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.

A little more on PyMC

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.

```python
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 variables children and parent variables using the `children` and `parents` attributes attached to variables.

```python
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`:
set([<pymc.distributions.Poisson 'data_generator' at 0x10c4be4d0>])
```

```
        Parents of `data_generator`:
        {'mu': <pymc.distributions.Exponential 'poisson_param' at 0x10c040dd0>}

        Children of `data_generator`:
        set([<pymc.PyMCObjects.Deterministic '(data_generator_add_1)' at 0x10c4bee50>])
```

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

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:

```
print "parameter.value =", parameter.value
print "data_generator.value =", data_generator.value
print "data_plus_one.value =", data_plus_one.value
```

```
parameter.value = 1.25374385648
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 (http://pymc-devs.github.com/pymc/distributions.html) 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 $size$ 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 β_i, $i = 1, \ldots, N$ you wish to model. Instead of creating arbitrary names and variables for each one, like:

```python
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:

```python
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.

```python
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
print

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 = 0.108
lambda_2.value = 0.500
tau.value = 5.000

After calling random() on the variables...
lambda_1.value = 2.366
lambda_2.value = 0.992
tau.value = 10.000
```

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.

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:

```python
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.
This is not completely true: elementary operations, like addition, exponentials etc. implicitly create
deterministic variables. For example, the following returns a deterministic variable:

```
type(lambda_1 + lambda_2)
```

```
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:

```
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 lambda1
    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.
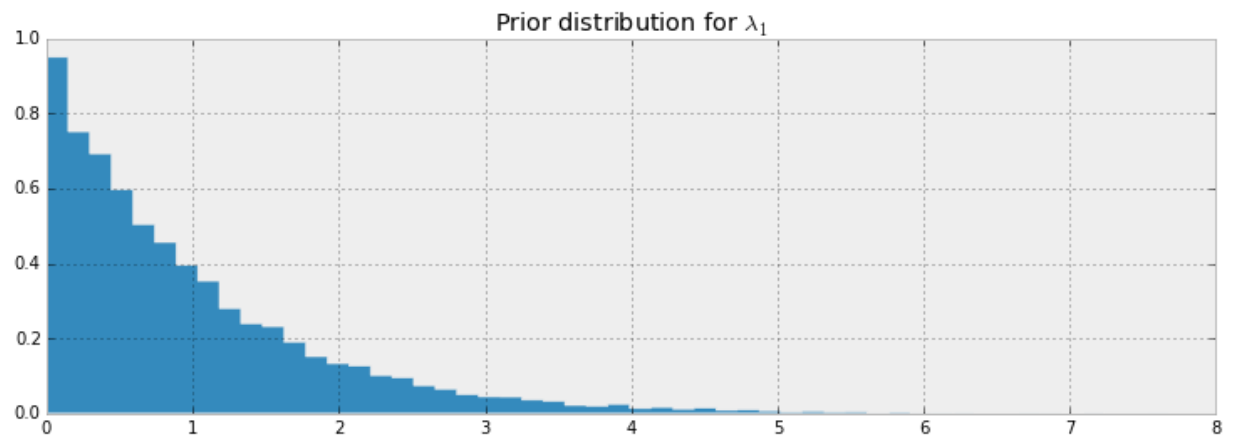
Including observations in the Model

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?"

```
%pylab inline --no-import-all
figsize(12.5, 4)

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

```
Populating the interactive namespace from numpy and matplotlib
```


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:

```
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.

```
# 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]
```

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 ;)

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

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 latter 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 (http://daft-pgm.org/))
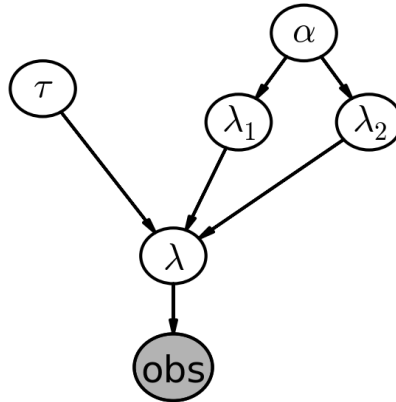
PyMC, and other probabilistic programming languages, have been designed to tell these data-generation *stories*. More generally, B. Cronin writes [5]:

> 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.

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 $\mathrm{DiscreteUniform}(0, 80)$:

```
tau = pm.rdiscrete_uniform(0, 80)
print tau
```
```
67
```

2. Draw λ_1 and λ_2 from an $\mathrm{Exp}(\alpha)$ distribution:

```
alpha = 1./20.
lambda_1, lambda_2 = pm.rexponential(alpha, 2)
print lambda_1, lambda_2
```
```
25.3697150118 21.1892828642
```

3. For days before $\tau$, represent the user's received SMS count by sampling from $\mathrm{Poi}(\lambda_1)$, and sample from $\mathrm{Poi}(\lambda_2)$ for days after $\tau$. For example:

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

4. Plot the artificial dataset:

