tied to us acting on the prediction. How should we measure the loss associated with the model's predictions, and subsequent future predictions? A squared-error loss is agnostic to the signage and would penalize a prediction of -0.01 equally as bad a prediction of 0.03:

$$(0.01 - (-0.01))^2 = (0.01 - 0.03)^2 = 0.004$$

If you had made a bet based on your model's prediction, you would have earned money with a prediction of 0.03, and lost money with a prediction of -0.01, yet our loss did not capture this. We need a better loss that takes into account the *sign* of the prediction and true value. We design a new loss that is better for financial applications below:

```
In [6]: figsize(12.5, 4)


        def stock_loss(true_return, yhat, alpha=100.):
            if true_return * yhat < 0:
                # opposite signs, not good
                return alpha * yhat ** 2 - np.sign(true_return) * yhat \
                    + abs(true_return)
            else:
                return abs(true_return - yhat)


        true_value = .05
        pred = np.linspace(-.04, .12, 75)
```

```
plt.plot(pred, [stock_loss(true_value, _p) for _p in pred],
         label="Loss associated with\n prediction if true value = 0.05", lw=3)
plt.vlines(0, 0, .25, linestyles="--")

plt.xlabel("prediction")
plt.ylabel("loss")
plt.xlim(-0.04, .12)
plt.ylim(0, 0.25)

true_value = -.02
plt.plot(pred, [stock_loss(true_value, _p) for _p in pred], alpha=0.6,
         label="Loss associated with\n prediction if true value = -0.02", lw=3)
plt.legend()
plt.title("Stock returns loss if true value = 0.05, -0.02");
```

Note the change in the shape of the loss as the prediction crosses zero. This loss reflects that the user really does not want to guess the wrong sign, especially be wrong *and* a large magnitude.

Why would the user care about the magnitude? Why is the loss not 0 for predicting the correct sign? Surely, if the return is 0.01 and we bet millions we will still be (very) happy.

Financial institutions treat downside risk, as in predicting a lot on the wrong side, and upside risk, as in predicting a lot on the right side, similarly. Both are seen as risky behaviour and discouraged. Hence why we have an increasing loss as we move further away from the true price. (With less extreme loss in the direction of the correct sign.)

We will perform a regression on a trading signal that we believe predicts future returns well. Our dataset is artificial, as most financial data is not even close to linear. Below, we plot the data along with the least-squares line.

```
In [7]: # Code to create artificial data
        N = 100
        X = 0.025 * np.random.randn(N)
        Y = 0.5 * X + 0.01 * np.random.randn(N)

        ls_coef_ = np.cov(X, Y)[0, 1] / np.var(X)
        ls_intercept = Y.mean() - ls_coef_ * X.mean()
```

```
plt.scatter(X, Y, c="k")
plt.xlabel("trading signal")
plt.ylabel("returns")
plt.title("Empirical returns vs trading signal")
plt.plot(X, ls_coef_ * X + ls_intercept, label="Least-squares line")
plt.xlim(X.min(), X.max())
plt.ylim(Y.min(), Y.max())
plt.legend(loc="upper left");
```

We perform a simple Bayesian linear regression on this dataset. We look for a model like:

$$R = \alpha + \beta x + \epsilon$$

where α, β are our unknown parameters and $\epsilon \sim \text{Normal}(0, 1/\tau)$. The most common priors on β and α are Normal priors. We will also assign a prior on τ, so that $\sigma = 1/\sqrt{\tau}$ is uniform over 0 to 100 (equivalently then $\tau = 1/\text{Uniform}(0, 100)^2$).

```
In [8]: import pymc as pm
        from pymc.Matplot import plot as mcplot

        std = pm.Uniform("std", 0, 100, trace=False)  # this needs to be explained.


        @pm.deterministic
        def prec(U=std):
            return 1.0 / (U) ** 2

        beta = pm.Normal("beta", 0, 0.0001)
        alpha = pm.Normal("alpha", 0, 0.0001)


        @pm.deterministic
        def mean(X=X, alpha=alpha, beta=beta):
```

```
        return alpha + beta * X

    obs = pm.Normal("obs", mean, prec, value=Y, observed=True)
    mcmc = pm.MCMC([obs, beta, alpha, std, prec])

    mcmc.sample(100000, 80000)
    mcplot(mcmc)
```

```
 [-----------------100%-----------------] 100000 of 100000 complete in 17.2 secPlotting prec
Plotting beta
Plotting alpha
```

Ch5_LossFunctions_PyMC3

January 4, 2018

1 Chapter 5

```
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) ____
```

1.0.1 Would you rather lose an arm or a leg?

Statisticians can be a sour bunch. Instead of considering their winnings, they only measure how much they have lost. In fact, they consider their wins as *negative losses*. But what's interesting is *how they measure their losses.*

For example, consider the following example:

> A meteorologist is predicting the probability of a possible hurricane striking his city. He estimates, with 95% confidence, that the probability of it *not* striking is between 99% - 100%. He is very happy with his precision and advises the city that a major evacuation is unnecessary. Unfortunately, the hurricane does strike and the city is flooded.

This stylized example shows the flaw in using a pure accuracy metric to measure outcomes. Using a measure that emphasizes estimation accuracy, while an appealing and *objective* thing to do, misses the point of why you are even performing the statistical inference in the first place: results of inference. The author Nassim Taleb of *The Black Swan* and *Antifragility* stresses the importance of the *payoffs* of decisions, *not the accuracy*. Taleb distills this quite succinctly: "I would rather be vaguely right than very wrong."

1.1 Loss Functions

We introduce what statisticians and decision theorists call *loss functions*. A loss function is a function of the true parameter, and an estimate of that parameter

$$L(\theta, \hat{\theta}) = f(\theta, \hat{\theta})$$

The important point of loss functions is that it measures how *bad* our current estimate is: the larger the loss, the worse the estimate is according to the loss function. A simple, and very common, example of a loss function is the *squared-error loss*:

$$L(\theta, \hat{\theta}) = (\theta - \hat{\theta})^2$$

The squared-error loss function is used in estimators like linear regression, UMVUEs and many areas of machine learning. We can also consider an asymmetric squared-error loss function, something like:

$$L(\theta, \hat{\theta}) = \begin{cases} (\theta - \hat{\theta})^2 & \hat{\theta} < \theta \\ \\ c(\theta - \hat{\theta})^2 & \hat{\theta} \geq \theta, \;\; 0 < c < 1 \end{cases}$$

which represents that estimating a value larger than the true estimate is preferable to estimating a value below. A situation where this might be useful is in estimating web traffic for the next month, where an over-estimated outlook is preferred so as to avoid an underallocation of server resources.

A negative property about the squared-error loss is that it puts a disproportionate emphasis on large outliers. This is because the loss increases quadratically, and not linearly, as the estimate moves away. That is, the penalty of being three units away is much less than being five units away, but the penalty is not much greater than being one unit away, though in both cases the magnitude of difference is the same:

$$\frac{1^2}{3^2} < \frac{3^2}{5^2}, \;\; \text{although} \;\; 3 - 1 = 5 - 3$$

This loss function imposes that large errors are *very* bad. A more *robust* loss function that increases linearly with the difference is the *absolute-loss*

$$L(\theta, \hat{\theta}) = |\theta - \hat{\theta}|$$

Other popular loss functions include:

- $L(\theta, \hat{\theta}) = \mathbb{1}_{\hat{\theta} \neq \theta}$ is the zero-one loss often used in machine learning classification algorithms.
- $L(\theta, \hat{\theta}) = -\hat{\theta} \log(\theta) - (1 - \hat{\theta}) \log(1 - \theta), \;\; \hat{\theta} \in 0, 1, \; \theta \in [0, 1]$, called the *log-loss*, also used in machine learning.

Historically, loss functions have been motivated from 1) mathematical convenience, and 2) they are robust to application, i.e., they are objective measures of loss. The first reason has really held back the full breadth of loss functions. With computers being agnostic to mathematical convenience, we are free to design our own loss functions, which we take full advantage of later in this Chapter.

With respect to the second point, the above loss functions are indeed objective, in that they are most often a function of the difference between estimate and true parameter, independent of signage or payoff of choosing that estimate. This last point, its independence of payoff, causes quite pathological results though. Consider our hurricane example above: the statistician equivalently predicted that the probability of the hurricane striking was between 0% to 1%. But if he had ignored being precise and instead focused on outcomes (99% chance of no flood, 1% chance of flood), he might have advised differently.

By shifting our focus from trying to be incredibly precise about parameter estimation to focusing on the outcomes of our parameter estimation, we can customize our estimates to be optimized for our application. This requires us to design new loss functions that reflect our goals and outcomes. Some examples of more interesting loss functions:

- $L(\theta, \hat{\theta}) = \frac{|\theta - \hat{\theta}|}{\theta(1-\theta)}, \;\; \hat{\theta}, \theta \in [0, 1]$ emphasizes an estimate closer to 0 or 1 since if the true value θ is near 0 or 1, the loss will be *very* large unless $\hat{\theta}$ is similarly close to 0 or 1. This loss function might be used by a political pundit whose job requires him or her to give confident "Yes/No" answers. This loss reflects that if the true parameter is close to 1 (for example, if a political outcome is very likely to occur), he or she would want to strongly agree as to not look like a skeptic.

- $L(\theta, \hat{\theta}) = 1 - \exp\left(-(\theta - \hat{\theta})^2\right)$ is bounded between 0 and 1 and reflects that the user is indifferent to sufficiently-far-away estimates. It is similar to the zero-one loss above, but not quite as penalizing to estimates that are close to the true parameter.

- Complicated non-linear loss functions can programmed:

```
def loss(true_value, estimate):
    if estimate*true_value > 0:
        return abs(estimate - true_value)
    else:
       return abs(estimate)*(estimate - true_value)**2
```

- Another example is from the book *The Signal and The Noise*. Weather forecasters have an interesting loss function for their predictions.

  > People notice one type of mistake — the failure to predict rain — more than other, false alarms. If it rains when it isn't supposed to, they curse the weatherman for ruining their picnic, whereas an unexpectedly sunny day is taken as a serendipitous bonus.
  >
  > [The Weather Channel's bias] is limited to slightly exaggerating the probability of rain when it is unlikely to occur — saying there is a 20 percent change when they know it is really a 5 or 10 percent chance — covering their butts in the case of an unexpected sprinkle.

As you can see, loss functions can be used for good and evil: with great power, comes great — well you know.

1.2 Loss functions in the real world

So far we have been under the unrealistic assumption that we know the true parameter. Of course if we knew the true parameter, bothering to guess an estimate is pointless. Hence a loss function is really only practical when the true parameter is unknown.

In Bayesian inference, we have a mindset that the unknown parameters are really random variables with prior and posterior distributions. Concerning the posterior distribution, a value drawn from it is a *possible* realization of what the true parameter could be. Given that realization, we can compute a loss associated with an estimate. As we have a whole distribution of what the unknown parameter could be (the posterior), we should be more interested in computing the *expected loss* given an estimate. This expected loss is a better estimate of the true loss than comparing the given loss from only a single sample from the posterior.

First it will be useful to explain a *Bayesian point estimate*. The systems and machinery present in the modern world are not built to accept posterior distributions as input. It is also rude to

hand someone over a distribution when all they asked for was an estimate. In the course of an individual's day, when faced with uncertainty we still act by distilling our uncertainty down to a single action. Similarly, we need to distill our posterior distribution down to a single value (or vector in the multivariate case). If the value is chosen intelligently, we can avoid the flaw of frequentist methodologies that mask the uncertainty and provide a more informative result.The value chosen, if from a Bayesian posterior, is a Bayesian point estimate.

Suppose $P(\theta|X)$ is the posterior distribution of θ after observing data X, then the following function is understandable as the *expected loss of choosing estimate $\hat{\theta}$ to estimate θ*:

$$l(\hat{\theta}) = E_\theta \left[\; L(\theta, \hat{\theta}) \; \right]$$

This is also known as the *risk* of estimate $\hat{\theta}$. The subscript θ under the expectation symbol is used to denote that θ is the unknown (random) variable in the expectation, something that at first can be difficult to consider.

We spent all of last chapter discussing how to approximate expected values. Given N samples $\theta_i, \; i = 1, ..., N$ from the posterior distribution, and a loss function L, we can approximate the expected loss of using estimate $\hat{\theta}$ by the Law of Large Numbers:

$$\frac{1}{N} \sum_{i=1}^{N} L(\theta_i, \hat{\theta}) \approx E_\theta \left[\; L(\theta, \hat{\theta}) \; \right] = l(\hat{\theta})$$

Notice that measuring your loss via an *expected value* uses more information from the distribution than the MAP estimate which, if you recall, will only find the maximum value of the distribution and ignore the shape of the distribution. Ignoring information can over-expose yourself to tail risks, like the unlikely hurricane, and leaves your estimate ignorant of how ignorant you really are about the parameter.

Similarly, compare this with frequentist methods, that traditionally only aim to minimize the error, and do not consider the *loss associated with the result of that error*. Compound this with the fact that frequentist methods are almost guaranteed to never be absolutely accurate. Bayesian point estimates fix this by planning ahead: your estimate is going to be wrong, you might as well err on the right side of wrong.

Example: Optimizing for the *Showcase* on *The Price is Right* Bless you if you are ever chosen as a contestant on the Price is Right, for here we will show you how to optimize your final price on the *Showcase*. For those who forget the rules:

1. Two contestants compete in *The Showcase*.
2. Each contestant is shown a unique suite of prizes.
3. After the viewing, the contestants are asked to bid on the price for their unique suite of prizes.
4. If a bid price is over the actual price, the bid's owner is disqualified from winning.
5. If a bid price is under the true price by less than $250, the winner is awarded both prizes.

The difficulty in the game is balancing your uncertainty in the prices, keeping your bid low enough so as to not bid over, and trying to bid close to the price.

Suppose we have recorded the *Showcases* from previous *The Price is Right* episodes and have *prior* beliefs about what distribution the true price follows. For simplicity, suppose it follows a Normal:

$$\text{True Price} \sim \text{Normal}(\mu_p, \sigma_p)$$

In a later chapter, we will actually use *real Price is Right Showcase data* to form the historical prior, but this requires some advanced PyMC3 use so we will not use it here. For now, we will assume $\mu_p = 35000$ and $\sigma_p = 7500$.

We need a model of how we should be playing the *Showcase*. For each prize in the prize suite, we have an idea of what it might cost, but this guess could differ significantly from the true price. (Couple this with increased pressure being onstage and you can see why some bids are so wildly off). Let's suppose your beliefs about the prices of prizes also follow Normal distributions:

$$\text{Prize}_i \sim \text{Normal}(\mu_i, \sigma_i), \quad i = 1, 2$$

This is really why Bayesian analysis is great: we can specify what we think a fair price is through the μ_i parameter, and express uncertainty of our guess in the σ_i parameter.

We'll assume two prizes per suite for brevity, but this can be extended to any number. The true price of the prize suite is then given by $\text{Prize}_1 + \text{Prize}_2 + \epsilon$, where ϵ is some error term.

We are interested in the updated True Price given we have observed both prizes and have belief distributions about them. We can perform this using PyMC3.

Lets make some values concrete. Suppose there are two prizes in the observed prize suite:

1. A trip to wonderful Toronto, Canada!
2. A lovely new snowblower!

We have some guesses about the true prices of these objects, but we are also pretty uncertain about them. I can express this uncertainty through the parameters of the Normals:

$$\text{snowblower} \sim \text{Normal}(3000, 500) \tag{1}$$
$$\tag{2}$$
$$\text{Toronto} \sim \text{Normal}(12000, 3000) \tag{3}$$
$$\tag{4}$$
$$\tag{5}$$

For example, I believe that the true price of the trip to Toronto is 12 000 dollars, and that there is a 68.2% chance the price falls 1 standard deviation away from this, i.e. my confidence is that there is a 68.2% chance the trip is in [9 000, 15 000].

