Module 10: Markov Chain Monte Carlo (MCMC) - Part 2

Introduction to PyMC3

In this section we demonstrate how the Python PyMC3 library can be applied for Bayesian inference analysis and for the estimation of model parameters using the Monte Carlo Markov Chain technique.

References:

- https://github.com/markdregan/Bayesian-Modelling-in-Python (https://github.com/markdregan/Bayesian-Modelling-in-Python)
- https://github.com/CamDavidsonPilon/Probabilistic-Programming-and-Bayesian-Methods-for-Hackers/ (https://github.com/CamDavidsonPilon/Probabilistic-Programming-and-Bayesian-Methods-for-Hackers/)
- https://docs.pymc.io (https://docs.pymc.io)

PyMC3 installation

- via pypi: `pip install pymc3`
- via conda-forge: `conda install -c conda-forge pymc3`
- the latest (may be unstable) version can be downloaded from github.com:
  `pip install git+https://github.com/pymc-devs/pymc3`

In [1]:
```
from IPython.display import Image

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pymc3 as pm
import seaborn as sns

%matplotlib inline
plt.style.use('bmh')
colors = ['#348ABD', '#A60628', '#7A68A6', '#467821', '#D55E00',
          '#CC79A7', '#56B4E9', '#009E73', '#F0E442', '#0072B2']

messages = pd.read_csv('./module10/hangout_chat_data.csv')
```

Bayesian method of estimating model parameters

In this example, a Bayesian approach is used to estimate model parameters. This dataset is taken from Mark Regan's chat dataset; it shows the time taken to respond to messages.

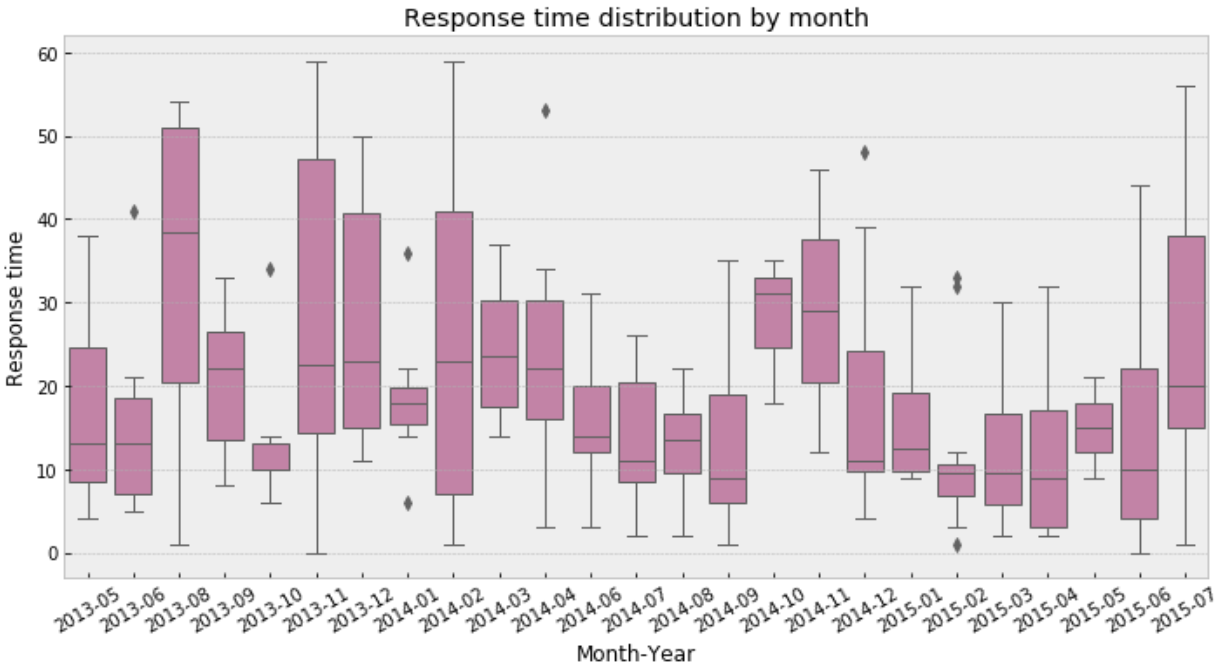
In the Bayesian approach the data is considered to be generated by a random process.

Because the response time is count data, it is natural to model it as a Poisson distribution. The goal here is to estimate the parameter of the distribution λ and represent the uncertainty in this parameter.

In other words we need to find the posterior distribution for the parameter λ using the Bayesian formula:

$$\overbrace{p(\lambda \mid Data)}^{\text{posterior}} = \frac{\overbrace{p(Data \mid \lambda)}^{\text{likelihood}} \cdot \overbrace{p(\lambda)}^{\text{prior}}}{\underbrace{p(Data)}_{\text{marginal likelihood}}}$$

In [2]:
```
he observed data distribution
re(figsize=(10,10))
ubplot(211)

t(messages['year_month'].unique())
messages['year_month'], y=messages['time_delay_seconds'], order=order, orien
le('Response time distribution by month')
bel('Month-Year')
bel('Response time')
(rotation=30)

ut()
```

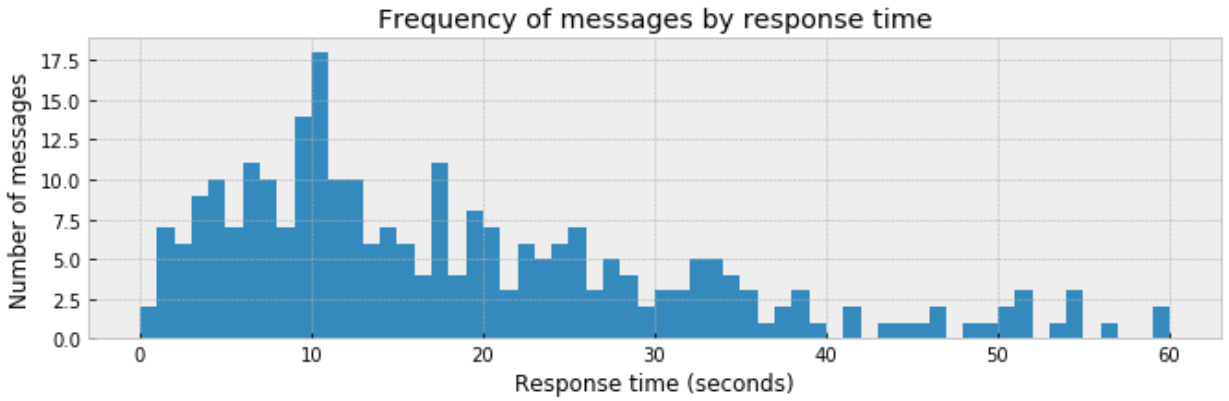


In [3]:
```
messages.describe()
```

Out[3]:

	message_length	num_participants	time_delay_seconds	time_delay_mins	day_of_week	is_we
count	266.000000	266.000000	266.000000	266.000000	266.000000	266.(
mean	31.500000	2.139098	18.041353	0.992481	2.357143	0.(
std	28.502003	0.498501	13.430821	0.086547	1.530904	0.2
min	1.000000	2.000000	0.000000	0.000000	0.000000	0.(
25%	10.000000	2.000000	8.000000	1.000000	1.000000	0.(
50%	23.000000	2.000000	14.000000	1.000000	2.000000	0.(
75%	43.750000	2.000000	25.000000	1.000000	3.000000	0.(
max	145.000000	4.000000	59.000000	1.000000	6.000000	1.(

In [4]:
```
fig = plt.figure(figsize=(11,3))
_ = plt.title('Frequency of messages by response time')
_ = plt.xlabel('Response time (seconds)')
_ = plt.ylabel('Number of messages')
_ = plt.hist(messages['time_delay_seconds'].values,
             range=[0, 60], bins=60, histtype='stepfilled')
```



To run this analysis, we need to choose the likelihood and the prior distribution.

- Observed data: counts of response time for each conversation;
- Likelihood: These data were generated by a random process which can be represented as a Poisson distribution;
- For the prior, the distribution of the Poisson distribution λ has to be chosen. From the observed data, we know that this parameter is between 0 and 60. We can start to model λ as a uniform distribution because we do not have an opinion as to where within this range to expect it.

To derive the posterior distribution, the MCMC sampler draws parameter λ values from the prior distribution and computes the likelihood that the observed data came from a distribution with these parameter values.

$$\overbrace{p(\lambda \mid Data)}^{posterior} \propto \overbrace{p(Data \mid \lambda)}^{likelihood} \cdot \overbrace{p(\lambda)}^{prior}$$

As the MCMC sampler draws values from the parameter priors, it computes the likelihood of these parameters given the data - and the sampling method (for example, Metropolis) will try to guide the sampler towards parameter values of higher probability.