```
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();
```
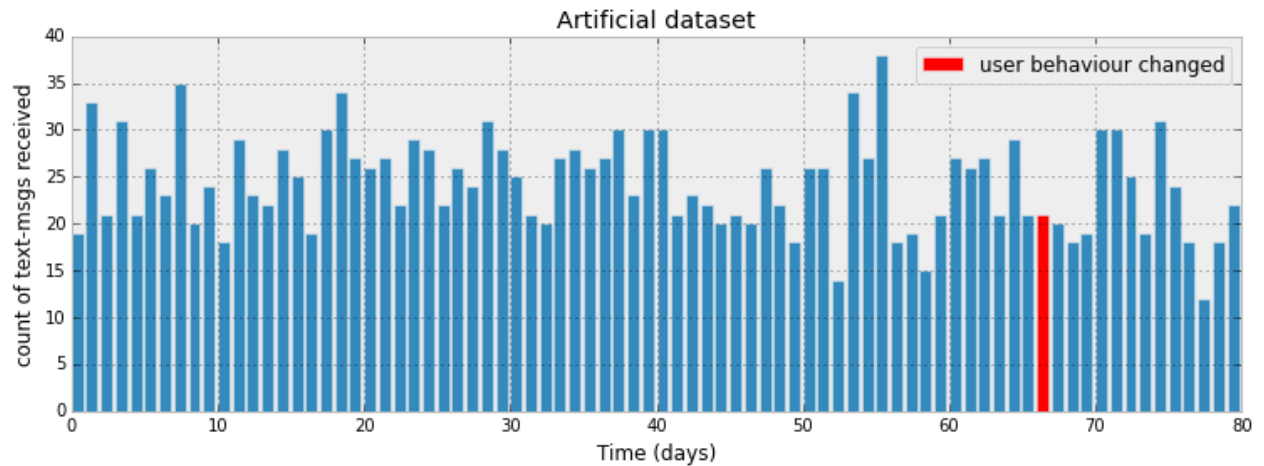

It is okay that our fictional dataset does not look like our observed dataset: the probability is incredibly small it indeed would. PyMC's engine is designed to find good parameters, λ_i, τ, that maximize this probability.

The ability to generate artificial 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:

```
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.title("More example of artificial datasets")
for i in range(4):
    plt.subplot(4, 1, i)
    plot_artificial_sms_dataset()
```
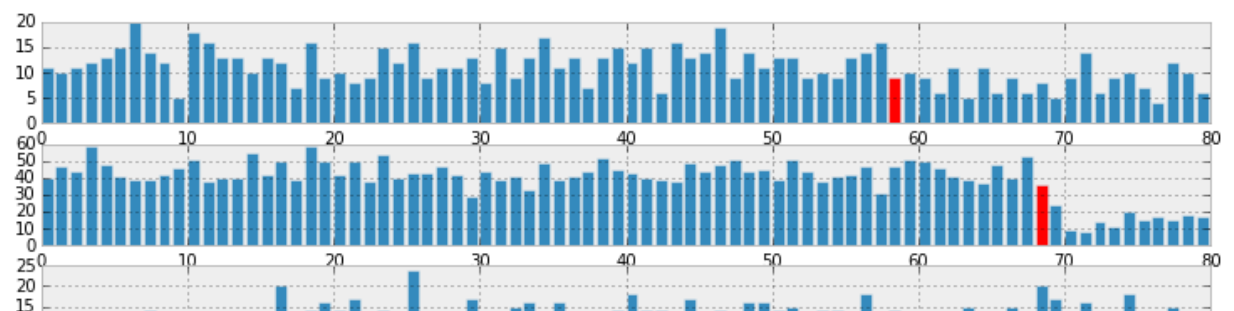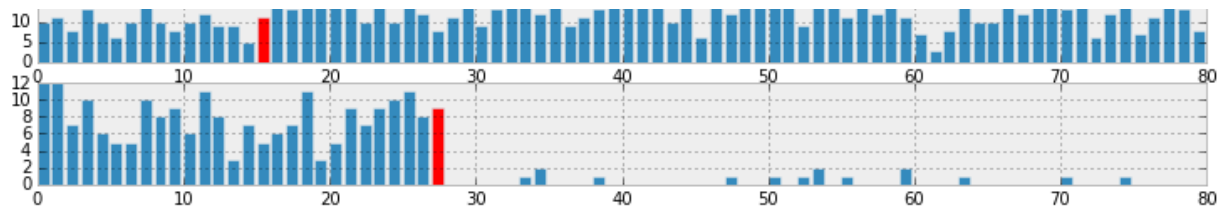
Later we will see how we use this to make predictions and test the appropriateness of our models.

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.

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 distrbutions 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]:

```python
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.

```python
#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]
58
```

The observed frequency is:

```python
# 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? 0.0387
Does this equal the true frequency? False
```

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

```python
#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
```

We plot the posterior distribution of the unknown p_A below:

```
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()
```

```
<matplotlib.legend.Legend at 0x10c403f10>
```


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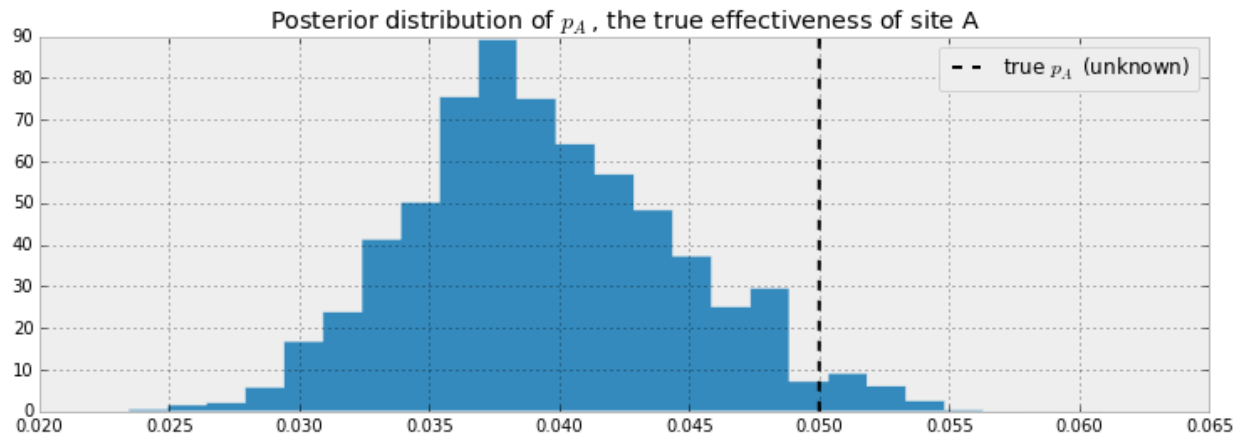
Our posterior distribution puts most weight near the true value of p_A, but also some weights in the tails. This is a measure of how uncertain we should be, given our observations. Try changing the number of observations, N, and observe how the posterior distribution changes.