We can create some PyMC3 code to perform inference on the true price of the suite.

```
In [2]: %matplotlib inline
        import scipy.stats as stats
        from IPython.core.pylabtools import figsize
        import numpy as np
        import matplotlib.pyplot as plt

        figsize(12.5, 9)

        norm_pdf = stats.norm.pdf
```

```
plt.subplot(311)
x = np.linspace(0, 60000, 200)
sp1 = plt.fill_between(x , 0, norm_pdf(x, 35000, 7500),
                color = "#348ABD", lw = 3, alpha = 0.6,
                label = "historical total prices")
p1 = plt.Rectangle((0, 0), 1, 1, fc=sp1.get_facecolor()[0])
plt.legend([p1], [sp1.get_label()])

plt.subplot(312)
x = np.linspace(0, 10000, 200)
sp2 = plt.fill_between(x , 0, norm_pdf(x, 3000, 500),
                 color = "#A60628", lw = 3, alpha = 0.6,
                 label="snowblower price guess")

p2 = plt.Rectangle((0, 0), 1, 1, fc=sp2.get_facecolor()[0])
plt.legend([p2], [sp2.get_label()])

plt.subplot(313)
x = np.linspace(0, 25000, 200)
sp3 = plt.fill_between(x , 0, norm_pdf(x, 12000, 3000),
                 color = "#7A68A6", lw = 3, alpha = 0.6,
                 label = "Trip price guess")
plt.autoscale(tight=True)
p3 = plt.Rectangle((0, 0), 1, 1, fc=sp3.get_facecolor()[0])
plt.legend([p3], [sp3.get_label()]);
```

In [3]:
```
import pymc3 as pm

data_mu = [3e3, 12e3]

data_std =  [5e2, 3e3]

mu_prior = 35e3
std_prior =  75e2
with pm.Model() as model:
    true_price = pm.Normal("true_price", mu=mu_prior, sd=std_prior)

    prize_1 = pm.Normal("first_prize", mu=data_mu[0], sd=data_std[0])
    prize_2 = pm.Normal("second_prize", mu=data_mu[1], sd=data_std[1])
    price_estimate = prize_1 + prize_2

    logp = pm.Normal.dist(mu=price_estimate, sd=(3e3)).logp(true_price)
    error = pm.Potential("error", logp)


    trace = pm.sample(50000, step=pm.Metropolis())
    burned_trace = trace[10000:]

price_trace = burned_trace["true_price"]
```

```
[-------100%-------] 50000 of 50000 in 8.2 sec. | SPS: 6119.0 | ETA: 0.0
```

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

import scipy.stats as stats

x = np.linspace(5000, 40000)
plt.plot(x, stats.norm.pdf(x, 35000, 7500), c = "k", lw = 2,
         label = "prior dist. of suite price")

_hist = plt.hist(price_trace, bins = 35, normed= True, histtype= "stepfilled")
plt.title("Posterior of the true price estimate")
plt.vlines(mu_prior, 0, 1.1*np.max(_hist[0]), label = "prior's mean",
           linestyles="--")
plt.vlines(price_trace.mean(), 0, 1.1*np.max(_hist[0]), \
           label = "posterior's mean", linestyles="-.")
plt.legend(loc = "upper left");
```

Notice that because of our two observed prizes and subsequent guesses (including uncertainty about those guesses), we shifted our mean price estimate down about $15,000 dollars from the previous mean price.

A frequentist, seeing the two prizes and having the same beliefs about their prices, would bid $\mu_1 + \mu_2 = 35000$, regardless of any uncertainty. Meanwhile, the *naive Bayesian* would simply pick the mean of the posterior distribution. But we have more information about our eventual outcomes; we should incorporate this into our bid. We will use the loss function above to find the *best* bid (*best* according to our loss).

What might a contestant's loss function look like? I would think it would look something like:

```
def showcase_loss(guess, true_price, risk = 80000):
    if true_price < guess:
        return risk
    elif abs(true_price - guess) <= 250:
        return -2*np.abs(true_price)
    else:
        return np.abs(true_price - guess - 250)
```

where `risk` is a parameter that defines of how bad it is if your guess is over the true price. A lower `risk` means that you are more comfortable with the idea of going over. If we do bid under and the difference is less than $250, we receive both prizes (modeled here as receiving twice the original prize). Otherwise, when we bid under the `true_price` we want to be as close as possible, hence the `else` loss is a increasing function of the distance between the guess and true price.

For every possible bid, we calculate the *expected loss* associated with that bid. We vary the `risk` parameter to see how it affects our loss:

```
In [5]: figsize(12.5, 7)
        #numpy friendly showdown_loss


        def showdown_loss(guess, true_price, risk = 80000):
                loss = np.zeros_like(true_price)
                ix = true_price < guess
                loss[~ix] = np.abs(guess - true_price[~ix])
                close_mask = [abs(true_price - guess) <= 250]
```

```
        loss[close_mask] = -2*true_price[close_mask]
        loss[ix] = risk
        return loss


guesses = np.linspace(5000, 50000, 70)
risks = np.linspace(30000, 150000, 6)
expected_loss = lambda guess, risk: \
    showdown_loss(guess, price_trace, risk).mean()

for _p in risks:
    results = [expected_loss(_g, _p) for _g in guesses]
    plt.plot(guesses, results, label = "%d"%_p)

plt.title("Expected loss of different guesses, \nvarious risk-levels of \
overestimating")
plt.legend(loc="upper left", title="Risk parameter")
plt.xlabel("price bid")
plt.ylabel("expected loss")
plt.xlim(5000, 30000);
```

1.2.1 Minimizing our losses

It would be wise to choose the estimate that minimizes our expected loss. This corresponds to the minimum point on each of the curves above. More formally, we would like to minimize our expected loss by finding the solution to

$$\arg\min_{\hat{\theta}} \quad E_\theta \left[\, L(\theta, \hat{\theta}) \, \right]$$

The minimum of the expected loss is called the *Bayes action*. We can solve for the Bayes action using Scipy's optimization routines. The function in `fmin` in `scipy.optimize` module uses an intelligent search to find a minimum (not necessarily a *global* minimum) of any uni- or multivariate function. For most purposes, `fmin` will provide you with a good answer.

We'll compute the minimum loss for the *Showcase* example above:

```
In [6]: import scipy.optimize as sop

        ax = plt.subplot(111)


        for _p in risks:
            _color = next(ax._get_lines.prop_cycler)
            _min_results = sop.fmin(expected_loss, 15000, args=(_p,),disp = False)
            _results = [expected_loss(_g, _p) for _g in guesses]
            plt.plot(guesses, _results , color = _color['color'])
            plt.scatter(_min_results, 0, s = 60, \
                        color= _color['color'], label = "%d"%_p)
            plt.vlines(_min_results, 0, 120000, color = _color['color'], linestyles="--")
            print("minimum at risk %d: %.2f" % (_p, _min_results))

        plt.title("Expected loss & Bayes actions of different guesses, \n \
        various risk-levels of overestimating")
        plt.legend(loc="upper left", scatterpoints = 1, title = "Bayes action at risk:")
        plt.xlabel("price guess")
        plt.ylabel("expected loss")
        plt.xlim(7000, 30000)
        plt.ylim(-1000, 80000);
```

```
minimum at risk 30000: 14723.45
minimum at risk 54000: 13500.92
minimum at risk 78000: 11900.78
minimum at risk 102000: 11649.08
minimum at risk 126000: 11649.08
minimum at risk 150000: 11329.30
```

As intuition suggests, as we decrease the risk threshold (care about overbidding less), we increase our bid, willing to edge closer to the true price. It is interesting how far away our optimized loss is from the posterior mean, which was about 20 000.

Suffice to say, in higher dimensions being able to eyeball the minimum expected loss is impossible. Hence why we require use of Scipy's `fmin` function.

1.2.2 Shortcuts

For some loss functions, the Bayes action is known in closed form. We list some of them below:

- If using the mean-squared loss, the Bayes action is the mean of the posterior distribution, i.e. the value
$$E_\theta\left[\theta\right]$$
minimizes $E_\theta\left[\ (\theta - \hat{\theta})^2\ \right]$. Computationally this requires us to calculate the average of the posterior samples [See chapter 4 on The Law of Large Numbers]

- Whereas the *median* of the posterior distribution minimizes the expected absolute-loss. The sample median of the posterior samples is an appropriate and very accurate approximation to the true median.

- In fact, it is possible to show that the MAP estimate is the solution to using a loss function that shrinks to the zero-one loss.

Maybe it is clear now why the first-introduced loss functions are used most often in the mathematics of Bayesian inference: no complicated optimizations are necessary. Luckily, we have machines to do the complications for us.

1.3 Machine Learning via Bayesian Methods

Whereas frequentist methods strive to achieve the best precision about all possible parameters, machine learning cares to achieve the best *prediction* among all possible parameters. Of course, one way to achieve accurate predictions is to aim for accurate predictions, but often your prediction measure and what frequentist methods are optimizing for are very different.

For example, least-squares linear regression is the most simple active machine learning algorithm. I say active as it engages in some learning, whereas predicting the sample mean is technically *simpler*, but is learning very little if anything. The loss that determines the coefficients of the regressors is a squared-error loss. On the other hand, if your prediction loss function (or score function, which is the negative loss) is not a squared-error, like AUC, ROC, precision, etc., your least-squares line will not be optimal for the prediction loss function. This can lead to prediction results that are suboptimal.

Finding Bayes actions is equivalent to finding parameters that optimize *not parameter accuracy* but an arbitrary performance measure, however we wish to define performance (loss functions, AUC, ROC, precision/recall etc.).

The next two examples demonstrate these ideas. The first example is a linear model where we can choose to predict using the least-squares loss or a novel, outcome-sensitive loss.

The second example is adapted from a Kaggle data science project. The loss function associated with our predictions is incredibly complicated.

Example: Financial prediction Suppose the future return of a stock price is very small, say 0.01 (or 1%). We have a model that predicts the stock's future price, and our profit and loss is directly tied to us acting on the prediction. How should we measure the loss associated with the model's predictions, and subsequent future predictions? A squared-error loss is agnostic to the signage and would penalize a prediction of -0.01 equally as bad a prediction of 0.03:

$$(0.01 - (-0.01))^2 = (0.01 - 0.03)^2 = 0.004$$

If you had made a bet based on your model's prediction, you would have earned money with a prediction of 0.03, and lost money with a prediction of -0.01, yet our loss did not capture this. We need a better loss that takes into account the *sign* of the prediction and true value. We design a new loss that is better for financial applications below:

```
In [7]: figsize(12.5, 4)


        def stock_loss(true_return, yhat, alpha = 100.):
            if true_return * yhat < 0:
                #opposite signs, not good
                return alpha*yhat**2 - np.sign(true_return)*yhat \
                                + abs(true_return)
            else:
                return abs(true_return - yhat)


        true_value = .05
        pred = np.linspace(-.04, .12, 75)
```

```
plt.plot(pred, [stock_loss(true_value, _p) for _p in pred], \
        label = "Loss associated with\n prediction if true value = 0.05", lw =3)
plt.vlines(0, 0, .25, linestyles="--")

plt.xlabel("prediction")
plt.ylabel("loss")
plt.xlim(-0.04, .12)
plt.ylim(0, 0.25)

true_value = -.02
plt.plot(pred, [stock_loss(true_value, _p) for _p in pred], alpha = 0.6, \
        label = "Loss associated with\n prediction if true value = -0.02", lw =3)
plt.legend()
plt.title("Stock returns loss if true value = 0.05, -0.02");
```

Note the change in the shape of the loss as the prediction crosses zero. This loss reflects that the user really does not want to guess the wrong sign, especially be wrong *and* a large magnitude.

Why would the user care about the magnitude? Why is the loss not 0 for predicting the correct sign? Surely, if the return is 0.01 and we bet millions we will still be (very) happy.

Financial institutions treat downside risk, as in predicting a lot on the wrong side, and upside risk, as in predicting a lot on the right side, similarly. Both are seen as risky behaviour and discouraged. Hence why we have an increasing loss as we move further away from the true price. (With less extreme loss in the direction of the correct sign.)

We will perform a regression on a trading signal that we believe predicts future returns well. Our dataset is artificial, as most financial data is not even close to linear. Below, we plot the data along with the least-squares line.

```
In [8]: ## Code to create artificial data
        N = 100
        X = 0.025*np.random.randn(N)
        Y = 0.5*X + 0.01*np.random.randn(N)

        ls_coef_ = np.cov(X, Y)[0,1]/np.var(X)
        ls_intercept = Y.mean() - ls_coef_*X.mean()
```

```
plt.scatter(X, Y, c="k")
plt.xlabel("trading signal")
plt.ylabel("returns")
plt.title("Empirical returns vs trading signal")
plt.plot(X, ls_coef_*X + ls_intercept, label = "Least-squares line")
plt.xlim(X.min(), X.max())
plt.ylim(Y.min(), Y.max())
plt.legend(loc="upper left");
```

We perform a simple Bayesian linear regression on this dataset. We look for a model like:

$$R = \alpha + \beta x + \epsilon$$

where α, β are our unknown parameters and $\epsilon \sim \text{Normal}(0, \sigma)$. The most common priors on β and α are Normal priors. We will also assign a prior on σ, so that σ is uniform over 0 to 100.

```
In [9]: import pymc3 as pm

        with pm.Model() as model:
            std = pm.Uniform("std", 0, 100)

            beta = pm.Normal("beta", mu=0, sd=100)
            alpha = pm.Normal("alpha", mu=0, sd=100)

            mean = pm.Deterministic("mean", alpha + beta*X)

            obs = pm.Normal("obs", mu=mean, sd=std, observed=Y)

            trace = pm.sample(100000, step=pm.Metropolis())
            burned_trace = trace[20000:]
```

```
Applied interval-transform to std and added transformed std_interval_ to model.
 [-------100%-------] 100000 of 100000 in 26.5 sec. | SPS: 3769.6 | ETA: 0.0
```

```
In [10]: pm.plots.traceplot(trace=burned_trace, varnames=["std", "beta", "alpha"])
         pm.plot_posterior(trace=burned_trace, varnames=["std", "beta", "alpha"], kde_plot=True
```

It appears the MCMC has converged so we may continue.
For a specific trading signal, call it x, the distribution of possible returns has the form:

$$R_i(x) = \alpha_i + \beta_i x + \epsilon$$

where $\epsilon \sim \text{Normal}(0, \sigma_i)$ and i indexes our posterior samples. We wish to find the solution to

$$\arg\min_r \quad E_{R(x)} \left[\, L(R(x), r) \, \right]$$

according to the loss given above. This r is our Bayes action for trading signal x. Below we plot the Bayes action over different trading signals. What do you notice?

In [11]:
```
figsize(12.5, 6)
from scipy.optimize import fmin


def stock_loss(price, pred, coef = 500):
    """vectorized for numpy"""
    sol = np.zeros_like(price)
    ix = price*pred < 0
    sol[ix] = coef*pred**2 - np.sign(price[ix])*pred + abs(price[ix])
    sol[~ix] = abs(price[~ix] - pred)
    return sol

std_samples = burned_trace["std"]
alpha_samples = burned_trace["alpha"]
beta_samples = burned_trace["beta"]

N = std_samples.shape[0]

noise = std_samples*np.random.randn(N)

possible_outcomes = lambda signal: alpha_samples + beta_samples*signal + noise


opt_predictions = np.zeros(50)
trading_signals =  np.linspace(X.min(), X.max(), 50)
for i, _signal in enumerate(trading_signals):
        _possible_outcomes = possible_outcomes(_signal)
        tomin = lambda pred: stock_loss(_possible_outcomes, pred).mean()
        opt_predictions[i] = fmin(tomin, 0, disp = False)


plt.xlabel("trading signal")
plt.ylabel("prediction")
plt.title("Least-squares prediction vs. Bayes action prediction")
plt.plot(X, ls_coef_*X + ls_intercept, label ="Least-squares prediction")
plt.xlim(X.min(), X.max())
plt.plot(trading_signals, opt_predictions, label ="Bayes action prediction")
plt.legend(loc="upper left");
```

What is interesting about the above graph is that when the signal is near 0, and many of the possible returns outcomes are possibly both positive and negative, our best (with respect to our loss) prediction is to predict close to 0, hence *take on no position*. Only when we are very confident do we enter into a position. I call this style of model a *sparse prediction*, where we feel uncomfortable with our uncertainty so choose not to act. (Compare with the least-squares prediction which will rarely, if ever, predict zero).

A good sanity check that our model is still reasonable: as the signal becomes more and more extreme, and we feel more and more confident about the positive/negativeness of returns, our position converges with that of the least-squares line.

The sparse-prediction model is not trying to *fit* the data the best (according to a *squared-error loss* definition of *fit*). That honor would go to the least-squares model. The sparse-prediction model is trying to find the best prediction *with respect to our* `stock_loss`*-defined loss*. We can turn this reasoning around: the least-squares model is not trying to *predict* the best (according to a `stock-loss` definition of *predict*). That honor would go the *sparse prediction* model. The least-squares model is trying to find the best fit of the data *with respect to the squared-error loss*.

Example: Kaggle contest on *Observing Dark World* A personal motivation for learning Bayesian methods was trying to piece together the winning solution to Kaggle's *Observing Dark Worlds* contest. From the contest's website:

> There is more to the Universe than meets the eye. Out in the cosmos exists a form of matter that outnumbers the stuff we can see by almost 7 to 1, and we don't know what it is. What we do know is that it does not emit or absorb light, so we call it Dark Matter. Such a vast amount of aggregated matter does not go unnoticed. In fact we observe that this stuff aggregates and forms massive structures called Dark Matter Halos. Although dark, it warps and bends spacetime such that any light from a background galaxy which passes close to the Dark Matter will have its path altered and changed. This bending causes the galaxy to appear as an ellipse in the sky.

The contest required predictions about where dark matter was likely to be. The winner, Tim Salimans, used Bayesian inference to find the best locations for the halos (interestingly, the second-place winner also used Bayesian inference). With Tim's permission, we provided his solution [1] here:

1. Construct a prior distribution for the halo positions $p(x)$, i.e. formulate our expectations about the halo positions before looking at the data.
2. Construct a probabilistic model for the data (observed ellipticities of the galaxies) given the positions of the dark matter halos: $p(e|x)$.
3. Use Bayes' rule to get the posterior distribution of the halo positions, i.e. use to the data to guess where the dark matter halos might be.
4. Minimize the expected loss with respect to the posterior distribution over the predictions for the halo positions: $\hat{x} = \arg\min_{\text{prediction}} E_{p(x|e)}[L(\text{prediction}, x)]$, i.e. tune our predictions to be as good as possible for the given error metric.

The loss function in this problem is very complicated. For the very determined, the loss function is contained in the file DarkWorldsMetric.py in the parent folder. Though I suggest not reading it all, suffice to say the loss function is about 160 lines of code — not something that can be written down in a single mathematical line. The loss function attempts to measure the accuracy of prediction, in a Euclidean distance sense, such that no shift-bias is present. More details can be found on the metric's main page.

We will attempt to implement Tim's winning solution using PyMC3 and our knowledge of loss functions.

1.3.1 The Data

The dataset is actually 300 separate files, each representing a sky. In each file, or sky, are between 300 and 720 galaxies. Each galaxy has an x and y position associated with it, ranging from 0 to 4200, and measures of ellipticity: e_1 and e_2. Information about what these measures mean can be found here, but for our purposes it does not matter besides for visualization purposes. Thus a typical sky might look like the following:

```
In [12]: from draw_sky2 import draw_sky

         n_sky = 3 #choose a file/sky to examine.
         data = np.genfromtxt("data/Train_Skies/Train_Skies/\
         Training_Sky%d.csv" % (n_sky),
                                dtype = None,
                                skip_header = 1,
                                delimiter = ",",
                                usecols = [1,2,3,4])
         print("Data on galaxies in sky %d."%n_sky)
         print("position_x, position_y, e_1, e_2 ")
         print(data[:3])

         fig = draw_sky(data)
         plt.title("Galaxy positions and ellipcities of sky %d." % n_sky)
         plt.xlabel("x-position")
         plt.ylabel("y-position");
```

```
Data on galaxies in sky 3.
position_x, position_y, e_1, e_2
[[  1.62690000e+02   1.60006000e+03   1.14664000e-01  -1.90326000e-01]
 [  2.27228000e+03   5.40040000e+02   6.23555000e-01   2.14979000e-01]
 [  3.55364000e+03   2.69771000e+03   2.83527000e-01  -3.01870000e-01]]
```

1.3.2 Priors

Each sky has one, two or three dark matter halos in it. Tim's solution details that his prior distribution of halo positions was uniform, i.e.

$$x_i \sim \text{Uniform}(0, 4200) \tag{6}$$
$$\tag{7}$$
$$y_i \sim \text{Uniform}(0, 4200), \quad i = 1, 2, 3 \tag{8}$$
$$\tag{9}$$
$$\tag{10}$$

Tim and other competitors noted that most skies had one large halo and other halos, if present, were much smaller. Larger halos, having more mass, will influence the surrounding galaxies more. He decided that the large halos would have a mass distributed as a *log*-uniform random variable between 40 and 180 i.e.

$$m_{\text{large}} = \log \text{Uniform}(40, 180)$$

and in PyMC3,