In [5]:
```
with pm.Model() as model:
    lambda_ = pm.Uniform('lambda', lower=0, upper=60)
    likelihood = pm.Poisson('likelihood', mu=lambda_, observed=messages['ti

    start = pm.find_MAP()
    step = pm.Metropolis()
    trace = pm.sample(100000, step, start=start, progressbar=True)
```

```
logp = -2,608.5, ||grad|| = 1,590.5: 100%|██████████| 7/7 [00:00<00:00, 2
126.47it/s]
Multiprocess sampling (4 chains in 4 jobs)
Metropolis: [lambda]
Sampling 4 chains, 0 divergences: 100%|██████████| 402000/402000 [00:43<0
0:00, 9285.03draws/s]
The number of effective samples is smaller than 10% for some parameters.
```

In the above code, our model is implemented with the uniform distribution for the prior distribution, and the Poisson distribution for the likelihood.

The PyMC3 method `find_MAP()` was used to find the most likely value of λ for the starting point. This is optional, but often speeds up the computation.

The step specifies what sampling method is used for guidance towards the most likely parameter values.

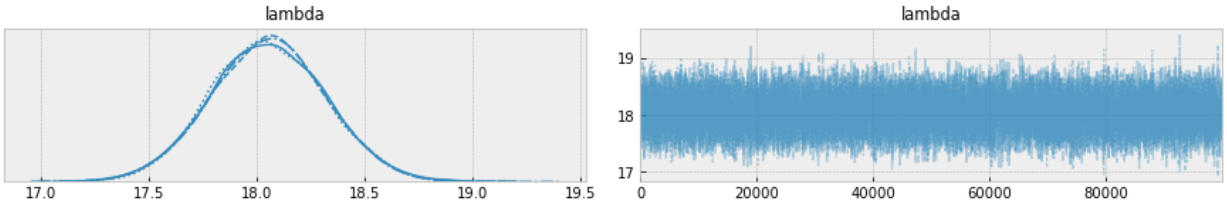
And finally, the PyMC3 function `sample()` gathers the samples (100,000 samples in this case) by traversing over the areas of the most likely parameter values. This is the main entry point to the MCMC sampling algorithms. This function takes a defined sampler passed as step, or if step is not passed will try to auto-assign the correct sampler and auto-initialize it.

The PyMC3 library has methods for graphical representation of the collected samples; below, the `traceplot` method is used to show the determined posterior distribution of λ.

The below plot (left) shows the posterior distribution of values collected for λ. In this case, we obtained a bell-shaped distribution for the parameter even though the prior is the uniform distribution. The mean is slightly over 18.0 sec and the obtained distribution provides a measure of uncertainty: credible values of λ are between 17.0 and 19.0 seconds.

This result shows the difference between the frequentist approach, where the parameter λ is estimated as the mean value of 'time_delay_seconds' and is given as a single value. In contrast, from Bayesian analysis, we get the parameter value with a measure of uncertainty. This measure of uncertainty is incredibly valuable, as we will see later. Note that the mean of posterior distribution is very close to the frequentist estimate.

In [6]:
```
lambda_trace = pm.traceplot(trace)
```



In [7]:
```
pm.summary(trace)
```

Out[7]:

	mean	sd	hpd_3%	hpd_97%	mcse_mean	mcse_sd	ess_mean	ess_sd	ess_bulk	es
lambda	18.045	0.261	17.545	18.527	0.001	0.001	31318.0	31318.0	31356.0	33

Model convergence

The *trace plots* show the history of a parameter value across iterations of the chain. It shows the value of parameters explored on each iteration step. If the model reached equilibrium there should not be any trends observed. The trace should look random around some constant value, should be jumping around and look like a "hairy caterpillar". Any trends (periodicity, upward or downward trend) indicate that the chain has not converged to stationary equilibrium state yet and the number of iteration has to be increased.

Also, it is recommended to inspect sample autocorrelation - the measure of correlation between successive samples in the chain. An autocorrelation plot should taper off to zero relatively quickly, and then oscillate around zero. If your autocorrelation plot does not taper off, the model selection (likelihood) and sampling methods should be revisited.

The *effective sample size* (ESS) shows the number of points that bring useful information. The effective sample size is usually given in the summary of MCMC simulation output. The ESS is related to autocorrelation. Since autocorrealtion measures the linear dependence of the current chain value on the past value (lags), it helps to estimate how much information is available. Due to autocorrelation Markov chain sampling 1000 iteration would contain less information about the distribution thag 1000 samples independently drawn from that distribution. Autocorrelation guides how iterations should be thinned out untile autocorrelation is close to 0

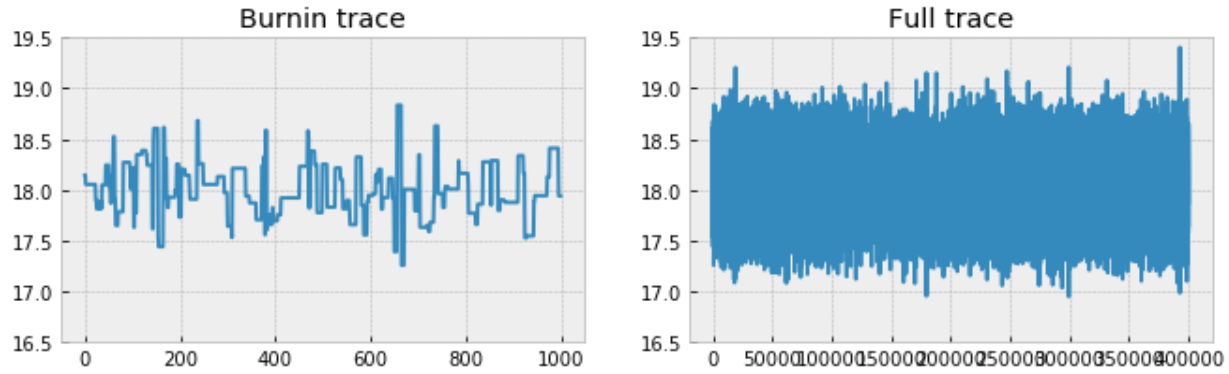
Burnin samples

In the above code the function `pm.find_MAP()` was used to find a good starting point for sampling. This function finds a maximum a posteriori (MAP) estimation and helps to start sampling in the area of high likelihood. But sometimes the wrong maximum can be detected, and it is recommended to discard the samples collected at the early stage (burnin samples) as they might be biased by the choice of starting point. The initial point of the chain does not affect the posterior distribution. For any starting point we expect the chain to find the bulk of posterior distribution and get the stationary distribution. However, if chain started far from the significant regions it would take longer to find stationary distribution. From trace plot inspection, especially the initial period, it is

possible to detect the time chain took to converge. The initial period is called *burn-in period* and should always be discarded form the posterior distributio. Sometimes, up to 90% of trace is discarded.

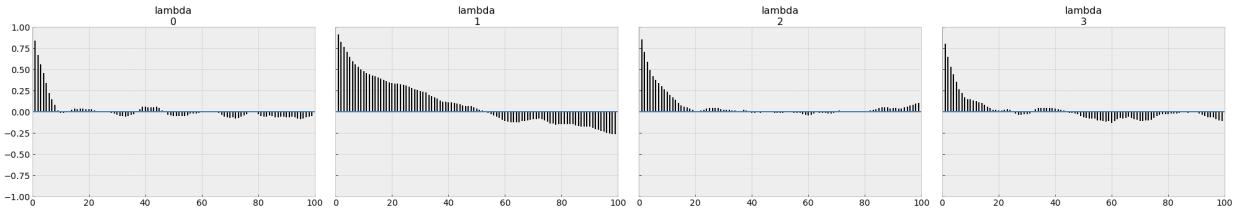
The standard practice is to simulate several chains at once. Each starting at diffeent initial value. The diagnostic method (Gelman-Rubin) was developed which calculates variability within chains, comapring that to variability between chains.