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 PyMC'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)

```
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 1 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 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0] ...
```

```python
print observations_A.mean()
print observations_B.mean()
```

```
0.0593333333333
0.0533333333333
```

```python
# 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
```

Below we plot the posterior distributions for the three unknowns:

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

```python
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")
```

<matplotlib.legend.Legend at 0x10cb19f50>

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:

```
# 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: 0.307
Probability site A is BETTER than site B: 0.693
```

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.

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.

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 $0 \le X \le N$). 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.

```
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):
```
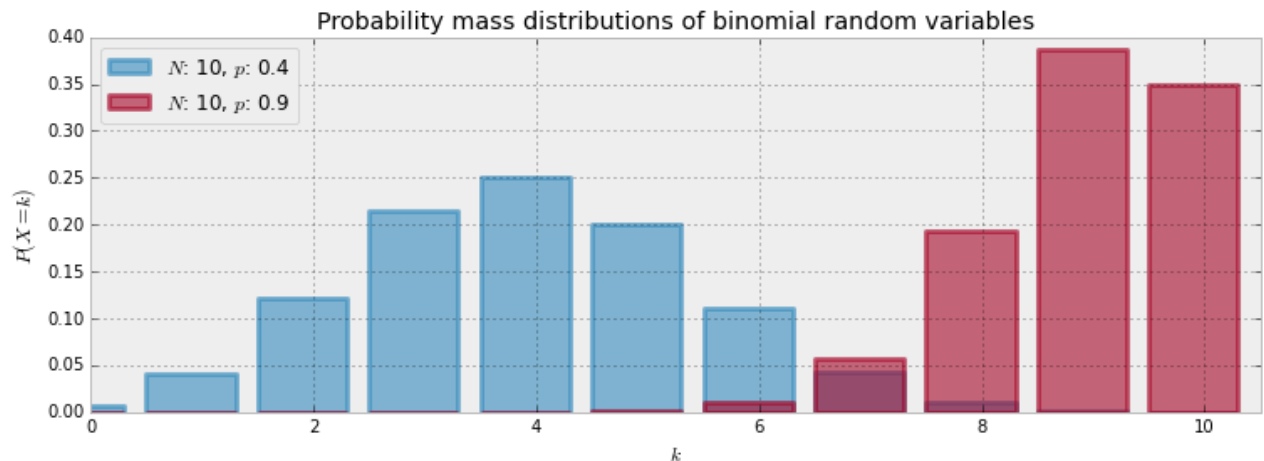
```
        N, p = parameters[i]
        _x = np.arange(N + 1)
        plt.bar(_x - 0.5, binomial.pmf(_x, N, p), color=colors[i],
                edgecolor=colors[i],
                alpha=0.6,
                label="$N$: %d, $p$: %.1f" % (N, p),
                linewidth=3)

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")
```

<matplotlib.text.Text at 0x10cde3f50>

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

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.

```python
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.

```python
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*.

```python
first_coin_flips = pm.Bernoulli("first_flips", 0.5, size=N)
print first_coin_flips.value
```

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

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

```python
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:

```python
@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.

```
observed_proportion.value
```

```
0.54000000000000004
```

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`:

```
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.

```
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
```

```
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()
```
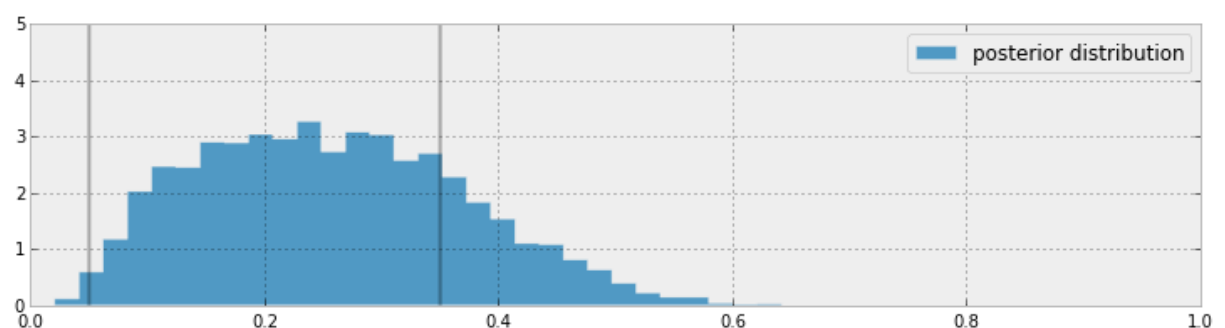
```
<matplotlib.legend.Legend at 0x10d10fd50>
```

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.

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:

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:

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

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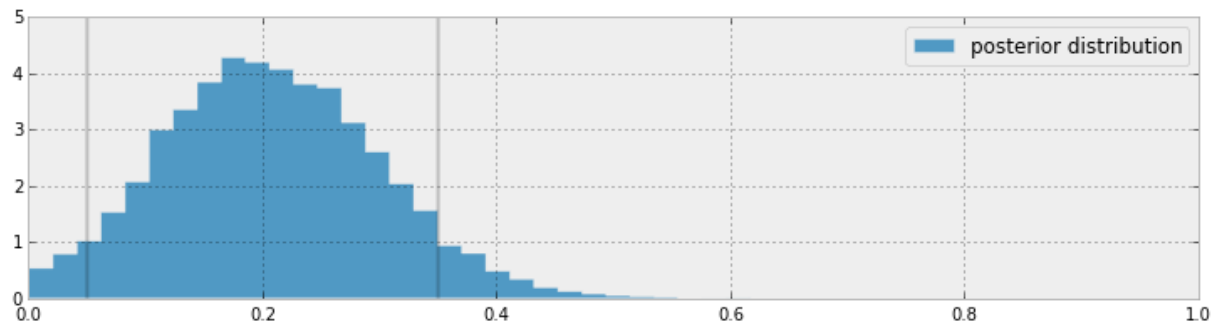
```
model = pm.Model([yes_responses, p_skewed, p])

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

```
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()
```

<matplotlib.legend.Legend at 0x10d8b9f10>

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:

```
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]):

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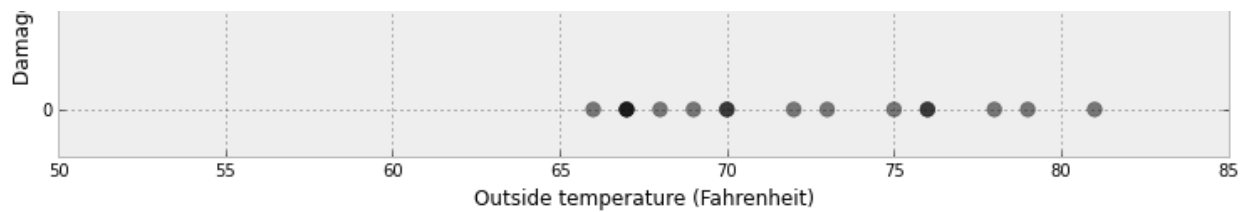
```
Temp (F), O-Ring failure?
[[ 66.    0.]
 [ 70.    1.]
 [ 69.    0.]
 [ 68.    0.]
 [ 67.    0.]
 [ 72.    0.]
 [ 73.    0.]
 [ 70.    0.]
 [ 57.    1.]
 [ 63.    1.]
 [ 70.    1.]
 [ 78.    0.]
 [ 67.    0.]
 [ 53.    1.]
 [ 67.    0.]
 [ 75.    0.]
 [ 70.    0.]
 [ 81.    0.]
 [ 76.    0.]
 [ 79.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

```
<matplotlib.text.Text at 0x10d8f14d0>
```


Defects of the Space Shuttle O-Rings vs temperature
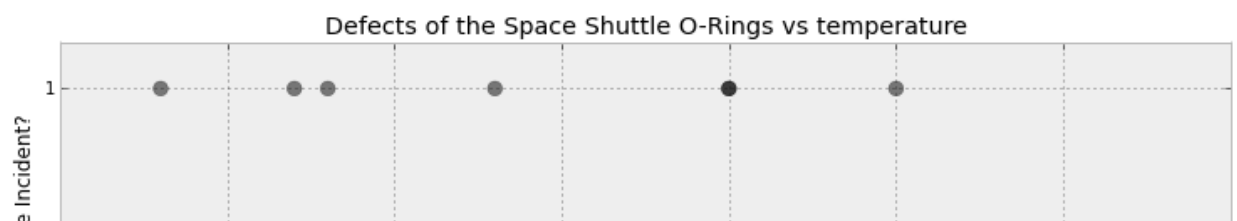
It looks clear that *the probability* of damage incidents occurring increases as the outside temperature decreases. We are interested in modeling the probability here because it does not look like there is a strict cutoff point between temperature and a damage incident occurring. The best we can do is 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 $\beta = 1, 3, -5$.

```
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()
```

```
<matplotlib.legend.Legend at 0x10d9eb050>
```
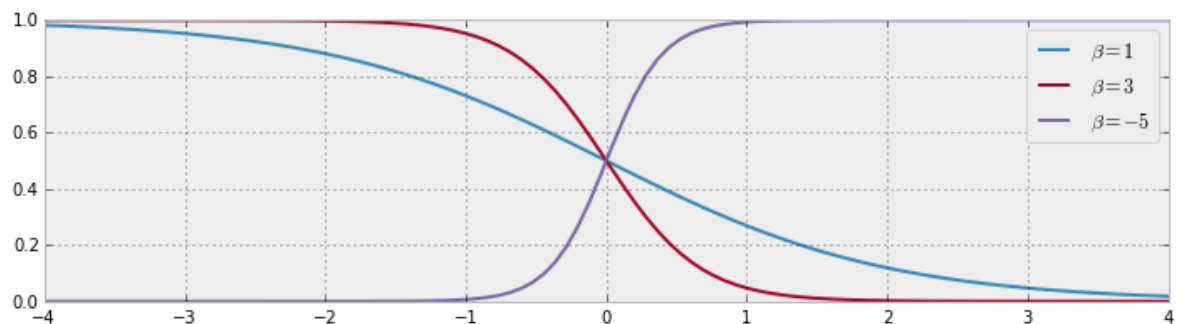

But something is missing. In the plot of the logistic function, the probability changes only near zero, but in our data above the probability changes around 65 to 70. We need to add a *bias* term to our logistic function:

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

Some plots are below, with differing α.

```
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")
```
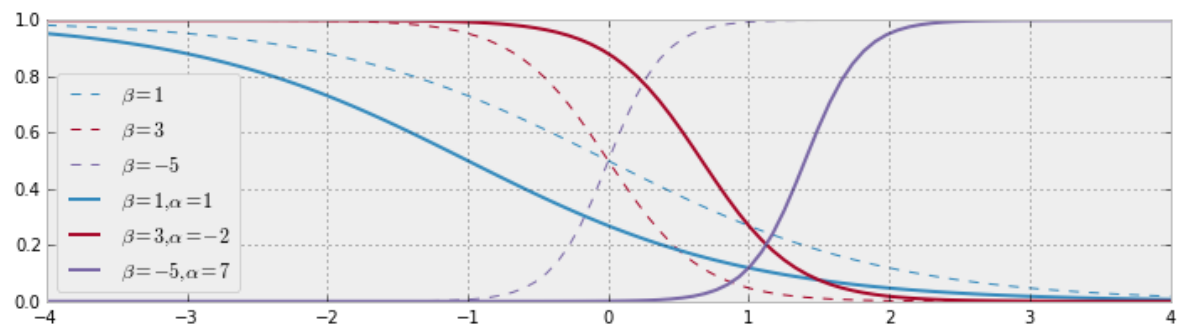
<matplotlib.legend.Legend at 0x10dae4290>

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.

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.

```
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")
```

<matplotlib.text.Text at 0x10db92550>
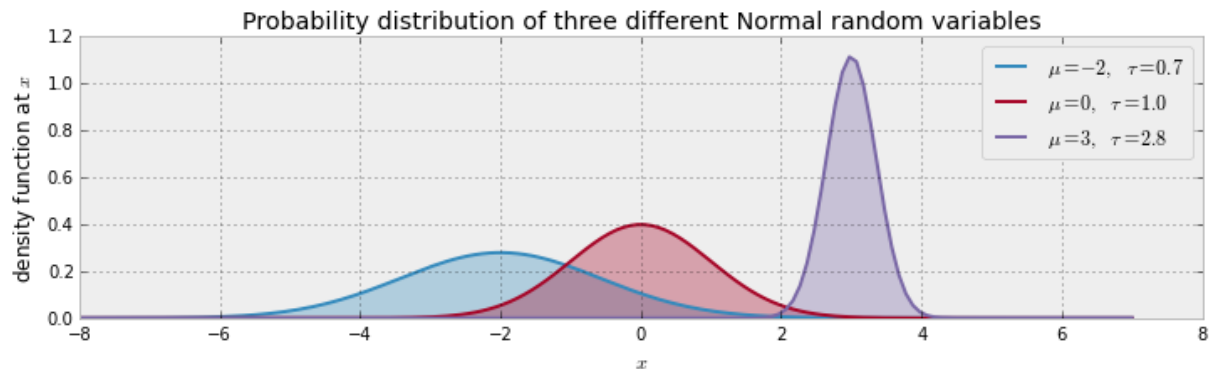
A Normal random variable can be take on any real number, but the variable is very likely to be relatively close to μ. In fact, the expected value of a Normal is equal to its μ parameter:

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

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:

```
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 Ber(p), is a random variable that takes value 1 with probability p, and 0 else. Thus, our model can look like:

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

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.

```
p.value
```

```
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])
```

```python
# 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
```

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:

```python
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()
```
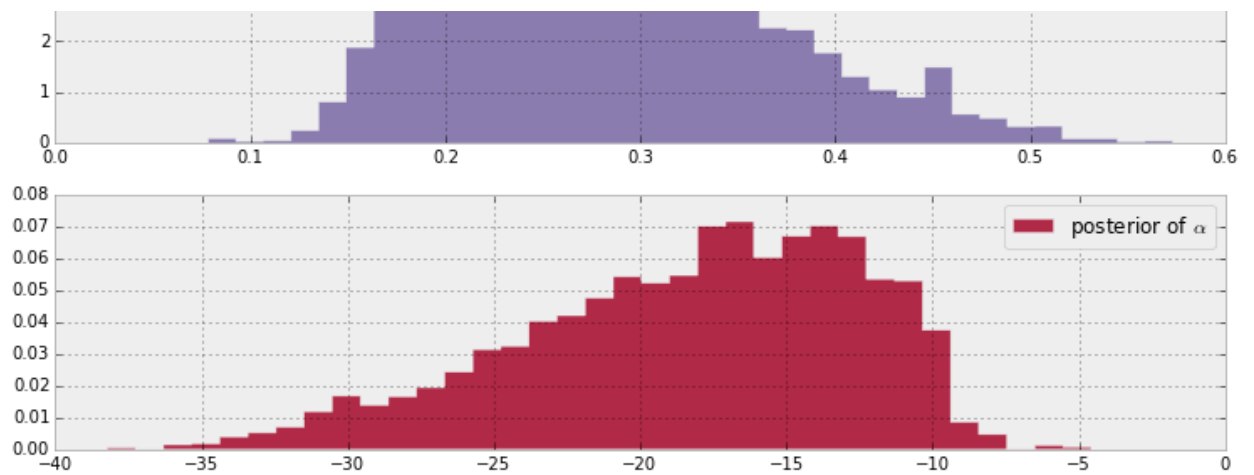
```
<matplotlib.legend.Legend at 0x10dd81d50>
```
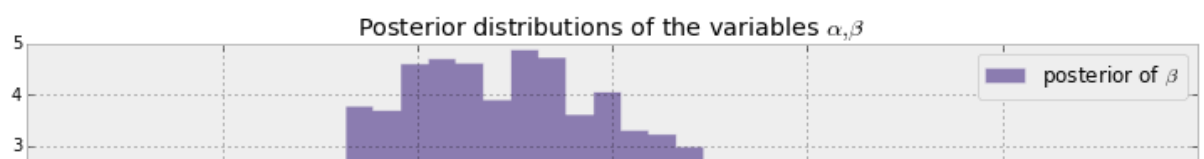
All samples of β are greater than 0. If instead the posterior was centered around 0, we may suspect that $\beta = 0$, implying that temperature has no effect on the probability of defect.

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 $p(t_i)$.