```
exp_mass_large = pm.Uniform("exp_mass_large", 40, 180)
mass_large = pm.Deterministic("mass_large", np.log(exp_max_large))
```

(This is what we mean when we say *log*-uniform.) For smaller galaxies, Tim set the mass to be the logarithm of 20. Why did Tim not create a prior for the smaller mass, nor treat it as a unknown? I believe this decision was made to speed up convergence of the algorithm. This is not too restrictive, as by construction the smaller halos have less influence on the galaxies.

Tim logically assumed that the ellipticity of each galaxy is dependent on the position of the halos, the distance between the galaxy and halo, and the mass of the halos. Thus the vector of ellipticity of each galaxy, $\mathbf{e}_i$, are *children* variables of the vector of halo positions $(\mathbf{x}, \mathbf{y})$, distance (which we will formalize), and halo masses.

Tim conceived a relationship to connect positions and ellipticity by reading literature and forum posts. He supposed the following was a reasonable relationship:

$$e_i | (\mathbf{x}, \mathbf{y}) \sim \text{Normal}(\sum_{j=\text{halo positions}} d_{i,j} m_j f(r_{i,j}), \sigma^2)$$

where $d_{i,j}$ is the *tangential direction* (the direction in which halo j bends the light of galaxy i), m_j is the mass of halo j, $f(r_{i,j})$ is a *decreasing function* of the Euclidean distance between halo j and galaxy i.

Tim's function f was defined:

$$f(r_{i,j}) = \frac{1}{\min(r_{i,j}, 240)}$$

for large halos, and for small halos

$$f(r_{i,j}) = \frac{1}{\min(r_{i,j}, 70)}$$

This fully bridges our observations and unknown. This model is incredibly simple, and Tim mentions this simplicity was purposefully designed: it prevents the model from overfitting.

1.3.3 Training & PyMC3 implementation

For each sky, we run our Bayesian model to find the posteriors for the halo positions — we ignore the (known) halo position. This is slightly different than perhaps traditional approaches to Kaggle competitions, where this model uses no data from other skies nor the known halo location. That does not mean other data are not necessary — in fact, the model was created by comparing different skies.

In [13]:
```
import pymc3 as pm
import theano.tensor as T

def euclidean_distance(x, y):
    return np.sqrt(((x - y)**2)).sum(axis=1)

def f_distance(gxy_pos, halo_pos, c):
    # foo_position should be a 2-d numpy array
    # T.maximum() provides our element-wise maximum as in NumPy, but instead for thea
    return T.maximum(euclidean_distance(gxy_pos, halo_pos), c)[:, None]

def tangential_distance(glxy_position, halo_position):
    # foo_position should be a 2-d numpy array
    delta = glxy_position - halo_position
    t = (2*T.arctan(delta[:,1]/delta[:,0]))
    return T.stack([-T.cos(t), -T.sin(t)], axis=1)


with pm.Model() as model:
    #set the size of the halo's mass
    mass_large = pm.Uniform("mass_large", 40, 180)

    #set the initial prior position of the halos, it's a 2-d Uniform dist.
    halo_position = pm.Uniform("halo_position", 0, 4200, shape=(1,2))

    mean = pm.Deterministic("mean", mass_large /\
            f_distance(T.as_tensor(data[:,:2]), halo_position, 240)*\
            tangential_distance(T.as_tensor(data[:,:2]), halo_position))

    ellpty = pm.Normal("ellipcity", mu=mean, tau=1./0.05, observed=data[:,2:])
```

```
Applied interval-transform to mass_large and added transformed mass_large_interval_ to model.
Applied interval-transform to halo_position and added transformed halo_position_interval_ to mo
```

In [14]:
```
with model:
    mu, sds, elbo = pm.variational.advi(n=50000)
    step = pm.NUTS(scaling=model.dict_to_array(sds), is_cov=True)
    trace = pm.sample(5000, step=step, start=mu)
```

```
Iteration 0 [0%]: ELBO = 58.1
Iteration 5000 [10%]: Average ELBO = 51.72
```

```
Iteration 10000 [20%]: Average ELBO = 69.89
Iteration 15000 [30%]: Average ELBO = 82.23
Iteration 20000 [40%]: Average ELBO = 102.96
Iteration 25000 [50%]: Average ELBO = 130.65
Iteration 30000 [60%]: Average ELBO = 145.73
Iteration 35000 [70%]: Average ELBO = 149.08
Iteration 40000 [80%]: Average ELBO = 149.4
Iteration 45000 [90%]: Average ELBO = 149.36
Finished [100%]: Average ELBO = 149.36
 [-------100%-------] 5000 of 5000 in 32.4 sec. | SPS: 154.4 | ETA: 0.0
```

We use ADVI here to find a good starting point and scaling for our NUTS sampler. NUTS is a "smarter" MCMC sampling method than Metropolis, so as a result we need fewer total samples for our chains to converge. ADVI follows a completely different methodology to fit a model that we will not get into here. We may cover ADVI, as well as NUTS, in-depth in a later chapter.

Below we plot a "heatmap" of the posterior distribution. (Which is just a scatter plot of the posterior, but we can visualize it as a heatmap.)

```
In [15]: t = trace["halo_position"].reshape(5000,2)

         fig = draw_sky(data)
         plt.title("Galaxy positions and ellipcities of sky %d." % n_sky)
         plt.xlabel("x-position")
         plt.ylabel("y-position")
         plt.scatter(t[:,0], t[:,1], alpha = 0.015, c = "r")
         plt.xlim(0, 4200)
         plt.ylim(0, 4200);
```

The most probable position reveals itself like a lethal wound.

Associated with each sky is another data point, located in `./data/Training_halos.csv` that holds the locations of up to three dark matter halos contained in the sky. For example, the night sky we trained on has halo locations:

```
In [16]: halo_data = np.genfromtxt("data/Training_halos.csv",
                                   delimiter = ",",
                                   usecols = [1, 2,3, 4,5,6,7,8,9],
                                   skip_header = 1)
         print(halo_data[n_sky])
```

```
[  1.00000000e+00   1.40861000e+03   1.68586000e+03   1.40861000e+03
   1.68586000e+03   0.00000000e+00   0.00000000e+00   0.00000000e+00
   0.00000000e+00]
```

The third and fourth column represent the true x and y position of the halo. It appears that the Bayesian method has located the halo within a tight vicinity.

```
In [17]: fig = draw_sky(data)
         plt.title("Galaxy positions and ellipcities of sky %d." % n_sky)
         plt.xlabel("x-position")
         plt.ylabel("y-position")
         plt.scatter(t[:,0], t[:,1], alpha = 0.015, c = "r")
         plt.scatter(halo_data[n_sky-1][3], halo_data[n_sky-1][4],
                     label = "True halo position",
                     c = "k", s = 70)
         plt.legend(scatterpoints = 1, loc = "lower left")
         plt.xlim(0, 4200)
         plt.ylim(0, 4200);

         print("True halo location:", halo_data[n_sky][3], halo_data[n_sky][4])
```

```
True halo location: 1408.61 1685.86
```

Perfect. Our next step is to use the loss function to optimize our location. A naive strategy would be to simply choose the mean:

```
In [18]: mean_posterior = t.mean(axis=0).reshape(1,2)
         print(mean_posterior)
```

```
[[ 2312.96009501  1127.87170923]]
```

```
In [19]: from DarkWorldsMetric import main_score

         _halo_data = halo_data[n_sky-1]

         nhalo_all =  _halo_data[0].reshape(1,1)
```

```
x_true_all = _halo_data[3].reshape(1,1)
y_true_all = _halo_data[4].reshape(1,1)
x_ref_all = _halo_data[1].reshape(1,1)
y_ref_all = _halo_data[2].reshape(1,1)
sky_prediction = mean_posterior

print("Using the mean:")
main_score(nhalo_all, x_true_all, y_true_all, \
            x_ref_all, y_ref_all, sky_prediction)

#what's a bad score?
random_guess = np.random.randint(0, 4200, size=(1,2))
print("Using a random location:", random_guess)
main_score(nhalo_all, x_true_all, y_true_all, \
            x_ref_all, y_ref_all, random_guess)
```

```
Using the mean:
Your average distance in pixels you are away from the true halo is 46.0082084242
Your average angular vector is 1.0
Your score for the training data is 1.04600820842
Using a random location: [[ 288 3167]]
Your average distance in pixels you are away from the true halo is 2908.49191694
Your average angular vector is 1.0
Your score for the training data is 3.90849191694
```

```
Out[19]: 3.9084919169390866
```

This is a good guess, it is not very far from the true location, but it ignores the loss function that was provided to us. We also need to extend our code to allow for up to two additional, *smaller* halos: Let's create a function for automatizing our PyMC3.

```
In [20]: def halo_posteriors(n_halos_in_sky, galaxy_data,samples = 5e5, burn_in = 500):
             #set the size of the halo's mass
             with pm.Model() as model:
                 mass_large = pm.Uniform("mass_large", 40, 180)

                 mass_small_1 = 20
                 mass_small_2 = 20

                 masses = np.array([mass_large,mass_small_1, mass_small_2], dtype=object)

                 #set the initial prior positions of the halos, it's a 2-d Uniform dist.
                 halo_positions = pm.Uniform("halo_positions", 0, 4200, shape=(n_halos_in_sky,

                 fdist_constants = np.array([240, 70, 70])

                 _sum = 0
                 for i in range(n_halos_in_sky):
```

```
                _sum += masses[i]/f_distance(data[:,:2], halo_positions[i, :], fdist_cons
                    tangential_distance(data[:,:2], halo_positions[i, :])

            mean = pm.Deterministic("mean", _sum)

            ellpty = pm.Normal("ellipcity", mu=mean, tau=1./0.05, observed=data[:,2:])

            mu, sds, elbo = pm.variational.advi(n=50000)
            step = pm.NUTS(scaling=model.dict_to_array(sds), is_cov=True)
            trace = pm.sample(samples, step=step, start=mu)

        burned_trace = trace[burn_in:]
        return burned_trace["halo_positions"]
```

In [21]:
```
n_sky = 215
data = np.genfromtxt("data/Train_Skies/Train_Skies/\
Training_Sky%d.csv" % (n_sky),
                      dtype = None,
                      skip_header = 1,
                      delimiter = ",",
                      usecols = [1,2,3,4])
```

In [22]:
```
#there are 3 halos in this file.
samples = 5000
traces = halo_posteriors(3, data, samples = samples, burn_in=500)
```

```
Applied interval-transform to mass_large and added transformed mass_large_interval_ to model.
Applied interval-transform to halo_positions and added transformed halo_positions_interval_ to
Iteration 0 [0%]: ELBO = 102.73
Iteration 5000 [10%]: Average ELBO = 113.4
Iteration 10000 [20%]: Average ELBO = 128.11
Iteration 15000 [30%]: Average ELBO = 131.97
Iteration 20000 [40%]: Average ELBO = 132.94
Iteration 25000 [50%]: Average ELBO = 133.52
Iteration 30000 [60%]: Average ELBO = 133.95
Iteration 35000 [70%]: Average ELBO = 134.24
Iteration 40000 [80%]: Average ELBO = 134.31
Iteration 45000 [90%]: Average ELBO = 134.43
Finished [100%]: Average ELBO = 134.31
 [-------100%-------] 5000 of 5000 in 621.1 sec. | SPS: 8.1 | ETA: -0.0
```

In [23]:
```
fig = draw_sky(data)
plt.title("Galaxy positions and ellipcities of sky %d." % n_sky)
plt.xlabel("x-position")
plt.ylabel("y-position")

colors = ["#467821", "#A60628", "#7A68A6"]

for i in range(traces.shape[1]):
```

```
    plt.scatter(traces[:, i, 0],  traces[:, i, 1], c = colors[i], alpha = 0.02)


for i in range(traces.shape[1]):
    plt.scatter(halo_data[n_sky-1][3 + 2*i], halo_data[n_sky-1][4 + 2*i],
            label = "True halo position", c = "k", s = 90)

#plt.legend(scatterpoints = 1)
plt.xlim(0, 4200)
plt.ylim(0, 4200);
```

This looks pretty good, though it took a long time for the system to (sort of) converge. Our optimization step would look something like this:

```
In [24]: _halo_data = halo_data[n_sky-1]
         print(traces.shape)

         mean_posterior = traces.mean(axis=0).reshape(1,6)
         print(mean_posterior)


         nhalo_all =  _halo_data[0].reshape(1,1)
         x_true_all = _halo_data[3].reshape(1,1)
         y_true_all = _halo_data[4].reshape(1,1)
         x_ref_all = _halo_data[1].reshape(1,1)
         y_ref_all = _halo_data[2].reshape(1,1)
         sky_prediction = mean_posterior


         print("Using the mean:")
         main_score([1], x_true_all, y_true_all, \
                    x_ref_all, y_ref_all, sky_prediction)

         #what's a bad score?
         random_guess = np.random.randint(0, 4200, size=(1,2))
         print("Using a random location:", random_guess)
         main_score([1], x_true_all, y_true_all, \
                    x_ref_all, y_ref_all, random_guess)
```

```
(4500, 3, 2)
[[ 3790.95827012  3794.60133909  3170.00252772  3142.31644835
   2265.01634741  3623.32277728]]
Using the mean:
Your average distance in pixels you are away from the true halo is 175.125422038
Your average angular vector is 1.0
Your score for the training data is 1.17512542204
Using a random location: [[1199 3589]]
Your average distance in pixels you are away from the true halo is 2522.2681726
Your average angular vector is 1.0
Your score for the training data is 3.5222681726
```

```
Out[24]: 3.5222681725978306
```

1.4 References

1. Antifragile: Things That Gain from Disorder. New York: Random House. 2012. ISBN 978-1-4000-6782-4.
2. Tim Saliman's solution to the Dark World's Contest
3. Silver, Nate. The Signal and the Noise: Why So Many Predictions Fail — but Some Don't. 1. Penguin Press HC, The, 2012. Print.

In [3]:
```
from IPython.core.display import HTML
def css_styling():
    styles = open("../styles/custom.css", "r").read()
    return HTML(styles)
css_styling()
```

Out[3]:
```
<IPython.core.display.HTML at 0x510f198>
```

Ch6_Priors_PyMC2

January 4, 2018

1 Chapter 6

This chapter of Bayesian Methods for Hackers focuses on the most debated and discussed part of Bayesian methodologies: how to choose an appropriate prior distribution. We also present how the prior's influence changes as our dataset increases, and an interesting relationship between priors and penalties on linear regression.

1.1 Getting our priorities straight

Up until now, we have mostly ignored our choice of priors. This is unfortunate as we can be very expressive with our priors, but we also must be careful about choosing them. This is especially true if we want to be objective, that is, not to express any personal beliefs in the priors.

1.1.1 Subjective vs Objective priors

Bayesian priors can be classified into two classes: *objective* priors, which aim to allow the data to influence the posterior the most, and *subjective* priors, which allow the practitioner to express his or her views into the prior.

What is an example of an objective prior? We have seen some already, including the *flat* prior, which is a uniform distribution over the entire possible range of the unknown. Using a flat prior implies that we give each possible value an equal weighting. Choosing this type of prior is invoking what is called "The Principle of Indifference", literally we have no prior reason to favor one value over another. Calling a flat prior over a restricted space an objective prior is not correct, though it seems similar. If we know p in a Binomial model is greater than 0.5, then $\text{Uniform}(0.5, 1)$ is not an objective prior (since we have used prior knowledge) even though it is "flat" over [0.5, 1]. The flat prior must be flat along the *entire* range of possibilities.

Aside from the flat prior, other examples of objective priors are less obvious, but they contain important characteristics that reflect objectivity. For now, it should be said that *rarely* is a objective prior *truly* objective. We will see this later.

Subjective Priors On the other hand, if we added more probability mass to certain areas of the prior, and less elsewhere, we are biasing our inference towards the unknowns existing in the former area. This is known as a subjective, or *informative* prior. In the figure below, the subjective prior reflects a belief that the unknown likely lives around 0.5, and not around the extremes. The objective prior is insensitive to this.

```
In [1]: %matplotlib inline
        import numpy as np
        from IPython.core.pylabtools import figsize
        import matplotlib.pyplot as plt
        import scipy.stats as stats

        figsize(12.5, 3)
        colors = ["#348ABD", "#A60628", "#7A68A6", "#467821"]

        x = np.linspace(0, 1)
        y1, y2 = stats.beta.pdf(x, 1, 1), stats.beta.pdf(x, 10, 10)

        p = plt.plot(x, y1,
                     label='An objective prior \n(uninformative, \n"Principle of Indifference"
        plt.fill_between(x, 0, y1, color=p[0].get_color(), alpha=0.3)

        p = plt.plot(x, y2,
                     label="A subjective prior \n(informative)")
        plt.fill_between(x, 0, y2, color=p[0].get_color(), alpha=0.3)

        p = plt.plot(x[25:], 2 * np.ones(25), label="another subjective prior")
        plt.fill_between(x[25:], 0, 2, color=p[0].get_color(), alpha=0.3)

        plt.ylim(0, 4)