In [8]:
```
fig = plt.figure(figsize=(11,3))
plt.subplot(121)
_ = plt.title('Burnin trace')
_ = plt.ylim(ymin=16.5, ymax=19.5)
_ = plt.plot(trace.get_values('lambda')[:1000])
fig = plt.subplot(122)
_ = plt.title('Full trace')
_ = plt.ylim(ymin=16.5, ymax=19.5)
_ = plt.plot(trace.get_values('lambda'))
```



In [9]:
```
_ = pm.autocorrplot(trace[:1000], varnames=['lambda'])
```

```
/Users/sergiynokhrin/opt/anaconda3/lib/python3.7/site-packages/pymc3/plots/__init__.py:21: UserWarning: Keyword argument `varnames` renamed to `var_names`, and will be removed in pymc3 3.8
  warnings.warn('Keyword argument `{old}` renamed to `{new}`, and will be removed in pymc3 3.8'.format(old=old, new=new))
```



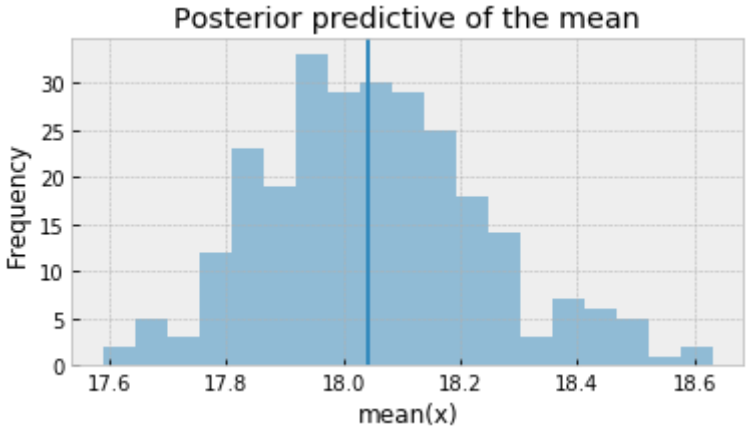
As a validation check of model performance, we shall generate the distribution of the response time.
The postrior predictive check is one of the ways to validate a model, this generate the data from the model using parameter drawn from the posterior distribution.
While `plot_posterior` function plots the posterior distributions of the model parameters, the `sample_posterior_predictive(trace, samples= N , model= ... )` will randomly draw N samples from the trace, which contains the posterior distribution. This will produce N sets of the model parameters; then for each sample of the parameters it will draw (or 'simulate') the data distribution.

This method will produce N 'simulated' datasets; these might be inspected individually or aggregate them into one distribution, for example by finding average of N distributions.
Note that the function returns a dictionary {'likelihood': array of N distributions [...], [...], ... [...]}

In [10]:
```
ppc_hangout = pm.sample_posterior_predictive(trace, samples=500, model=mode
```

```
/Users/sergiynokhrin/opt/anaconda3/lib/python3.7/site-packages/pymc3/samp
ling.py:1247: UserWarning: samples parameter is smaller than nchains time
s ndraws, some draws and/or chains may not be represented in the returned
posterior predictive sample
  "samples parameter is smaller than nchains times ndraws, some draws "
100%|██████████| 500/500 [00:00<00:00, 2258.80it/s]
```

In [11]:
```
# evaluating distribution of posterior mean parameter vs true sample mean
_, ax = plt.subplots(figsize=(6, 3))
ax.hist(ppc_hangout['likelihood'].mean(axis=0), bins=19,alpha=0.5)
ax.axvline(messages['time_delay_seconds'].values.mean())
ax.set(title='Posterior predictive of the mean', xlabel='mean(x)', ylabel='
```



A/B test

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.

Here, we shall use fake data about sales from site A and B

In [12]:
```
import scipy.stats as stats

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

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

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

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

Now, pretend that purchase rate from site A and site B unknown. All we get is the arrays of data:

- site A had 1500 visits and some number of purchase (1 for yes, 0 for no);
- site B saw 750 visitors and some of them made purchase.

For frequentist this is a problem about two proportions: proportion_A and proportion_B, finding them from data. To answer is the difference is statistically significant, we will find std. error, **assume** that sampling distribution is Normal and will report that with 95% confidence the purchase rate for site A is proportion_A +/- 1.96 * Std. Error and for site B proportion_B +/- 1.96 * Std. Error.

In [13]:
```
proportion_A = np.sum(observations_A)/1500
proportion_B = np.sum(observations_B)/750

print("Proportion for site A is %.3f" % proportion_A)
print("Proportion for site B is %.3f" % proportion_B)
```

```
Proportion for site A is 0.046
Proportion for site B is 0.043
```

For standard error we may use formula:

$$\sigma_{\widehat{p_A}-\widehat{p_B}} = SE_{\widehat{p_A}-\widehat{p_B}} = \sqrt{SE^2_{\widehat{p_A}} + SE^2_{\widehat{p_B}}} = \sqrt{\frac{p_A(1-p_A)}{n_A} + \frac{p_B(1-p_B)}{n_B}}$$

And then assuming the Normal distribution, we may run hypothesis testing:

- the null hypothesis - there no difference between site A and site B: p_A - p_B = 0
- alternative hypothesis p_A and p_B are different.

Next, we calculate z-score for the difference of two proportions with the std.error found above:

$$z = \frac{(\widehat{p_A} - \widehat{p_B}) - (0)}{\sqrt{\frac{p_A(1-p_A)}{n_A} + \frac{p_B(1-p_B)}{n_B}}}$$

and find p-value to test this hypothesis.

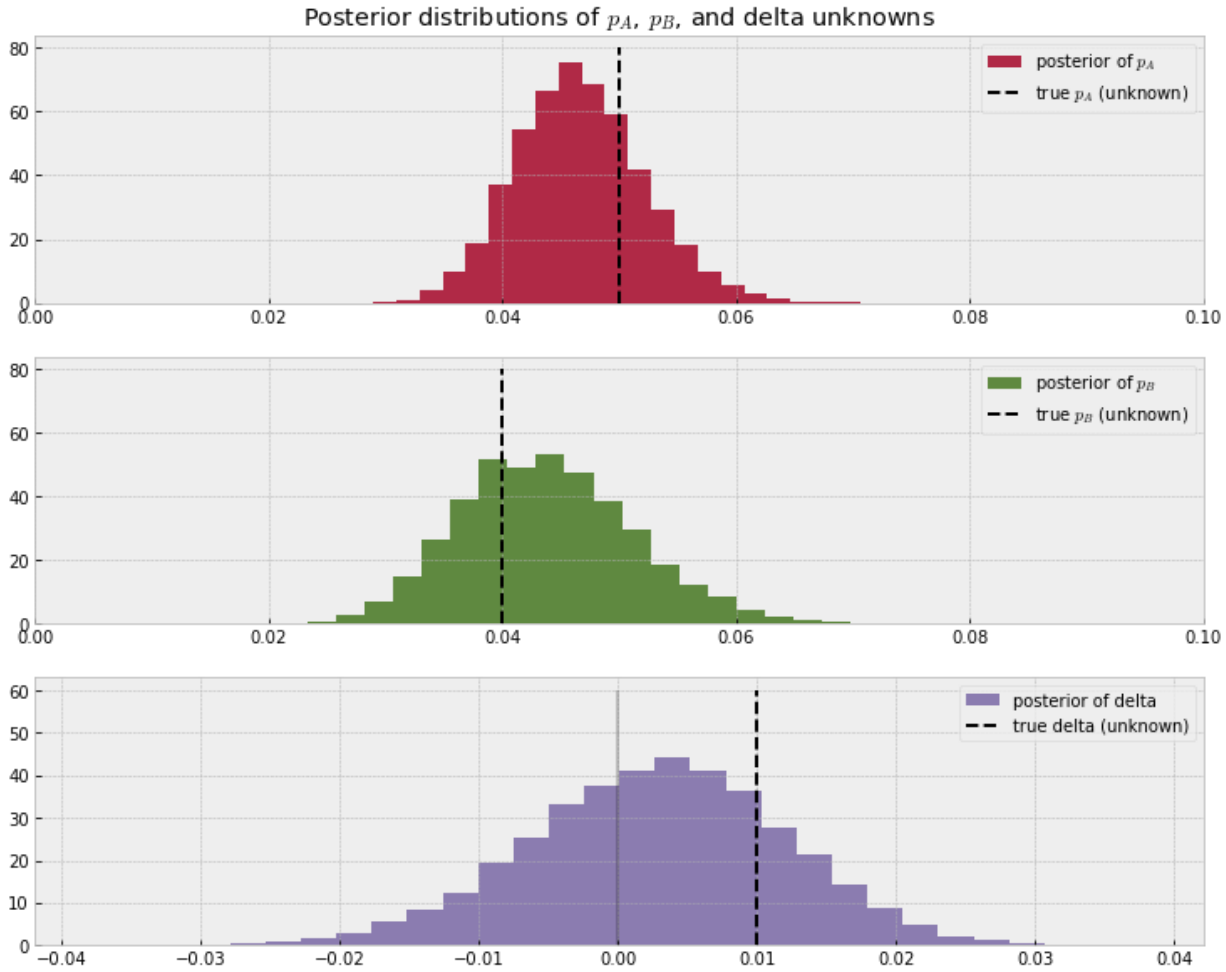
Bayesian would not make any assumption about sampling distribution. We just intoduce parameter p_A for site A and p_B for site B and we know that we have a Bernoulli trial with p_A and p_B for observetions. Purchase is made with corresponding probability or not. Bernoulli trial is a single parameter distribution and 'low-informative' priors intoduced - Uniform(0,1). We just say that this probability is a number between 0 and 1 and let data to shape our distribution.

In [14]:
```
# Set up the pymc3 model. Assume Uniform priors for p_A and p_B.
with pm.Model() as model:
    p_A = pm.Uniform("p_A", 0, 1)
    p_B = pm.Uniform("p_B", 0, 1)

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


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

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

```
Multiprocess sampling (4 chains in 4 jobs)
CompoundStep
>Metropolis: [p_B]
>Metropolis: [p_A]
Sampling 4 chains, 0 divergences: 100%|██████████| 82000/82000 [00:11<00:
00, 7178.84draws/s]
The number of effective samples is smaller than 25% for some parameters.
```

In [15]:
```
p_A_samples = burned_trace["p_A"]
p_B_samples = burned_trace["p_B"]
delta_samples = burned_trace["delta"]
```

In [16]:
```
%matplotlib inline
from IPython.core.pylabtools import figsize
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", density=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", density=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", density=True)
plt.vlines(true_p_A - true_p_B, 0, 60, linestyle="--",
           label="true delta (unknown)")
plt.vlines(0, 0, 60, color="black", alpha=0.2)
plt.legend(loc="upper right");
```



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

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

In [17]:
```
# Count the number of samples less than 0, i.e. the area under the curve
# before 0, represent the probability that site A is worse than site B.
print("Probability site A is WORSE than site B: %.3f" % \
    np.mean(delta_samples < 0))

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

```
Probability site A is WORSE than site B: 0.373
Probability site A is BETTER than site B: 0.627
```

Logistic regression with parameters estimated from Bayesian inference

In [18]:
```
import theano.tensor as tt
import numpy as np
```

In [19]:
```
challenger_data = np.genfromtxt("./module10/challenger_data.csv", skip_head
```

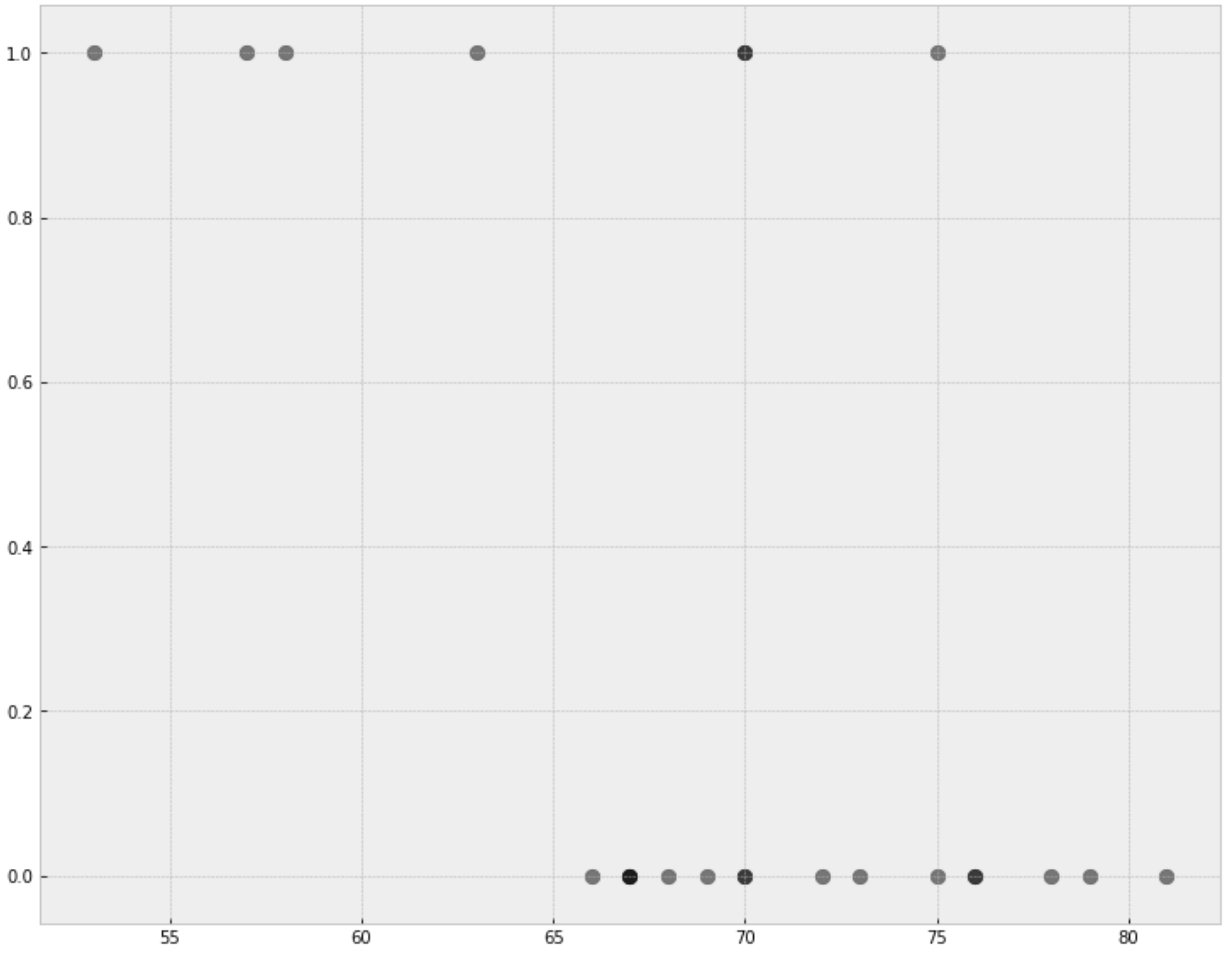
In [20]:
```
challenger_data = challenger_data[~np.isnan(challenger_data[:,1])]
challenger_data
```

Out[20]:
```
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.]])
```

Of the previous 24 flights, data were available on failures of the O-rings on 23. 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.

In [21]:
```
plt.scatter(challenger_data[:,0], challenger_data[:, 1], s=75, color="k", a
```

Out[21]:
```
<matplotlib.collections.PathCollection at 0x7f95f8f676d0>
```



It looks clear that the probability of damage incidents occurring increases as the outside

temperature decreases. The best we can do is ask "At temperature t, what is the probability of a damage incident?".

We need a function of temperature, call it p(t), that is bounded between 0 and 1, and gradually changes as we increase temperature.

The logistic function is the most popular choice.

In [22]:
```
def logistic(x, beta, alpha=0):
    return 1.0 / (1.0 + np.exp(np.dot(beta, x) + alpha))
```

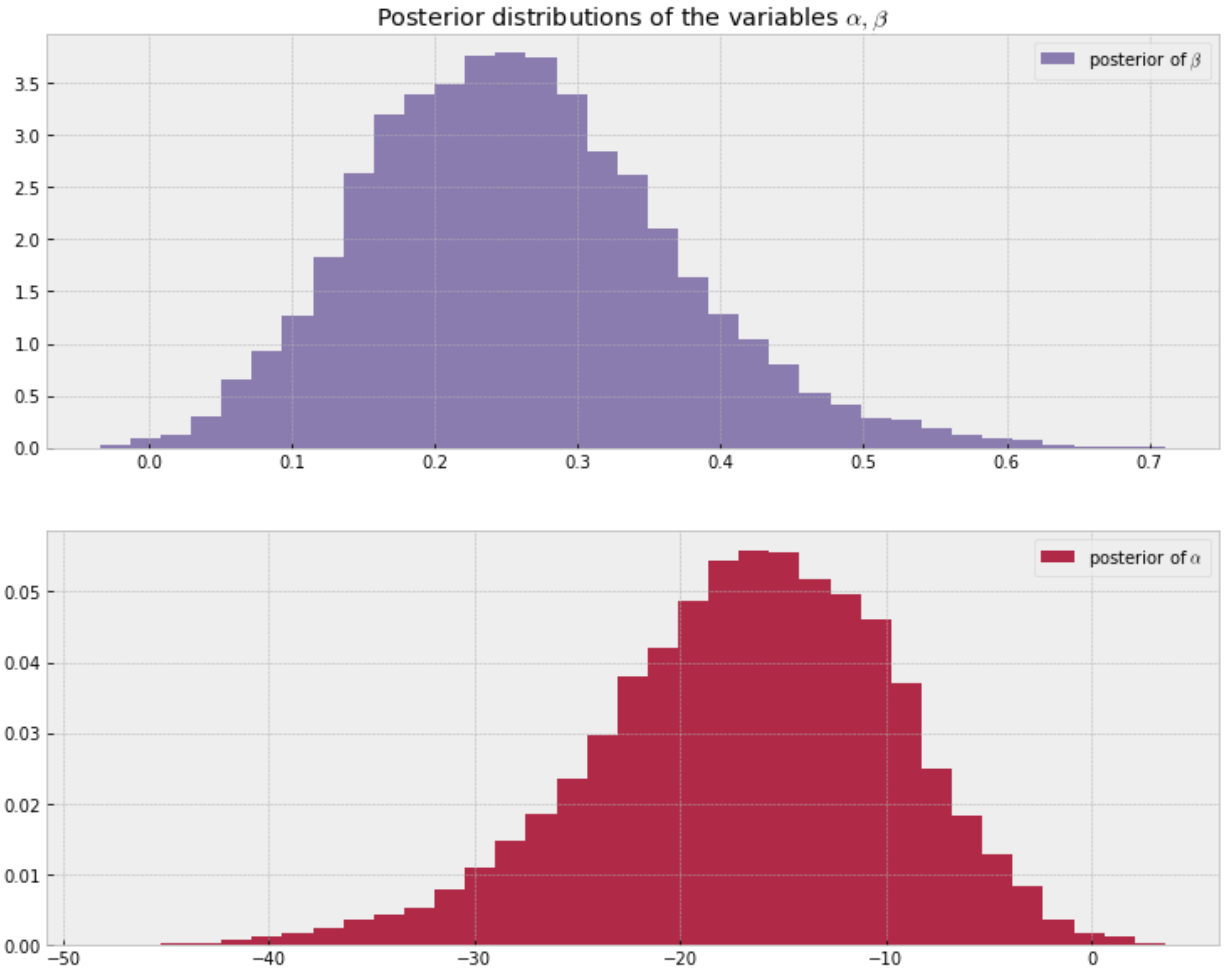
In [23]:
```
temperature  = challenger_data[:, 0]
D = challenger_data[:,1]

with pm.Model() as model_logit:
    beta = pm.Normal("beta", mu=0, tau=0.0001, testval = 0)
    alpha = pm.Normal("alpha", mu=0, tau=0.001, testval = 0)
    p = pm.Deterministic("p", 1.0/(1.0 + tt.exp(beta*temperature + alpha)))