```python
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)
```

```python
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")
```

```
<matplotlib.text.Text at 0x10dd85f50>
```
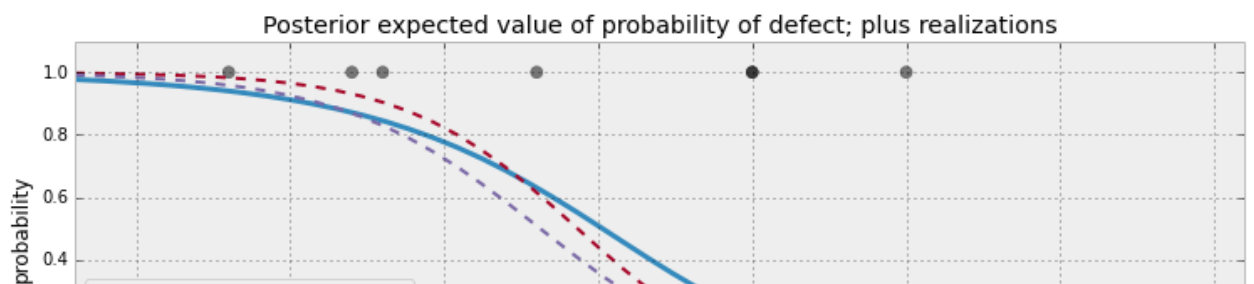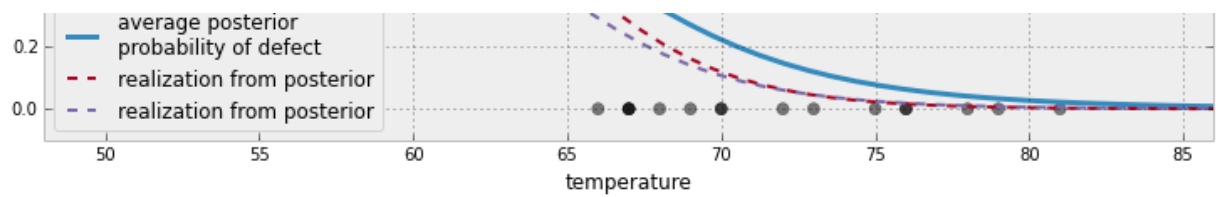
Above we also plotted two possible realizations of what the actual underlying system might be. Both are equally likely as any other draw. The blue line is what occurs when we average all the 20000 possible dotted lines together.

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.

```python
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. $t$")
```
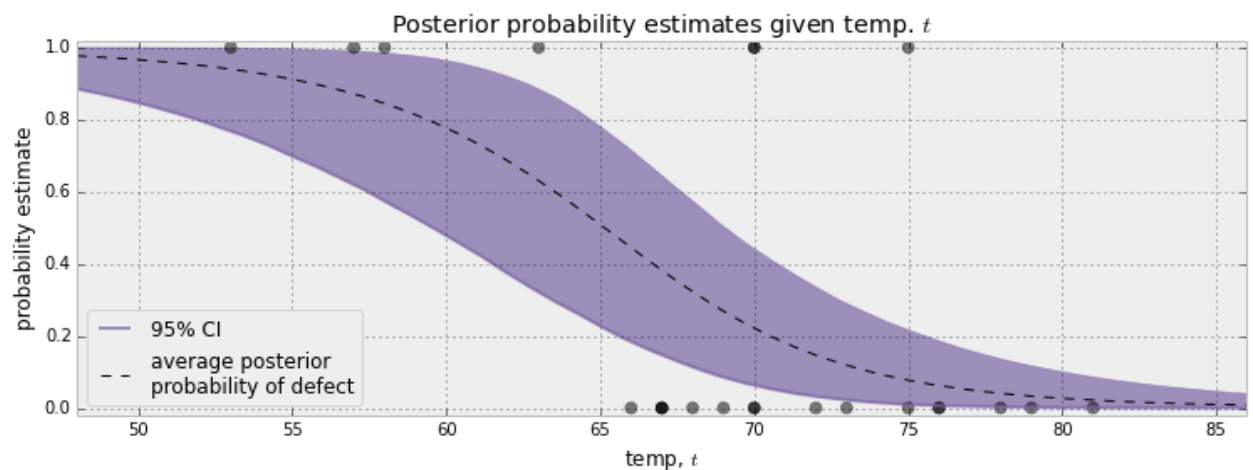
<matplotlib.text.Text at 0x10ea026d0>

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.

## 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.