        plt.ylim(0, 4)
        leg = plt.legend(loc="upper left")
        leg.get_frame().set_alpha(0.4)
        plt.title("Comparing objective vs. subjective priors for an unknown probability");
```

The choice of a subjective prior does not always imply that we are using the practitioner's subjective opinion: more often the subjective prior was once a posterior to a previous problem, and now the practitioner is updating this posterior with new data. A subjective prior can also be used to inject *domain knowledge* of the problem into the model. We will see examples of these two situations later.

1.1.2 Decision, decisions...

The choice, either *objective* or *subjective* mostly depends on the problem being solved, but there are a few cases where one is preferred over the other. In instances of scientific research, the choice of an objective prior is obvious. This eliminates any biases in the results, and two researchers who might have differing prior opinions would feel an objective prior is fair. Consider a more extreme situation:

> A tobacco company publishes a report with a Bayesian methodology that retreated 60 years of medical research on tobacco use. Would you believe the results? Unlikely. The researchers probably chose a subjective prior that too strongly biased results in their favor.

Unfortunately, choosing an objective prior is not as simple as selecting a flat prior, and even today the problem is still not completely solved. The problem with naively choosing the uniform prior is that pathological issues can arise. Some of these issues are pedantic, but we delay more serious issues to the Appendix of this Chapter (TODO).

We must remember that choosing a prior, whether subjective or objective, is still part of the modeling process. To quote Gelman [5]:

> ...after the model has been fit, one should look at the posterior distribution and see if it makes sense. If the posterior distribution does not make sense, this implies that additional prior knowledge is available that has not been included in the model, and that contradicts the assumptions of the prior distribution that has been used. It is then appropriate to go back and alter the prior distribution to be more consistent with this external knowledge.

If the posterior does not make sense, then clearly one had an idea what the posterior *should* look like (not what one *hopes* it looks like), implying that the current prior does not contain all the prior information and should be updated. At this point, we can discard the current prior and choose a more reflective one.

Gelman [4] suggests that using a uniform distribution with large bounds is often a good choice for objective priors. Although, one should be wary about using Uniform objective priors with large bounds, as they can assign too large of a prior probability to non-intuitive points. Ask yourself: do you really think the unknown could be incredibly large? Often quantities are naturally biased towards 0. A Normal random variable with large variance (small precision) might be a better choice, or an Exponential with a fat tail in the strictly positive (or negative) case.

If using a particularly subjective prior, it is your responsibility to be able to explain the choice of that prior, else you are no better than the tobacco company's guilty parties.

1.1.3 Empirical Bayes

While not a true Bayesian method, *empirical Bayes* is a trick that combines frequentist and Bayesian inference. As mentioned previously, for (almost) every inference problem there is a Bayesian method and a frequentist method. The significant difference between the two is that Bayesian methods have a prior distribution, with hyperparameters α, while empirical methods do not have any notion of a prior. Empirical Bayes combines the two methods by using frequentist methods to select α, and then proceeds with Bayesian methods on the original problem.

A very simple example follows: suppose we wish to estimate the parameter μ of a Normal distribution, with $\sigma = 5$. Since μ could range over the whole real line, we can use a Normal

distribution as a prior for μ. How to select the prior's hyperparameters, denoted (μ_p, σ_p^2)? The σ_p^2 parameter can be chosen to reflect the uncertainty we have. For μ_p, we have two options:

1. Empirical Bayes suggests using the empirical sample mean, which will center the prior around the observed empirical mean:

$$\mu_p = \frac{1}{N} \sum_{i=0}^{N} X_i$$

2. Traditional Bayesian inference suggests using prior knowledge, or a more objective prior (zero mean and fat standard deviation).

Empirical Bayes can be argued as being semi-objective, since while the choice of prior model is ours (hence subjective), the parameters are solely determined by the data.

Personally, I feel that Empirical Bayes is *double-counting* the data. That is, we are using the data twice: once in the prior, which will influence our results towards the observed data, and again in the inferential engine of MCMC. This double-counting will understate our true uncertainty. To minimize this double-counting, I would only suggest using Empirical Bayes when you have *lots* of observations, else the prior will have too strong of an influence. I would also recommend, if possible, to maintain high uncertainty (either by setting a large σ_p^2 or equivalent.)

Empirical Bayes also violates a theoretical axiom in Bayesian inference. The textbook Bayesian algorithm of:

prior $\Rightarrow$ *observed data* $\Rightarrow$ *posterior*

is violated by Empirical Bayes, which instead uses

observed data $\Rightarrow$ *prior* $\Rightarrow$ *observed data* $\Rightarrow$ *posterior*

Ideally, all priors should be specified *before* we observe the data, so that the data does not influence our prior opinions (see the volumes of research by Daniel Kahneman *et. al* about anchoring).

1.2 Useful priors to know about

1.2.1 The Gamma distribution

A Gamma random variable, denoted $X \sim \text{Gamma}(\alpha, \beta)$, is a random variable over the positive real numbers. It is in fact a generalization of the Exponential random variable, that is:

$$\text{Exp}(\beta) \sim \text{Gamma}(1, \beta)$$

This additional parameter allows the probability density function to have more flexibility, hence allowing the practitioner to express his or her subjective priors more accurately. The density function for a $\text{Gamma}(\alpha, \beta)$ random variable is:

$$f(x \mid \alpha, \beta) = \frac{\beta^\alpha x^{\alpha-1} e^{-\beta x}}{\Gamma(\alpha)}$$

where $\Gamma(\alpha)$ is the Gamma function, and for differing values of (α, β) looks like:

```
In [2]: figsize(12.5, 5)
        gamma = stats.gamma

        parameters = [(1, 0.5), (9, 2), (3, 0.5), (7, 0.5)]
        x = np.linspace(0.001, 20, 150)
        for alpha, beta in parameters:
            y = gamma.pdf(x, alpha, scale=1. / beta)
            lines = plt.plot(x, y, label="(%.1f,%.1f)" % (alpha, beta), lw=3)
            plt.fill_between(x, 0, y, alpha=0.2, color=lines[0].get_color())
            plt.autoscale(tight=True)

        plt.legend(title=r"$\alpha, \beta$ - parameters");
```

1.2.2 The Wishart distribution

Until now, we have only seen random variables that are scalars. Of course, we can also have *random matrices*! Specifically, the Wishart distribution is a distribution over all positive semi-definite matrices. Why is this useful to have in our arsenal? (Proper) covariance matrices are positive-definite, hence the Wishart is an appropriate prior for covariance matrices. We can't really visualize a distribution of matrices, so I'll plot some realizations from the 5×5 (above) and 20×20 (below) Wishart distribution:

```
In [3]: import pymc as pm

        n = 4
        for i in range(10):
            ax = plt.subplot(2, 5, i + 1)
            if i >= 5:
                n = 15
            plt.imshow(pm.rwishart(n + 1, np.eye(n)), interpolation="none",
                       cmap=plt.cm.hot)
```

```
    ax.axis("off")

plt.suptitle("Random matrices from a Wishart Distribution");
```

One thing to notice is the symmetry of these matrices. The Wishart distribution can be a little troubling to deal with, but we will use it in an example later.

1.2.3 The Beta distribution

You may have seen the term `beta` in previous code in this book. Often, I was implementing a Beta distribution. The Beta distribution is very useful in Bayesian statistics. A random variable X has a Beta distribution, with parameters (α, β), if its density function is:

$$f_X(x|\ \alpha, \beta) = \frac{x^{(\alpha-1)}(1-x)^{(\beta-1)}}{B(\alpha, \beta)}$$

where B is the Beta function (hence the name). The random variable X is only allowed in [0,1], making the Beta distribution a popular distribution for decimal values, probabilities and proportions. The values of α and β, both positive values, provide great flexibility in the shape of the distribution. Below we plot some distributions:

```
In [4]: figsize(12.5, 5)

        params = [(2, 5), (1, 1), (0.5, 0.5), (5, 5), (20, 4), (5, 1)]

        x = np.linspace(0.01, .99, 100)
        beta = stats.beta
        for a, b in params:
            y = beta.pdf(x, a, b)
            lines = plt.plot(x, y, label="(%.1f,%.1f)" % (a, b), lw=3)
            plt.fill_between(x, 0, y, alpha=0.2, color=lines[0].get_color())
            plt.autoscale(tight=True)
```

```
plt.ylim(0)
plt.legend(loc='upper left', title="(a,b)-parameters");
```

One thing I'd like the reader to notice is the presence of the flat distribution above, specified by parameters $(1, 1)$. This is the Uniform distribution. Hence the Beta distribution is a generalization of the Uniform distribution, something we will revisit many times.

There is an interesting connection between the Beta distribution and the Binomial distribution. Suppose we are interested in some unknown proportion or probability p. We assign a $\text{Beta}(\alpha, \beta)$ prior to p. We observe some data generated by a Binomial process, say $X \sim \text{Binomial}(N, p)$, with p still unknown. Then our posterior *is again a Beta distribution*, i.e. $p|X \sim \text{Beta}(\alpha + X, \beta + N - X)$. Succinctly, one can relate the two by "a Beta prior with Binomial observations creates a Beta posterior". This is a very useful property, both computationally and heuristically.

In light of the above two paragraphs, if we start with a $\text{Beta}(1, 1)$ prior on p (which is a Uniform), observe data $X \sim \text{Binomial}(N, p)$, then our posterior is $\text{Beta}(1 + X, 1 + N - X)$.

Example: Bayesian Multi-Armed Bandits *Adapted from an example by Ted Dunning of MapR Technologies*

> Suppose you are faced with N slot machines (colourfully called multi-armed bandits). Each bandit has an unknown probability of distributing a prize (assume for now the prizes are the same for each bandit, only the probabilities differ). Some bandits are very generous, others not so much. Of course, you don't know what these probabilities are. By only choosing one bandit per round, our task is devise a strategy to maximize our winnings.

Of course, if we knew the bandit with the largest probability, then always picking this bandit would yield the maximum winnings. So our task can be phrased as "Find the best bandit, and as quickly as possible".

The task is complicated by the stochastic nature of the bandits. A suboptimal bandit can return many winnings, purely by chance, which would make us believe that it is a very profitable bandit. Similarly, the best bandit can return many duds. Should we keep trying losers then, or give up?

A more troublesome problem is, if we have a found a bandit that returns *pretty good* results, do we keep drawing from it to maintain our *pretty good score*, or do we try other bandits in hopes of finding an *even-better* bandit? This is the exploration vs. exploitation dilemma.

1.2.4 Applications

The Multi-Armed Bandit problem at first seems very artificial, something only a mathematician would love, but that is only before we address some applications:

- Internet display advertising: companies have a suite of potential ads they can display to visitors, but the company is not sure which ad strategy to follow to maximize sales. This is similar to A/B testing, but has the added advantage of naturally minimizing strategies that do not work (and generalizes to A/B/C/D... strategies)
- Ecology: animals have a finite amount of energy to expend, and following certain behaviours has uncertain rewards. How does the animal maximize its fitness?
- Finance: which stock option gives the highest return, under time-varying return profiles.
- Clinical trials: a researcher would like to find the best treatment, out of many possible treatment, while minimizing losses.
- Psychology: how does punishment and reward affect our behaviour? How do humans learn?

Many of these questions above are fundamental to the application's field.

It turns out the *optimal solution* is incredibly difficult, and it took decades for an overall solution to develop. There are also many approximately-optimal solutions which are quite good. The one I wish to discuss is one of the few solutions that can scale incredibly well. The solution is known as *Bayesian Bandits*.

1.2.5 A Proposed Solution

Any proposed strategy is called an *online algorithm* (not in the internet sense, but in the continuously-being-updated sense), and more specifically a reinforcement learning algorithm. The algorithm starts in an ignorant state, where it knows nothing, and begins to acquire data by testing the system. As it acquires data and results, it learns what the best and worst behaviours are (in this case, it learns which bandit is the best). With this in mind, perhaps we can add an additional application of the Multi-Armed Bandit problem:

- Psychology: how does punishment and reward affect our behaviour? How do humans learn?

The Bayesian solution begins by assuming priors on the probability of winning for each bandit. In our vignette we assumed complete ignorance of these probabilities. So a very natural prior is the flat prior over 0 to 1. The algorithm proceeds as follows:

For each round:

1. Sample a random variable X_b from the prior of bandit b, for all b.
2. Select the bandit with largest sample, i.e. select $B = \text{argmax } X_b$.
3. Observe the result of pulling bandit B, and update your prior on bandit B.
4. Return to 1.

That's it. Computationally, the algorithm involves sampling from N distributions. Since the initial priors are $\text{Beta}(\alpha = 1, \beta = 1)$ (a uniform distribution), and the observed result X (a win or loss, encoded 1 and 0 respectfully) is Binomial, the posterior is a $\text{Beta}(\alpha = 1 + X, \beta = 1 + 1X)$.

To answer our question from before, this algorithm suggests that we should not discard losers, but we should pick them at a decreasing rate as we gather confidence that there exist *better* bandits. This follows because there is always a non-zero chance that a loser will achieve the status of B, but the probability of this event decreases as we play more rounds (see figure below).

Below we implement Bayesian Bandits using two classes, `Bandits` that defines the slot machines, and `BayesianStrategy` which implements the above learning strategy.

```
In [5]: from pymc import rbeta


        class Bandits(object):

            """
            This class represents N bandits machines.

            parameters:
                p_array: a (n,) Numpy array of probabilities >0, <1.

            methods:
                pull( i ): return the results, 0 or 1, of pulling
                           the ith bandit.
            """

            def __init__(self, p_array):
                self.p = p_array
                self.optimal = np.argmax(p_array)

            def pull(self, i):
                # i is which arm to pull
                return np.random.rand() < self.p[i]

            def __len__(self):
                return len(self.p)


        class BayesianStrategy(object):

            """
            Implements a online, learning strategy to solve
            the Multi-Armed Bandit problem.

            parameters:
                bandits: a Bandit class with .pull method

            methods:
```

```
        sample_bandits(n): sample and train on n pulls.

    attributes:
        N: the cumulative number of samples
        choices: the historical choices as a (N,) array
        bb_score: the historical score as a (N,) array
    """

    def __init__(self, bandits):

        self.bandits = bandits
        n_bandits = len(self.bandits)
        self.wins = np.zeros(n_bandits)
        self.trials = np.zeros(n_bandits)
        self.N = 0
        self.choices = []
        self.bb_score = []

    def sample_bandits(self, n=1):

        bb_score = np.zeros(n)
        choices = np.zeros(n)

        for k in range(n):
            # sample from the bandits's priors, and select the largest sample
            choice = np.argmax(rbeta(1 + self.wins, 1 + self.trials - self.wins))

            # sample the chosen bandit
            result = self.bandits.pull(choice)

            # update priors and score
            self.wins[choice] += result
            self.trials[choice] += 1
            bb_score[k] = result
            self.N += 1
            choices[k] = choice

        self.bb_score = np.r_[self.bb_score, bb_score]
        self.choices = np.r_[self.choices, choices]
        return
```

Below we visualize the learning of the Bayesian Bandit solution.

```
In [6]: figsize(11.0, 10)

        beta = stats.beta
        x = np.linspace(0.001, .999, 200)
```

```
def plot_priors(bayesian_strategy, prob, lw=3, alpha=0.2, plt_vlines=True):
    # plotting function
    wins = bayesian_strategy.wins
    trials = bayesian_strategy.trials
    for i in range(prob.shape[0]):
        y = beta(1 + wins[i], 1 + trials[i] - wins[i])
        p = plt.plot(x, y.pdf(x), lw=lw)
        c = p[0].get_markeredgecolor()
        plt.fill_between(x, y.pdf(x), 0, color=c, alpha=alpha,
                         label="underlying probability: %.2f" % prob[i])
        if plt_vlines:
            plt.vlines(prob[i], 0, y.pdf(prob[i]),
                       colors=c, linestyles="--", lw=2)
        plt.autoscale(tight="True")
        plt.title("Posteriors After %d pull" % bayesian_strategy.N +
                  "s" * (bayesian_strategy.N > 1))
        plt.autoscale(tight=True)
    return
```

```
In [7]: hidden_prob = np.array([0.85, 0.60, 0.75])
        bandits = Bandits(hidden_prob)
        bayesian_strat = BayesianStrategy(bandits)

        draw_samples = [1, 1, 3, 10, 10, 25, 50, 100, 200, 600]

        for j, i in enumerate(draw_samples):
            plt.subplot(5, 2, j + 1)
            bayesian_strat.sample_bandits(i)
            plot_priors(bayesian_strat, hidden_prob)
            # plt.legend()
            plt.autoscale(tight=True)
        plt.tight_layout()
```

Note that we don't really care how accurate we become about the inference of the hidden probabilities — for this problem we are more interested in choosing the best bandit (or more accurately, becoming *more confident* in choosing the best bandit). For this reason, the distribution of the red bandit is very wide (representing ignorance about what that hidden probability might be) but we are reasonably confident that it is not the best, so the algorithm chooses to ignore it.

From the above, we can see that after 1000 pulls, the majority of the "blue" function leads the pack, hence we will almost always choose this arm. This is good, as this arm is indeed the best.

Below is a D3 app that demonstrates our algorithm updating/learning three bandits. The first figure shows the raw counts of pulls and wins, and the second figure is a dynamically updating plot. I encourage you to try to guess which bandit is optimal, prior to revealing the true probabilities, by selecting the `arm buttons`.

```
In [8]: from IPython.core.display import HTML

        # try executing the below command twice if the first time doesn't work
        HTML(filename="BanditsD3.html")

Out[8]: <IPython.core.display.HTML at 0x106822050>
```

Deviations of the observed ratio from the highest probability is a measure of performance. For example,in the long run, optimally we can attain the reward/pull ratio of the maximum bandit probability. Long-term realized ratios less than the maximum represent inefficiencies. (Realized ratios larger than the maximum probability is due to randomness, and will eventually fall below).

1.2.6 A Measure of *Good*

We need a metric to calculate how well we are doing. Recall the absolute *best* we can do is to always pick the bandit with the largest probability of winning. Denote this best bandit's probability of w_{opt}. Our score should be relative to how well we would have done had we chosen the best bandit from the beginning. This motivates the *total regret* of a strategy, defined as:

$$R_T = \sum_{i=1}^{T} \left(w_{opt} - w_{B(i)} \right) \tag{1}$$

$$\tag{2}$$

$$= Tw^* - \sum_{i=1}^{T} w_{B(i)} \tag{3}$$

where $w_{B(i)}$ is the probability of a prize of the chosen bandit in the ith round. A total regret of 0 means the strategy is attaining the best possible score. This is likely not possible, as initially our algorithm will often make the wrong choice. Ideally, a strategy's total regret should flatten as it learns the best bandit. (Mathematically, we achieve $w_{B(i)} = w_{opt}$ often)

Below we plot the total regret of this simulation, including the scores of some other strategies:

1. Random: randomly choose a bandit to pull. If you can't beat this, just stop.
2. Largest Bayesian credible bound: pick the bandit with the largest upper bound in its 95% credible region of the underlying probability.
3. Bayes-UCB algorithm: pick the bandit with the largest *score*, where score is a dynamic quantile of the posterior (see [4])
4. Mean of posterior: choose the bandit with the largest posterior mean. This is what a human player (sans computer) would likely do.
5. Largest proportion: pick the bandit with the current largest observed proportion of winning.

The code for these are in the `other_strats.py`, where you can implement your own strategy very easily.