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

    start = pm.find_MAP()
    step = pm.Metropolis()
    trace =  pm.sample(200000, start=start, step=step, progressbar=True)
    burned_trace = trace[100000::2]
```

```
logp = -20.175, ||grad|| = 9.907: 100%|██████████| 27/27 [00:00<00:00, 25
57.79it/s]
Multiprocess sampling (4 chains in 4 jobs)
CompoundStep
>Metropolis: [alpha]
>Metropolis: [beta]
Sampling 4 chains, 0 divergences: 100%|██████████| 802000/802000 [01:50<0
0:00, 7276.64draws/s]
The number of effective samples is smaller than 10% for some parameters.
```

To connect probabilities to observed data, we use a Bernoulli random variable with parameter p. It takes value 1 with probability p, and 0 otherwise.

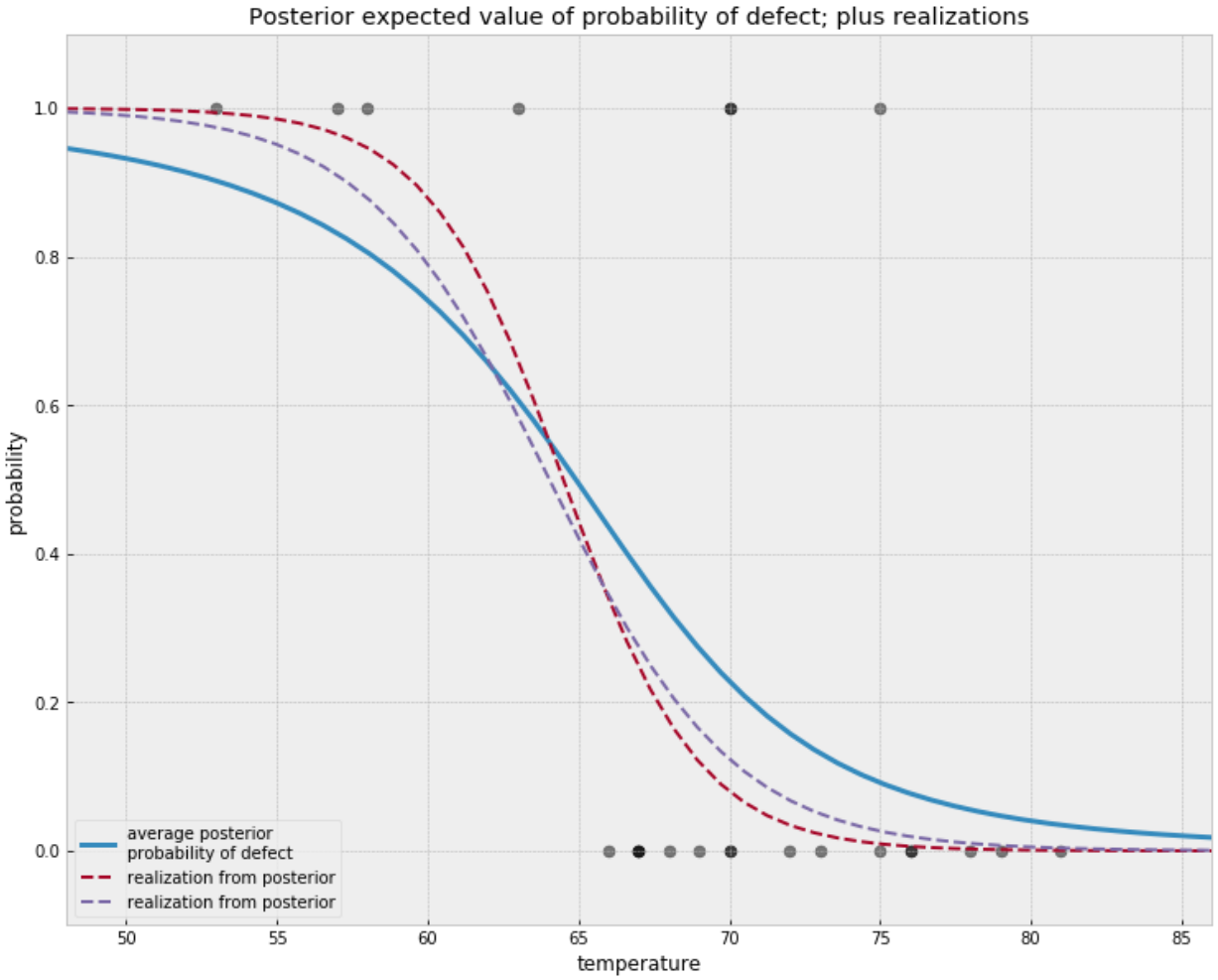
In [24]:
```
alpha_samples = burned_trace["alpha"][:, None]
beta_samples = burned_trace["beta"][:, None]


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();
```

```
/Users/sergiynokhrin/opt/anaconda3/lib/python3.7/site-packages/ipykernel_
launcher.py:8: MatplotlibDeprecationWarning:
The 'normed' kwarg was deprecated in Matplotlib 2.1 and will be removed i
n 3.1. Use 'density' instead.

/Users/sergiynokhrin/opt/anaconda3/lib/python3.7/site-packages/ipykernel_
launcher.py:13: MatplotlibDeprecationWarning:
The 'normed' kwarg was deprecated in Matplotlib 2.1 and will be removed i
n 3.1. Use 'density' instead.
  del sys.path[0]
```



In [25]:
```
t = np.linspace(temperature.min() - 5, temperature.max()+5, 50)[:, None]
p_t = logistic(t.T, beta_samples, alpha_samples)

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

In [26]:
```
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");
```



Bayesian linear regression

In [27]:
```
from sklearn.linear_model import LinearRegression
```

In [28]:
```
bike_sharing = pd.read_csv('./module10/bikes_sharing.csv', header=0, sep= '
```

In [29]:
```
bike_sharing.isnull().values.any()
```

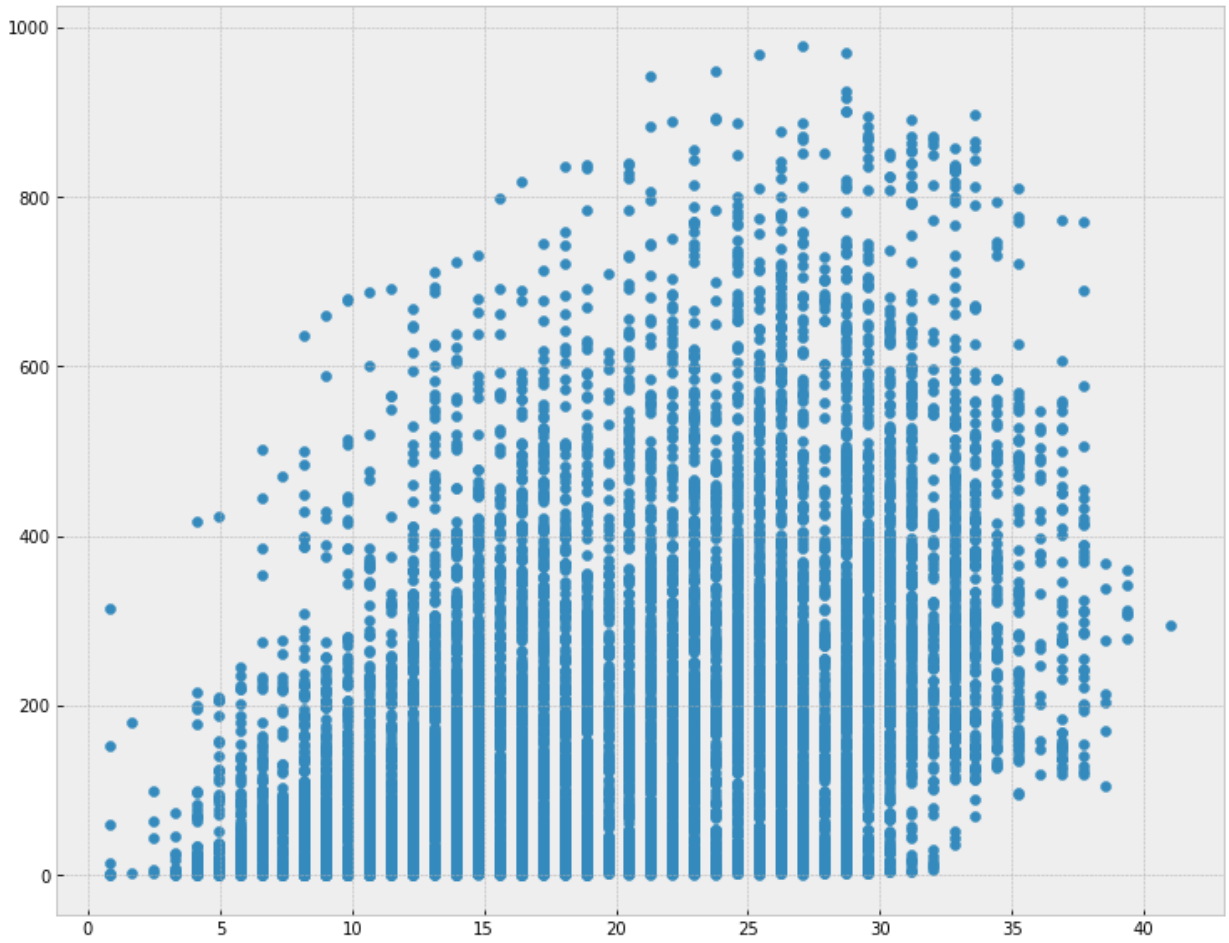
Out[29]: False

In [30]:
```
bike_sharing.shape
```

Out[30]:
```
(10886, 12)
```

In [31]:
```
plt.scatter(x=bike_sharing['temp'], y=bike_sharing['count'])
```

Out[31]:
```
<matplotlib.collections.PathCollection at 0x7f95f8b6a210>
```

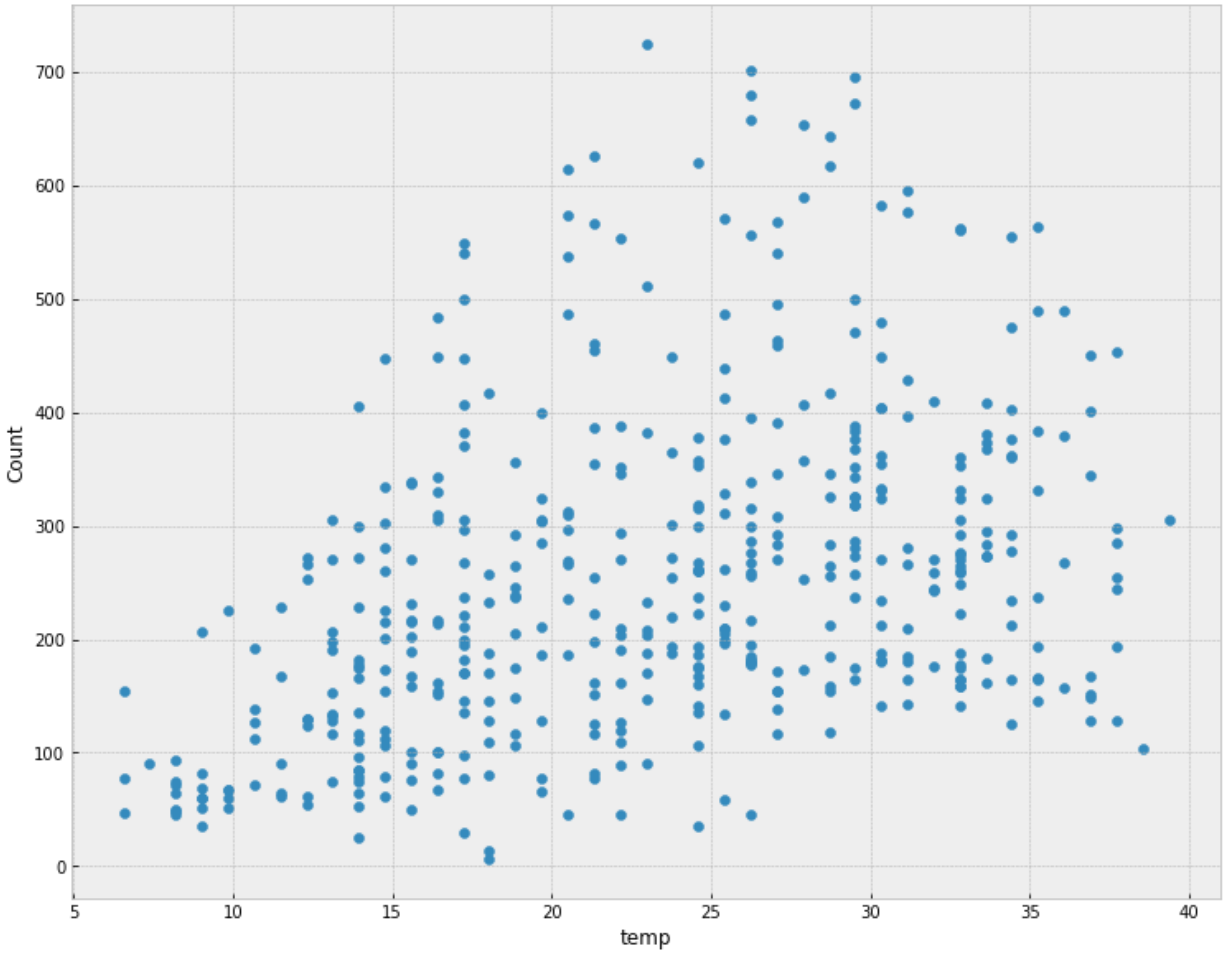


In [32]:
```
bike_sharing.datetime = bike_sharing.datetime.apply(pd.to_datetime)

bike_sharing['month'] = bike_sharing.datetime.apply(lambda x : x.month)
bike_sharing['hour'] = bike_sharing.datetime.apply(lambda x : x.hour)
```

In [33]:
```
# we want to predice bike renting at certain time (15:00 - 15:59)

bike_sharing_15 = bike_sharing.loc[bike_sharing['hour'] == 15]
```

In [34]:
```
plt.scatter(x=bike_sharing_15['temp'], y=bike_sharing_15['count'])

plt.xlabel("temp")
plt.ylabel('Count')
```

Out[34]:
```
Text(0, 0.5, 'Count')
```



In [35]:
```
# Preparation of X and y for simulation

X=bike_sharing_15.loc[:,['humidity', 'atemp']]
y = bike_sharing_15.loc[:, "count"]
```

In [36]:
```
lin_reg=LinearRegression()
lin_reg.fit(X.atemp.values.reshape(-1, 1), y)
```

Out[36]:
```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=
False)
```

In [37]:
```
print('Intercept:', lin_reg.intercept_)
print('Slope:', lin_reg.coef_)
```

```
Intercept: 78.64994444157625
Slope: [6.54614017]
```

In [38]:
```
lin_reg.intercept_+lin_reg.coef_[0]*20
```

Out[38]:
```
209.57274780495794
```

Implement MCMC to find the posterior distribution of the model parameters. Rather than a single point estimate of the model weights, Bayesian linear regression will give us a posterior distribution for the model weights.

In [39]:
```
 pm.Model() as linear_model:
#prior for each parameter of linear model, including variance of observatio
intercept = pm.Normal('Intercept', mu = 0, sd = 10)

slope = pm.Normal('slope', mu = 0, sd = 10)

sigma = pm.HalfNormal('sigma', sd = 10)

# Estimate
# mean value at each X value is given by linear model with Intercept and Sl
mean = intercept + slope * X.loc[:, 'atemp']

# the likelihood is Normal distribution of observation around the mean give
Y_obs = pm.Normal('Y_obs', mu = mean, sd = sigma, observed = y.values)

#Sampler
step = pm.NUTS()

linear_trace = pm.sample(20000, step=step, progressbar=True)
```

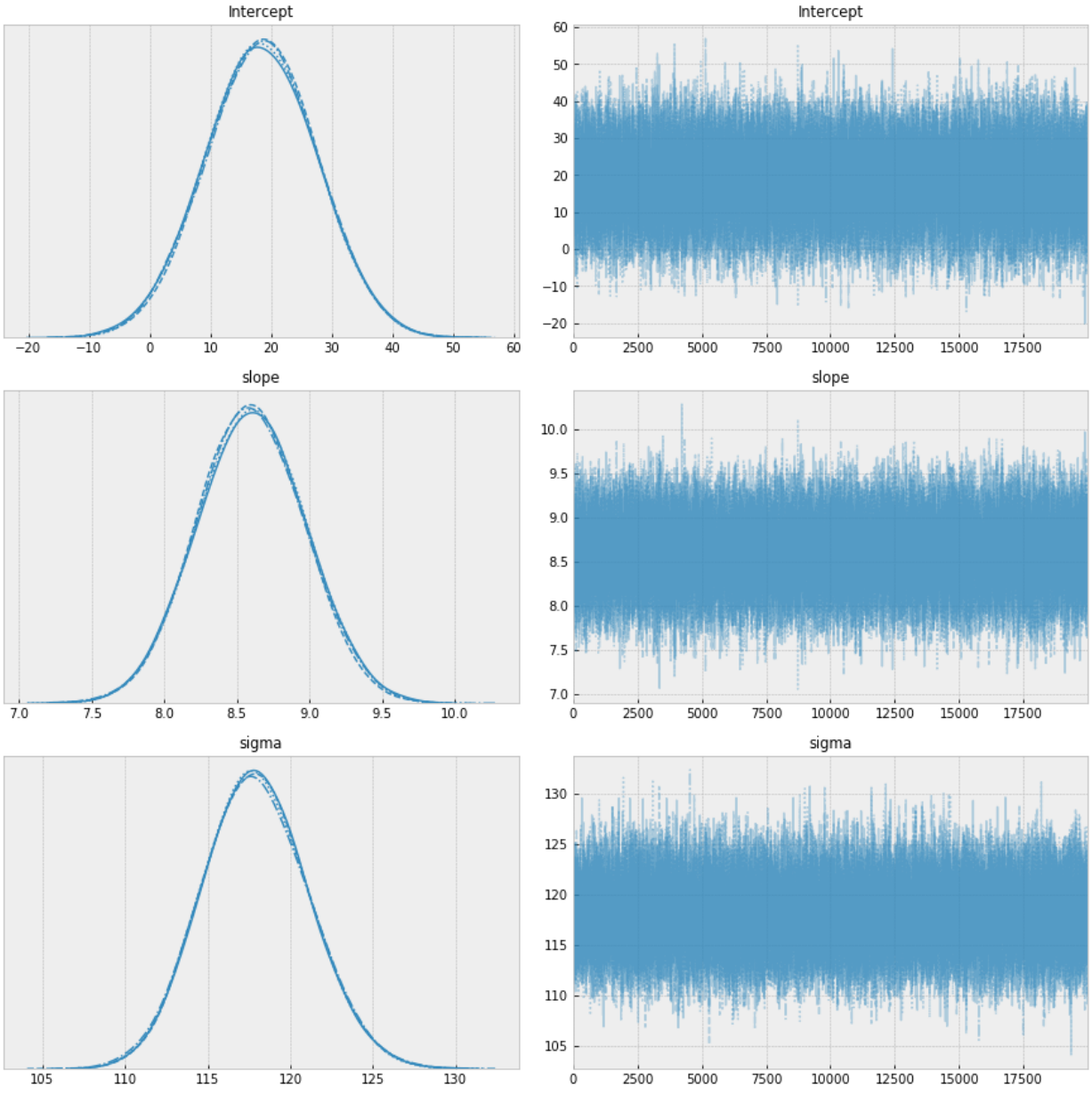
```
Multiprocess sampling (4 chains in 4 jobs)
NUTS: [sigma, slope, Intercept]
Sampling 4 chains, 0 divergences: 100%|██████████| 82000/82000 [00:32<00:
00, 2524.25draws/s]
The acceptance probability does not match the target. It is 0.89628795067
51434, but should be close to 0.8. Try to increase the number of tuning s
teps.
```

In [40]:
```
# For NUTS sample the points acceptance is usually higher than that in Metr
# the acceptance can be checked with 'mean_tree_accept'

linear_trace.mean_tree_accept.mean()
```

Out[40]:
```
0.878316994986686
```

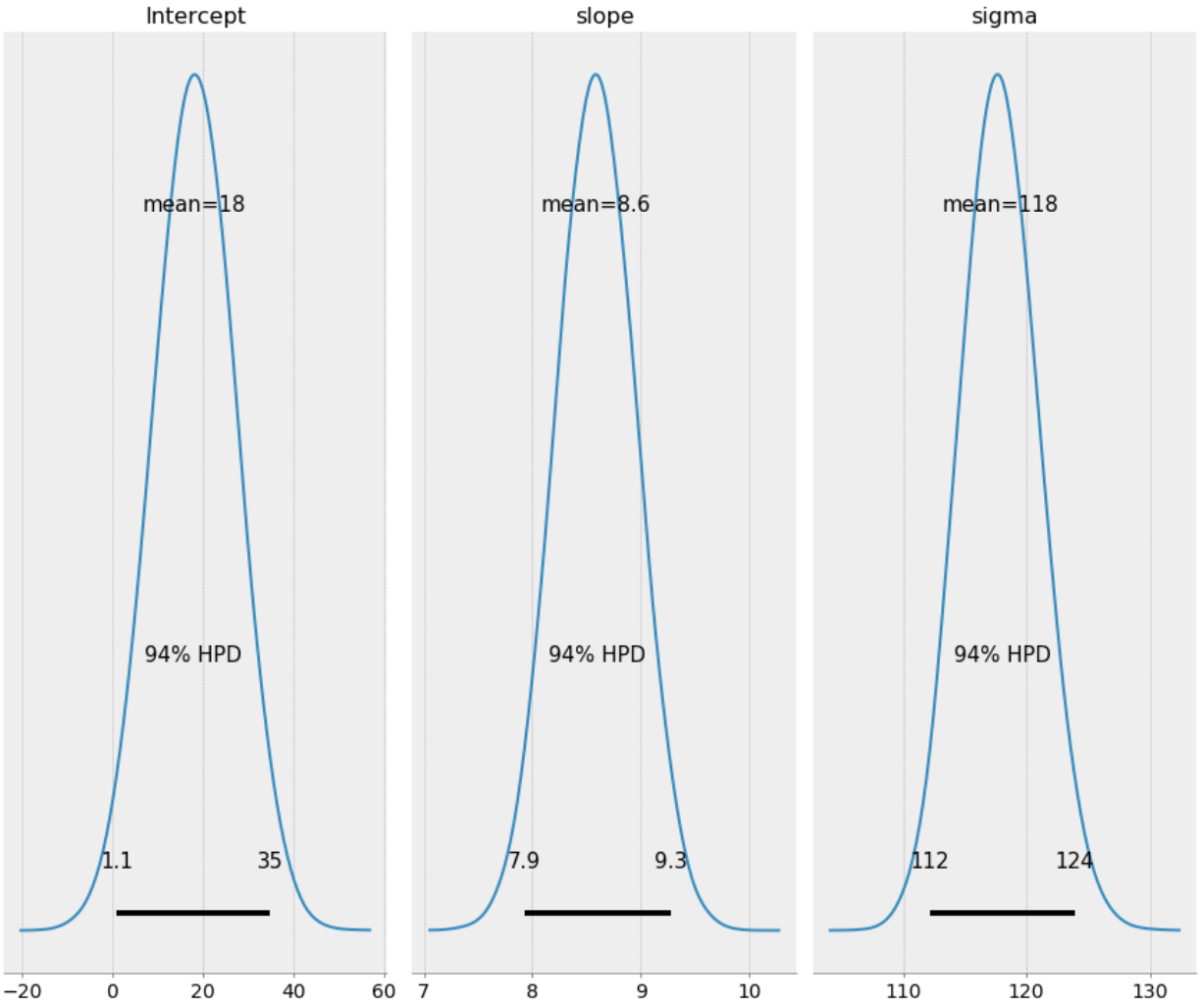
In [41]:
```
pm.traceplot(linear_trace, figsize=(12,12))
```

Out[41]:
```
array([[<matplotlib.axes._subplots.AxesSubplot object at 0x7f955b3036d0>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f95f8dde350
>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x7f9559c85790>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f95611a7490
>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x7f95f8b91890>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f956115f6d0
>]],
      dtype=object)
```



In [43]:
```
pm.plot_posterior(linear_trace, figsize=(12,10))
```

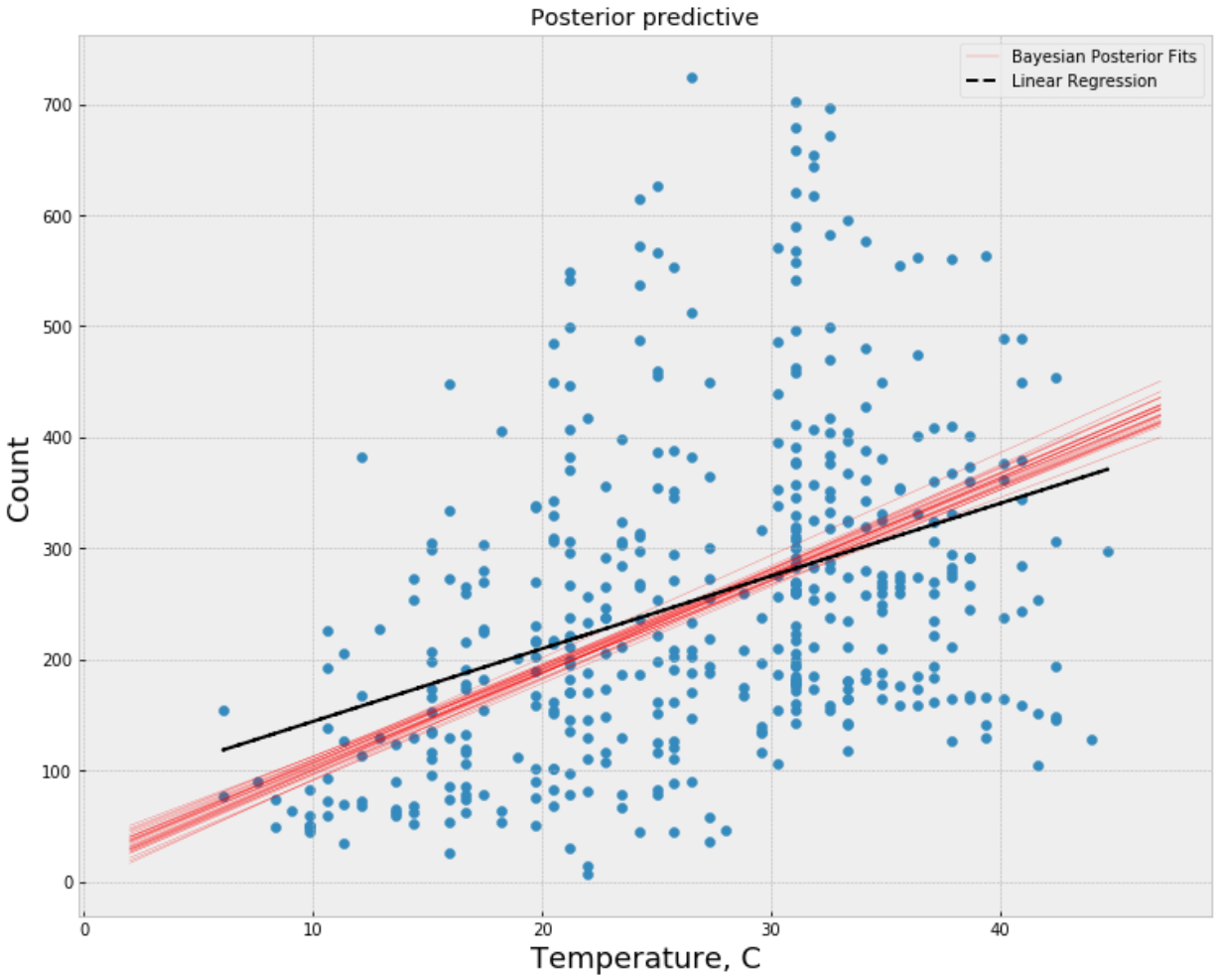
Out[43]:
```
array([<matplotlib.axes._subplots.AxesSubplot object at 0x7f9529a9ae90>,
       <matplotlib.axes._subplots.AxesSubplot object at 0x7f9527b5a790>,
       <matplotlib.axes._subplots.AxesSubplot object at 0x7f9529c43a10>],
      dtype=object)
```



Prediction of response

In [44]:
```
For using posterior distributi for data simulation we may use posterior samp
.plot_posterior_predictive_glm(linear_trace, eval=np.linspace(2,47, 100),
                               color='red', label = 'Bayesian Posterior Fits
                               lm= lambda x, sample:
                               sample['Intercept'] + sample['slope']*x)

t.plot(X.atemp.values.reshape(-1, 1), lin_reg.predict(X.atemp.values.reshape
t.scatter(x=bike_sharing_15["atemp"], y=bike_sharing_15["count"])
t.ylabel('Count', size = 18)
t.xlabel('Temperature, C', size = 18)
t.legend()
```

Out[44]: <matplotlib.legend.Legend at 0x7f95275c5090>



In [45]:
```
pm.summary(linear_trace)
```

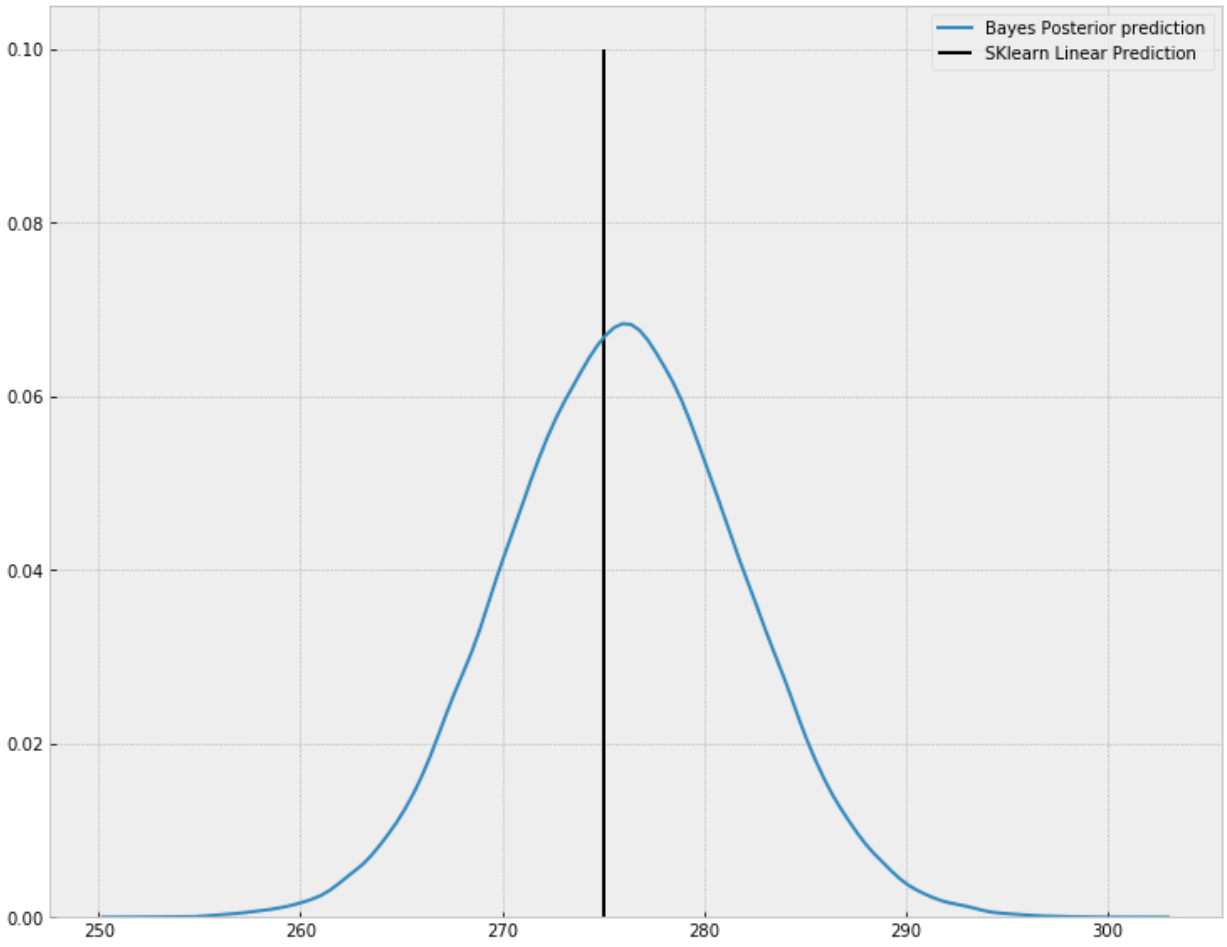
Out[45]:

	mean	sd	hpd_3%	hpd_97%	mcse_mean	mcse_sd	ess_mean	ess_sd	ess_bulk
Intercept	18.219	8.982	1.086	34.808	0.049	0.034	34072.0	33899.0	34049.0
slope	8.592	0.363	7.926	9.281	0.002	0.001	33343.0	33329.0	33315.0
sigma	117.870	3.146	112.146	123.917	0.015	0.011	44265.0	44236.0	44285.0

Prediction for single point

In [46]:
```
single_point_prediction = linear_trace['Intercept'] + linear_trace['slope']

plt.vlines(x=lin_reg.intercept_+lin_reg.coef_[0]*30, ymin=0, ymax=0.1,
           label='SKlearn Linear Prediction', )
sns.kdeplot(single_point_prediction, label='Bayes Posterior prediction')
```

Out[46]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f9529d81b90>
```



You have reached the end of this module.

If you have any questions, please reach out to your peers using the discussion boards. If you and your peers are unable to come to a suitable conclusion, do not hesitate to reach out to your instructor on the designated discussion board.

When you are comfortable with the content, and have practiced to your satisfaction, you may proceed to any related assignments, and to the next module.