```
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")
```

```
<matplotlib.text.Text at 0x10e9fc2d0>
```
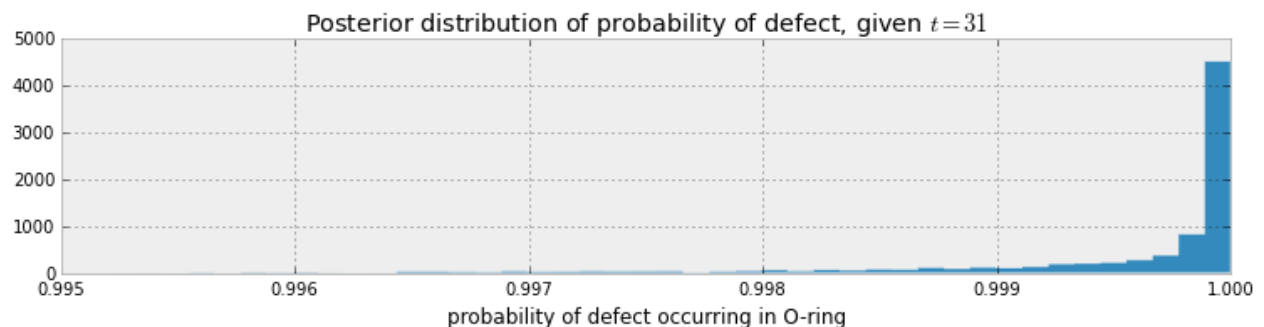

Is our model appropriate?

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, value=D, observed=True)
```

Hence we create:

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

Let's simulate 10 000:

```
simulated = pm.Bernoulli("bernoulli_sim", p)
N = 10000

mcmc = pm.MCMC([simulated, alpha, beta, observed])
mcmc.sample(N)
```
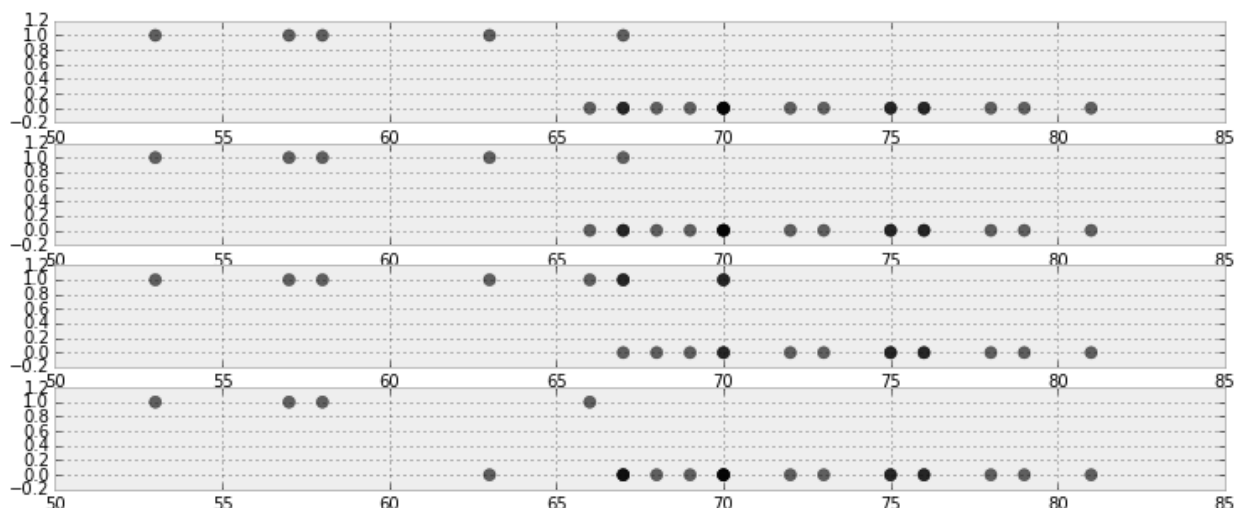
```
[****************100%******************]  10000 of 10000 complete
```

```
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 = 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 (http://stats.stackexchange.com/questions/53078/how-to-visualize-bayesian-goodness-of-fit-for-logistic-regression)!).

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 (http://mdwardlab.com/sites/default/files/GreenhillWardSacks.pdf), 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:

```
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
0.47                     |  0
0.19                     |  1
0.25                     |  0
0.32                     |  0
0.39                     |  0
0.11                     |  0
0.09                     |  0
0.20                     |  0
0.93                     |  1
0.69                     |  1
0.20                     |  1
0.02                     |  0
0.40                     |  0
0.98                     |  1
0.40                     |  0
0.05                     |  0
0.20                     |  0
0.01                     |  0
0.04                     |  0
0.01                     |  0
0.05                     |  1
0.04                     |  0
0.91                     |  1
```

Next we sort each column by the posterior probabilities:

```
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
0.01  |   0
0.01  |   0
0.02  |   0
0.04  |   0
0.04  |   0
0.05  |   1
0.05  |   0
0.09  |   0
0.11  |   0
0.19  |   1
0.20  |   0
0.20  |   0
0.20  |   1
0.25  |   0
0.32  |   0
0.39  |   0
0.40  |   0
0.40  |   0
0.47  |   0
0.69  |   1
0.91  |   1
0.93  |   1
0.98  |   1
```

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

```
from separation_plot import separation_plot

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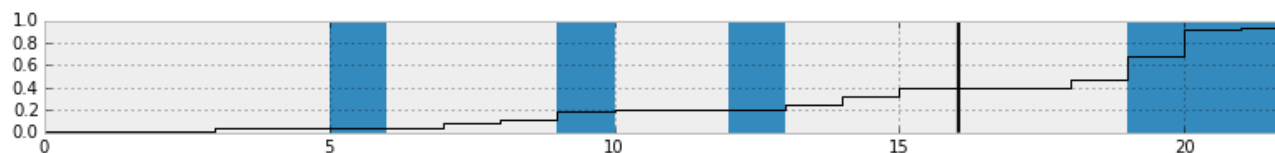

The snaking-line is the sorted probabilities, blue bars denote defects, and empty space (or grey bars for the optimistic readers) denote non-defects. As the probability rises, we see more and more defects occur. On the right hand side, the plot suggests that as the posterior probability is large (line 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.

```
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")
```
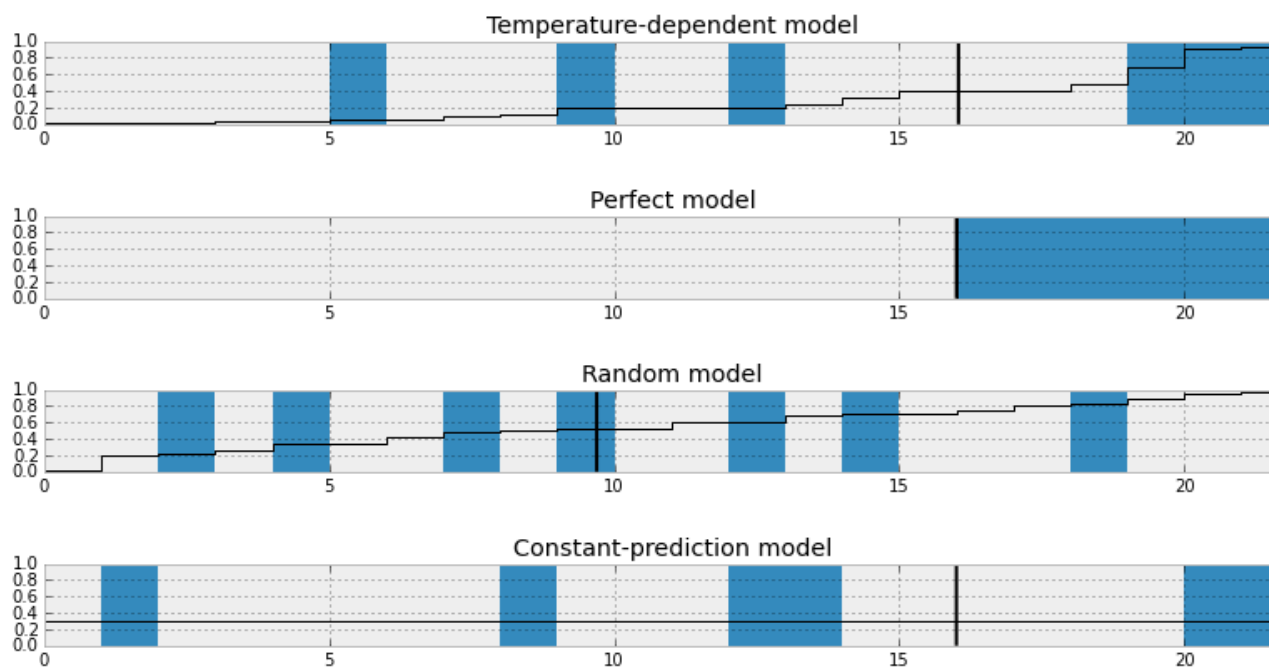
```
<matplotlib.text.Text at 0x10f0c9c50>
```


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?

```python
#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"$\beta$")
```

```
<matplotlib.text.Text at 0x10f007250>
```
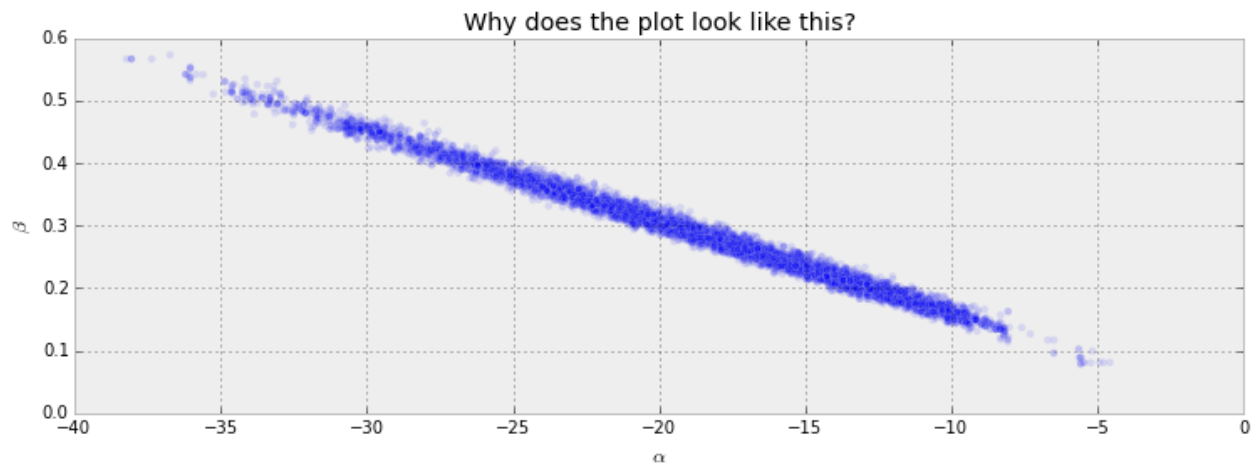

References

- [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 (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 (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 (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.

```python
from IPython.core.display import HTML


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

```
        return HTML(styles)
css_styling()
```

Back to top