```
In [9]: figsize(12.5, 5)
        from other_strats import GeneralBanditStrat, bayesian_bandit_choice, max_mean, lower_c
                                 upper_credible_choice, random_choice, ucb_bayes, Bandits

        # define a harder problem
        hidden_prob = np.array([0.15, 0.2, 0.1, 0.05])
        bandits = Bandits(hidden_prob)

        # define regret
```

```
def regret(probabilities, choices):
    w_opt = probabilities.max()
    return (w_opt - probabilities[choices.astype(int)]).cumsum()

# create new strategies
strategies = [upper_credible_choice,
              bayesian_bandit_choice,
              ucb_bayes,
              max_mean,
              random_choice]
algos = []
for strat in strategies:
    algos.append(GeneralBanditStrat(bandits, strat))
```

```
In [10]: # train 10000 times
         for strat in algos:
             strat.sample_bandits(10000)

         #test and plot
         for i, strat in enumerate(algos):
             _regret = regret(hidden_prob, strat.choices)
             plt.plot(_regret, label=strategies[i].__name__, lw=3)

         plt.title("Total Regret of Bayesian Bandits Strategy vs. Random guessing")
         plt.xlabel("Number of pulls")
         plt.ylabel("Regret after $n$ pulls");
         plt.legend(loc="upper left");
```

Like we wanted, Bayesian bandits and other strategies have decreasing rates of regret, representing that we are achieving optimal choices. To be more scientific so as to remove any possible luck in the above simulation, we should instead look at the *expected total regret*:

$$\bar{R_T} = E[R_T]$$

It can be shown that any *sub-optimal* strategy's expected total regret is bounded below logarithmically. Formally:

$$E[R_T] = \Omega\,(\,\log(T)\,)$$

Thus, any strategy that matches logarithmic-growing regret is said to "solve" the Multi-Armed Bandit problem [3].

Using the Law of Large Numbers, we can approximate Bayesian Bandit's expected total regret by performing the same experiment many times (500 times, to be fair):

```
In [14]: # this can be slow, so I recommend NOT running it.
         trials = 200
         expected_total_regret = np.zeros((1000, 3))

         for i_strat, strat in enumerate(strategies[:-2]):
             for i in range(trials):
                 general_strat = GeneralBanditStrat(bandits, strat)
                 general_strat.sample_bandits(1000)
                 _regret = regret(hidden_prob, general_strat.choices)
                 expected_total_regret[:, i_strat] += _regret

             plt.plot(expected_total_regret[:, i_strat] / trials, lw=3, label=strat.__name__)

         plt.title("Expected Total Regret of Multi-armed Bandit strategies")
         plt.xlabel("Number of pulls")
         plt.ylabel("Exepected Total Regret \n after $n$ pulls");
         plt.legend(loc="upper left");
```

```
In [15]: plt.figure()
         [pl1, pl2, pl3] = plt.plot(expected_total_regret[:, [0, 1, 2]], lw=3)
         plt.xscale("log")
         plt.legend([pl1, pl2, pl3],
                    ["Upper Credible Bound", "Bayesian Bandit", "UCB-Bayes"],
                    loc="upper left")
         plt.ylabel("Exepected Total Regret \n after $\log{n}$ pulls");
         plt.title("log-scale of above");
         plt.ylabel("Exepected Total Regret \n after $\log{n}$ pulls");
```

1.2.7 Extending the algorithm

Because of the Bayesian Bandits algorithm's simplicity, it is easy to extend. Some possibilities are:

- If interested in the *minimum* probability (eg: where prizes are a bad thing), simply choose $B = \text{argmin } X_b$ and proceed.
- Adding learning rates: Suppose the underlying environment may change over time. Technically the standard Bayesian Bandit algorithm would self-update itself (awesome) by noting that what it thought was the best is starting to fail more often. We can motivate the algorithm to learn changing environments quicker by simply adding a *rate* term upon updating:

```
self.wins[ choice ] = rate*self.wins[ choice ] + result
self.trials[ choice ] = rate*self.trials[ choice ] + 1
```

If `rate < 1`, the algorithm will *forget* its previous wins quicker and there will be a downward pressure towards ignorance. Conversely, setting `rate > 1` implies your algorithm will act more risky, and bet on earlier winners more often and be more resistant to changing environments.

- Hierarchical algorithms: We can setup a Bayesian Bandit algorithm on top of smaller bandit algorithms. Suppose we have N Bayesian Bandit models, each varying in some behavior

(for example different `rate` parameters, representing varying sensitivity to changing environments). On top of these N models is another Bayesian Bandit learner that will select a sub-Bayesian Bandit. This chosen Bayesian Bandit will then make an internal choice as to which machine to pull. The super-Bayesian Bandit updates itself depending on whether the sub-Bayesian Bandit was correct or not.

- Extending the rewards, denoted y_a for bandit a, to random variables from a distribution $f_{y_a}(y)$ is straightforward. More generally, this problem can be rephrased as "Find the bandit with the largest expected value", as playing the bandit with the largest expected value is optimal. In the case above, f_{y_a} was Bernoulli with probability p_a, hence the expected value for a bandit is equal to p_a, which is why it looks like we are aiming to maximize the probability of winning. If f is not Bernoulli, and it is non-negative, which can be accomplished a priori by shifting the distribution (we assume we know f), then the algorithm behaves as before:

For each round,

1. Sample a random variable X_b from the prior of bandit b, for all b.
2. Select the bandit with largest sample, i.e. select bandit $B = \text{argmax } X_b$.
3. Observe the result,$R \sim f_{y_b}$, of pulling bandit B, and update your prior on bandit B.
4. Return to 1

The issue is in the sampling of the X_b drawing phase. With Beta priors and Bernoulli observations, we have a Beta posterior — this is easy to sample from. But now, with arbitrary distributions f, we have a non-trivial posterior. Sampling from these can be difficult.

- There has been some interest in extending the Bayesian Bandit algorithm to commenting systems. Recall in Chapter 4, we developed a ranking algorithm based on the Bayesian lower-bound of the proportion of upvotes to the total number of votes. One problem with this approach is that it will bias the top rankings towards older comments, since older comments naturally have more votes (and hence the lower-bound is tighter to the true proportion). This creates a positive feedback cycle where older comments gain more votes, hence are displayed more often, hence gain more votes, etc. This pushes any new, potentially better comments, towards the bottom. J. Neufeld proposes a system to remedy this that uses a Bayesian Bandit solution.

His proposal is to consider each comment as a Bandit, with the number of pulls equal to the number of votes cast, and number of rewards as the number of upvotes, hence creating a Beta$(1 + U, 1 + D)$ posterior. As visitors visit the page, samples are drawn from each bandit/comment, but instead of displaying the comment with the max sample, the comments are ranked according to the ranking of their respective samples. From J. Neufeld's blog [7]:

> [The] resulting ranking algorithm is quite straightforward, each new time the comments page is loaded, the score for each comment is sampled from a Beta$(1 + U, 1 + D)$, comments are then ranked by this score in descending order... This randomization has a unique benefit in that even untouched comments $(U = 0, D = 0)$ have some chance of being seen even in threads with 5000+ comments (something that is not happening now), but, at the same time, the user is not likely to be inundated with rating these new comments.

Just for fun, though the colors explode, we watch the Bayesian Bandit algorithm learn 35 different options.

```
In [16]: figsize(12.0, 8)
         beta = stats.beta
         hidden_prob = beta.rvs(1, 13, size=35)
         print(hidden_prob)
         bandits = Bandits(hidden_prob)
         bayesian_strat = BayesianStrategy(bandits)

         for j, i in enumerate([100, 200, 500, 1300]):
             plt.subplot(2, 2, j + 1)
             bayesian_strat.sample_bandits(i)
             plot_priors(bayesian_strat, hidden_prob, lw=2, alpha=0.0, plt_vlines=False)
             # plt.legend()
             plt.xlim(0, 0.5)
```

```
[ 0.0431  0.0745  0.1187  0.0098  0.0945  0.0438  0.0442  0.0059  0.0749
  0.023   0.0543  0.025   0.1231  0.0148  0.0164  0.2688  0.0073  0.0564
  0.0031  0.0698  0.0478  0.1657  0.0091  0.0384  0.2236  0.1548  0.0562
  0.0209  0.024   0.0197  0.0788  0.0572  0.1207  0.0405  0.0679]
```

1.3 Eliciting expert prior

Specifying a subjective prior is how practitioners incorporate domain knowledge about the problem into our mathematical framework. Allowing domain knowledge is useful for many reasons, for example:

- Aids the speed of MCMC convergence. For example, if we know the unknown parameter is strictly positive, then we can restrict our attention there, hence saving time that would otherwise be spent exploring negative values.
- More accurate inference. By weighing prior values near the true unknown value higher, we are narrowing our eventual inference (by making the posterior tighter around the unknown)
- Express our uncertainty better. See the *Price is Right* problem in Chapter 5.

Of course, practitioners of Bayesian methods are not experts in every field, so we must turn to domain experts to craft our priors. We must be careful with how we elicit these priors though. Some things to consider:

1. From experience, I would avoid introducing Betas, Gammas, etc. to non-Bayesian practitioners. Furthermore, non-statisticians can get tripped up by how a continuous probability function can have a value exceeding one.

2. Individuals often neglect the rare *tail-events* and put too much weight around the mean of distribution.

3. Related to above is that almost always individuals will under-emphasize the uncertainty in their guesses.

Eliciting priors from non-technical experts is especially difficult. Rather than introduce the notion of probability distributions, priors, etc. that may scare an expert, there is a much simpler solution.

1.3.1 Trial roulette method

The *trial roulette method* [8] focuses on building a prior distribution by placing counters (think casino chips) on what the expert thinks are possible outcomes. The expert is given N counters (say $N = 20$) and is asked to place them on a pre-printed grid, with bins representing intervals. Each column would represent their belief of the probability of getting the corresponding bin result. Each chip would represent an $\frac{1}{N} = 0.05$ increase in the probability of the outcome being in that interval. For example [9]:

> A student is asked to predict the mark in a future exam. The figure below shows a completed grid for the elicitation of a subjective probability distribution. The horizontal axis of the grid shows the possible bins (or mark intervals) that the student was asked to consider. The numbers in top row record the number of chips per bin. The completed grid (using a total of 20 chips) shows that the student believes there is a 30% chance that the mark will be between 60 and 64.9.

From this, we can fit a distribution that captures the expert's choice. Some reasons in favor of using this technique are:

Ch6_Priors_PyMC3

January 4, 2018

1 Chapter 6

```
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)
```

This chapter of Bayesian Methods for Hackers focuses on the most debated and discussed part of Bayesian methodologies: how to choose an appropriate prior distribution. We also present how the prior's influence changes as our dataset increases, and an interesting relationship between priors and penalties on linear regression.

1.1 Getting our priorities straight

Up until now, we have mostly ignored our choice of priors. This is unfortunate as we can be very expressive with our priors, but we also must be careful about choosing them. This is especially true if we want to be objective, that is, not to express any personal beliefs in the priors.

1.1.1 Subjective vs Objective priors

Bayesian priors can be classified into two classes: *objective* priors, which aim to allow the data to influence the posterior the most, and *subjective* priors, which allow the practitioner to express his or her views into the prior.

What is an example of an objective prior? We have seen some already, including the *flat* prior, which is a uniform distribution over the entire possible range of the unknown. Using a flat prior implies that we give each possible value an equal weighting. Choosing this type of prior is invoking what is called "The Principle of Indifference", literally we have no prior reason to favor one value over another. Calling a flat prior over a restricted space an objective prior is not correct, though it seems similar. If we know p in a Binomial model is greater than 0.5, then $\text{Uniform}(0.5, 1)$ is not an objective prior (since we have used prior knowledge) even though it is "flat" over [0.5, 1]. The flat prior must be flat along the *entire* range of possibilities.

Aside from the flat prior, other examples of objective priors are less obvious, but they contain important characteristics that reflect objectivity. For now, it should be said that *rarely* is a objective prior *truly* objective. We will see this later.

Subjective Priors On the other hand, if we added more probability mass to certain areas of the prior, and less elsewhere, we are biasing our inference towards the unknowns existing in the former area. This is known as a subjective, or *informative* prior. In the figure below, the subjective prior reflects a belief that the unknown likely lives around 0.5, and not around the extremes. The objective prior is insensitive to this.

```
In [1]: %matplotlib inline
        import numpy as np
        import scipy.stats as stats
        from IPython.core.pylabtools import figsize
        import matplotlib.pyplot as plt

        figsize(12.5,3)
        colors = ["#348ABD", "#A60628", "#7A68A6", "#467821"]

        x = np.linspace(0,1)
        y1, y2 = stats.beta.pdf(x, 1,1), stats.beta.pdf(x, 10,10)

        p = plt.plot(x, y1,
            label='An objective prior \n(uninformative, \n"Principle of Indifference")')
        plt.fill_between(x, 0, y1, color = p[0].get_color(), alpha = 0.3)

        p = plt.plot(x,y2 ,
             label = "A subjective prior \n(informative)")
        plt.fill_between(x, 0, y2, color = p[0].get_color(), alpha = 0.3)

        p = plt.plot(x[25:], 2*np.ones(25), label = "another subjective prior")
        plt.fill_between(x[25:], 0, 2, color = p[0].get_color(), alpha = 0.3)

        plt.ylim(0,4)

        plt.ylim(0, 4)
        leg = plt.legend(loc = "upper left")
        leg.get_frame().set_alpha(0.4)
        plt.title("Comparing objective vs. subjective priors for an unknown probability");
```

The choice of a subjective prior does not always imply that we are using the practitioner's subjective opinion: more often the subjective prior was once a posterior to a previous problem, and now the practitioner is updating this posterior with new data. A subjective prior can also be used to inject *domain knowledge* of the problem into the model. We will see examples of these two situations later.

1.1.2 Decision, decisions...

The choice, either *objective* or *subjective* mostly depends on the problem being solved, but there are a few cases where one is preferred over the other. In instances of scientific research, the choice of an objective prior is obvious. This eliminates any biases in the results, and two researchers who might have differing prior opinions would feel an objective prior is fair. Consider a more extreme situation:

> A tobacco company publishes a report with a Bayesian methodology that retreated 60 years of medical research on tobacco use. Would you believe the results? Unlikely. The researchers probably chose a subjective prior that too strongly biased results in their favor.

Unfortunately, choosing an objective prior is not as simple as selecting a flat prior, and even today the problem is still not completely solved. The problem with naively choosing the uniform prior is that pathological issues can arise. Some of these issues are pedantic, but we delay more serious issues to the Appendix of this Chapter (TODO).

We must remember that choosing a prior, whether subjective or objective, is still part of the modeling process. To quote Gelman [5]:

> ... after the model has been fit, one should look at the posterior distribution and see if it makes sense. If the posterior distribution does not make sense, this implies that additional prior knowledge is available that has not been included in the model, and that contradicts the assumptions of the prior distribution that has been used. It is then appropriate to go back and alter the prior distribution to be more consistent with this external knowledge.

If the posterior does not make sense, then clearly one had an idea what the posterior *should* look like (not what one *hopes* it looks like), implying that the current prior does not contain all the prior information and should be updated. At this point, we can discard the current prior and choose a more reflective one.

Gelman [4] suggests that using a uniform distribution with large bounds is often a good choice for objective priors. Although, one should be wary about using Uniform objective priors with large bounds, as they can assign too large of a prior probability to non-intuitive points. Ask yourself: do you really think the unknown could be incredibly large? Often quantities are naturally biased towards 0. A Normal random variable with large variance (small precision) might be a better choice, or an Exponential with a fat tail in the strictly positive (or negative) case.

If using a particularly subjective prior, it is your responsibility to be able to explain the choice of that prior, else you are no better than the tobacco company's guilty parties.

1.1.3 Empirical Bayes

While not a true Bayesian method, *empirical Bayes* is a trick that combines frequentist and Bayesian inference. As mentioned previously, for (almost) every inference problem there is a Bayesian

method and a frequentist method. The significant difference between the two is that Bayesian methods have a prior distribution, with hyperparameters α, while empirical methods do not have any notion of a prior. Empirical Bayes combines the two methods by using frequentist methods to select α, and then proceeds with Bayesian methods on the original problem.

A very simple example follows: suppose we wish to estimate the parameter μ of a Normal distribution, with $\sigma = 5$. Since μ could range over the whole real line, we can use a Normal distribution as a prior for μ. How to select the prior's hyperparameters, denoted (μ_p, σ_p^2)? The σ_p^2 parameter can be chosen to reflect the uncertainty we have. For μ_p, we have two options:

1. Empirical Bayes suggests using the empirical sample mean, which will center the prior around the observed empirical mean:

$$\mu_p = \frac{1}{N} \sum_{i=0}^{N} X_i$$

2. Traditional Bayesian inference suggests using prior knowledge, or a more objective prior (zero mean and fat standard deviation).

Empirical Bayes can be argued as being semi-objective, since while the choice of prior model is ours (hence subjective), the parameters are solely determined by the data.

Personally, I feel that Empirical Bayes is *double-counting* the data. That is, we are using the data twice: once in the prior, which will influence our results towards the observed data, and again in the inferential engine of MCMC. This double-counting will understate our true uncertainty. To minimize this double-counting, I would only suggest using Empirical Bayes when you have *lots* of observations, else the prior will have too strong of an influence. I would also recommend, if possible, to maintain high uncertainty (either by setting a large σ_p^2 or equivalent.)

Empirical Bayes also violates a theoretical axiom in Bayesian inference. The textbook Bayesian algorithm of:

prior $\Rightarrow$ *observed data* $\Rightarrow$ *posterior*

is violated by Empirical Bayes, which instead uses

observed data $\Rightarrow$ *prior* $\Rightarrow$ *observed data* $\Rightarrow$ *posterior*

Ideally, all priors should be specified *before* we observe the data, so that the data does not influence our prior opinions (see the volumes of research by Daniel Kahneman *et. al* about anchoring).

1.2 Useful priors to know about

1.2.1 The Gamma distribution

A Gamma random variable, denoted $X \sim \text{Gamma}(\alpha, \beta)$, is a random variable over the positive real numbers. It is in fact a generalization of the Exponential random variable, that is:

$$\text{Exp}(\beta) \sim \text{Gamma}(1, \beta)$$

This additional parameter allows the probability density function to have more flexibility, hence allowing the practitioner to express his or her subjective priors more accurately. The density function for a $\text{Gamma}(\alpha, \beta)$ random variable is:

$$f(x \mid \alpha, \beta) = \frac{\beta^\alpha x^{\alpha-1} e^{-\beta x}}{\Gamma(\alpha)}$$

where $\Gamma(\alpha)$ is the Gamma function, and for differing values of (α, β) looks like:

```
In [2]: figsize(12.5, 5)
        gamma = stats.gamma

        parameters = [(1, 0.5), (9, 2), (3, 0.5), (7, 0.5)]
        x = np.linspace(0.001 ,20, 150)
        for alpha, beta in parameters:
            y = gamma.pdf(x, alpha, scale=1./beta)
            lines = plt.plot(x, y, label = "(%.1f,%.1f)"%(alpha,beta), lw = 3)
            plt.fill_between(x, 0, y, alpha = 0.2, color = lines[0].get_color())
            plt.autoscale(tight=True)

        plt.legend(title=r"$\alpha, \beta$ - parameters");
```

1.2.2 The Wishart distribution

Until now, we have only seen random variables that are scalars. Of course, we can also have *random matrices*! Specifically, the Wishart distribution is a distribution over all positive semi-definite matrices. Why is this useful to have in our arsenal? (Proper) covariance matrices are positive-definite, hence the Wishart is an appropriate prior for covariance matrices. We can't really visualize a distribution of matrices, so I'll plot some realizations from the 5×5 (above) and 20×20 (below) Wishart distribution:

