[1]:
```
%matplotlib inline
import matplotlib

from IPython.display import set_matplotlib_formats
set_matplotlib_formats('pdf')

# make figures better for projector:
font = {'weight':'normal','size':20}
matplotlib.rc('font', **font)
matplotlib.rc('figure', figsize=(8.0, 6.0))
matplotlib.rc('xtick', labelsize=16)
matplotlib.rc('ytick', labelsize=16)
matplotlib.rc('legend',**{'fontsize':16})

import warnings
warnings.filterwarnings('ignore')
```

[2]:
```
import numpy as np
import matplotlib.pyplot as plt
import random
import scipy, scipy.stats
from IPython.display import Image

# some helper functions, to be used later:

import textwrap
def print_attribute_names(obj, keep_private=False, width=80, delimiter="   "):
    """Nicely display attributes of object obj."""
    attrs = [d for d in obj.__dir__() if type(d) == type("")]
    if not keep_private:
        attrs = [d for d in attrs if not d.startswith("__")]
    print(textwrap.fill(delimiter.join(attrs)))

def print_doc(obj,  cutoff=None):
    print(textwrap.dedent(obj.__doc__).strip()[:cutoff])

def plot_mat(X,Y,Z, fig=None, extent=None):
    if fig is None:
        fig = plt.gcf() # get current fig
    ax = fig.gca()
    #ax.contour(X,Y, Z)
    ax.imshow(Z, interpolation='none', origin='lower',
              extent=extent)
    if extent is not None:
        ax.set_xlim(extent[0:2])
        ax.set_ylim(extent[2:] )

def plot_truth(x_true,y_true, fig=None):
    if fig is None:
        fig = plt.gcf() # get current fig
    ax = fig.gca()
```

```
    ax.scatter(x_true, y_true, c="w", s=50, edgecolor="k",zorder=1e9)
```

DS1 Lecture 26

James Bagrow, james.bagrow@uvm.edu, http://bagrow.com

Previously:

1. Bayesian inference
   - Is a coin biased?
   - text messaging data, change in rate?
2. Markov Chain Monte Carlo (MCMC)

Today:

1. Picture of an MCMC trace
2. Doing MCMC with PyMC
3. (Time permitting) Biased coin with MCMC and PyMC
4. Text messages model with MCMC
5. (Time permitting) Logistic regression
   - Challenger accident

Recall from class

We went on a very long discussion on bayesian inference and markov chain monte carlo.

- What is the likelihood? What is the prior (or prior distribution)? What is the posterior?
- How to study the posterior when we can't even compute it?
  - MCMC samples from the posterior

For our text messages data:

We proposed to model the number of texts per day as a poisson, $C_t \sim \mathrm{Pois}(\lambda)$ where λ took on one of two rates, λ_1 or λ_2, depending on whether $t < \tau$ or $t \geq \tau$.

We drew many samples from the posterior distribution (our *trace*) to give us an estimate of the probability of this model given the data.

Visualizing the MCMC trace

When we first discussed the statistical model, we made a picture of the prior and posterior by plotting a matrix of many different values of λ_1 and λ_2. We also dropped τ from this cartoon by forcing it to be in the middle of the data.

- As mentioned then, *sweeping* through multiple parameter values is too expensive in practice. Imagine trying to compute every combination of parameter values for a 10-parameter model. What about a 100-parameter model?

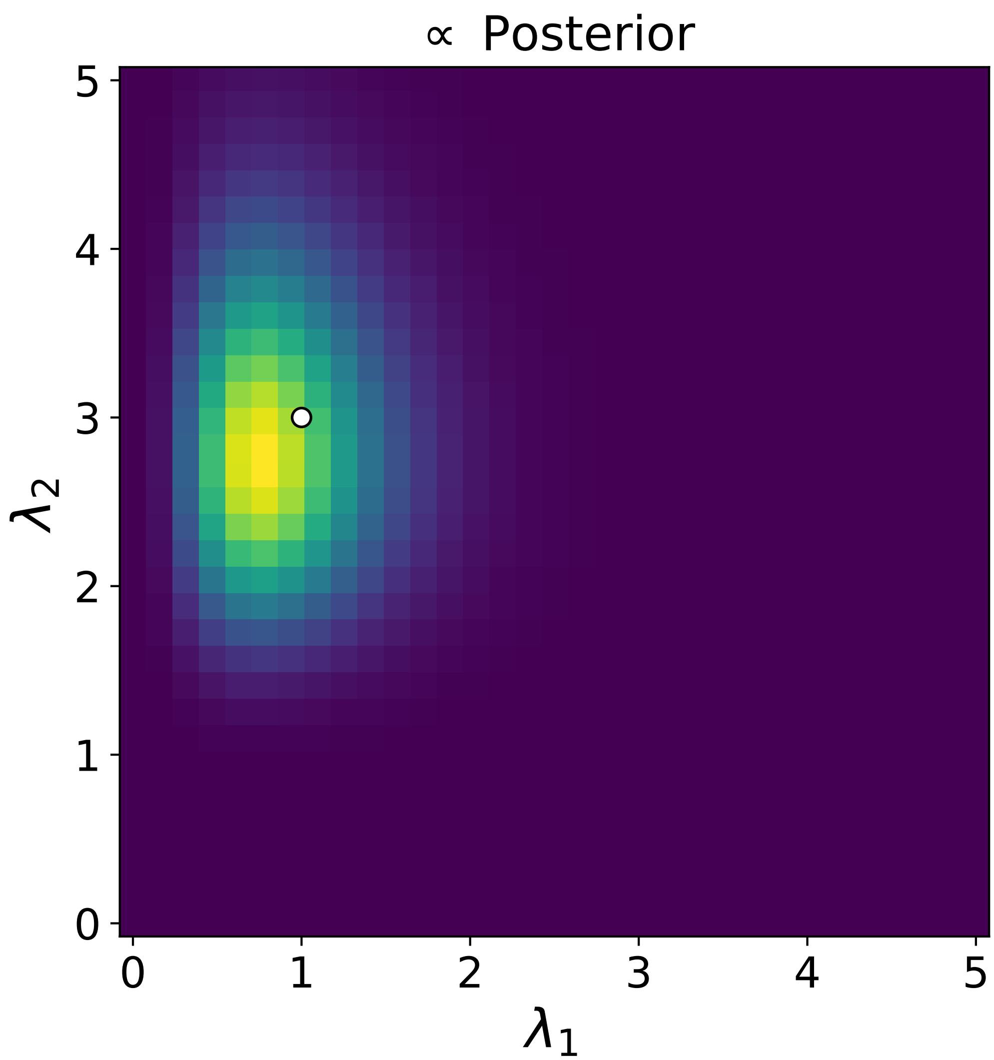
Here's the code from that lecture

[3]:
```
# build synthetic data, no tau:
N = 5 # 2N = number of data points
num_bins = 33 # for parameter sweeps
lambda_1_true,lambda_2_true = 1,3
```

```python
data_b = scipy.stats.poisson.rvs(lambda_1_true, size=(N, 1))
data_a = scipy.stats.poisson.rvs(lambda_2_true, size=(N, 1))
data = np.append(data_b, data_a)
T = len(data)

# exponential prior:
L1 = L2 = np.linspace(0.00, 5, num_bins) # parameter sweeps
exp_L1 = scipy.stats.expon.pdf(L1, scale=3)
exp_L2 = scipy.stats.expon.pdf(L2, scale=10)
Exp_Prior = np.dot(exp_L2[:, None], exp_L1[None, :])

# likelihood of data using poisson distributions:
Pois = scipy.stats.poisson.pmf
like_db = np.array([Pois(data_b, lam1).prod() for lam1 in L1])
like_da = np.array([Pois(data_a, lam2).prod() for lam2 in L2])
Likelihood = np.dot(like_da[:, None], like_db[None, :])

# This is *proportional* to posterior:
Posterior = np.nan_to_num(Likelihood * Exp_Prior) # element-wise product
```

Here's a plot of that posterior, annotated with the true (unknown) value of λ_1 and λ_2:

[4]:
```python
# plotting:
s = (L1[1]- L1[0])/2 # shift to align matrix "pixel" centers
extent = (min(L1)-s,max(L1)+s,min(L2)-s,max(L2)+s)
tfsize = 18

#plt.subplot(121)
plot_mat(L1, L2, Posterior, extent=extent)
plot_truth(lambda_1_true, lambda_2_true)
plt.title(r"$\propto$ Posterior", fontsize=tfsize)
plt.xlabel(r"$\lambda_1$")
plt.ylabel(r"$\lambda_2$")
plt.show()
```



OK, now let's do a small MCMC sample, 4000 steps with no burnin or thinning:

- Also: see how quickly we can build the model in PyMC:

[5]:
```
# MCMC sampling using PyMC:
import pymc3 as pm

# PyMC priors (no tau):
alpha = 1.0 / data.mean()

mod = pm.Model()
with mod:
    lam  = pm.Exponential('lam', lam=alpha, shape=2)
    grp = (np.arange(T) > T/2) * 1 # remember, no tau
    y_obs = pm.Poisson('y_obs', mu=lam[grp], observed=data)

    #step =  pm.Slice([lam])
    step = pm.Metropolis([lam], tune=False, scaling=0.05)
    #step = pm.NUTS() # default, No U-Turn Sampler
```

```
    # sample from the posterior using one or more "step" methods:
    trace = pm.sample(step=step, draws=10000, chains=1, tune=300,
                      start={'lam':[4.4,0.4]},
                      discard_tuned_samples=False)

# Get sampled values (the trace):
lambda_samples = trace.get_values('lam')

lambda_1_samples = lambda_samples[:,0]
lambda_2_samples = lambda_samples[:,1]
```

```
Sequential sampling (1 chains in 1 job)
Metropolis: [lam]
Sampling chain 0, 0 divergences: 100%|      | 10300/10300 [00:02<00:00, 5026.21it/s]
Only one chain was sampled, this makes it impossible to run some convergence checks
```

[6]:
```
lambda_samples[:10]
```

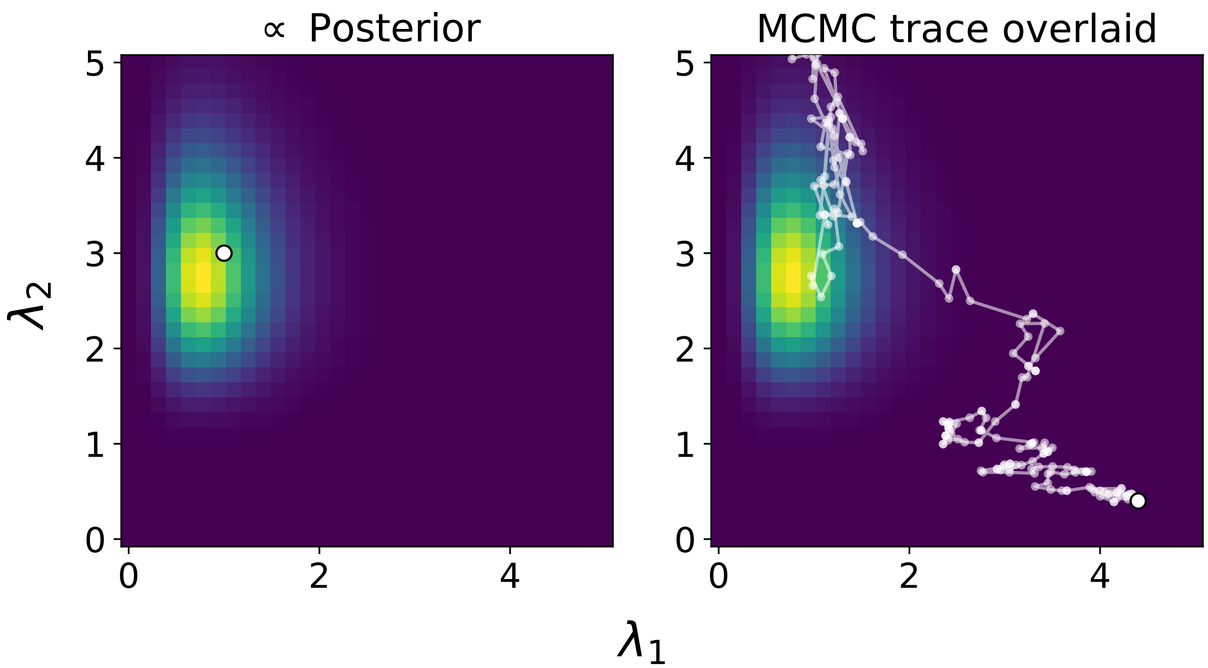
[6]:
```
array([[4.29045584, 0.41938919],
       [4.17335911, 0.42732034],
       [4.14490558, 0.38998755],
       [4.14490558, 0.38998755],
       [4.00123402, 0.4496602 ],
       [4.2171389 , 0.47361679],
       [4.27633261, 0.44872134],
       [4.2936542 , 0.45964969],
       [4.09920264, 0.47106115],
       [4.08128581, 0.46967071]])
```

Here's the posterior on the left and the posterior with the MCMC "trace" superimposed on the right:

- Only the first 200 steps:

[7]:
```
# plotting:
fig = plt.figure(figsize=(9,5))
s = (L1[1]- L1[0])/2 # shift to align matrix "pixel" centers
extent = (min(L1)-s,max(L1)+s,min(L2)-s,max(L2)+s)
tfsize = 18

plt.subplot(121)
plot_mat(L1, L2, Posterior, extent=extent)
plot_truth(lambda_1_true, lambda_2_true)
plt.title(r"$\propto$ Posterior", fontsize=tfsize)

plt.subplot(122)
plot_mat(L1, L2, Posterior, extent=extent)
num = 200
plt.plot(lambda_1_samples[:num], lambda_2_samples[:num],
         '.-w', zorder=1e8, alpha=0.55)
plot_truth(4.4,0.4)
plt.title("MCMC trace overlaid", fontsize=tfsize)
```

```
fig.text(0.5, 0.06, r"$\lambda_1$",
         fontsize=22, ha='center', va='center')
fig.text(0.06, 0.5, r"$\lambda_2$",
         fontsize=22, ha='center', va='center',
         rotation='vertical');
```

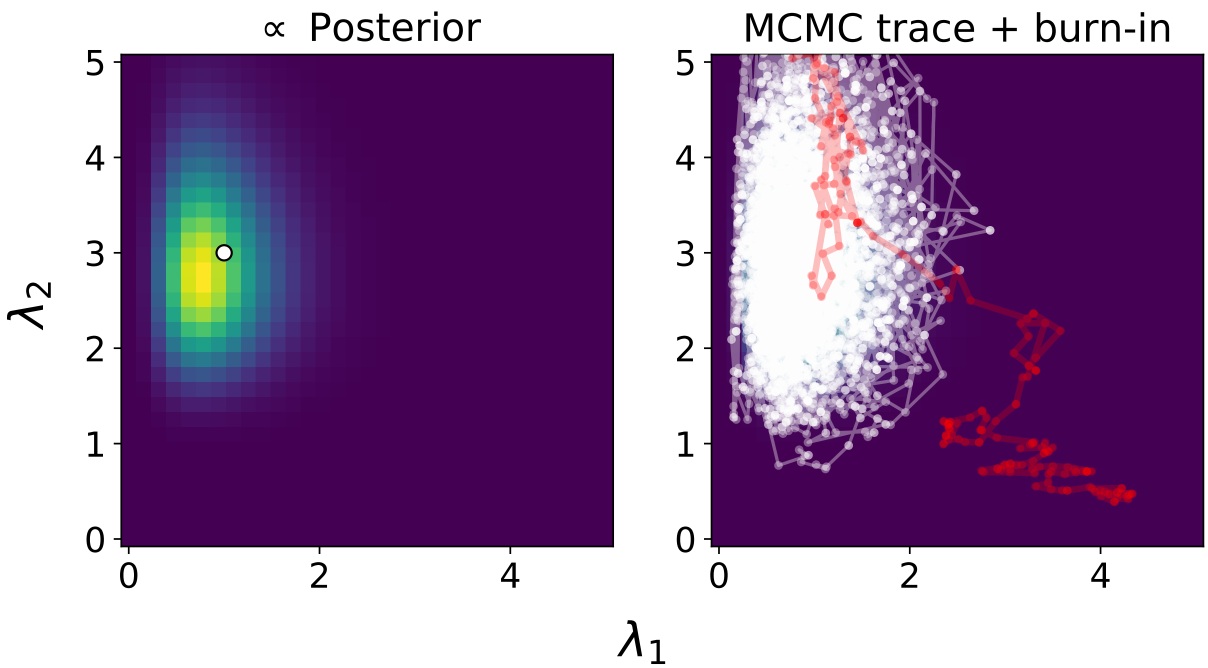


So we see that the MCMC "walker" starts way off from the high-probability regions of posterior, wanders around for a while until it gets closer to the high-probability region.

Now let's plot the rest of the MCMC trace, highlighting this initial movement with a different color:

[8]:
```
fig = plt.figure(figsize=(9,5))

plt.subplot(121)
plot_mat(L1, L2, Posterior, extent=extent)
plot_truth(lambda_1_true, lambda_2_true)
plt.title(r"$\propto$ Posterior", fontsize=tfsize)

plt.subplot(122)
plot_mat(L1, L2, Posterior, extent=extent)
plt.plot(lambda_1_samples[num:], lambda_2_samples[num:],
         '.-w', zorder=1e8, alpha=0.35)
plt.plot(lambda_1_samples[:num], lambda_2_samples[:num],
         '.-r', zorder=1e8, alpha=0.25, lw=3)
plt.title(r"MCMC trace + burn-in", fontsize=tfsize)

fig.text(0.5, 0.06, r"$\lambda_1$",
         fontsize=22, ha='center', va='center')
fig.text(0.06, 0.5, r"$\lambda_2$",
```

```
        fontsize=22, ha='center', va='center',
        rotation='vertical');
```



(The above illustration is a bit unrealistic as I forced the step method to perform poorly to emphasize the failure mode. The defaults work much better.)

Notice how much better "mixed" the points are after we get to the high-probability region of the posterior!

The notion of MCMC, where we choose a random parameter θ and jump to a nearby parameter θ' with probability proportional to the **ratio** of our estimates of the posterior at those locations, should make intuitive sense.

- But we haven't actually implemented this idea. We should code it up to REALLY learn in.

Instead let's go through a common Python library, creatively named PyMC.

Diagnosing an MCMC trace

If everything is working, the trace should be a collection of iid draws from the posterior. That means they should be independent from one another. But since one point is used to generate the next point, the trace will exhibit **serial correlation**

- Burn in - delete the early values which may be abnormal
- Thinning - keep only every kth value, to break serial correlations

Diagnostics include

Plotting the trace — does it look random?

Plotting the distribution of the trace — does it look normally distributed?

Autocorrelation — Pearson correlation between values of the trace at different times:

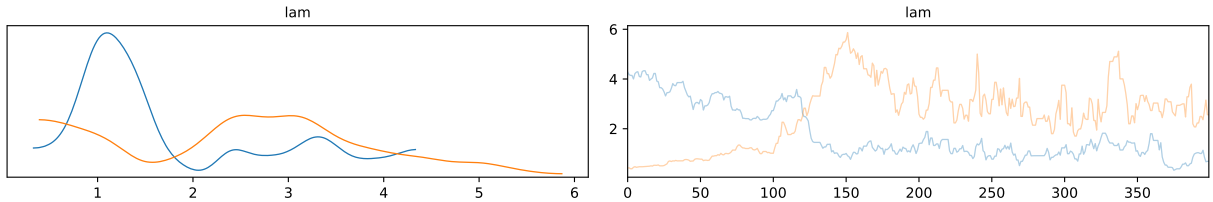
$$R(\tau) = \frac{\mathrm{E}\left[\left(X_t - \mu\right)\left(X_{t+\tau} - \mu\right)\right]}{\sigma^2}$$

Does the trace look uncorrelated?

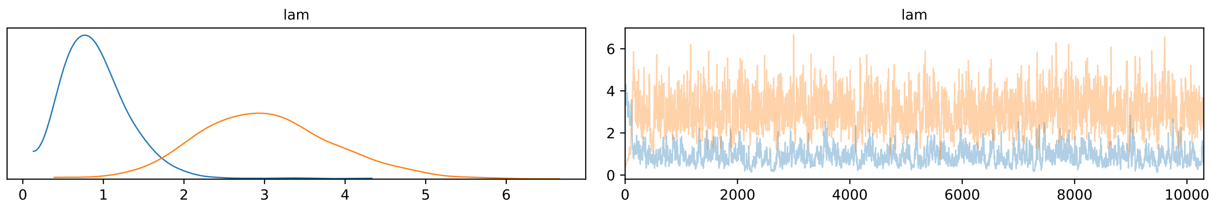
Model checking by eye:

No burn-in and no thinning

[9]:
```
pm.traceplot(trace[:400]);
```



[10]:
```
pm.traceplot(trace);
```



Bayesian inference with PyMC

We started Bayes with the coin, how to tell if a coin was biased or not. This is a rare example where we can write down the EXACT posterior distribution.

Let's teach ourselves a bit of PyMC using this same problem, so we can take our solution and compare to the MCMC samples from the posterior.

- First, let's build some coin flip data:

[11]:
```
n_flips = 100
n_heads = 60

data = [0]*(n_flips-n_heads) + [1]*n_heads
random.shuffle(data) # not really needed

print(data[:15], ". . .")
```

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

OK, now let's do the inference.

We need to build a model for the data. Recall since all flips are iid, we are dealing with n_{flips} **bernoulli** random variables. We used this for our likelihood which was a product → binomial distribution.

We need a prior distribution as well. The tells us the probability for a particular value of coin bias p before we look at data. We assumed all p's were equally likely, making it a uniform random variable, or $p \sim U(0, 1)$.

From previous lecture:

[12]:
```python
Image(filename='screenshot_coinFlip_notes.png')
```

[12]:

Let's apply this to our coin problem:

prior : $\boxed{f(p)=1}$ → uniform prior, all values of p equally likely

$$\Pr(\underset{\text{"model"}}{p} \mid \underset{\text{"data"}}{k \text{ heads}}) = \frac{\Pr(k \text{ heads} \mid p)\, f(p)}{\int_0^1 \Pr(k \text{ heads} \mid r)\, f(r)\, dr}$$

$$= \frac{\binom{n}{k} p^k (1-p)^{n-k}}{\int_0^1 \binom{n}{k} r^k (1-r)^{n-k} dr}$$

$$= \frac{p^k (1-p)^{n-k}}{\int_0^1 r^k (1-r)^{n-k} dr} \leftarrow \text{integrate this ?}$$

Beta distribution
$$= \frac{p^k (1-p)^{n-k}}{B(k+1, n-k+1)} \leftarrow \text{Beta function} \leftrightarrow \text{or use binomial theorem b/c } n, k \text{ are integers}$$

$$= \frac{(n+1)!}{k!(n-k)!} p^k (1-p)^{n-k} = (n+1)\binom{n}{k} p^k (1-p)^{n-k}$$

$\boxed{2}$

→ These probability distributions are actually implemented in PyMC as python objects!

[13]:
```python
import pymc3 as pm

# this is our prior distribution for coin flip prob p:
coins = pm.Model()
with coins:
    p = pm.Uniform("p", lower=0, upper=1)
```

This object p is a subclass of a PyMC class called Stochastic, representing random variables.

- Let's see some of the things p gives us:

[14]:
```python
print_attribute_names( p )
```

```
tag   type   owner   index   name   auto_name   transformation   model
distribution   dshape   dsize   transformed   scaling   random
```

```
_repr_latex_   init_value   _is_nonzero   T   transpose   shape   size
any   all   reshape   dimshuffle   flatten   ravel   diagonal
transfer   arccos   arccosh   arcsin   arcsinh   arctan   arctanh
ceil   cos   cosh   deg2rad   exp   exp2   expm1   floor   log   log10
log1p   log2   rad2deg   sin   sinh   sqrt   tan   tanh   trunc
astype   take   copy   ndim   broadcastable   dtype   dot   sum   prod
norm   mean   var   std   min   max   argmin   argmax   nonzero
nonzero_values   sort   argsort   clip   conj   conjugate   repeat
round   trace   get_scalar_constant_value   zeros_like   ones_like
cumsum   cumprod   searchsorted   ptp   swapaxes   fill   choose
squeeze   compress   real   imag   clone   get_parents   eval
construction_observers   append_construction_observer
remove_construction_observer   notify_construction_observers
```

We can draw values from this distribution using its random method:

[15]:
```
print(p.random(), p.random(), p.random())
```

```
0.46665171092971647 0.9036173548217922 0.9805214262571803
```

PyMC provides **many** random variables/probability distributions:

[16]:
```
print_attribute_names( pm.distributions)
```

```
shape_utils   distribution   transforms   special   dist_math
continuous   multivariate   timeseries   Uniform   Flat   HalfFlat
TruncatedNormal   Normal   Beta   Kumaraswamy   Exponential   Laplace
StudentT   Cauchy   HalfCauchy   Gamma   Weibull   HalfStudentT
Lognormal   ChiSquared   HalfNormal   Wald   Pareto   InverseGamma
ExGaussian   VonMises   SkewNormal   Triangular   Gumbel   Logistic
LogitNormal   Interpolated   Rice   discrete   Binomial   BetaBinomial
Bernoulli   DiscreteWeibull   Poisson   NegativeBinomial
ConstantDist   Constant   ZeroInflatedPoisson
ZeroInflatedNegativeBinomial   ZeroInflatedBinomial   DiscreteUniform
Geometric   Categorical   OrderedLogistic   DensityDist   Distribution
Continuous   Discrete   NoDistribution   TensorType   draw_values
generate_samples   simulator   Simulator   mixture   Mixture
NormalMixture   MvNormal   MatrixNormal   KroneckerNormal   MvStudentT
Dirichlet   Multinomial   Wishart   WishartBartlett   LKJCholeskyCov
LKJCorr   AR1   AR   GaussianRandomWalk   GARCH11
MvGaussianRandomWalk   MvStudentTRandomWalk   bound   Bound
```

And **documentation** is available (though it's often not great):

[17]:
```
print_doc(pm.Uniform)
```

```
Continuous uniform log-likelihood.

The pdf of this distribution is

.. math::

   f(x \mid lower, upper) = \frac{1}{upper-lower}
```

```
.. plot::

    import matplotlib.pyplot as plt
    import numpy as np
    plt.style.use('seaborn-darkgrid')
    x = np.linspace(-3, 3, 500)
    ls = [0., -2]
    us = [2., 1]
    for l, u in zip(ls, us):
        y = np.zeros(500)
        y[(x<u) & (x>l)] = 1.0/(u-l)
        plt.plot(x, y, label='lower = {}, upper = {}'.format(l, u))
    plt.xlabel('x', fontsize=12)
    plt.ylabel('f(x)', fontsize=12)
    plt.ylim(0, 1)
    plt.legend(loc=1)
    plt.show()

========  =====================================
Support   :math:`x \in [lower, upper]`
Mean      :math:`\dfrac{lower + upper}{2}`
Variance  :math:`\dfrac{(upper - lower)^2}{12}`
========  =====================================

Parameters
----------
lower: float
    Lower limit.
upper: float
    Upper limit.
```

(You would access this interactively by running pm.Uniform? in IPython)

Adding data

OK, so we can make probability distributions. What about data?

- We need to compute **likelihoods**

Data are stored in **the same** so-called distribution ``objects'' (WEIRD!)

Here's the data for the coin flip, in the form PyMC uses:

[18]:
```python
with coins:
    obs = pm.Bernoulli("obs", p, observed=data)
```

Notice a few things:

1. The parameter for the Bernoulli distribution is p. This is our uniform prior for the coin bias! These function calls, where one distribution is called **inside** another, are what link the statistical model together.
2. observed=data. These are how the data are incorporated. We assume it's a Bernoulli variable but its **value** is fixed to the data. This way PyMC can use the **same code**

to compute both Bernoulli random variables and the likelihood of a bunch of data given/assuming a Bernoulli.

The second part can take a while to wrap your head around!

[19]:
```
example = pm.Model()
with example:
    berRnd = pm.Bernoulli("berRnd", p)
    berObs = pm.Bernoulli("berObs", p, observed=1)
```

Easy!

And here we can set up the materials for running MCMC:

How to perform the sample:

[20]:
```
print_doc( pm.sample, cutoff=1000 )
```

```
Draw samples from the posterior using the given step methods.

    Multiple step methods are supported via compound step methods.

    Parameters
    ----------
    draws : int
        The number of samples to draw. Defaults to 500. The number of tuned samples are
discarded
        by default. See ``discard_tuned_samples``.
    init : str
        Initialization method to use for auto-assigned NUTS samplers.

        * auto : Choose a default initialization method automatically.
          Currently, this is ``'jitter+adapt_diag'``, but this can change in the future.
          If you depend on the exact behaviour, choose an initialization method
explicitly.
        * adapt_diag : Start with a identity mass matrix and then adapt a diagonal based
on the
          variance of the tuning samples. All chains use the test value (usually the
prior mean)
          as starting point.
        * jitter+adapt_diag : Same as ``adapt_diag``\, but add uniform jitter in [-1, 1]
to the
          starting
```

How to extract the samples (or **trace**) once MCMC is finished:

Let's do the sampling!

[21]:
```
with coins:
    trace = pm.sample(30000)
```

```
Auto-assigning NUTS sampler…
Initializing NUTS using jitter+adapt_diag…
Multiprocess sampling (4 chains in 4 jobs)
NUTS: [p]
```

Sampling 4 chains, 0 divergences: 100%| | 122000/122000 [00:22<00:00,
5445.38draws/s]

Cool! A little progress bar! (When you run this yourself you can watch it fill up!)

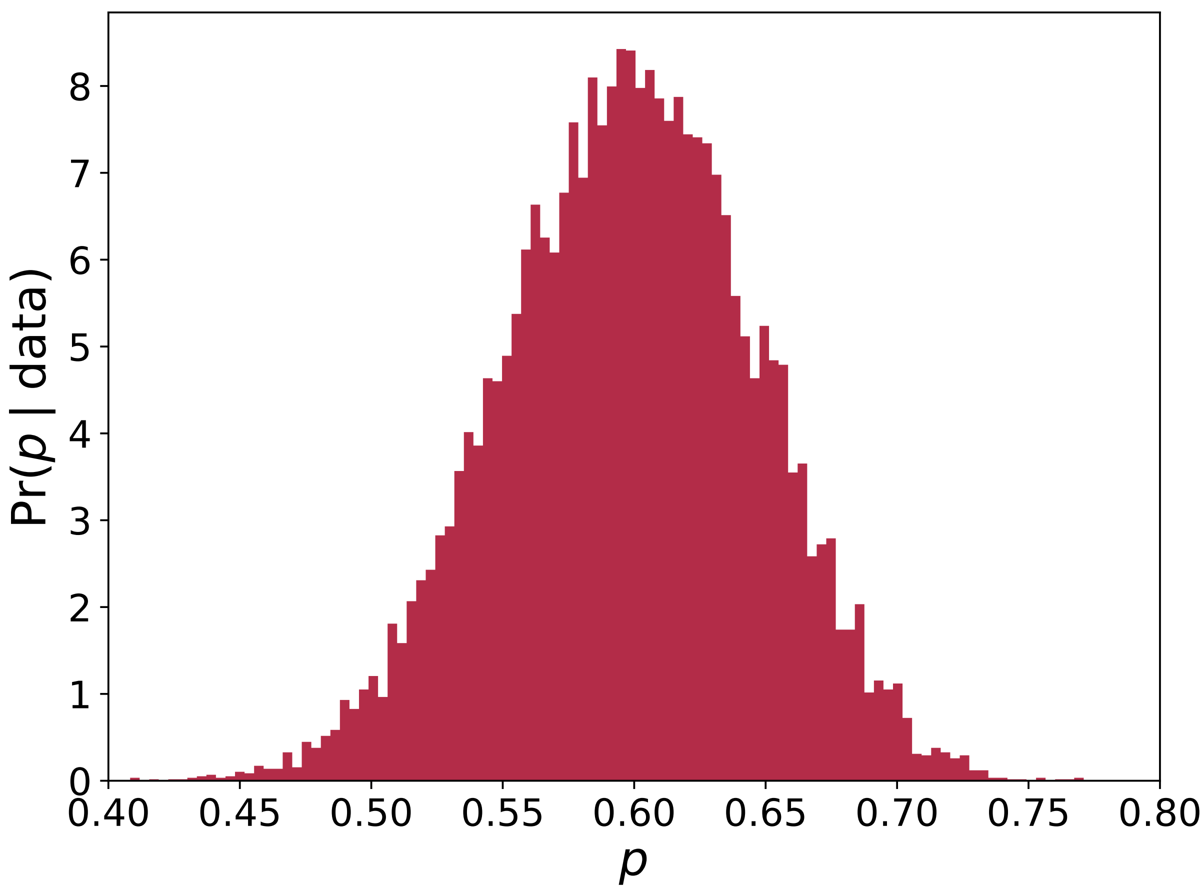
We can now put the trace into a simple numpy array:

[22]:
```
p_samples = trace.get_values("p", burn=10000,thin=5)

print(type(p_samples))
print(p_samples.shape)
print(p_samples[:20])
```

```
<class 'numpy.ndarray'>
(16000,)
[0.70705071 0.53215432 0.5659377  0.49401561 0.62830127 0.54592507
 0.6939224  0.52826051 0.53804334 0.66729895 0.578572   0.58708797
 0.51811652 0.67876187 0.62524115 0.63609212 0.67190192 0.62322972
 0.63007496 0.56809781]
```

And we can look at the **posterior** of p (the probability of p given the coin flips) with just a **histogram**:

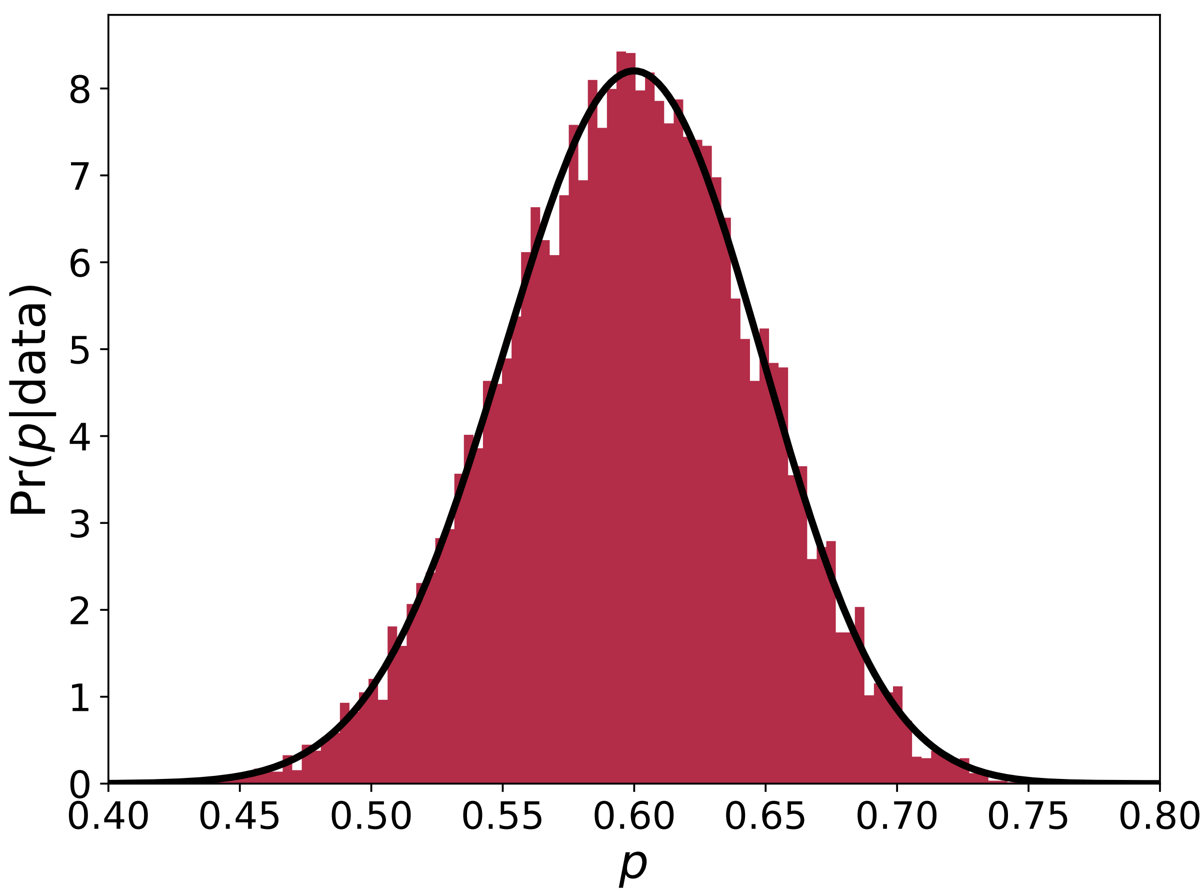
[23]:
```
plt.hist(p_samples, bins=100,
         histtype='stepfilled', alpha=0.85,
         color="#A60628", normed=True, edgecolor="none")
plt.xlabel("$p$"); plt.ylabel(r"Pr($p \mid$data)")
plt.xlim(0.4,0.8);
```



And how does this compare to our analytic equation (derived in class) for the posterior?

[24]:
```
plt.hist(p_samples, bins=100,
         histtype='stepfilled', alpha=0.85,
         label="posterior", color="#A60628", normed=True, edgecolor="none")
plt.xlabel("$p$"); plt.ylabel("Pr($p$|data)")

# plot exact expression:
x = np.linspace(0,1,500)

n = n_flips
k = n_heads
f = scipy.special.factorial
print(n, k)

y = f(n+1)/f(k)/f(n-k) * x**k * (1-x)**(n-k) # derived in a prev lecture
plt.plot(x,y, 'k-', lw=3)
plt.xlim(0.4,0.8);
```

```
100 60
```



BAM!!!

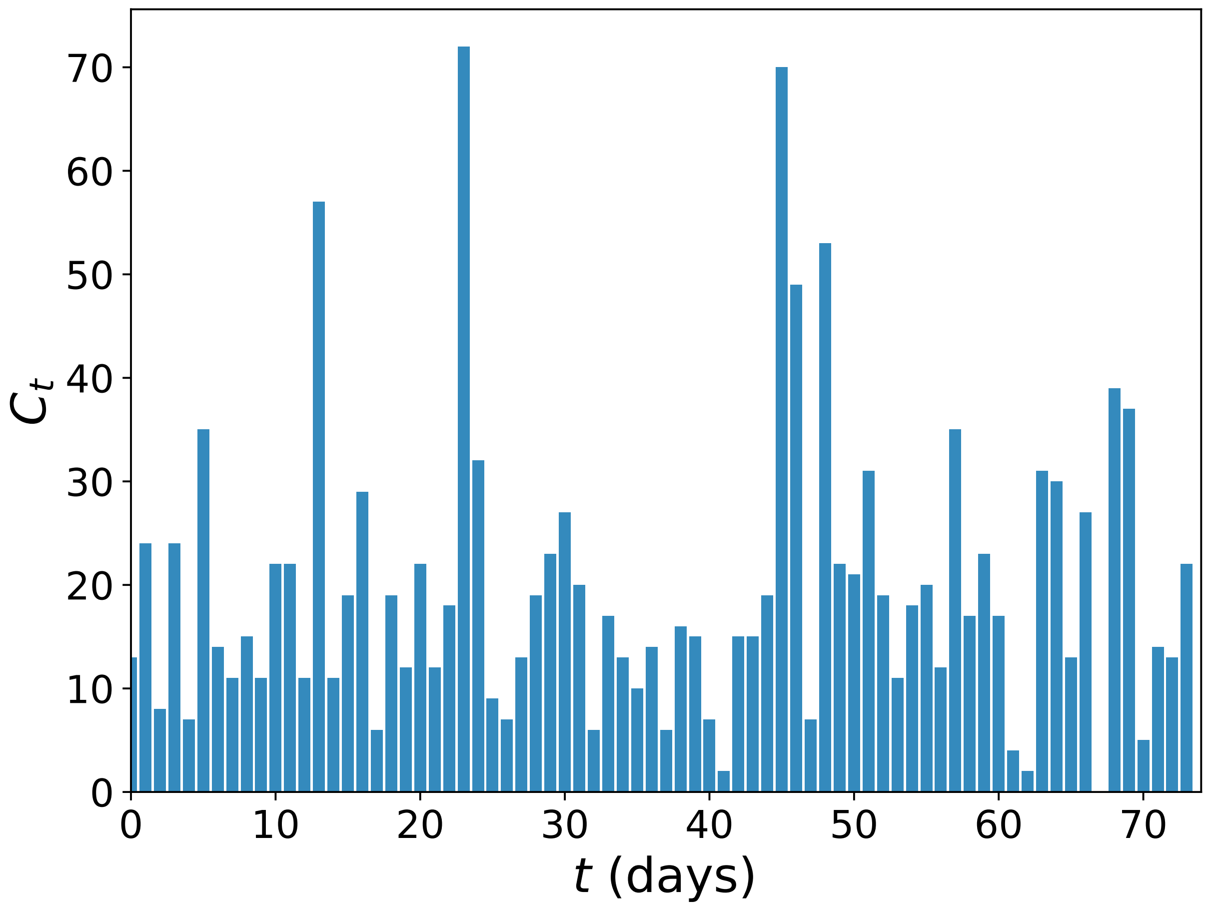
Looking good!

Text Message Rates

OK, now let's look at what we did for the text messages.

- Load the data:

[25]:
```
C = np.loadtxt("txtdata.csv")
T = len(C)
plt.bar(np.arange(T), C, color="#348ABD", ec='none')
plt.xlabel("$t$ (days)")
plt.ylabel("$C_t$")
plt.xlim(0, T);
```



Recall, here we had a more complicated model:

$$C_t \sim \text{Pois}(\lambda_t)$$

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

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

$$\lambda_1 \sim \text{Exp}(\alpha)$$
$$\lambda_2 \sim \text{Exp}(\alpha)$$

and α was fixed from the data ($\alpha = E[C_t]^{-1}$).

The λ_1, λ_2 act like our p in the coin flip problem. Their priors are exponentials, whereas the p was uniform.

- But what about the piecewise function for λ_t?

Here's how we build this model in PyMC.

1. First, let's build our priors for the three parameters (λ_1, λ_2, and τ):

[26]:
```
# Build the three prior probability distributions
# using PyMC objects
alpha = 1.0 / C.mean() # Recall C is the numpy vector
                       # that holds our txt counts
print("1/alpha =", 1.0/alpha)

mod = pm.Model()
with mod:
    # Prior for tau:
    tau = pm.DiscreteUniform('tau', 0, T)

    # Prior for lambda1, lambda2 (note shape=2):
    lam  = pm.Exponential('lam', lam=alpha, shape=2)
    grp = (np.arange(T) > tau) * 1 # 0 if t <= tau, 1 if t > tau

    # Likelihood, Poisson w/ time-dependent rate:
    y_obs = pm.Poisson('y_obs', mu=lam[grp], observed=C)

    # algorithm(s) for sampling from posterior:
    #step1 =  pm.Slice([tau])
    step2 = pm.Metropolis([lam])
    #step2 = pm.NUTS() # default, No U-Turn Sampler

    # sample from the posterior using one or more "step" methods:
    trace = pm.sample(step=[step2], draws=15000, chains=1,tune=1000)
```

```
1/alpha = 19.743243243243242

Sequential sampling (1 chains in 1 job)
CompoundStep
>Metropolis: [lam]
>Metropolis: [tau]
```

```
Sampling chain 0, 0 divergences: 100%|          | 16000/16000 [00:04<00:00, 3427.81it/s]
Only one chain was sampled, this makes it impossible to run some convergence checks
```

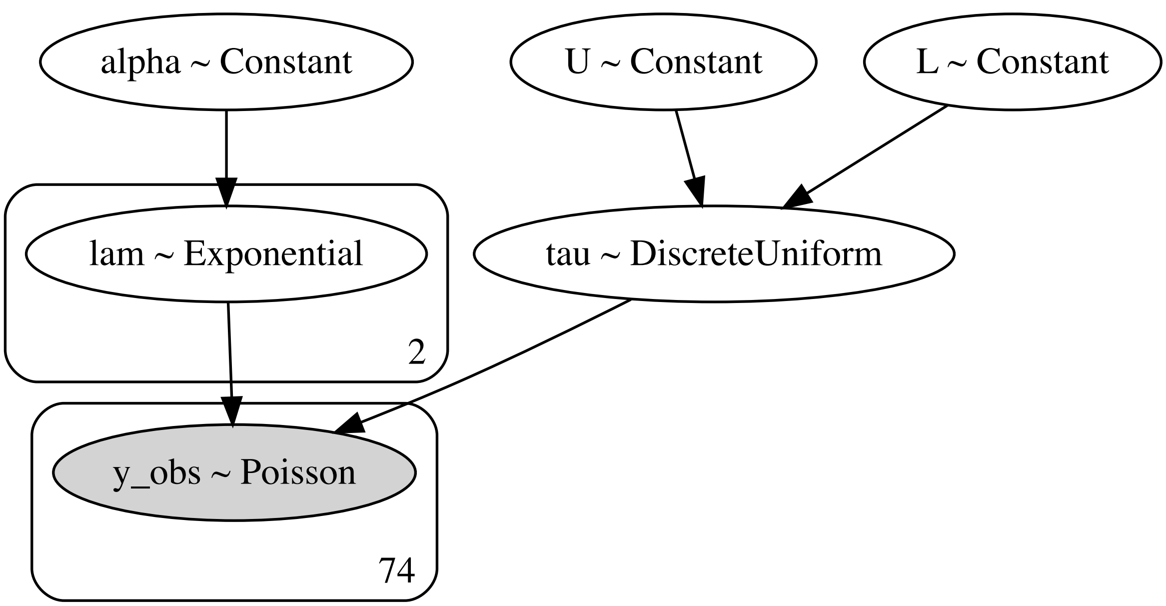
Aside: PyMC actually provides a **nice tool for drawing** the dependencies between the random variables:

- (This may not run on your machine, you need to install some graph-drawing dependencies)

[27]:
```
# Build the three prior probability distributions
# using PyMC objects
alpha = 1.0 / C.mean() # Recall C is the numpy vector
                       # that holds our txt counts
print("1/alpha =", 1.0/alpha)

mod_viz = pm.Model()
with mod_viz:
    # Prior for tau:
    L = pm.Constant("L",0)
    U = pm.Constant("U",T)
    tau = pm.DiscreteUniform('tau', L, U)

    # Prior for lambda1, lambda2 (note shape=2):
    alpha = pm.Constant('alpha',alpha)
    lam  = pm.Exponential('lam', lam=alpha, shape=2)
    grp = (np.arange(T) > tau) * 1 # 0 if t <= tau, 1 if t > tau

    # Likelihood, Poisson w/ time-dependent rate:
    y_obs = pm.Poisson('y_obs', mu=lam[grp], observed=C)

pm.model_to_graphviz(mod_viz)
```

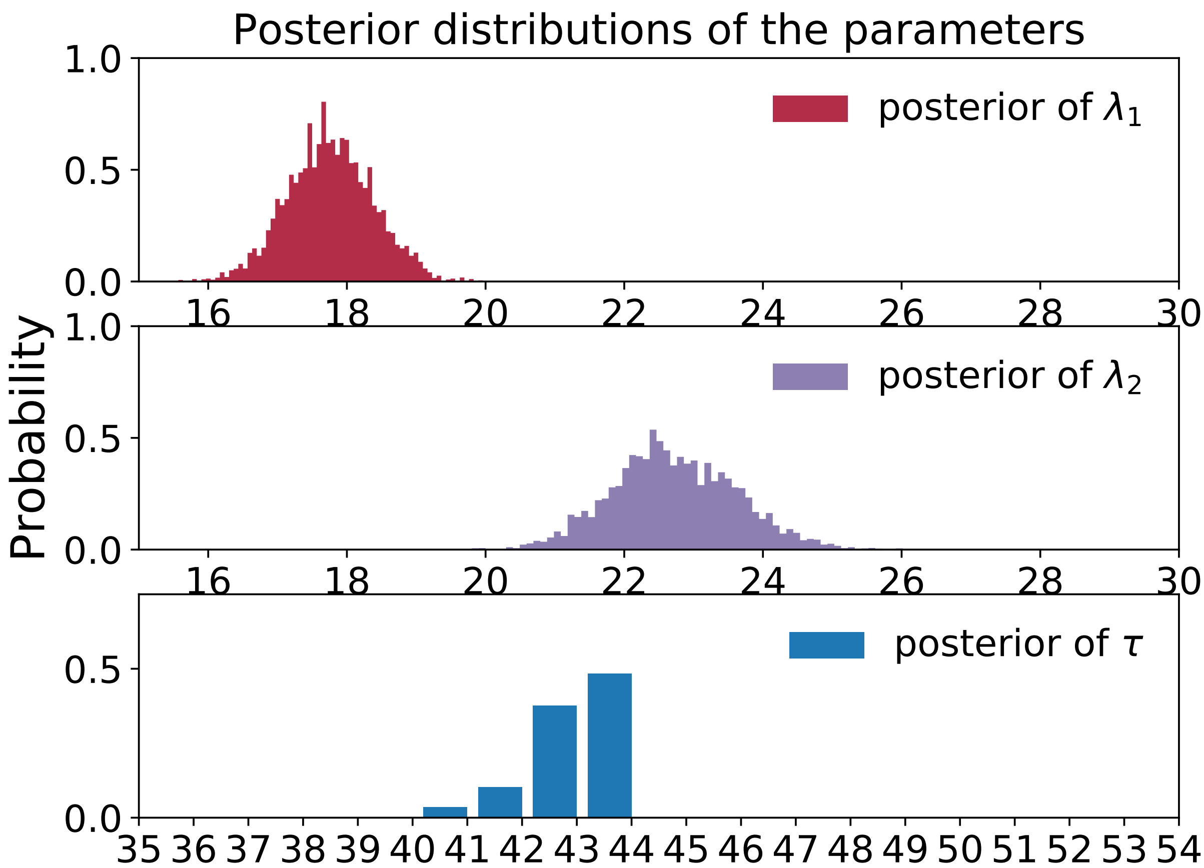
```
1/alpha = 19.743243243243242
```

[27]:



[28]:
```
# pull the sample values out with mcmc.trace
lambda_samples = trace.get_values('lam')
lambda_1_samples = lambda_samples[:,0]
lambda_2_samples = lambda_samples[:,1]
tau_samples = trace.get_values('tau')
```

[29]:
```
# Plot the posterior distributions

fig = plt.figure(figsize=(8,6))
ax = plt.subplot(311)
ax.set_autoscaley_on(False)

plt.hist(lambda_1_samples, histtype='stepfilled', bins='auto', alpha=0.85,
         label="posterior of $\lambda_1$", color="#A60628", normed=True)
plt.legend(loc="upper right", frameon=False);
plt.title(r"Posterior distributions of the parameters", fontsize=18);
plt.xlim([15, 30])
plt.xlabel("$\lambda_1$ value"); # actually behind the next subplot..

ax = plt.subplot(312)
ax.set_autoscaley_on(False)
plt.hist(lambda_2_samples, histtype='stepfilled', bins='auto', alpha=0.85,
         label="posterior of $\lambda_2$", color="#7A68A6", normed=True)
plt.legend(loc="upper right", frameon=False);
plt.xlim([15, 30])
plt.xlabel("$\lambda_2$ value", fontsize=14);
plt.ylabel("Probability");


plt.subplot(313)
import collections
taus,Ntaus = zip(*sorted( collections.Counter(tau_samples).items() ))
taus = [t-0.4 for t in taus] # shift bar locations
Ptaus = [1.0*n/sum(Ntaus) for n in Ntaus]
plt.bar(taus,Ptaus, color='C0', label=r"posterior of $\tau$");
plt.xticks(np.arange(T));

plt.legend(loc="upper right", frameon=False);
plt.ylim([0, .75]);
plt.xlim([35, len(C) - 20]);
#plt.xlabel(r"$\tau$ (in days)");

plt.savefig("text_msgs_posterior.png");
```



So we are now experts on PyMC?

- Maybe not, but we may know just enough to **be dangerous**

Logistic regression

Recall:

Logistic regression models a binary y variable ($y = 0$ or $y = 1$) by assuming the probability that $y = 1$ is a sigmoid of a linear function of x:

$$P(y = 1 \mid x) = \frac{1}{1 + \exp\big(-(\alpha + \beta x)\big)}$$

The parameters α and β control where the sigmoid changes from 0 to 1 and how steep the change is.

Logistic regression generalizes to multiple x variables as well:

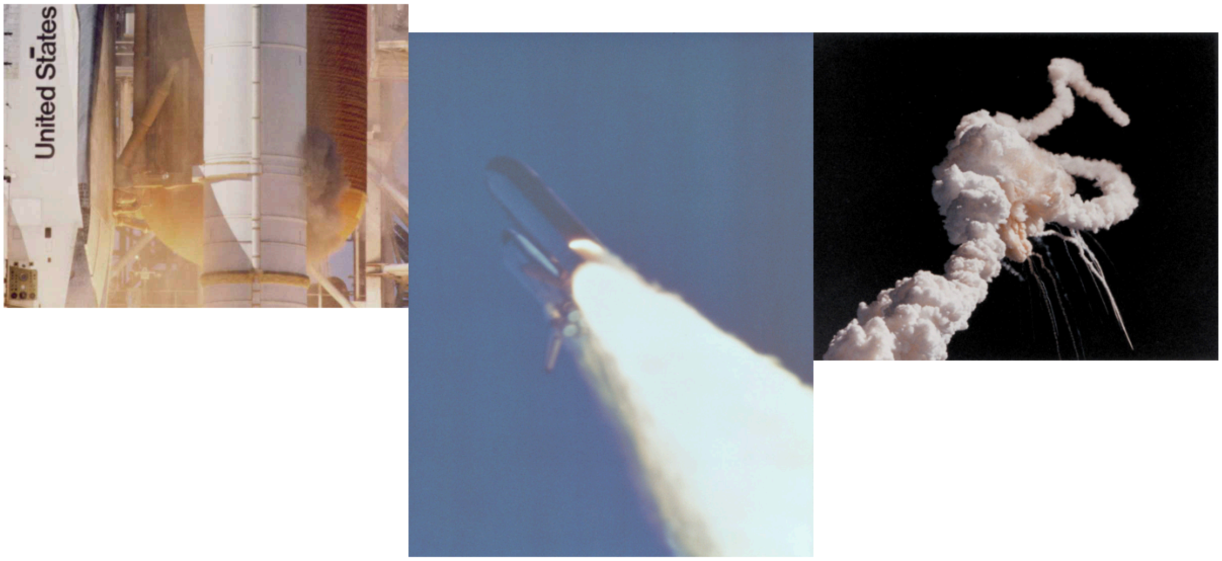
$$P(y = 1 \mid x_1, x_2, \ldots, x_p) = \frac{1}{1 + \exp\big(-(\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p)\big)}$$

Example problem, the **Challenger disaster**:

STS-51-L

[30]:
```
Image('STS-51-L_images.png')
```

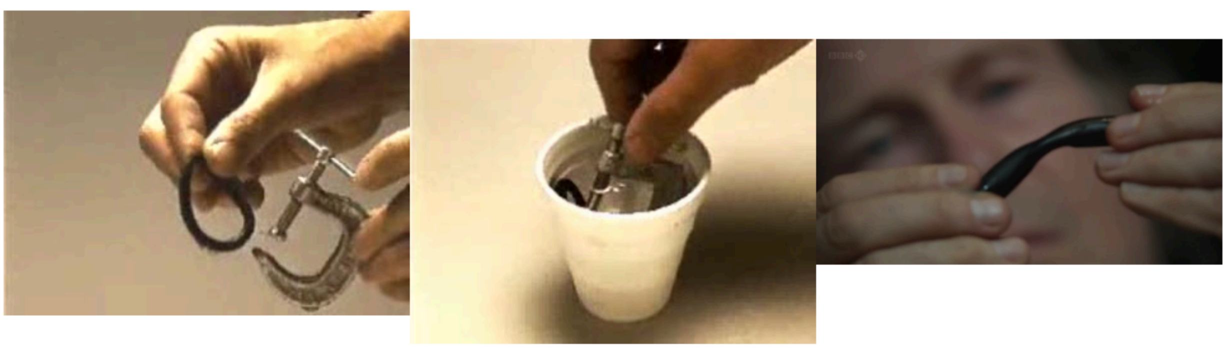
[30]:



The Challenger space shuttle mission was lost (with seven deaths) because a rubber O-ring failed to seal off fuel inside one of the rocket boosters:

[31]:
```
Image("O-ring_images.png", width=600)
```

[31]:

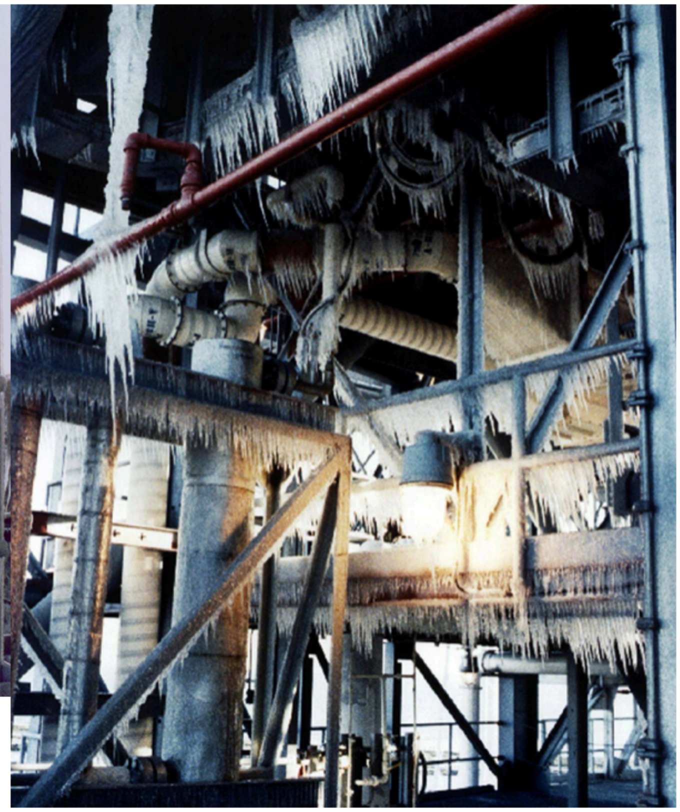


Cold temperatures caused the rubber to lose its springiness

Was it cold on the day that Challenger launched?

[32]:
```
Image("STS-51-L_ice_images.png",width=600)
```

[32]:

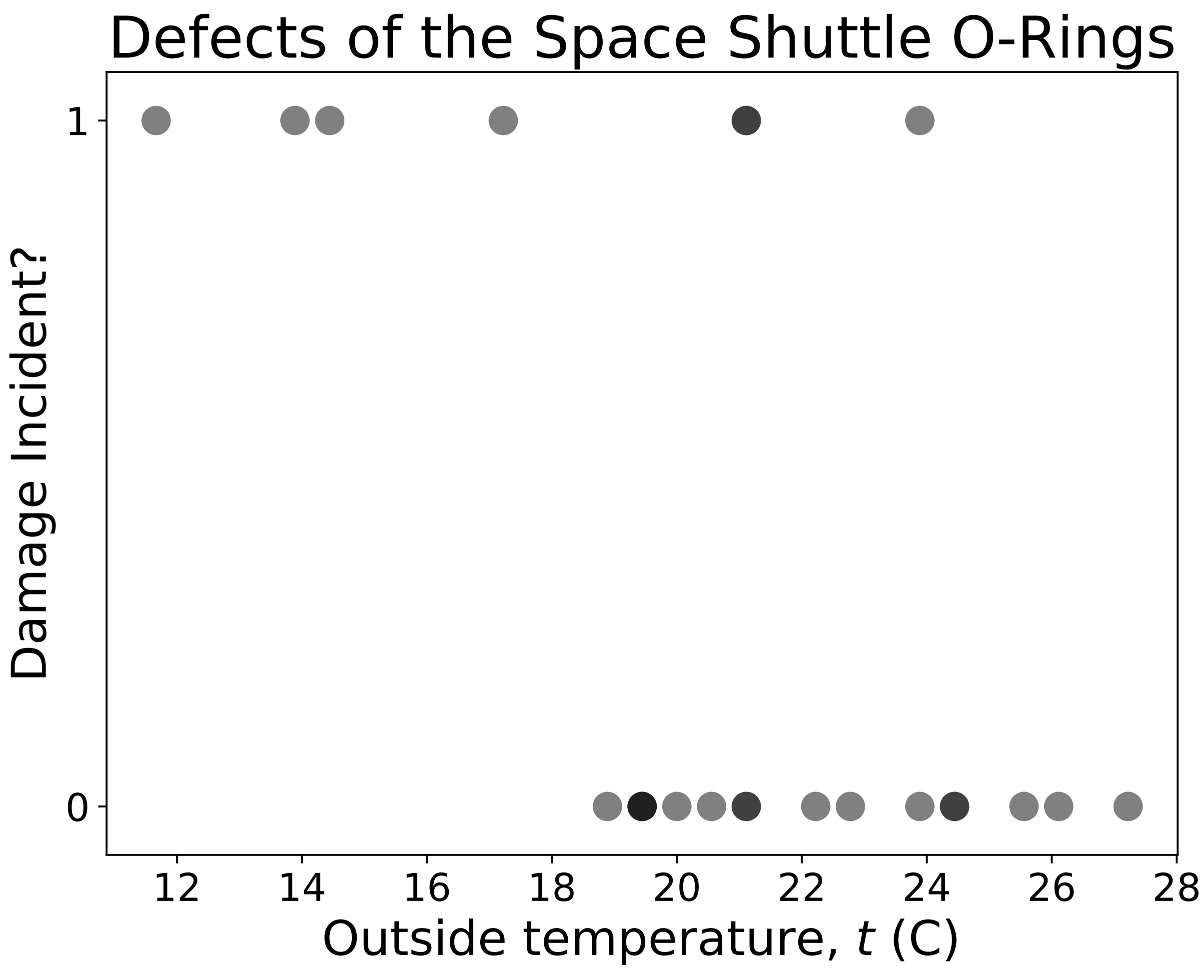
Data

We have **damage reports** for **23 different engine tests** detailing whether or not the *O-ring failed* as a function of temperature on the launch pad:

[33]:
```
challenger_data = np.array([[ 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.]])

challenger_data[:,0] = (challenger_data[:,0] - 32) * 5./9
```

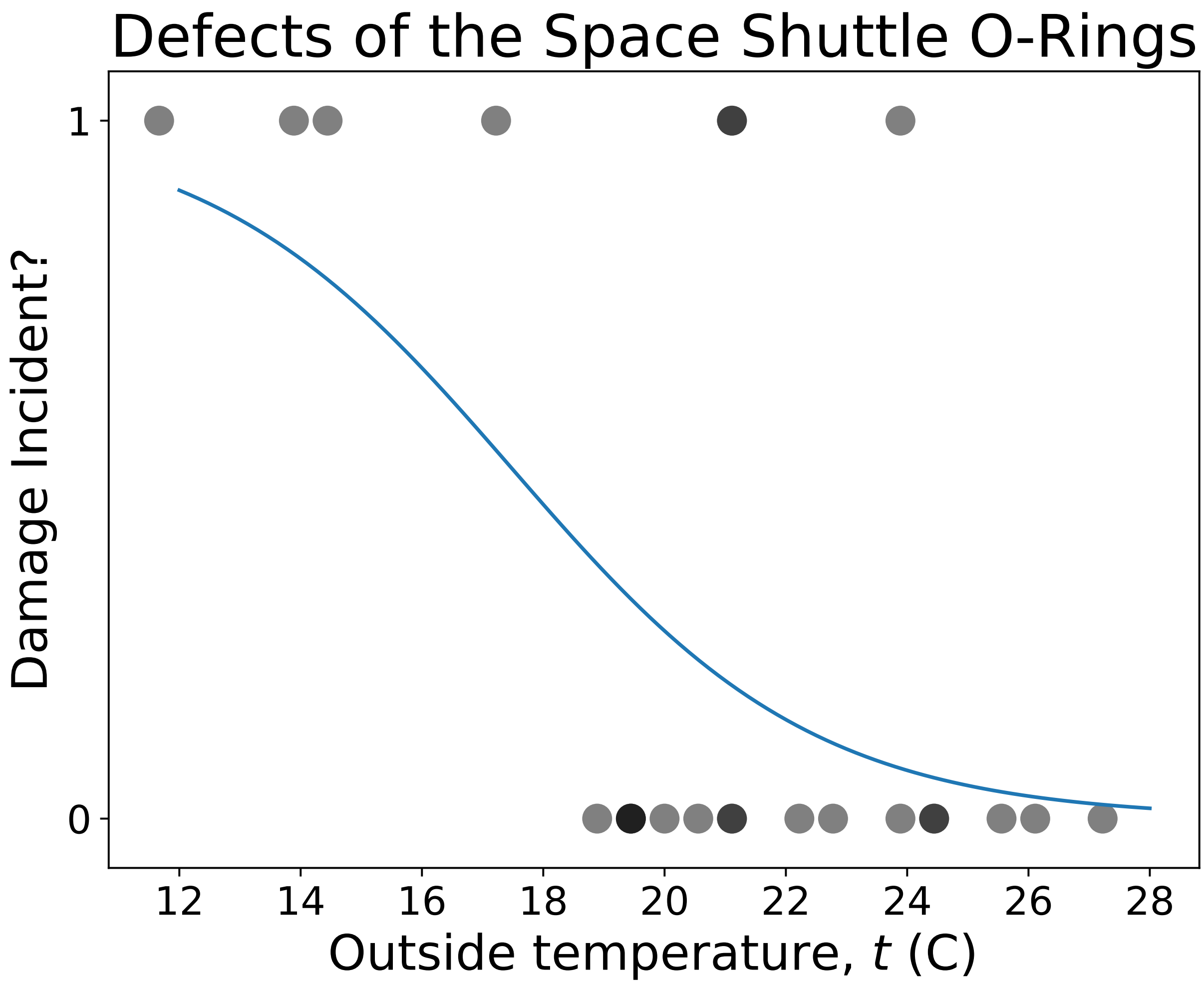
[34]:
```
plt.scatter(challenger_data[:, 0], challenger_data[:, 1], s=150, color="k",
            alpha=0.5, edgecolors='none');
plt.yticks([0, 1])
plt.ylabel("Damage Incident?")
plt.xlabel(r"Outside temperature, $t$ (C)")
plt.title("Defects of the Space Shuttle O-Rings");
```



And here's a sigmoid function overlaid on top, motivating logistic regression:

[35]:
```
alpha = 7 # guessed parameters, not a fit!
beta = -0.4
sigmoid_x = np.linspace(12,28,100)
sigmoid_y = 1 / (1 + np.exp(-(alpha + beta*sigmoid_x)))

plt.scatter(challenger_data[:, 0], challenger_data[:, 1], s=150, color="k",
            alpha=0.5, edgecolors='none');
```

```
plt.plot(sigmoid_x, sigmoid_y)
plt.yticks([0, 1])
plt.ylabel("Damage Incident?")
plt.xlabel(r"Outside temperature, $t$ (C)")
plt.title("Defects of the Space Shuttle O-Rings");
```



The sigmoid function smoothly varies from 0 to 1 (or 1 to 0, in this case). The fit parameters (α and β) let us tune the position and steepness of the transition from 0 to 1.

Historical context:

- The night before Challenger launched, NASA and the aerospace companies had a three-hour teleconference to determine whether it would be too cold to launch.
- At that time, they only had seven data points (the top row) to look at, and they decided it was OK to launch.
- We will see that this was a mistake (hindsight!)

Bayesian logistic regression

Typically, logistic regression parameters are estimated using maximum likelihood techniques (recall from prior lecture), but let's see how to figure out these parameters using a **Bayesian approach**

Here is the PyMC setup:

[36]:
```
# data:
temperature = challenger_data[:, 0]
D = challenger_data[:, 1] # defect or not?


with pm.Model() as model_simple:
    alpha = pm.Normal('alpha', mu=0, sd=1/0.001)
    beta = pm.Normal('beta',   mu=0, sd=1/0.001)

    mu = alpha + pm.math.dot(temperature, beta)
    theta = pm.Deterministic('theta', pm.math.sigmoid(mu))

    dec_bnd = pm.Deterministic('dec_bnd', -alpha/beta) # decision boundary

    y_1 = pm.Bernoulli('y_1', p=theta, observed=D)

    trace = pm.sample(1000, tune=1000)
```

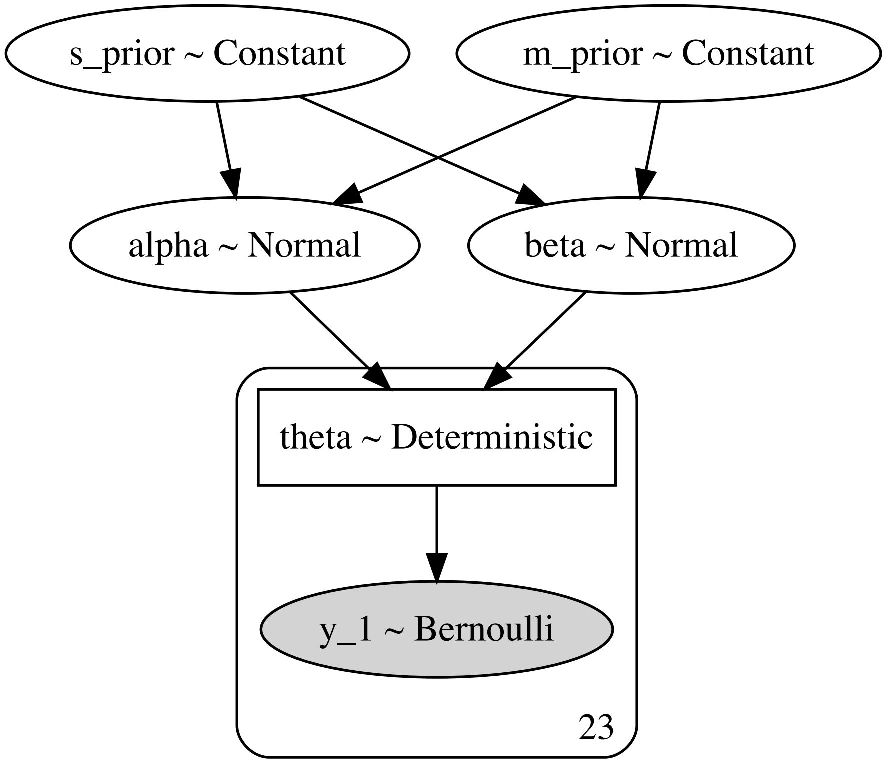
```
Auto-assigning NUTS sampler…
Initializing NUTS using jitter+adapt_diag…
Multiprocess sampling (4 chains in 4 jobs)
NUTS: [beta, alpha]
Sampling 4 chains, 0 divergences: 100%|      | 8000/8000 [00:05<00:00,
1478.98draws/s]
The acceptance probability does not match the target. It is 0.7119541719568686, but
should be close to 0.8. Try to increase the number of tuning steps.
The acceptance probability does not match the target. It is 0.6645167126663026, but
should be close to 0.8. Try to increase the number of tuning steps.
The number of effective samples is smaller than 10% for some parameters.
```

We've got the trace, but before we investigate it, let's look at the model:

[37]:
```
mod_viz = pm.Model()
with mod_viz:
    m_pr = pm.Constant("m_prior",0)
    s_pr = pm.Constant("s_prior",1/0.001)

    alpha = pm.Normal('alpha', mu=m_pr, sd=s_pr)
    beta  = pm.Normal('beta',  mu=m_pr, sd=s_pr)

    mu = alpha + pm.math.dot(temperature, beta)
    theta = pm.Deterministic('theta', pm.math.sigmoid(mu))

    y_1 = pm.Bernoulli('y_1', p=theta, observed=D)
```

```
pm.model_to_graphviz(mod_viz)
```

[37]:



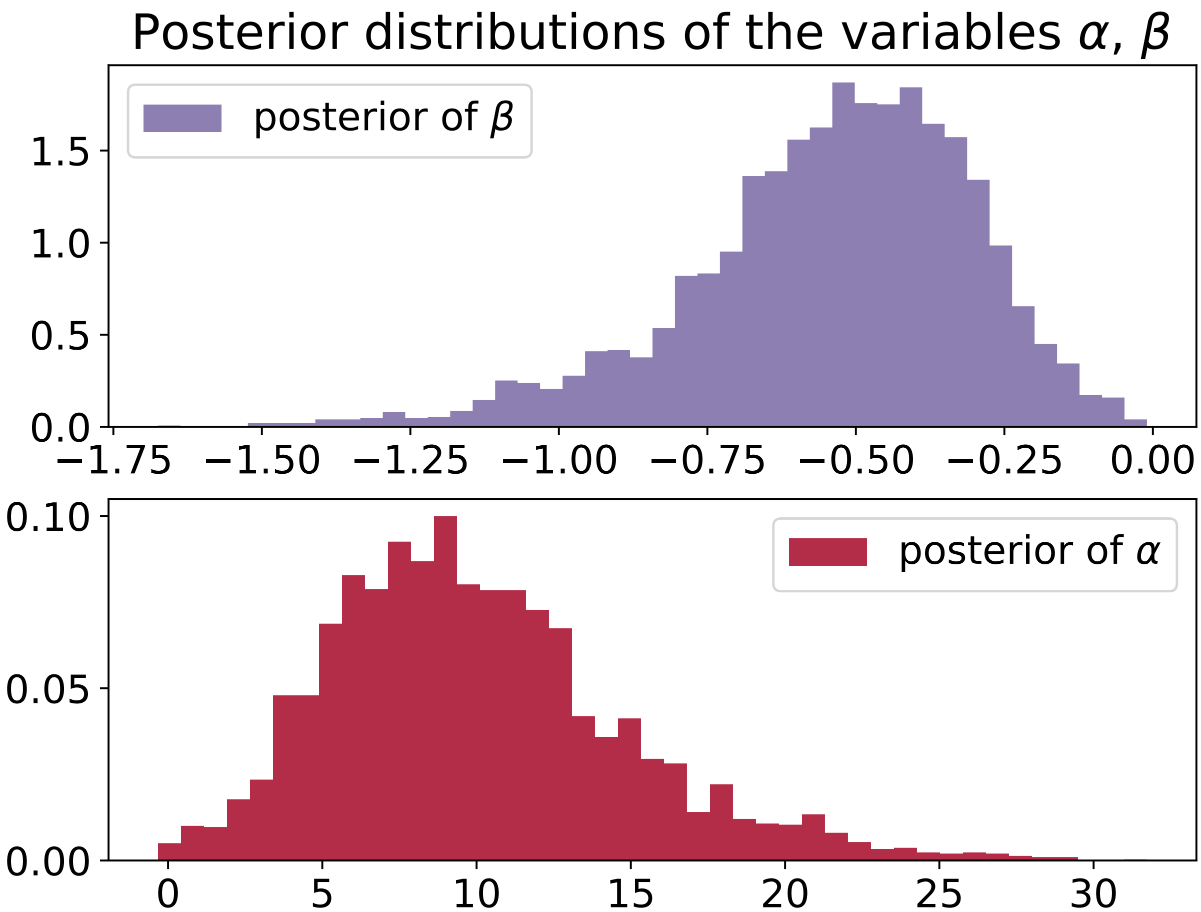
Now extract the trace and plot the posteriors:

[38]:
```
alpha_samples = trace.get_values('alpha')
beta_samples  = trace.get_values('beta')

print(len(alpha_samples), alpha_samples.shape)
print(len(beta_samples), beta_samples.shape)
```

```
4000 (4000,)
4000 (4000,)
```

[39]:
```
# histogram of the samples:
fig = plt.figure(figsize=(8,6))

plt.subplot(211)
plt.title(r"Posterior distributions of the variables $\alpha$, $\beta$", fontsize=20)
plt.hist(beta_samples, bins='auto',
         histtype='stepfilled', alpha=0.85,
         label=r"posterior of $\beta$", color="#7A68A6", normed=True)
plt.legend(loc="upper left")

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

```
plt.legend();
```



Remember we need to interpret α and β in terms of the logistic function:

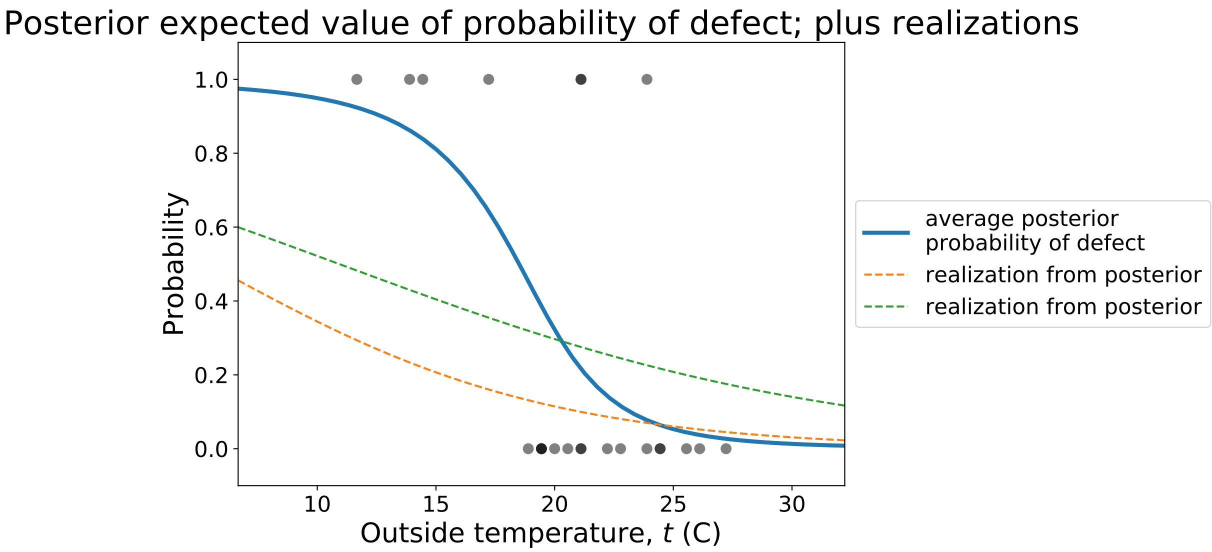
$$P(y = 1 \mid t) = \frac{1}{1 + \exp\big(-(\alpha + \beta t)\big)}$$

Let's take all our samples and plot the logistic curves:

[40]:
```
def logistic(x, alpha, beta):
    return 1.0 / ( 1.0 + np.exp(-(alpha + np.dot(beta, x))) )

# note the np.dot, we're doing many curves at once using some matrix operations
```

[41]:
```
t = np.linspace(temperature.min() - 5, temperature.max()+5, 50)[:, None]

# all logistic curves in a big matrix:
p_t = logistic(t.T, alpha_samples[:, None], beta_samples[:, None])

# the average probability at each time:
mean_prob_t = p_t.mean(axis=0)

plt.plot(t, mean_prob_t, lw=3,   label="average posterior\nprobability of defect")
```

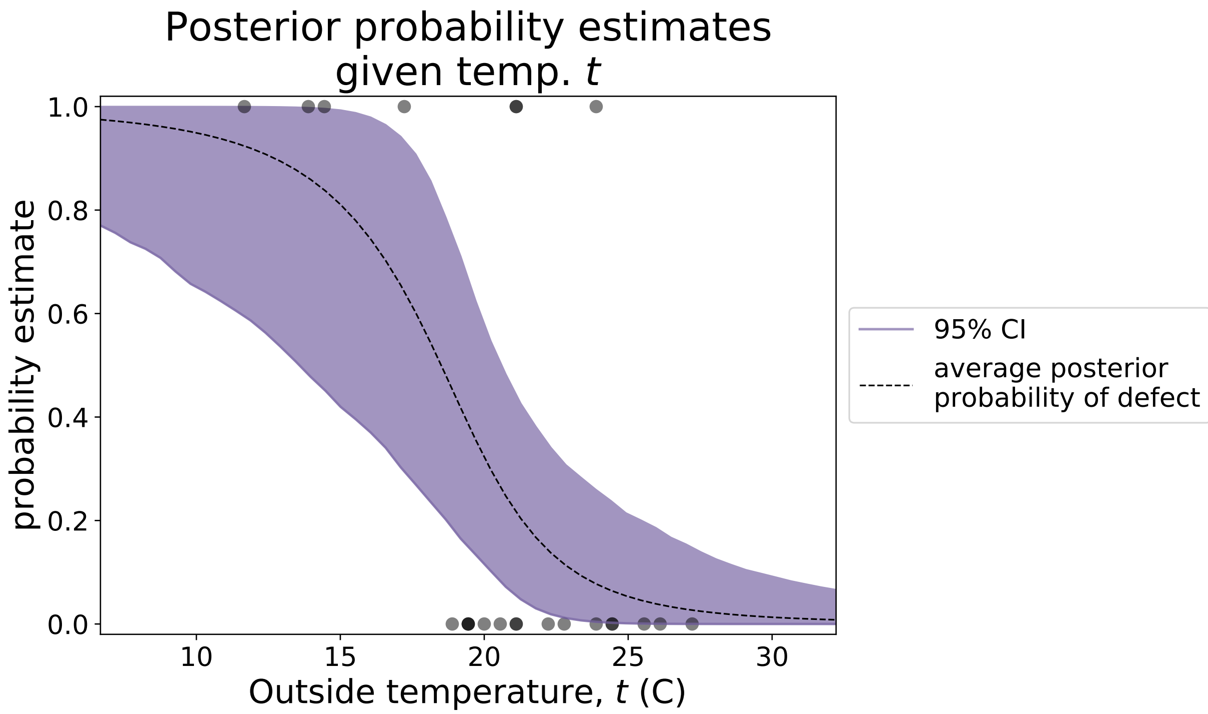
```
plt.plot(t, p_t[0, :],  ls="--", label="realization from posterior")
plt.plot(t, p_t[9, :],  ls="--", label="realization from posterior")
plt.scatter(temperature, D, color="k", s=50, alpha=0.5)
plt.title("Posterior expected value of probability of defect; \
plus realizations")
plt.legend(loc='center left', bbox_to_anchor=(1, 0.5))
plt.ylim(-0.1, 1.1)
plt.xlim(t.min(), t.max())
plt.ylabel("Probability")
plt.xlabel(r"Outside temperature, $t$ (C)");
```



Since we have so many different curves, let's plot the average and a 95% **confidence interval**:

[42]:
```
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")
# this is sometimes known as an "equal tail interval" and may not be
# the best choice (see "highest posterior density" for more)

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='center left', bbox_to_anchor=(1, 0.5))
plt.scatter(temperature, D, color="k", s=50, alpha=0.5)
plt.xlabel(r"Outside temperature, $t$ (C)")

plt.ylabel("probability estimate")
```

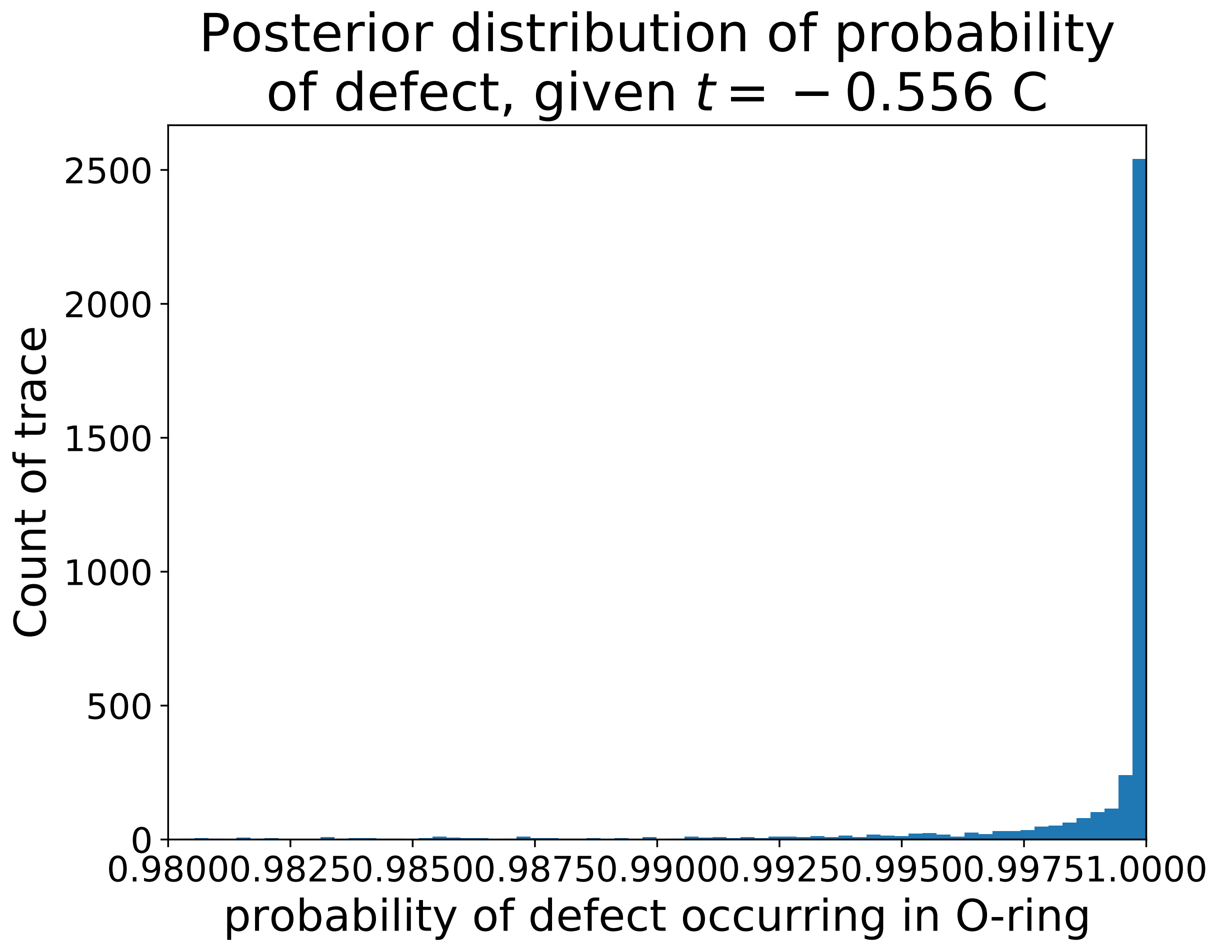
```
plt.title("Posterior probability estimates\ngiven temp. $t$");
```



So we see that the probability for a defect is pretty close to 1 for temperatures around 50 F (10 C). What about the day of the shuttle launch, which was 31 F?

We can plug a single value in for t and look at how p is distributed over our posterior sample of α and β:

[43]:
```
temp_launch = (31-32)*5./9
prob_at_31 = logistic(temp_launch, alpha_samples, beta_samples)

plt.hist(prob_at_31, bins=2000, ec='none')#, histtype='stepfilled')
plt.xlim(0.98, 1)
plt.title("Posterior distribution of probability\nof defect, given $t = %0.3f$ C" %␣
 ↪temp_launch)
plt.xlabel("probability of defect occurring in O-ring")
plt.ylabel("Count of trace"); # number of trace samples
```



According to this result, it was a bad idea to launch.

Lastly, how does Bayesian regression compare to traditional logistic regression?

- Traditional logistic regression finds the individual parameters that make the **likelihood** as large as possible. This is called Maximum Likelihood Estimation, and is done (in this case) using a numerical optimization scheme. Doing this is called a **point estimate** because it gives a *single value* for each parameter, in contrast to sampling *distributions* of parameters from the posterior.

[44]:
```
import statsmodels.api as sm

df = pd.DataFrame(challenger_data, columns=["temp","failure"])
df["constant"] = 1.0
print(df.head())
print()

logit = sm.Logit(df['failure'], df[['constant','temp']])

# fit the model
result = logit.fit()
```

```python
print(result.summary2())
print()
# odds ratios and 95% CI
params = result.params
conf = result.conf_int()
conf['OR'] = params
conf.columns = ['2.5%', '97.5%', 'OR']
print(np.exp(conf))
```

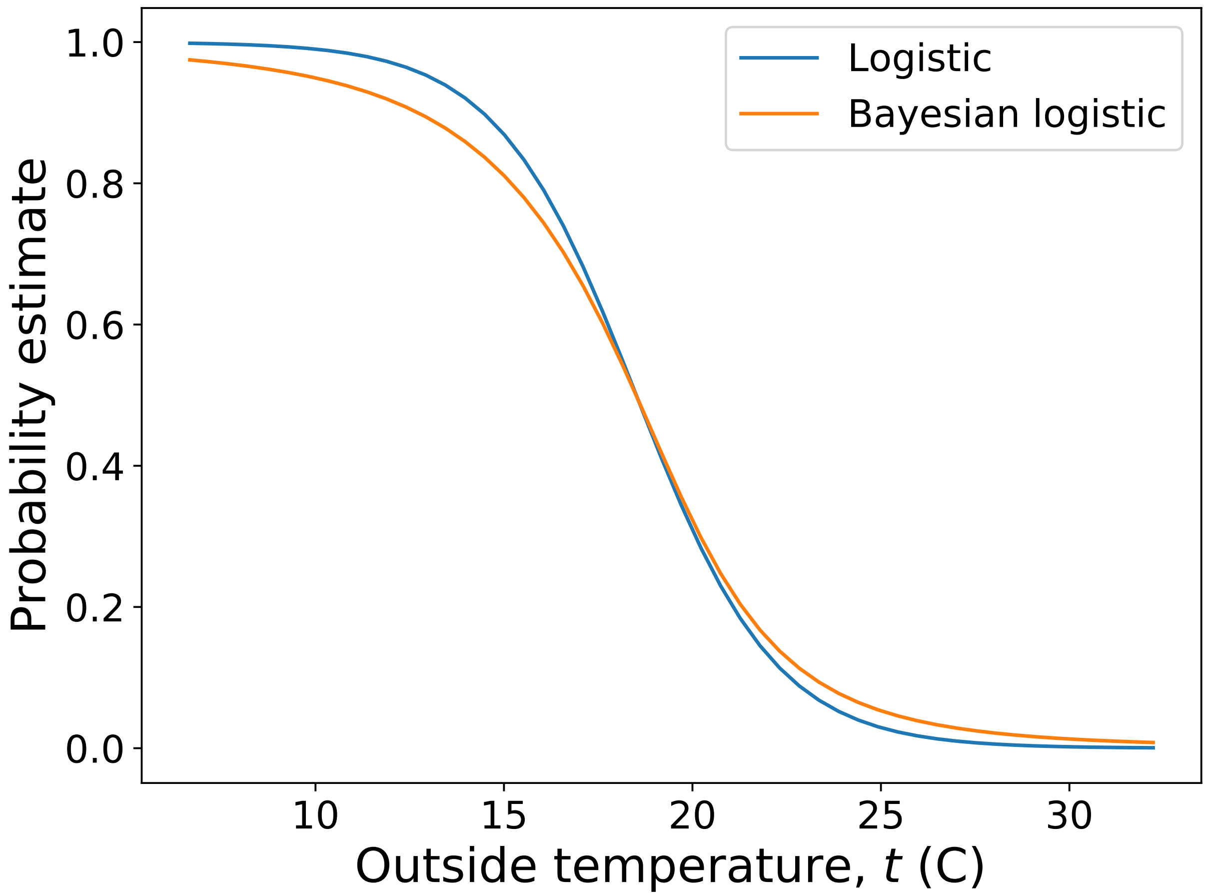
```
        temp  failure  constant
0  18.888889      0.0       1.0
1  21.111111      1.0       1.0
2  20.555556      0.0       1.0
3  20.000000      0.0       1.0
4  19.444444      0.0       1.0

Optimization terminated successfully.
         Current function value: 0.441635
         Iterations 7
                         Results: Logit
================================================================
Model:              Logit            Pseudo R-squared: 0.281
Dependent Variable: failure          AIC:              24.3152
Date:               2020-04-06 20:12 BIC:              26.5862
No. Observations:   23               Log-Likelihood:   -10.158
Df Model:           1                LL-Null:          -14.134
Df Residuals:       21               LLR p-value:      0.0048035
Converged:          1.0000           Scale:            1.0000
No. Iterations:     7.0000
-----------------------------------------------------------------
              Coef.  Std.Err.    z     P>|z|   [0.025   0.975]
-----------------------------------------------------------------
constant      7.6137   3.9334  1.9356  0.0529  -0.0957  15.3231
temp         -0.4179   0.1948 -2.1450  0.0320  -0.7997  -0.0360
================================================================


              2.5%         97.5%           OR
constant  0.908758  4.515668e+06  2025.747065
temp      0.449444  9.646003e-01     0.658433
```

[45]:

```python
# Now compare the average of the posterior samples to the
# above coefficients of the logistic regression fit:
print(params)
print(beta_samples.mean(), alpha_samples.mean())
```

```
constant    7.613694
temp       -0.417893
dtype: float64
-0.5401711300931675 9.998553370566441
```

[46]:
```python
prob_t_logistic = logistic(t.T, alpha_samples.mean(), beta_samples.mean() )

plt.plot(t, prob_t_logistic.T,  label="Logistic")
plt.plot(t, mean_prob_t,        label="Bayesian logistic")

plt.xlabel(r"Outside temperature, $t$ (C)")
plt.ylabel("Probability estimate")
plt.legend();
```



Actually pretty close! The Bayesian is a little more conservative in that it has a slower transition from high to low probability

Summary

So now we know how to do Bayes!

- Not fully discussed, but needed in practice - how to diagnose if the MCMC sample is converging (in distribution) and what to do about it if it isn't.
- How do we know that MCMC is sampling from the correct posterior?