```
In [3]: n = 4
        for i in range(10):
            ax = plt.subplot(2, 5, i+1)
            if i >= 5:
```

```
        n = 15
    plt.imshow(stats.wishart.rvs(n+1, np.eye(n)), interpolation="none",
               cmap = "hot")
    ax.axis("off")

plt.suptitle("Random matrices from a Wishart Distribution");
```

One thing to notice is the symmetry of these matrices. The Wishart distribution can be a little troubling to deal with, but we will use it in an example later.

1.2.3 The Beta distribution

You may have seen the term `beta` in previous code in this book. Often, I was implementing a Beta distribution. The Beta distribution is very useful in Bayesian statistics. A random variable X has a Beta distribution, with parameters (α, β), if its density function is:

$$f_X(x|\ \alpha, \beta) = \frac{x^{(\alpha-1)}(1-x)^{(\beta-1)}}{B(\alpha, \beta)}$$

where B is the Beta function (hence the name). The random variable X is only allowed in [0,1], making the Beta distribution a popular distribution for decimal values, probabilities and proportions. The values of α and β, both positive values, provide great flexibility in the shape of the distribution. Below we plot some distributions:

```
In [4]: figsize(12.5, 5)

        params = [(2, 5), (1, 1), (0.5, 0.5), (5, 5), (20, 4), (5, 1)]

        x = np.linspace(0.01, .99, 100)
        beta = stats.beta
        for a, b in params:
            y = beta.pdf(x, a, b)
```

```
        lines = plt.plot(x, y, label = "(%.1f,%.1f)"%(a,b), lw = 3)
        plt.fill_between(x, 0, y, alpha = 0.2, color = lines[0].get_color())
        plt.autoscale(tight=True)
    plt.ylim(0)
    plt.legend(loc = 'upper left', title="(a,b)-parameters");
```

One thing I'd like the reader to notice is the presence of the flat distribution above, specified by parameters $(1, 1)$. This is the Uniform distribution. Hence the Beta distribution is a generalization of the Uniform distribution, something we will revisit many times.

There is an interesting connection between the Beta distribution and the Binomial distribution. Suppose we are interested in some unknown proportion or probability p. We assign a Beta(α, β) prior to p. We observe some data generated by a Binomial process, say $X \sim$ Binomial(N, p), with p still unknown. Then our posterior *is again a Beta distribution*, i.e. $p|X \sim$ Beta$(\alpha + X, \beta + N - X)$. Succinctly, one can relate the two by "a Beta prior with Binomial observations creates a Beta posterior". This is a very useful property, both computationally and heuristically.

In light of the above two paragraphs, if we start with a Beta$(1, 1)$ prior on p (which is a Uniform), observe data $X \sim$ Binomial(N, p), then our posterior is Beta$(1 + X, 1 + N - X)$.

Example: Bayesian Multi-Armed Bandits *Adapted from an example by Ted Dunning of MapR Technologies*

> Suppose you are faced with N slot machines (colourfully called multi-armed bandits). Each bandit has an unknown probability of distributing a prize (assume for now the prizes are the same for each bandit, only the probabilities differ). Some bandits are very generous, others not so much. Of course, you don't know what these probabilities are. By only choosing one bandit per round, our task is devise a strategy to maximize our winnings.

Of course, if we knew the bandit with the largest probability, then always picking this bandit would yield the maximum winnings. So our task can be phrased as "Find the best bandit, and as quickly as possible".

The task is complicated by the stochastic nature of the bandits. A suboptimal bandit can return many winnings, purely by chance, which would make us believe that it is a very profitable bandit. Similarly, the best bandit can return many duds. Should we keep trying losers then, or give up?

A more troublesome problem is, if we have found a bandit that returns *pretty good* results, do we keep drawing from it to maintain our *pretty good score*, or do we try other bandits in hopes of finding an *even-better* bandit? This is the exploration vs. exploitation dilemma.

1.2.4 Applications

The Multi-Armed Bandit problem at first seems very artificial, something only a mathematician would love, but that is only before we address some applications:

- Internet display advertising: companies have a suite of potential ads they can display to visitors, but the company is not sure which ad strategy to follow to maximize sales. This is similar to A/B testing, but has the added advantage of naturally minimizing strategies that do not work (and generalizes to A/B/C/D... strategies)
- Ecology: animals have a finite amount of energy to expend, and following certain behaviours has uncertain rewards. How does the animal maximize its fitness?
- Finance: which stock option gives the highest return, under time-varying return profiles.
- Clinical trials: a researcher would like to find the best treatment, out of many possible treatment, while minimizing losses.
- Psychology: how does punishment and reward affect our behaviour? How do humans learn?

Many of these questions above are fundamental to the application's field.

It turns out the *optimal solution* is incredibly difficult, and it took decades for an overall solution to develop. There are also many approximately-optimal solutions which are quite good. The one I wish to discuss is one of the few solutions that can scale incredibly well. The solution is known as *Bayesian Bandits*.

1.2.5 A Proposed Solution

Any proposed strategy is called an *online algorithm* (not in the internet sense, but in the continuously-being-updated sense), and more specifically a reinforcement learning algorithm. The algorithm starts in an ignorant state, where it knows nothing, and begins to acquire data by testing the system. As it acquires data and results, it learns what the best and worst behaviours are (in this case, it learns which bandit is the best). With this in mind, perhaps we can add an additional application of the Multi-Armed Bandit problem:

- Psychology: how does punishment and reward affect our behaviour? How do humans learn?

The Bayesian solution begins by assuming priors on the probability of winning for each bandit. In our vignette we assumed complete ignorance of these probabilities. So a very natural prior is the flat prior over 0 to 1. The algorithm proceeds as follows:

For each round:

1. Sample a random variable X_b from the prior of bandit b, for all b.
2. Select the bandit with largest sample, i.e. select $B = \text{argmax} \;\; X_b$.

3. Observe the result of pulling bandit B, and update your prior on bandit B.
4. Return to 1.

That's it. Computationally, the algorithm involves sampling from N distributions. Since the initial priors are Beta$(\alpha = 1, \beta = 1)$ (a uniform distribution), and the observed result X (a win or loss, encoded 1 and 0 respectfully) is Binomial, the posterior is a Beta$(\alpha = 1 + X, \beta = 1 + 1X)$.

To answer our question from before, this algorithm suggests that we should not discard losers, but we should pick them at a decreasing rate as we gather confidence that there exist *better* bandits. This follows because there is always a non-zero chance that a loser will achieve the status of B, but the probability of this event decreases as we play more rounds (see figure below).

Below we implement Bayesian Bandits using two classes, `Bandits` that defines the slot machines, and `BayesianStrategy` which implements the above learning strategy.

```
In [5]: rand = np.random.rand

        class Bandits(object):
            """
            This class represents N bandits machines.

            parameters:
                p_array: a (n,) Numpy array of probabilities >0, <1.

            methods:
                pull( i ): return the results, 0 or 1, of pulling
                           the ith bandit.
            """
            def __init__(self, p_array):
                self.p = p_array
                self.optimal = np.argmax(p_array)

            def pull(self, i):
                #i is which arm to pull
                return np.random.rand() < self.p[i]

            def __len__(self):
                return len(self.p)


        class BayesianStrategy(object):
            """
            Implements a online, learning strategy to solve
            the Multi-Armed Bandit problem.

            parameters:
                bandits: a Bandit class with .pull method

            methods:
                sample_bandits(n): sample and train on n pulls.
```

```
        attributes:
            N: the cumulative number of samples
            choices: the historical choices as a (N,) array
            bb_score: the historical score as a (N,) array
        """

        def __init__(self, bandits):

            self.bandits = bandits
            n_bandits = len(self.bandits)
            self.wins = np.zeros(n_bandits)
            self.trials = np.zeros(n_bandits)
            self.N = 0
            self.choices = []
            self.bb_score = []


        def sample_bandits(self, n=1):

            bb_score = np.zeros(n)
            choices = np.zeros(n)

            for k in range(n):
                #sample from the bandits's priors, and select the largest sample
                choice = np.argmax(np.random.beta(1 + self.wins, 1 + self.trials - self.wi

                #sample the chosen bandit
                result = self.bandits.pull(choice)

                #update priors and score
                self.wins[choice] += result
                self.trials[choice] += 1
                bb_score[k] = result
                self.N += 1
                choices[k] = choice

            self.bb_score = np.r_[self.bb_score, bb_score]
            self.choices = np.r_[self.choices, choices]
            return
```

Below we visualize the learning of the Bayesian Bandit solution.

```
In [6]: figsize(11.0, 10)

        beta = stats.beta
        x = np.linspace(0.001,.999,200)
```

```
def plot_priors(bayesian_strategy, prob, lw = 3, alpha = 0.2, plt_vlines = True):
    ## plotting function
    wins = bayesian_strategy.wins
    trials = bayesian_strategy.trials
    for i in range(prob.shape[0]):
        y = beta(1+wins[i], 1 + trials[i] - wins[i])
        p = plt.plot(x, y.pdf(x), lw = lw)
        c = p[0].get_markeredgecolor()
        plt.fill_between(x,y.pdf(x),0, color = c, alpha = alpha,
                         label="underlying probability: %.2f" % prob[i])
        if plt_vlines:
            plt.vlines(prob[i], 0, y.pdf(prob[i]) ,
                       colors = c, linestyles = "--", lw = 2)
        plt.autoscale(tight = "True")
        plt.title("Posteriors After %d pull" % bayesian_strategy.N +\
                    "s"*(bayesian_strategy.N > 1))
        plt.autoscale(tight=True)
    return
```

```
In [7]: hidden_prob = np.array([0.85, 0.60, 0.75])
        bandits = Bandits(hidden_prob)
        bayesian_strat = BayesianStrategy(bandits)

        draw_samples = [1, 1, 3, 10, 10, 25, 50, 100, 200, 600]

        for j,i in enumerate(draw_samples):
            plt.subplot(5, 2, j+1)
            bayesian_strat.sample_bandits(i)
            plot_priors(bayesian_strat, hidden_prob)
            #plt.legend()
            plt.autoscale(tight = True)
        plt.tight_layout()
```

Note that we don't really care how accurate we become about the inference of the hidden probabilities — for this problem we are more interested in choosing the best bandit (or more accurately, becoming *more confident* in choosing the best bandit). For this reason, the distribution of the red bandit is very wide (representing ignorance about what that hidden probability might be) but we are reasonably confident that it is not the best, so the algorithm chooses to ignore it.

From the above, we can see that after 1000 pulls, the majority of the "blue" function leads the pack, hence we will almost always choose this arm. This is good, as this arm is indeed the best.

Below is a D3 app that demonstrates our algorithm updating/learning three bandits. The first figure are the raw counts of pulls and wins, and the second figure is a dynamically updating plot. I encourage you to try to guess which bandit is optimal, prior to revealing the true probabilities, by selecting the `arm buttons`.

```
In [8]: from IPython.core.display import HTML

        #try executing the below command twice if the first time doesn't work
        HTML(filename = "BanditsD3.html")

Out[8]: <IPython.core.display.HTML object>
```

Deviations of the observed ratio from the highest probability is a measure of performance. For example,in the long run, optimally we can attain the reward/pull ratio of the maximum bandit probability. Long-term realized ratios less than the maximum represent inefficiencies. (Realized ratios larger than the maximum probability is due to randomness, and will eventually fall below).

1.2.6 A Measure of *Good*

We need a metric to calculate how well we are doing. Recall the absolute *best* we can do is to always pick the bandit with the largest probability of winning. Denote this best bandit's probability by w_{opt}. Our score should be relative to how well we would have done had we chosen the best bandit from the beginning. This motivates the *total regret* of a strategy, defined:

$$R_T = \sum_{i=1}^{T} \left(w_{opt} - w_{B(i)} \right) \tag{1}$$

$$\tag{2}$$

$$= Tw^* - \sum_{i=1}^{T} w_{B(i)} \tag{3}$$

where $w_{B(i)}$ is the probability of a prize of the chosen bandit in the i round. A total regret of 0 means the strategy is matching the best possible score. This is likely not possible, as initially our algorithm will often make the wrong choice. Ideally, a strategy's total regret should flatten as it learns the best bandit. (Mathematically, we achieve $w_{B(i)} = w_{opt}$ often)

Below we plot the total regret of this simulation, including the scores of some other strategies:

1. Random: randomly choose a bandit to pull. If you can't beat this, just stop.
2. Largest Bayesian credible bound: pick the bandit with the largest upper bound in its 95% credible region of the underlying probability.
3. Bayes-UCB algorithm: pick the bandit with the largest *score*, where score is a dynamic quantile of the posterior (see [4])
4. Mean of posterior: choose the bandit with the largest posterior mean. This is what a human player (sans computer) would likely do.
5. Largest proportion: pick the bandit with the current largest observed proportion of winning.

The code for these are in the `other_strats.py`, where you can implement your own very easily.

```
In [9]: figsize(12.5, 5)
        from other_strats import *

        #define a harder problem
        hidden_prob = np.array([0.15, 0.2, 0.1, 0.05])
        bandits = Bandits(hidden_prob)

        #define regret
        def regret(probabilities, choices):
            w_opt = probabilities.max()
            return (w_opt - probabilities[choices.astype(int)]).cumsum()
```

```
#create new strategies
strategies= [upper_credible_choice,
            bayesian_bandit_choice,
            ucb_bayes ,
            max_mean,
            random_choice]
algos = []
for strat in strategies:
    algos.append(GeneralBanditStrat(bandits, strat))
```

```
In [10]: #train 10000 times
         for strat in algos:
             strat.sample_bandits(10000)

         #test and plot
         for i,strat in enumerate(algos):
             _regret = regret(hidden_prob, strat.choices)
             plt.plot(_regret, label = strategies[i].__name__, lw = 3)

         plt.title("Total Regret of Bayesian Bandits Strategy vs. Random guessing")
         plt.xlabel("Number of pulls")
         plt.ylabel("Regret after $n$ pulls");
         plt.legend(loc = "upper left");
```

Like we wanted, Bayesian bandits and other strategies have decreasing rates of regret, representing we are achieving optimal choices. To be more scientific so as to remove any possible luck in the above simulation, we should instead look at the *expected total regret*:

$$\bar{R}_T = E[R_T]$$

It can be shown that any *sub-optimal* strategy's expected total regret is bounded below logarithmically. Formally,

$$E[R_T] = \Omega\,(\,\log(T)\,)$$

Thus, any strategy that matches logarithmic-growing regret is said to "solve" the Multi-Armed Bandit problem [3].

Using the Law of Large Numbers, we can approximate Bayesian Bandit's expected total regret by performing the same experiment many times (500 times, to be fair):

```
In [11]: #this can be slow, so I recommend NOT running it.

         trials = 500
         expected_total_regret = np.zeros((10000, 3))

         for i_strat, strat in enumerate(strategies[:-2]):
             for i in range(trials):
                 general_strat = GeneralBanditStrat(bandits, strat)
                 general_strat.sample_bandits(10000)
                 _regret =  regret(hidden_prob, general_strat.choices)
                 expected_total_regret[:,i_strat] += _regret
             plt.plot(expected_total_regret[:,i_strat]/trials, lw =3, label = strat.__name__)

         plt.title("Expected Total Regret of Multi-armed Bandit strategies")
         plt.xlabel("Number of pulls")
         plt.ylabel("Exepected Total Regret \n after $n$ pulls");
         plt.legend(loc = "upper left");
```

```
In [12]: plt.figure()
         [pl1, pl2, pl3] = plt.plot(expected_total_regret[:, [0,1,2]], lw = 3)
         plt.xscale("log")
         plt.legend([pl1, pl2, pl3],
                    ["Upper Credible Bound", "Bayesian Bandit", "UCB-Bayes"],
```

```
                loc="upper left")
plt.ylabel("Exepected Total Regret \n after $\log{n}$ pulls");
plt.title( "log-scale of above" );
plt.ylabel("Exepected Total Regret \n after $\log{n}$ pulls");
```

1.2.7 Extending the algorithm

Because of the Bayesian Bandits algorithm's simplicity, it is easy to extend. Some possibilities:

- If interested in the *minimum* probability (eg: where prizes are a bad thing), simply choose $B = \text{argmin } X_b$ and proceed.
- Adding learning rates: Suppose the underlying environment may change over time. Technically the standard Bayesian Bandit algorithm would self-update itself (awesome) by noting that what it thought was the best is starting to fail more often. We can motivate the algorithm to learn changing environments quicker by simply adding a *rate* term upon updating:

```
self.wins[choice] = rate*self.wins[choice] + result
self.trials[choice] = rate*self.trials[choice] + 1
```

If `rate < 1`, the algorithm will *forget* its previous wins quicker and there will be a downward pressure towards ignorance. Conversely, setting `rate > 1` implies your algorithm will act more risky, and bet on earlier winners more often and be more resistant to changing environments.

- Hierarchical algorithms: We can setup a Bayesian Bandit algorithm on top of smaller bandit algorithms. Suppose we have N Bayesian Bandit models, each varying in some behavior (for example different `rate` parameters, representing varying sensitivity to changing environments). On top of these N models is another Bayesian Bandit learner that will select a sub-Bayesian Bandit. This chosen Bayesian Bandit will then make an internal choice as to which machine to pull. The super-Bayesian Bandit updates itself depending on whether the sub-Bayesian Bandit was correct or not.

- Extending the rewards, denoted y_a for bandit a, to random variables from a distribution $f_{y_a}(y)$ is straightforward. More generally, this problem can be rephrased as "Find the bandit with the largest expected value", as playing the bandit with the largest expected value is optimal. In the case above, f_{y_a} was Bernoulli with probability p_a, hence the expected value for a bandit is equal to p_a, which is why it looks like we are aiming to maximize the probability of winning. If f is not Bernoulli, and it is non-negative, which can be accomplished apriori by shifting the distribution (we assume we know f), then the algorithm behaves as before:

For each round,

1. Sample a random variable X_b from the prior of bandit b, for all b.
2. Select the bandit with largest sample, i.e. select bandit $B = \text{argmax } X_b$.
3. Observe the result,$R \sim f_{y_a}$, of pulling bandit B, and update your prior on bandit B.
4. Return to 1

The issue is in the sampling of X_b drawing phase. With Beta priors and Bernoulli observations, we have a Beta posterior — this is easy to sample from. But now, with arbitrary distributions f, we have a non-trivial posterior. Sampling from these can be difficult.

- There has been some interest in extending the Bayesian Bandit algorithm to commenting systems. Recall in Chapter 4, we developed a ranking algorithm based on the Bayesian lower-bound of the proportion of upvotes to total votes. One problem with this approach is that it will bias the top rankings towards older comments, since older comments naturally have more votes (and hence the lower-bound is tighter to the true proportion). This creates a positive feedback cycle where older comments gain more votes, hence are displayed more often, hence gain more votes, etc. This pushes any new, potentially better comments, towards the bottom. J. Neufeld proposes a system to remedy this that uses a Bayesian Bandit solution.

His proposal is to consider each comment as a Bandit, with the number of pulls equal to the number of votes cast, and number of rewards as the number of upvotes, hence creating a Beta$(1 + U, 1 + D)$ posterior. As visitors visit the page, samples are drawn from each bandit/comment, but instead of displaying the comment with the max sample, the comments are ranked according to the ranking of their respective samples. From J. Neufeld's blog [7]:

> [The] resulting ranking algorithm is quite straightforward, each new time the comments page is loaded, the score for each comment is sampled from a Beta$(1 + U, 1 + D)$, comments are then ranked by this score in descending order... This randomization has a unique benefit in that even untouched comments $(U = 1, D = 0)$ have some chance of being seen even in threads with 5000+ comments (something that is not happening now), but, at the same time, the user is not likely to be inundated with rating these new comments.

Just for fun, though the colors explode, we watch the Bayesian Bandit algorithm learn 15 different options.

```
In [13]: figsize(12.0, 8)
         beta = stats.beta
         hidden_prob = beta.rvs(1,13, size = 35)
```

```
print(hidden_prob)
bandits = Bandits(hidden_prob)
bayesian_strat = BayesianStrategy(bandits)

for j,i in enumerate([100, 200, 500, 1300]):
    plt.subplot(2, 2, j+1)
    bayesian_strat.sample_bandits(i)
    plot_priors(bayesian_strat, hidden_prob, lw = 2, alpha = 0.0, plt_vlines=False)
    #plt.legend()
    plt.xlim(0, 0.5)
```

```
[  6.78509844e-03   9.68721665e-02   3.16101371e-02   8.88059449e-02
   3.32812651e-02   6.59572621e-02   7.91635546e-02   5.97822577e-02
   1.17088549e-01   1.29945195e-05   3.66798062e-02   5.77077187e-02
   4.32774140e-02   6.94914246e-02   1.22741733e-01   5.13528129e-02
   3.29414904e-01   5.13320236e-02   5.35031763e-02   1.57610420e-02
   1.94570205e-02   1.11759388e-01   3.23349076e-02   2.04068995e-02
   1.47822753e-01   8.24022697e-03   3.20395660e-04   4.45643230e-03
   6.42090321e-03   7.29322919e-02   8.18486095e-02   5.05066236e-03
   1.73946201e-01   6.48018322e-02   7.70657954e-03]
```

1.3 Eliciting expert prior

Specifying a subjective prior is how practitioners incorporate domain knowledge about the problem into our mathematical framework. Allowing domain knowledge is useful for many reasons:

- Aids speeds of MCMC convergence. For example, if we know the unknown parameter is strictly positive, then we can restrict our attention there, hence saving time that would otherwise be spent exploring negative values.
- More accurate inference. By weighing prior values near the true unknown value higher, we are narrowing our eventual inference (by making the posterior tighter around the unknown)
- Express our uncertainty better. See the *Price is Right* problem in Chapter 5.

plus many other reasons. Of course, practitioners of Bayesian methods are not experts in every field, so we must turn to domain experts to craft our priors. We must be careful with how we elicit these priors though. Some things to consider:

1. From experience, I would avoid introducing Betas, Gammas, etc. to non-Bayesian practitioners. Furthermore, non-statisticians can get tripped up by how a continuous probability function can have a value exceeding one.

2. Individuals often neglect the rare *tail-events* and put too much weight around the mean of distribution.

3. Related to above is that almost always individuals will under-emphasize the uncertainty in their guesses.

Eliciting priors from non-technical experts is especially difficult. Rather than introduce the notion of probability distributions, priors, etc. that may scare an expert, there is a much simpler solution.

1.3.1 Trial roulette method

The *trial roulette method* [8] focuses on building a prior distribution by placing counters (think casino chips) on what the expert thinks are possible outcomes. The expert is given N counters (say $N = 20$) and is asked to place them on a pre-printed grid, with bins representing intervals. Each column would represent their belief of the probability of getting the corresponding bin result. Each chip would represent an $\frac{1}{N} = 0.05$ increase in the probability of the outcome being in that interval. For example [9]:

> A student is asked to predict the mark in a future exam. The figure below shows a completed grid for the elicitation of a subjective probability distribution. The horizontal axis of the grid shows the possible bins (or mark intervals) that the student was asked to consider. The numbers in top row record the number of chips per bin. The completed grid (using a total of 20 chips) shows that the student believes there is a 30% chance that the mark will be between 60 and 64.9.

From this, we can fit a distribution that captures the expert's choice. Some reasons in favor of using this technique are:

1. Many questions about the shape of the expert's subjective probability distribution can be answered without the need to pose a long series of questions to the expert - the statistician can simply read off density above or below any given point, or that between any two points.

2. During the elicitation process, the experts can move around the chips if unsatisfied with the way they placed them initially - thus they can be sure of the final result to be submitted.

3. It forces the expert to be coherent in the set of probabilities that are provided. If all the chips are used, the probabilities must sum to one.

4. Graphical methods seem to provide more accurate results, especially for participants with modest levels of statistical sophistication.

Example: Stock Returns Take note stock brokers: you're doing it wrong. When choosing which stocks to pick, an analyst will often look at the *daily return* of the stock. Suppose S_t is the price of the stock on day t, then the daily return on day t is :

$$r_t = \frac{S_t - S_{t-1}}{S_{t-1}}$$

The *expected daily return* of a stock is denoted $\mu = E[r_t]$. Obviously, stocks with high expected returns are desirable. Unfortunately, stock returns are so filled with noise that it is very hard to estimate this parameter. Furthermore, the parameter might change over time (consider the rises and falls of AAPL stock), hence it is unwise to use a large historical dataset.

Historically, the expected return has been estimated by using the sample mean. This is a bad idea. As mentioned, the sample mean of a small sized dataset has enormous potential to be very wrong (again, see Chapter 4 for full details). Thus Bayesian inference is the correct procedure here, since we are able to see our uncertainty along with probable values.

For this exercise, we will be examining the daily returns of the AAPL, GOOG, MSFT and AMZN. Before we pull in the data, suppose we ask our a stock fund manager (an expert in finance, but see [10]),

> What do you think the return profile looks like for each of these companies?

Our stock broker, without needing to know the language of Normal distributions, or priors, or variances, etc. creates four distributions using the trial roulette method above. Suppose they look enough like Normals, so we fit Normals to them. They may look like:

```
In [14]: figsize(11., 5)
         colors = ["#348ABD", "#A60628", "#7A68A6", "#467821"]

         normal = stats.norm
         x = np.linspace(-0.15, 0.15, 100)

         expert_prior_params = {"AAPL":(0.05, 0.03),
                                "GOOG":(-0.03, 0.04),
                                "TSLA": (-0.02, 0.01),
                                "AMZN": (0.03, 0.02),
                                }

         for i, (name, params) in enumerate(expert_prior_params.items()):
             plt.subplot(2, 2, i+1)
             y = normal.pdf(x, params[0], scale = params[1])
             #plt.plot( x, y, c = colors[i] )
```

```
        plt.fill_between(x, 0, y, color = colors[i], linewidth=2,
                         edgecolor = colors[i], alpha = 0.6)
        plt.title(name + " prior")
        plt.vlines(0, 0, y.max(), "k","--", linewidth = 0.5)
        plt.xlim(-0.15, 0.15)
    plt.tight_layout()
```

Note that these are subjective priors: the expert has a personal opinion on the stock returns of each of these companies, and is expressing them in a distribution. He's not wishful thinking -- he's introducing domain knowledge.

In order to better model these returns, we should investigate the *covariance matrix* of the returns. For example, it would be unwise to invest in two stocks that are highly correlated, since they are likely to tank together (hence why fund managers suggest a diversification strategy). We will use the *Wishart distribution* for this, introduced earlier.

Let's get some historical data for these stocks. We will use the covariance of the returns as a starting point for our Wishart random variable. This is not empirical bayes (as we will go over later) because we are only deciding the starting point, not influencing the parameters.

```
In [15]: # I wish I could have used Pandas as a prereq for this book, but oh well.
         import datetime
         import collections
         import ystockquote as ysq
         import pandas as pd

         n_observations = 100 # we will truncate the the most recent 100 days.

         stocks = ["AAPL", "GOOG", "TSLA", "AMZN"]

         enddate = "2015-04-27"
         startdate = "2012-09-01"
```

```
CLOSE = 6

stock_closes = pd.DataFrame()

for stock in stocks:
    x = np.array(ysq.get_historical_prices(stock, startdate, enddate))
    stock_series = pd.Series(x[1:,CLOSE].astype(float), name=stock)
    stock_closes[stock] = stock_series

stock_closes = stock_closes[::-1]
stock_returns = stock_closes.pct_change()[1:][-n_observations:]

dates = list(map(lambda x: datetime.datetime.strptime(x, "%Y-%m-%d"), x[1:n_observati
```

And here let's form our basic model:

```
In [16]: import pymc3 as pm
         import theano.tensor as tt
         from theano.tensor.nlinalg import matrix_inverse, diag, matrix_dot

         prior_mu = np.array([x[0] for x in expert_prior_params.values()])
         prior_std = np.array([x[1] for x in expert_prior_params.values()])

         init = stock_returns.cov()

         with pm.Model() as model:
             cov_matrix = pm.WishartBartlett("covariance", np.diag(prior_std**2), 10, testval =

             mu = pm.Normal("returns", mu=prior_mu, sd=1, shape=4)

Applied log-transform to c and added transformed c_log_ to model.
Added new variable c to model diagonal of Wishart.
Added new variable z to model off-diagonals of Wishart.
```

Here are the returns for our chosen stocks:

```
In [17]: figsize(12.5, 4)

         cum_returns = np.cumprod(1 + stock_returns) - 1
         cum_returns.index = dates[::-1]
         cum_returns.plot()

         plt.legend(loc = "upper left")
         plt.title("Return space")
         plt.ylabel("Return of $1 on first date, x100%");
```

```
In [18]: figsize(11., 5 )

         for i, _stock in enumerate(stocks):
             plt.subplot(2,2,i+1)
             plt.hist(stock_returns[_stock], bins=20,
                      normed = True, histtype="stepfilled",
                      color=colors[i], alpha=0.7)
             plt.title(_stock + " returns")
             plt.xlim(-0.15, 0.15)

         plt.tight_layout()
         plt.suptitle("Histogram of daily returns", size =14);
```

Below we perform the inference on the posterior mean return and posterior covariance matrix.

```
In [19]: with model:
             obs = pm.MvNormal("observed returns", mu=mu, cov=cov_matrix, observed=stock_return
```

```
    step = pm.NUTS()
    trace = pm.sample(5000, step=step)
```

```
 [-------100%-------] 5000 of 5000 in 40.4 sec. | SPS: 123.8 | ETA: 0.0
```

```
In [20]: figsize(12.5,4)

         #examine the mean return first.
         mu_samples = trace["returns"]

         for i in range(4):
             plt.hist(mu_samples[:,i], alpha = 0.8 - 0.05*i, bins = 30,
                      histtype="stepfilled", normed=True,
                      label = "%s" % stock_returns.columns[i])

         plt.vlines(mu_samples.mean(axis=0), 0, 500, linestyle="--", linewidth = .5)

         plt.title("Posterior distribution of $\mu$, daily stock returns")
         plt.legend();
```

(Plots like these are what inspired the book's cover.)

What can we say about the results above? Clearly TSLA has been a strong performer, and our analysis suggests that it has an almost 1% daily return! Similarly, most of the distribution of AAPL is negative, suggesting that its *true daily return* is negative.

You may not have immediately noticed, but these variables are a whole order of magnitude *less* than our priors on them. For example, to put these one the same scale as the above prior distributions:

```
In [21]: figsize(11.0,3)
         for i in range(4):
             plt.subplot(2,2,i+1)
             plt.hist(mu_samples[:,i], alpha = 0.8 - 0.05*i, bins = 30,
                      histtype="stepfilled", normed=True, color = colors[i],
                      label = "%s" % stock_returns.columns[i])
```

```
        plt.title("%s" % stock_returns.columns[i])
        plt.xlim(-0.15, 0.15)

    plt.suptitle("Posterior distribution of daily stock returns")
    plt.tight_layout()
```

Why did this occur? Recall how I mentioned that finance has a very very low signal to noise ratio. This implies an environment where inference is much more difficult. One should be careful about over-interpreting these results: notice (in the first figure) that each distribution is positive at 0, implying that the stock may return nothing. Furthermore, the subjective priors influenced the results. From the fund managers point of view, this is good as it reflects his updated beliefs about the stocks, whereas from a neutral viewpoint this can be too subjective of a result.

Below we show the posterior correlation matrix, and posterior standard deviations. An important caveat to know is that the Wishart distribution models the *inverse covariance matrix*, so we must invert it to get the covariance matrix. We also normalize the matrix to acquire the *correlation matrix*. Since we cannot plot hundreds of matrices effectively, we settle by summarizing the posterior distribution of correlation matrices by showing the *mean posterior correlation matrix* (defined on line 2).

```
In [22]: cov_samples = trace["covariance"]
         mean_covariance_matrix = cov_samples.mean(axis=0)

         def cov2corr(A):
             """
             covariance matrix to correlation matrix.
             """
             d = np.sqrt(A.diagonal())
             A = ((A.T/d).T)/d
             #A[ np.diag_indices(A.shape[0]) ] = np.ones( A.shape[0] )
             return A


         plt.subplot(1,2,1)
         plt.imshow(cov2corr(mean_covariance_matrix) , interpolation="none",
                        cmap = "hot")
         plt.xticks(np.arange(4), stock_returns.columns)
         plt.yticks(np.arange(4), stock_returns.columns)
```

```
plt.colorbar(orientation="vertical")
plt.title("(mean posterior) Correlation Matrix")

plt.subplot(1,2,2)
plt.bar(np.arange(4), np.sqrt(np.diag(mean_covariance_matrix)),
        color = "#348ABD", alpha = 0.7)
plt.xticks(np.arange(4) + 0.5, stock_returns.columns);
plt.title("(mean posterior) standard deviations of daily stock returns")

plt.tight_layout();
```

Looking at the above figures, we can say that likely TSLA has an above-average volatility (looking at the return graph this is quite clear). The correlation matrix shows that there are not strong correlations present, but perhaps GOOG and AMZN express a higher correlation (about 0.30).

With this Bayesian analysis of the stock market, we can throw it into a Mean-Variance optimizer (which I cannot stress enough, do not use with frequentist point estimates) and find the minimum. This optimizer balances the tradeoff between a high return and high variance.

$$w_{opt} = \max_{w} \frac{1}{N} \left(\sum_{i=0}^{N} \mu_i^T w - \frac{\lambda}{2} w^T \Sigma_i w \right)$$

where μ_i and Σ_i are the ith posterior estimate of the mean returns and the covariance matrix. This is another example of loss function optimization.

1.3.2 Protips for the Wishart distribution

If you plan to be using the Wishart distribution, read on. Else, feel free to skip this.

In the problem above, the Wishart distribution behaves pretty nicely. Unfortunately, this is rarely the case. The problem is that estimating an NxN covariance matrix involves estimating $\frac{1}{2}N(N-1)$ unknowns. This is a large number even for modest N. Personally, I've tried performing a similar simulation as above with $N = 23$ stocks, and ended up giving considering that I was requesting my MCMC simulation to estimate at least $\frac{1}{2}23 * 22 = 253$ additional unknowns (plus the other interesting unknowns in the problem). This is not easy for MCMC. Essentially, you are asking you MCMC to traverse 250+ dimensional space. And the problem seemed so innocent initially! Below are some tips, in order of supremacy:

1. Use conjugancy if it applies. See section below.

2. Use a good starting value. What might be a good starting value? Why, the data's sample covariance matrix is! Note that this is not empirical Bayes: we are not touching the prior's parameters, we are modifying the starting value of the MCMC. Due to numerical instability, it is best to truncate the floats in the sample covariance matrix down a few degrees of precision (e.g. instability can cause unsymmetrical matrices, which can cause PyMC3 to cry.).

3. Provide as much domain knowledge in the form of priors, if possible. I stress *if possible*. It is likely impossible to have an estimate about each $\frac{1}{2}N(N-1)$ unknown. In this case, see number 4.

4. Use empirical Bayes, i.e. use the sample covariance matrix as the prior's parameter.

5. For problems where N is very large, nothing is going to help. Instead, ask, do I really care about *every* correlation? Probably not. Further ask yourself, do I really really care about correlations? Possibly not. In finance, we can set an informal hierarchy of what we might be interested in the most: first a good estimate of μ, the variances along the diagonal of the covariance matrix are secondly important, and finally the correlations are least important. So, it might be better to ignore the $\frac{1}{2}(N-1)(N-2)$ correlations and instead focus on the more important unknowns.

Another thing to note is that the implementation of the Wishart distribution has changed in from PyMC to PyMC3. Wishart distribution matrices are required to have certain mathematical characteristics that are very restrictive. This makes it so that it is impossible for MCMC methods to propose matrices that will be accepted in our sampling procedure. With our model here we sample the Bartlett decomposition of a Wishart distribution matrix and use that to calculate our samples for the covariance matrix (http://en.wikipedia.org/wiki/Wishart_distribution#Bartlett_decomposition).

1.4 Conjugate Priors

Recall that a Beta prior with Binomial data implies a Beta posterior. Graphically:

$$\underbrace{\text{Beta}}_{\text{prior}} \cdot \overbrace{\text{Binomial}}^{\text{data}} = \overbrace{\text{Beta}}^{\text{posterior}}$$

Notice the Beta on both sides of this equation (no, you cannot cancel them, this is not a *real* equation). This is a really useful property. It allows us to avoid using MCMC, since the posterior is known in closed form. Hence inference and analytics are easy to derive. This shortcut was the heart of the Bayesian Bandit algorithm above. Fortunately, there is an entire family of distributions that have similar behaviour.

Suppose X comes from, or is believed to come from, a well-known distribution, call it f_α, where α are possibly unknown parameters of f. f could be a Normal distribution, or Binomial distribution, etc. For particular distributions f_α, there may exist a prior distribution p_β, such that:

$$\overbrace{p_\beta}^{\text{prior}} \cdot \overbrace{f_\alpha(X)}^{\text{data}} = \overbrace{p_{\beta'}}^{\text{posterior}}$$

where β' is a different set of parameters *but p is the same distribution as the prior*. A prior p that satisfies this relationship is called a *conjugate prior*. As I mentioned, they are useful computationally, as we can avoided approximate inference using MCMC and go directly to the posterior. This sounds great, right?

Unfortunately, not quite. There are a few issues with conjugate priors.

1. The conjugate prior is not objective. Hence only useful when a subjective prior is required. It is not guaranteed that the conjugate prior can accommodate the practitioner's subjective opinion.

2. There typically exist conjugate priors for simple, one dimensional problems. For larger problems, involving more complicated structures, hope is lost to find a conjugate prior. For smaller models, Wikipedia has a nice table of conjugate priors.

Really, conjugate priors are only useful for their mathematical convenience: it is simple to go from prior to posterior. I personally see conjugate priors as only a neat mathematical trick, and offer little insight into the problem at hand.

1.5 Jefferys Priors

Earlier, we talked about objective priors rarely being *objective*. Partly what we mean by this is that we want a prior that doesn't bias our posterior estimates. The flat prior seems like a reasonable choice as it assigns equal probability to all values.

But the flat prior is not transformation invariant. What does this mean? Suppose we have a random variable $\mathbf{X}$ from Bernoulli(θ). We define the prior on $p(\theta) = 1$.

```
In [23]: figsize(12.5, 5)

         x = np.linspace(0.000 ,1, 150)
         y = np.linspace(1.0, 1.0, 150)
         lines = plt.plot(x, y, color="#A60628", lw = 3)
         plt.fill_between(x, 0, y, alpha = 0.2, color = lines[0].get_color())
         plt.autoscale(tight=True)
         plt.ylim(0, 2);
```

Now, let's transform θ with the function $\psi = log\frac{\theta}{1-\theta}$. This is just a function to stretch θ across the real line. Now how likely are different values of ψ under our transformation.

```
In [24]: figsize(12.5, 5)

         psi = np.linspace(-10 ,10, 150)
         y = np.exp(psi) / (1 + np.exp(psi))**2
         lines = plt.plot(psi, y, color="#A60628", lw = 3)
         plt.fill_between(psi, 0, y, alpha = 0.2, color = lines[0].get_color())
         plt.autoscale(tight=True)
         plt.ylim(0, 1);
```

Oh no! Our function is no longer flat. It turns out flat priors do carry information in them after all. The point of Jeffreys Priors is to create priors that don't accidentally become informative when you transform the variables you placed them originally on.

Jeffreys Priors are defined as:

$$p_J(\theta) \propto \mathbf{I}(\theta)^{\frac{1}{2}}$$

$$\mathbf{I}(\theta) = -\mathbb{E}\left[\frac{d^2 \log p(X|\theta)}{d\theta^2}\right]$$

I being the *Fisher information*

1.6 Effect of the prior as N increases

In the first chapter, I proposed that as the amount of our observations or data increases, the influence of the prior decreases. This is intuitive. After all, our prior is based on previous information, and eventually enough new information will shadow our previous information's value. The smothering of the prior by enough data is also helpful: if our prior is significantly wrong, then the self-correcting nature of the data will present to us a *less wrong*, and eventually *correct*, posterior.

We can see this mathematically. First, recall Bayes Theorem from Chapter 1 that relates the prior to the posterior. The following is a sample from What is the relationship between sample size and the influence of prior on posterior?[1] on CrossValidated.

> The posterior distribution for a parameter θ, given a data set $\mathbf{X}$ can be written as
>
> $$p(\theta|\mathbf{X}) \propto \underbrace{p(\mathbf{X}|\theta)}_{likelihood} \cdot \overbrace{p(\theta)}^{prior}$$
>
> or, as is more commonly displayed on the log scale,
>
> $$\log(p(\theta|\mathbf{X})) = c + L(\theta;\mathbf{X}) + \log(p(\theta))$$
>
> The log-likelihood, $L(\theta;\mathbf{X}) = \log\left(p(\mathbf{X}|\theta)\right)$, **scales with the sample size**, since it is a function of the data, while the prior density does not. Therefore, as the sample size increases, the absolute value of $L(\theta;\mathbf{X})$ is getting larger while $\log(p(\theta))$ stays fixed (for a fixed value of θ), thus the sum $L(\theta;\mathbf{X}) + \log(p(\theta))$ becomes more heavily influenced by $L(\theta;\mathbf{X})$ as the sample size increases.

There is an interesting consequence not immediately apparent. As the sample size increases, the chosen prior has less influence. Hence inference converges regardless of chosen prior, so long as the areas of non-zero probabilities are the same.

Below we visualize this. We examine the convergence of two posteriors of a Binomial's parameter θ, one with a flat prior and the other with a biased prior towards 0. As the sample size increases, the posteriors, and hence the inference, converge.

```
In [25]: figsize(12.5, 15)

         p = 0.6
         beta1_params = np.array([1.,1.])
         beta2_params = np.array([2,10])
         beta = stats.beta

         x = np.linspace(0.00, 1, 125)
         data = stats.bernoulli.rvs(p, size=500)

         plt.figure()
         for i,N in enumerate([0,4,8, 32,64, 128, 500]):
             s = data[:N].sum()
             plt.subplot(8,1,i+1)
             params1 = beta1_params + np.array([s, N-s])
             params2 = beta2_params + np.array([s, N-s])
             y1,y2 = beta.pdf(x, *params1), beta.pdf( x, *params2)
             plt.plot(x,y1, label = r"flat prior", lw =3)
             plt.plot(x, y2, label = "biased prior", lw= 3)
             plt.fill_between(x, 0, y1, color ="#348ABD", alpha = 0.15)
```

```
plt.fill_between(x, 0, y2, color ="#A60628", alpha = 0.15)
plt.legend(title = "N=%d" % N)
plt.vlines(p, 0.0, 7.5, linestyles = "--", linewidth=1)
#plt.ylim( 0, 10)#
```

Keep in mind, not all posteriors will "forget" the prior this quickly. This example was just to show that *eventually* the prior is forgotten. The "forgetfulness" of the prior as we become awash in more and more data is the reason why Bayesian and Frequentist inference eventually converge as well.

1.6.1 Bayesian perspective of Penalized Linear Regressions

There is a very interesting relationship between a penalized least-squares regression and Bayesian priors. A penalized linear regression is a optimization problem of the form:

$$\text{argmin}_\beta \ \ (Y - X\beta)^T(Y - X\beta) + f(\beta)$$

for some function f (typically a norm like $||\cdot||_p^p$).

We will first describe the probabilistic interpretation of least-squares linear regression. Denote our response variable Y, and features are contained in the data matrix X. The standard linear model is:

$$Y = X\beta + \epsilon \tag{4}$$

where $\epsilon \sim \text{Normal}(\mathbf{0}, \sigma\mathbf{I})$. Simply, the observed Y is a linear function of X (with coefficients β) plus some noise term. Our unknown to be determined is β. We use the following property of Normal random variables:

$$\mu' + \text{Normal}(\mu, \sigma) \sim \text{Normal}(\mu' + \mu, \sigma)$$

to rewrite the above linear model as:

$$Y = X\beta + \text{Normal}(\mathbf{0}, \sigma\mathbf{I}) \tag{5}$$

$$\tag{6}$$

$$Y = \text{Normal}(X\beta, \sigma\mathbf{I}) \tag{7}$$

$$\tag{8}$$

$$\tag{9}$$

In probabilistic notation, denote $f_Y(y \mid \beta)$ the probability distribution of Y, and recalling the density function for a Normal random variable (see here):

$$f_Y(Y \mid \beta, X) = L(\beta \mid X, Y) = \frac{1}{\sqrt{2\pi\sigma}} \exp\left(\frac{1}{2\sigma^2}(Y - X\beta)^T(Y - X\beta)\right)$$

This is the likelihood function for β. Taking the log:

$$\ell(\beta) = K - c(Y - X\beta)^T(Y - X\beta)$$

where K and $c > 0$ are constants. Maximum likelihood techniques wish to maximize this for β,

$$\hat{\beta} = \text{argmax}_\beta \ \ -(Y - X\beta)^T(Y - X\beta)$$

Equivalently we can *minimize the negative* of the above:

$$\hat{\beta} = \text{argmin}_\beta \ \ (Y - X\beta)^T(Y - X\beta)$$

This is the familiar least-squares linear regression equation. Therefore we showed that the solution to a linear least-squares is the same as the maximum likelihood assuming Normal noise. Next we extend this to show how we can arrive at penalized linear regression by a suitable choice of prior on β.

Penalized least-squares In the above, once we have the likelihood, we can include a prior distribution on β to derive to the equation for the posterior distribution:

$$P(\beta|Y, X) = L(\beta|\ X, Y)p(\beta)$$

where $p(\beta)$ is a prior on the elements of β. What are some interesting priors?

1. If we include *no explicit* prior term, we are actually including an uninformative prior, $P(\beta) \propto 1$, think of it as uniform over all numbers.

2. If we have reason to believe the elements of β are not too large, we can suppose that *a priori*:

$$\beta \sim \text{Normal}(\mathbf{0}, \lambda \mathbf{I})$$

The resulting posterior density function for β is *proportional to*:

$$\exp\left(\frac{1}{2\sigma^2}(Y - X\beta)^T(Y - X\beta)\right)\exp\left(\frac{1}{2\lambda^2}\beta^T\beta\right)$$

and taking the log of this, and combining and redefining constants, we arrive at:

$$\ell(\beta) \propto K - (Y - X\beta)^T(Y - X\beta) - \alpha\beta^T\beta$$

we arrive at the function we wish to maximize (recall the point that maximizes the posterior distribution is the MAP, or *maximum a posterior*):

$$\hat{\beta} = \text{argmax}_\beta \quad -(Y - X\beta)^T(Y - X\beta) - \alpha\ \beta^T\beta$$

Equivalently, we can minimize the negative of the above, and rewriting $\beta^T\beta = ||\beta||_2^2$:

$$\hat{\beta} = \text{argmin}_\beta \ \ (Y - X\beta)^T(Y - X\beta) + \alpha\ ||\beta||_2^2$$

This above term is exactly Ridge Regression. Thus we can see that ridge regression corresponds to the MAP of a linear model with Normal errors and a Normal prior on β.

3. Similarly, if we assume a *Laplace* prior on β, ie.

$$f_\beta(\beta) \propto \exp\left(-\lambda||\beta||_1\right)$$

and following the same steps as above, we recover:

$$\hat{\beta} = \text{argmin}_\beta \ \ (Y - X\beta)^T(Y - X\beta) + \alpha\ ||\beta||_1$$

which is LASSO regression. Some important notes about this equivalence. The sparsity that is a result of using a LASSO regularization is not a result of the prior assigning high probability to sparsity. Quite the opposite actually. It is the combination of the $||\cdot||_1$ function and using the MAP that creates sparsity on β: purely a geometric argument. The prior does contribute to an overall shrinking of the coefficients towards 0 though. An interesting discussion of this can be found in [2].

For an example of Bayesian linear regression, see Chapter 4's example on financial losses.

References

1. Macro, . "What is the relationship between sample size and the influence of prior on posterior?." 13 Jun 2013. StackOverflow, Online Posting to Cross-Validated. Web. 25 Apr. 2013.

2. Starck, J.-L., , et al. "Sparsity and the Bayesian Perspective." Astronomy & Astrophysics. (2013): n. page. Print.

3. Kuleshov, Volodymyr, and Doina Precup. "Algorithms for the multi-armed bandit problem." Journal of Machine Learning Research. (2000): 1-49. Print.

4. Gelman, Andrew. "Prior distributions for variance parameters in hierarchical models." Bayesian Analysis. 1.3 (2006): 515-533. Print.

5. Gelman, Andrew, and Cosma R. Shalizi. "Philosophy and the practice of Bayesian statistics." British Journal of Mathematical and Statistical Psychology. (2012): n. page. Web. 17 Apr. 2013.

6. http://jmlr.csail.mit.edu/proceedings/papers/v22/kaufmann12/kaufmann12.pdf

7. James, Neufeld. "Reddit's "best" comment scoring algorithm as a multi-armed bandit task." Simple ML Hacks. Blogger, 09 Apr 2013. Web. 25 Apr. 2013.

8. Oakley, J. E., Daneshkhah, A. and O'Hagan, A. Nonparametric elicitation using the roulette method. Submitted to Bayesian Analysis.

9. "Eliciting priors from experts." 19 Jul 2010. StackOverflow, Online Posting to Cross-Validated. Web. 1 May. 2013. http://stats.stackexchange.com/questions/1/eliciting-priors-from-experts.

10. Taleb, Nassim Nicholas (2007), The Black Swan: The Impact of the Highly Improbable, Random House, ISBN 978-1400063512


```
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 0x5175fd0>
```

DontOverfit

January 4, 2018

0.1 Implementation of Salisman's Don't Overfit submission

From Kaggle >In order to achieve this we have created a simulated data set with 200 variables and 20,000 cases. An 'equation' based on this data was created in order to generate a Target to be predicted. Given the all 20,000 cases, the problem is very easy to solve – but you only get given the Target value of 250 cases – the task is to build a model that gives the best predictions on the remaining 19,750 cases.

```
In [30]: import gzip
         import requests
         import zipfile

         url = "https://dl.dropbox.com/s/lnly9gw8pb1xhir/overfitting.zip"


         results = requests.get(url)
```

```
In [31]: import StringIO
         z = zipfile.ZipFile(StringIO.StringIO(results.content))
         # z.extractall()
```

```
In [32]: z.extractall()
```

```
In [38]: z.namelist()
```

```
Out[38]: ['overfitting.csv']
```

```
In [69]: d = z.open('overfitting.csv')
         d.readline()
```

```
Out[69]: 'case_id,train,Target_Practice,Target_Leaderboard,Target_Evaluate,var_1,var_2,var_3,va
```

```
In [70]: import numpy as np
```

```
In [63]: M = np.fromstring(d.read(), sep=",")
```

```
In [71]: len(d.read())
```

```
Out[71]: 23919756
```

```
In [75]: np.fromstring?

In [ ]:

In [4]: data = np.loadtxt("overfitting.csv", delimiter=",", skiprows=1)

In [5]: print """
        There are also 5 other fields,

        case_id - 1 to 20,000, a unique identifier for each row

        train - 1/0, this is a flag for the first 250 rows which are the training dataset

        Target_Practice - we have provided all 20,000 Targets for this model, so you can devel

        Target_Leaderboard - only 250 Targets are provided. You submit your predictions for th

        Target_Evaluate - again only 250 Targets are provided. Those competitors who beat the

        """

        data.shape
```

```
There are also 5 other fields,

case_id - 1 to 20,000, a unique identifier for each row

train - 1/0, this is a flag for the first 250 rows which are the training dataset

Target_Practice - we have provided all 20,000 Targets for this model, so you can develop your m

Target_Leaderboard - only 250 Targets are provided. You submit your predictions for the remaini

Target_Evaluate - again only 250 Targets are provided. Those competitors who beat the 'benchma
```

```
Out[5]: (20000L, 205L)

In [6]: ix_training = data[:, 1] == 1
        ix_testing = data[:, 1] == 0

        training_data = data[ix_training, 5:]
        testing_data = data[ix_testing, 5:]

        training_labels = data[ix_training, 2]
        testing_labels = data[ix_testing, 2]
```

```
print "training:", training_data.shape, training_labels.shape
print "testing: ", testing_data.shape, testing_labels.shape
```

```
training: (250L, 200L) (250L,)
testing:  (19750L, 200L) (19750L,)
```

0.2 Develop Tim's model

He mentions that the X variables are from a Uniform distribution. Let's investigate this:

```
In [7]: figsize(12, 4)
```

```
In [8]: hist(training_data.flatten())
        print training_data.shape[0] * training_data.shape[1]
```

```
50000
```

looks pretty right

```
In [127]: import pymc as pm

          to_include = pm.Bernoulli("to_include", 0.5, size=200)
```

```
In [128]: coef = pm.Uniform("coefs", 0, 1, size=200)
```

```
In [129]: @pm.deterministic
          def Z(coef=coef, to_include=to_include, data=training_data):
              ym = np.dot(to_include * training_data, coef)
              return ym - ym.mean()
```

```
In [130]: @pm.deterministic
          def T(z=Z):
              return 0.45 * (np.sign(z) + 1.1)
```

```
In [132]: obs = pm.Bernoulli("obs", T, value=training_labels, observed=True)

          model = pm.Model([to_include, coef, Z, T, obs])
          map_ = pm.MAP(model)
          map_.fit()
```

```
Warning: Stochastic to_include's value is neither numerical nor array with floating-point dtype
```

```
In [133]: mcmc = pm.MCMC(model)
```

```
In [176]: mcmc.sample(100000, 90000, 1)
```

```
[****************100%******************]  100000 of 100000 complete
```

```
In [177]: (np.round(T.value) == training_labels).mean()
```

```
Out[177]: 0.73999999999999999
```

```
In [178]: t_trace = mcmc.trace("T")[:]
          (np.round(t_trace[-500:-400, :]).mean(axis=0) == training_labels).mean()
```

```
Out[178]: 0.72399999999999998
```

```
In [179]: t_mean = np.round(t_trace).mean(axis=1)
```

```
In [180]: imshow(t_trace[-10000:, :], aspect="auto")
          colorbar()
```

```
Out[180]: <matplotlib.colorbar.Colorbar instance at 0x0000000013270208>
```

```
In [181]: figsize(23, 8)
          coef_trace = mcmc.trace("coefs")[:]
          imshow(coef_trace[-10000:, :], aspect="auto", cmap=pyplot.cm.RdBu, interpolation="no
```

Out[181]: <matplotlib.image.AxesImage at 0x19ce2780>

In [183]:
```
include_trace = mcmc.trace("to_include")[:]
```

In [184]:
```
figsize(23, 8)
imshow(include_trace[-10000:, :], aspect="auto", interpolation="none")
```

Out[184]: <matplotlib.image.AxesImage at 0x18d8ef60>

In [ ]:

MachineLearning

January 4, 2018

List of topics to cover:

- Bayesian solution to overfitting
- Salisman's solution to the Don't Overfit
- Predictive distributions; "how do I evaluate testing data?"
- model fitting, BIC + visualization tools
- Gaussian Processes

Would be nice/cool to cover:

- classification models (using the books text)
- Bayesian networks?

In [6]:

In [6]:

In [6]:

In [6]:

In [10]:
```
from IPython.core.display import HTML


def css_styling():
    styles = open("../styles/custom.css", "r").read()
    return HTML(styles)
css_styling()
```

Out[10]:
```
<IPython.core.display.HTML at 0x5bf0518>
```

In [ ]: