# El Fin (Final Exam) for AMPTH-2017/APMA E-207

**Harvard University**
**Fall 2018**
**Instructors: Rahul Dave**
**Due Date:** Monday, December 17th, 2018 at 11:59pm

**Instructions:**

- Upload your final answers in the form of a Jupyter notebook containing all work to Canvas.
- Structure your notebook and your work to maximize readability.

## Collaborators

**Group 44 - Braavos**

**Joe Davison**
**Anna Davydova**
**Michael S. Emanuel**
**Dylan Randle**

In [183]:
```
# core
import numpy as np
import pandas as pd
import scipy.stats
# pymc3 and theano
import pymc3 as pm
from pymc3.variational.callbacks import CheckParametersConvergence
import theano.tensor as tt
from theano.tensor.nnet import softmax
# torch
import torch
import torch.nn as nn
from torch.autograd import Variable
# Plotting
import matplotlib as mpl
import matplotlib.pyplot as plt
import seaborn.apionly as sns
import arviz as az
from IPython.display import display
```

In [184]:
```
# Turn off deprecation warning (too noisy)
import warnings
warnings.filterwarnings("ignore", category=DeprecationWarning)
warnings.filterwarnings("ignore", category=FutureWarning)
warnings.filterwarnings("ignore", category=mpl.MatplotlibDeprecationWarning)
```

In [218]:
```
# Additional imports
from numpy import log, exp
from scipy.stats import logistic, multivariate_normal
import pickle
import tqdm
import warnings
from typing import List, Dict, Callable
```

In [4]:
```
# Utilities for serializing variables
def load_vartbl(fname: str) -> Dict:
    """Load a dictionary of variables from a pickled file"""
    try:
        with open(fname, 'rb') as fh:
            vartbl = pickle.load(fh)
    except:
        vartbl = dict()
    return vartbl


def save_vartbl(vartbl: Dict, fname: str) -> None:
    """Save a dictionary of variables to the given file with pickle"""
    with open(fname, 'wb') as fh:
        pickle.dump(vartbl, fh)
```

# Q1: GLMs with correlation

## The dataset: A Bangladesh Contraception use census

This problem is based on one-two (12H1 and continuations ) from your textbook. The data is in the file `bangladesh.csv`. These data are from the 1988 Bangladesh Fertility Survey. Each row is one of 1934 women. There are six variables:

- (1) `district` : ID number of administrative district each woman resided in
- (2) `use.contraception` : An indicator (0/1) of whether the woman was using contraception
- (3) `urban` : An indicator (0/1) of whether the woman lived in a city, as opposed to living in a rural area
- (4) `woman` : a number indexing a single woman in this survey
- (5) `living.chidren` : the number of children living with a woman
- (6) `age.centered` : a continuous variable representing the age of the woman with the sample mean subtracted

We need to make sure that the cluster variable, district, is a contiguous set of integers, so that we can use the index to differentiate the districts easily while sampling ((look at the Chimpanzee models we did in lab to understand the indexing). So create a new contiguous integer index to represent the districts. Give it a new column in the dataframe, such as `district.id`.

You will be investigating the dependence of contraception use on the district in which the survey was done. Specifically, we will want to regularize estimates from those districts where very few women were surveyed. We will further want to investigate whether the areas of residence (urban or rural) within a district impacts a woman's use of contraception.

Feel free to indulge in any exploratory visualization which helps you understand the dataset better.

In [5]:
```
# Serialize variables for GLM quesion
fname: str = 'glm.pickle'
vartbl: Dict = load_vartbl(fname)
```

In [6]:

```
# Load the dataset
df = pd.read_csv('bangladesh.csv', sep=';')
# Generate map of distinct districts
district_id_map = {d:i for i, d in enumerate(sorted(set(df.district)))}
# Add new column district_id to the dataset; these are contiguous integers 0-59
df['district_id'] = df.district.map(district_id_map)

# Rename the columns to avoid the confusing '.' in column names
df.rename(axis='columns', inplace=True, mapper=
    {'use.contraception':'use_contraception',
     'living.children':'living_children',
     'age.centered':'age_centered',
     })

# The number of districts
num_districts: int = len(district_id_map)

# Create a dataset aggregated by district (sufficient statistic for the by district model)
agg_tbl = {
        'woman': ['count'],
        'use_contraception': ['sum']
        }
df_district = df.groupby(by=df.district_id).agg(agg_tbl)
df_district.columns = ["_".join(x) for x in df_district.columns.ravel()]
# The number of women sampled in each district
district_count = df_district.woman_count

# Set the number of samples for this problem (used in multiple parts)
num_samples: int = 11000
num_tune: int = 1000

# Chart formatting
mpl.rcParams.update({'font.size': 16})
```

In [7]:
```
# Review the dataset
display(df.sample(25).sort_values(by=['woman']))
```

| | woman | district | use_contraception | living_children | age_centered | urban | district_id |
|---|---|---|---|---|---|---|---|
| **366** | 367 | 11 | 0 | 2 | -5.5599 | 0 | 10 |
| **367** | 368 | 11 | 0 | 2 | 18.4400 | 0 | 10 |
| **370** | 371 | 11 | 0 | 1 | -12.5590 | 0 | 10 |
| **456** | 457 | 14 | 1 | 4 | 2.4400 | 1 | 13 |
| **464** | 465 | 14 | 1 | 4 | 14.4400 | 1 | 13 |
| **680** | 681 | 19 | 0 | 2 | 5.4400 | 0 | 18 |
| **687** | 688 | 19 | 1 | 4 | 3.4400 | 0 | 18 |
| **748** | 749 | 23 | 0 | 3 | -4.5599 | 0 | 22 |
| **754** | 755 | 23 | 0 | 3 | -5.5600 | 0 | 22 |
| **762** | 763 | 23 | 0 | 4 | 8.4400 | 0 | 22 |
| **782** | 783 | 25 | 1 | 2 | 9.4400 | 1 | 24 |
| **831** | 832 | 25 | 0 | 1 | -5.5599 | 0 | 24 |
| **844** | 845 | 26 | 1 | 4 | 5.4400 | 0 | 25 |
| **943** | 944 | 28 | 0 | 4 | 6.4400 | 0 | 27 |
| **1035** | 1036 | 30 | 0 | 1 | -9.5599 | 0 | 29 |
| **1182** | 1183 | 35 | 1 | 3 | -0.5599 | 0 | 34 |
| **1281** | 1282 | 40 | 1 | 4 | 1.4400 | 1 | 39 |
| **1345** | 1346 | 43 | 0 | 1 | 1.4400 | 1 | 42 |
| **1379** | 1380 | 43 | 1 | 1 | -10.5590 | 0 | 42 |
| **1410** | 1411 | 44 | 1 | 3 | 16.4400 | 0 | 43 |
| **1686** | 1687 | 52 | 0 | 2 | -7.5599 | 0 | 51 |
| **1718** | 1719 | 52 | 1 | 4 | 1.4400 | 0 | 51 |
| **1745** | 1746 | 56 | 1 | 4 | 13.4400 | 1 | 54 |
| **1908** | 1909 | 61 | 0 | 2 | -7.5599 | 0 | 59 |
| **1914** | 1915 | 61 | 0 | 2 | -7.5599 | 0 | 59 |

# Part A

We will use `use.contraception` as a Bernoulli response variable.

When we say "fit" below, we mean, specify the model, plot its graph, sample from it, do some tests, and forest-plot and summarize the posteriors, at the very least.

**A1** Fit a traditional "fixed-effects" model which sets up district-specific intercepts, each with its own Normal(0, 10) prior. That is, the intercept is modeled something like

```
alpha_district = pm.Normal('alpha_district', 0, 10, shape=num_districts)
p=pm.math.invlogit(alpha_district[df.district_id])
```

Why should there not be any overall intercept in this model?

**A2** Fit a multi-level "varying-effects" model with an overall intercept `alpha`, and district-specific intercepts `alpha_district`. Assume that the overall intercept has a Normal(0, 10) prior, while the district specific intercepts are all drawn from the **same** normal distribution with mean 0 and standard deviation $\sigma$. Let $\sigma$ be drawn from HalfCauchy(2). The setup of this model is similar to the per-chimanzee models in the prosocial chimanzee labs.

**A3** What does a posterior-predictive sample in this model look like? What is the difference between district specific posterior predictives and woman specific posterior predictives. In other words, how might you model the posterior predictive for a new woman being from a particular district vs that os a new woman in the entire sample? This is a word answer; no programming

required.

**A4** Plot the predicted proportions of women in each district using contraception against the id of the district, in both models. How do these models disagree? Look at the extreme values of predicted contraceptive use in the fixed effects model. How is the disagreement in these cases?

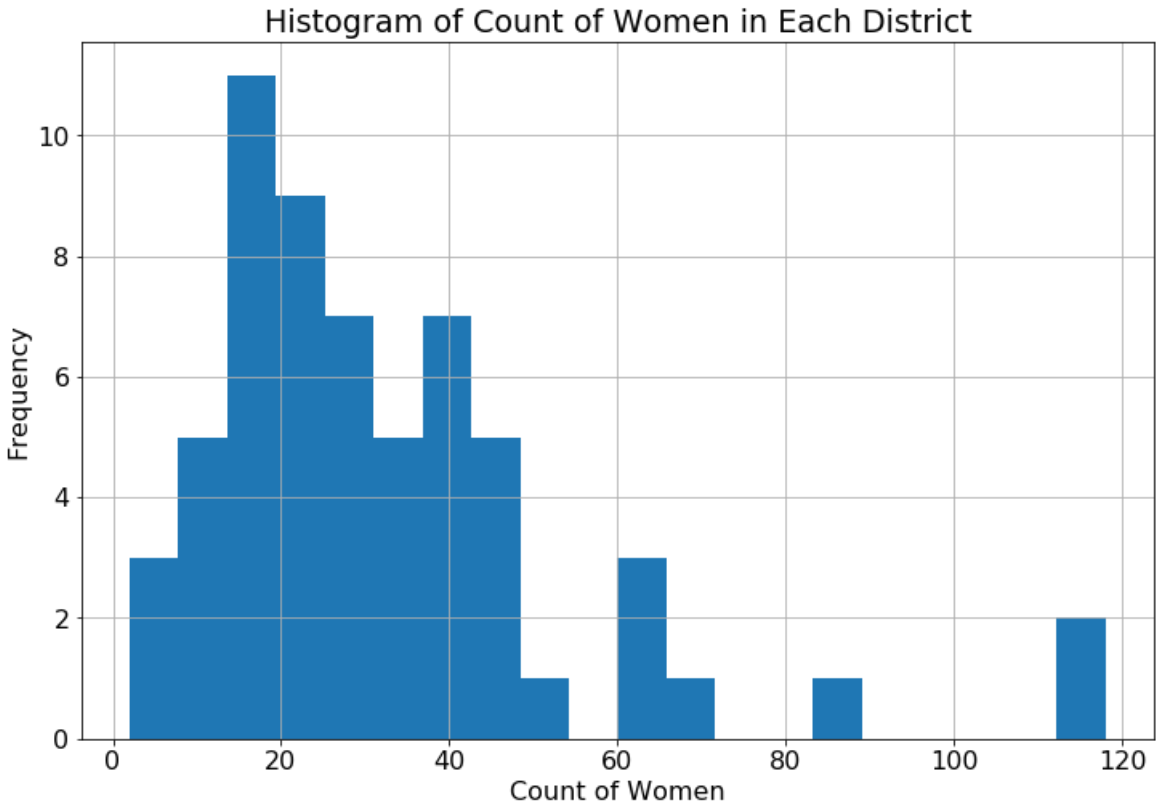
**A5** Plot the absolute value of the difference in probability of contraceptive use against the number of women sampled in each district. What do you see?

**A1** Fit a traditional "fixed-effects" model which sets up district-specific intercepts, each with its own Normal(0, 10) prior. That is, the intercept is modeled something like

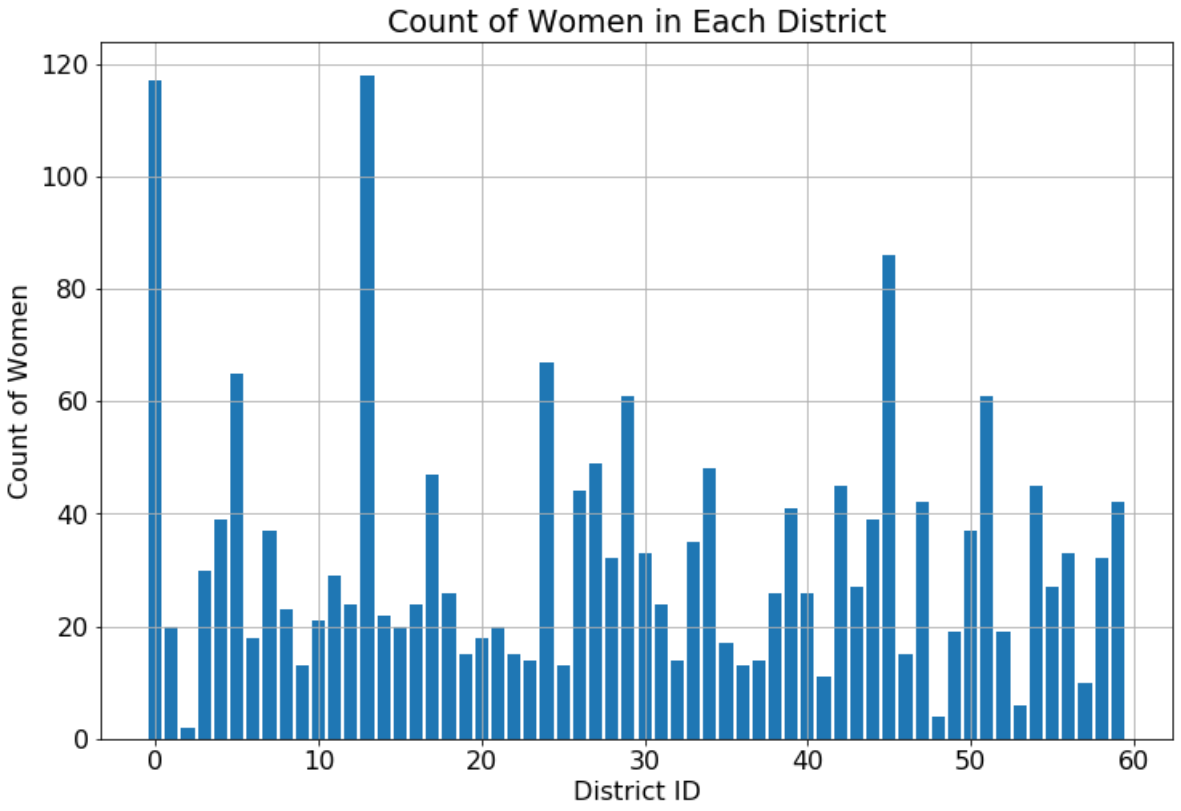
alpha_district = pm.Normal('alpha_district', 0, 10, shape=num_districts) p=pm.math.invlogit(alpha_district[df.district_id])

**View Data Aggregeated by District**

In [8]:
```
fig, ax = plt.subplots(figsize=[12,8])
ax.set_title('Histogram of Count of Women in Each District')
ax.set_xlabel('Count of Women')
ax.set_ylabel('Frequency')
ax.hist(df_district.woman_count, bins=20)
ax.grid()
```



In [9]:
```
fig, ax = plt.subplots(figsize=[12,8])
ax.set_title('Count of Women in Each District')
ax.set_xlabel('District ID')
ax.set_ylabel('Count of Women')
ax.bar(df_district.index.values, df_district.woman_count)
ax.grid()
```




We can see that there are three districts with small sample sizes. These will be hard to estimate in this model.

**Specify the Fixed Effects Model and Draw Samples**

In [10]:
```
# Define the fixed-effects model
with pm.Model() as model_fe:
    # Set the prior for the intercept in each district
    alpha_district = pm.Normal(name='alpha_district', mu=0.0, sd=10.0, shape=num_districts)
    # Set the probability that each woman uses contraception in this model
    # It depends only on the district she lives in
    p = pm.math.invlogit(alpha_district[df.district_id])
    # The response variable - whether this woman used contraception; modeled as Bernoulli
    # Bind this to the observed values
    use_contraception = pm.Bernoulli('use_contraception', p=p, observed=df['use_contraception'])

# Sample from the fixed-effects model
try:
    trace_fe = vartbl['trace_fe']
    print(f'Loaded samples for the Fixed Effects model in trace_fe.')
except:
    with model_fe:
        trace_fe = pm.sample(draws=num_samples, tune=num_tune, chains=2, cores=1)
    vartbl['trace_fe'] = trace_fe
    save_vartbl(vartbl, fname)
```

```
Loaded samples for the Fixed Effects model in trace_fe.
```

**Tabular Summary of the 60 alpha Parameters for Fixed Effects Model**

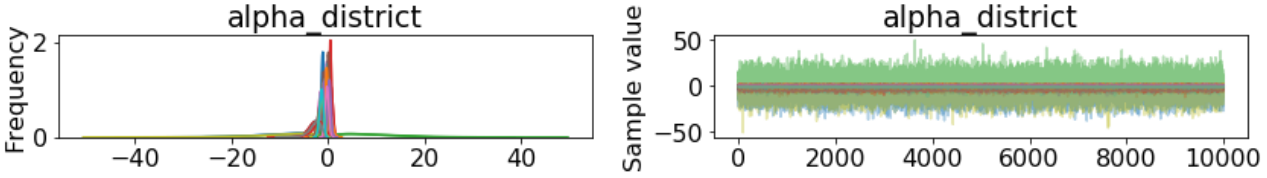
In [11]:
```
summary_fe = pm.summary(trace_fe)
display(summary_fe)
```

| | mean | sd | mc_error | hpd_2.5 | hpd_97.5 | n_eff | Rhat |
|---|---|---|---|---|---|---|---|
| **alpha_district__0** | -1.073243 | 0.214788 | 0.001066 | -1.489540 | -0.647595 | 40041.165286 | 0.999961 |
| **alpha_district__1** | -0.654165 | 0.484018 | 0.002597 | -1.645399 | 0.248680 | 33574.777705 | 0.999966 |
| **alpha_district__2** | 8.584608 | 6.050171 | 0.055936 | -0.730116 | 20.555641 | 13994.398200 | 0.999950 |
| **alpha_district__3** | -0.001951 | 0.371934 | 0.002184 | -0.747240 | 0.722498 | 31487.460838 | 0.999958 |
| **alpha_district__4** | -0.595484 | 0.334658 | 0.002170 | -1.246854 | 0.060581 | 28057.634456 | 1.000217 |
| **alpha_district__5** | -0.899699 | 0.276824 | 0.001459 | -1.453256 | -0.372018 | 31972.995839 | 0.999955 |
| **alpha_district__6** | -1.020319 | 0.547027 | 0.003179 | -2.081698 | 0.061998 | 28085.334737 | 0.999950 |
| **alpha_district__7** | -0.511668 | 0.341304 | 0.001938 | -1.164860 | 0.168269 | 37431.975288 | 0.999950 |
| **alpha_district__8** | -0.863216 | 0.468712 | 0.002341 | -1.845981 | 0.004499 | 35307.774063 | 0.999957 |
| **alpha_district__9** | -2.962574 | 1.267900 | 0.010468 | -5.516910 | -0.766858 | 15446.697879 | 0.999993 |
| **alpha_district__10** | -10.355359 | 5.564674 | 0.043709 | -21.548781 | -2.209755 | 12347.004756 | 0.999990 |
| **alpha_district__11** | -0.662113 | 0.394520 | 0.002002 | -1.455576 | 0.086338 | 34955.718166 | 0.999955 |
| **alpha_district__12** | -0.350043 | 0.423649 | 0.002511 | -1.167743 | 0.484508 | 32092.203844 | 1.000036 |
| **alpha_district__13** | 0.524344 | 0.187886 | 0.001107 | 0.161535 | 0.897190 | 28154.800120 | 0.999991 |
| **alpha_district__14** | -0.586845 | 0.451111 | 0.002155 | -1.485561 | 0.281917 | 32479.376209 | 0.999960 |
| **alpha_district__15** | 0.208197 | 0.461724 | 0.002885 | -0.726643 | 1.102990 | 25455.899028 | 0.999971 |
| **alpha_district__16** | -0.930461 | 0.456843 | 0.002402 | -1.843420 | -0.046071 | 33873.440248 | 0.999961 |
| **alpha_district__17** | -0.676373 | 0.311371 | 0.001526 | -1.289353 | -0.068114 | 31653.577539 | 0.999984 |
| **alpha_district__18** | -0.489764 | 0.412273 | 0.002375 | -1.301367 | 0.332398 | 32723.340991 | 0.999957 |
| **alpha_district__19** | -0.428284 | 0.543707 | 0.002990 | -1.481585 | 0.646021 | 32065.554125 | 1.000034 |
| **alpha_district__20** | -0.479888 | 0.499031 | 0.003032 | -1.481272 | 0.463296 | 32757.175814 | 0.999962 |
| **alpha_district__21** | -1.480272 | 0.591250 | 0.003361 | -2.689370 | -0.390734 | 29058.167570 | 0.999951 |
| **alpha_district__22** | -1.091934 | 0.606208 | 0.003446 | -2.296482 | 0.081861 | 30487.906597 | 0.999955 |
| **alpha_district__23** | -3.031626 | 1.271177 | 0.011377 | -5.572781 | -0.773350 | 14306.845481 | 0.999955 |
| **alpha_district__24** | -0.213181 | 0.245808 | 0.001337 | -0.701960 | 0.261617 | 34262.407382 | 0.999969 |
| **alpha_district__25** | -0.505243 | 0.593344 | 0.003168 | -1.677708 | 0.661977 | 31515.961288 | 0.999958 |
| **alpha_district__26** | -1.549604 | 0.401173 | 0.002284 | -2.361461 | -0.787539 | 32684.428575 | 0.999952 |
| **alpha_district__27** | -1.154341 | 0.335609 | 0.001709 | -1.785608 | -0.471857 | 31395.245410 | 0.999954 |
| **alpha_district__28** | -0.969577 | 0.401680 | 0.002363 | -1.736690 | -0.184705 | 31599.101330 | 0.999953 |
| **alpha_district__29** | -0.034008 | 0.255064 | 0.001430 | -0.537786 | 0.460711 | 34028.542480 | 0.999953 |
| **alpha_district__30** | -0.186481 | 0.353942 | 0.001945 | -0.872118 | 0.518062 | 34878.614060 | 0.999952 |
| **alpha_district__31** | -1.411969 | 0.526794 | 0.002953 | -2.430754 | -0.378501 | 31759.674989 | 1.000019 |
| **alpha_district__32** | -0.301578 | 0.560301 | 0.002899 | -1.390674 | 0.826071 | 39294.262443 | 1.000073 |
| **alpha_district__33** | 0.668846 | 0.364173 | 0.002083 | -0.028010 | 1.394163 | 33354.214010 | 1.000099 |
| **alpha_district__34** | 0.001328 | 0.290297 | 0.001585 | -0.552498 | 0.574295 | 31928.479913 | 0.999953 |
| **alpha_district__35** | -0.644477 | 0.521937 | 0.002751 | -1.660441 | 0.385284 | 33682.391932 | 0.999979 |
| **alpha_district__36** | 0.167207 | 0.572602 | 0.003150 | -0.954464 | 1.291115 | 38440.501773 | 1.000077 |
| **alpha_district__37** | -0.991909 | 0.614256 | 0.003336 | -2.238986 | 0.138079 | 30788.807086 | 0.999970 |
| **alpha_district__38** | 0.002934 | 0.396218 | 0.002021 | -0.792044 | 0.770239 | 33377.420233 | 0.999950 |
| **alpha_district__39** | -0.151940 | 0.322332 | 0.001842 | -0.781386 | 0.483093 | 34681.970612 | 0.999953 |
| **alpha_district__40** | 0.000719 | 0.396936 | 0.001976 | -0.784867 | 0.773010 | 35954.130977 | 0.999964 |
| **alpha_district__41** | 0.205175 | 0.623212 | 0.003110 | -1.002151 | 1.452739 | 36251.615562 | 1.000024 |
| **alpha_district__42** | 0.136827 | 0.299931 | 0.001996 | -0.429883 | 0.754656 | 32013.817925 | 0.999954 |

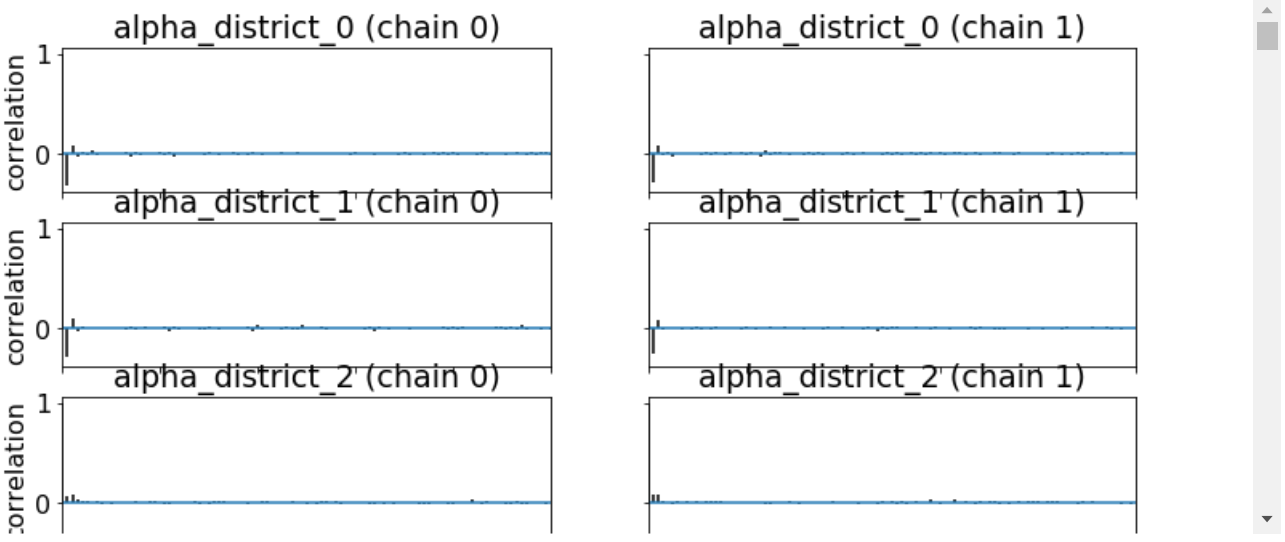
| | mean | sd | mc_error | hpd_2.5 | hpd_97.5 | n_eff | Rhat |
|---|---|---|---|---|---|---|---|
| **alpha_district__43** | -1.312320 | 0.480202 | 0.002675 | -2.240538 | -0.380542 | 26873.776141 | 1.000014 |
| **alpha_district__44** | -0.711040 | 0.338823 | 0.001870 | -1.363454 | -0.035772 | 33706.569816 | 0.999958 |
| **alpha_district__45** | 0.094381 | 0.214862 | 0.001234 | -0.318773 | 0.521524 | 30404.757599 | 1.000037 |
| **alpha_district__46** | -0.141399 | 0.531339 | 0.002862 | -1.214813 | 0.878446 | 30922.566820 | 0.999972 |
| **alpha_district__47** | 0.098111 | 0.313338 | 0.001547 | -0.550550 | 0.688669 | 35513.434149 | 0.999972 |
| **alpha_district__48** | -9.089031 | 5.843864 | 0.051160 | -20.763048 | -0.381613 | 14848.091404 | 0.999958 |
| **alpha_district__49** | -0.108294 | 0.469539 | 0.002522 | -1.041823 | 0.808073 | 37308.668435 | 0.999951 |
| **alpha_district__50** | -0.169634 | 0.333420 | 0.001988 | -0.816870 | 0.486635 | 31053.334034 | 0.999951 |
| **alpha_district__51** | -0.231538 | 0.258708 | 0.001527 | -0.748822 | 0.265638 | 32576.070989 | 1.000044 |
| **alpha_district__52** | -0.328996 | 0.478678 | 0.003003 | -1.269329 | 0.624714 | 27640.896795 | 0.999995 |
| **alpha_district__53** | -2.030885 | 1.320174 | 0.008250 | -4.778716 | 0.288938 | 18013.496646 | 0.999961 |
| **alpha_district__54** | 0.322471 | 0.307308 | 0.001578 | -0.268446 | 0.926554 | 38292.805896 | 1.000025 |
| **alpha_district__55** | -1.562131 | 0.523514 | 0.002925 | -2.612649 | -0.583326 | 26757.224745 | 0.999959 |
| **alpha_district__56** | -0.187849 | 0.355089 | 0.001843 | -0.891192 | 0.498663 | 36147.828427 | 0.999971 |
| **alpha_district__57** | -2.658778 | 1.286981 | 0.008792 | -5.201481 | -0.329533 | 18071.813140 | 0.999951 |
| **alpha_district__58** | -1.323432 | 0.439579 | 0.002024 | -2.153384 | -0.452722 | 33608.731019 | 0.999977 |
| **alpha_district__59** | -1.339060 | 0.385473 | 0.002508 | -2.107087 | -0.608050 | 30579.340233 | 0.999967 |

**Plot Graphs for Fixed Effects Model: Traceplot, Autocorrplot**

In [12]:
```
fig = pm.traceplot(trace_fe)
```




In [13]:
```
fig = pm.autocorrplot(trace_fe)
```

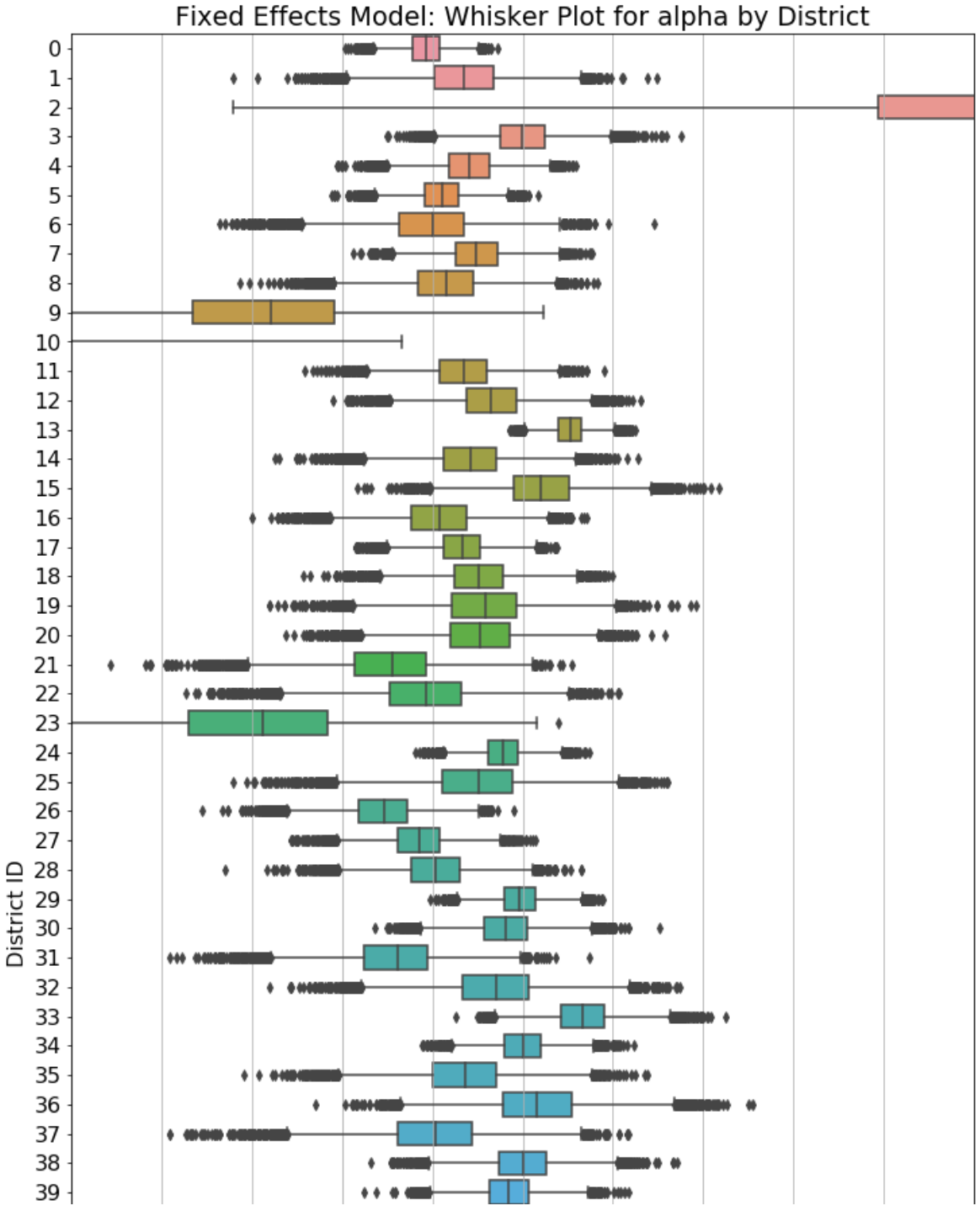


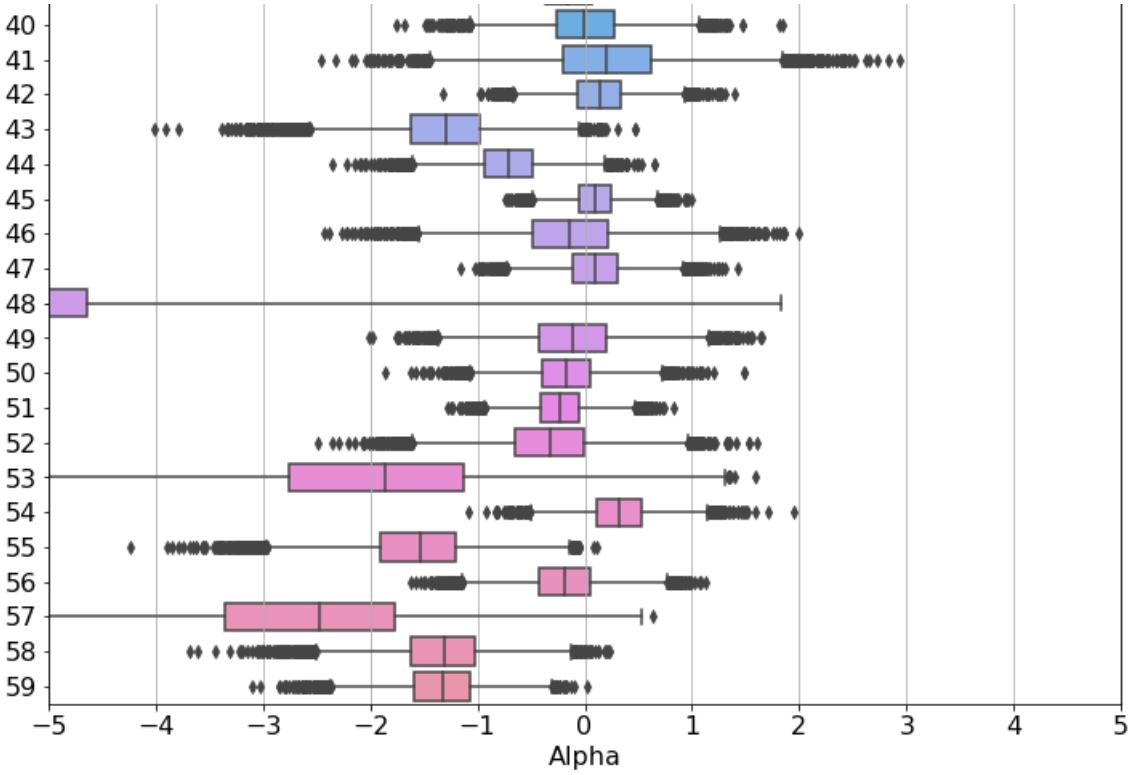

With 60 variables (one for each district) it can be a bit difficult to visualize what's going on. The autocorr plots look OK with very low correlations after the first handful of steps. Most of the alpha parameters for each district are clustered together. A few of them have very wide bases; these are the districts with small number of samples.

**Whisker Plot of Posterior Intercepts for Each District**

In [14]:
```
# Samples of alpha as a an Nx60 array
alpha_samples_fe = trace_fe.get_values('alpha_district')
# Arrange the alpha samples into a dataframe for plotting
col_names_fe = [f'alpha_{i}' for i in range(num_districts)]
df_alpha_samples_fe = pd.DataFrame(data=alpha_samples_fe, columns = col_names_fe)

# Generate a whisker plot
mpl.rcParams.update({'font.size': 16})
fig, ax = plt.subplots(figsize=[12,24])
ax.set_title('Fixed Effects Model: Whisker Plot for alpha by District')
ax.set_xlabel('Alpha')
ax.set_ylabel('District ID')
ax.set_xlim(-5, 5)
ax.set_xticks(np.arange(-5,6))
ax.grid()
sns.boxplot(data=alpha_samples_fe, orient='h', ax=ax)
```

Out[14]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1553c977c88>
```






Unlike the earlier plots, this one is very readable. It shows the alpha parameter (intercept) for each district. A high alpha means women in that district are more likely to use contraception.

**Forest Plot for Fixed Effects Model**

(omitted in the interest of brevity; essentially the same as the plot above. All RHat close to 1.)

In [17]:
```
fig = plt.figure(figsize=(12,20))
gs = pm.forestplot(trace_fe)
gs.figure = fig
ax1, ax2 = fig.axes
ax1 = ax1.set_xlim(0.8, 1.2)
plt.close(fig)
```

**Tests for Fixed Effects Model**

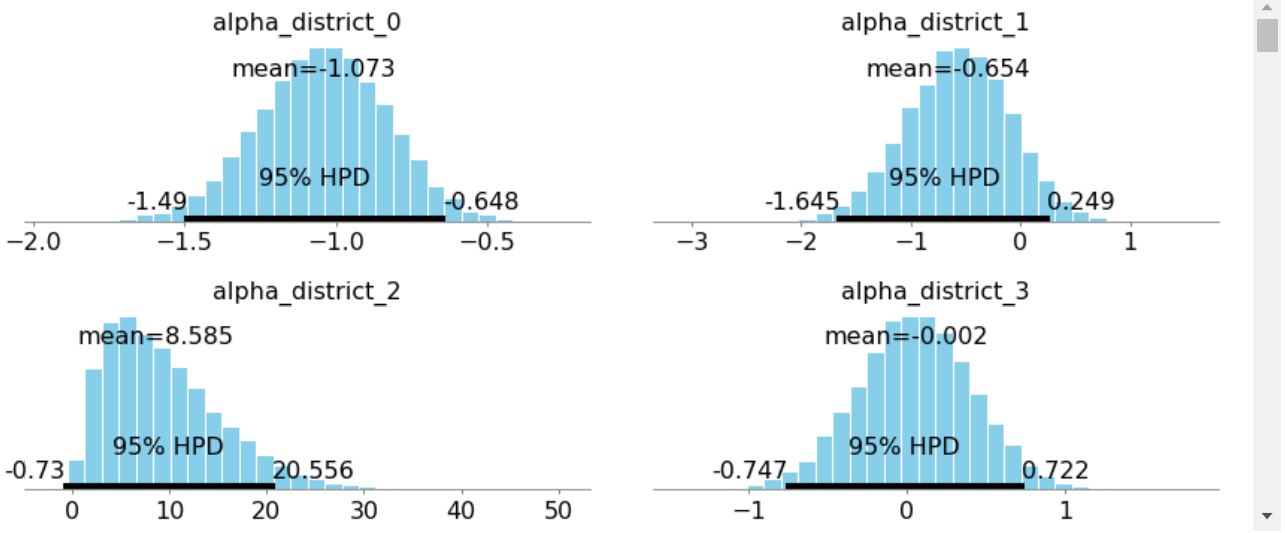
In [18]:
```
# Check n_eff, the number of effective draws
n_eff = summary_fe['n_eff']
n_eff_mean = np.mean(n_eff)
n_eff_min = np.min(n_eff)
print(f'Mean of n_eff for {num_districts} alpha parameters is {n_eff_mean:0.0f}')
print(f'Min of n_eff is {n_eff_min:0.0f}')

# Check Rhat, the correlation between of parameter estimates between chains
rhat = summary_fe['Rhat']
rhat_mean = np.mean(rhat)
rhat_min = np.min(rhat)
print(f'\nMean of RHat for {num_districts} alpha parameters is {rhat_mean:0.3f}')
print(f'Min of Rhat is {rhat_min:0.3f}')
```

```
Mean of n_eff for 60 alpha parameters is 30700
Min of n_eff is 12347

Mean of RHat for 60 alpha parameters is 1.000
Min of Rhat is 1.000
```

The number of effective draws is high and the correlations between chains are good. These results seem fine, and don't indicate any sampling problems. Also there were no warnings from the samplers.

**Plot Posterior for Fixed Effects Model: alpha for Each District**

In [19]:
```
fig = pm.plot_posterior(trace_fe)
```




**Summarize the Posteriors for Fixed Effects Model**

We can start summarizing the posteriors with the table displayed well above with summary statistics for each parameter. It's a lot to take in because there are 60 districts. Let's do some higher level summaries below.

In [20]:
```
# Compute the weighted average of the mean and sd of alpha
# weighting is by the number of women in each district
alpha_mean_wtd = np.average(a=summary_fe['mean'], weights=district_count)
alpha_sd_wtd = np.average(a=summary_fe['sd'], weights=district_count)

print(f'Weighted average of mean and sd alpha, weighted by number of women in each district:')
print(f'alpha_mean_wtd: {alpha_mean_wtd:0.4f}')
print(f'alpha_sd_wtd:    {alpha_sd_wtd:0.4f}')
```

```
Weighted average of mean and sd alpha, weighted by number of women in each district:
alpha_mean_wtd: -0.6052
alpha_sd_wtd:    0.4454
```

This is telling us that overall, women in this survey are more likely not to use contraception (mean alpha weighted by size is negative). The size of the negative alpha is somewhat larger than the weighted average standard deviation, so the effect has a reasonable size. This lines up with the overall mean rate of contraception use in the surve, 39.25%.

**Why should there not be any overall intercept in this model?**

This model fits a parameter $\alpha$ to each district the determines the probability $p$ that women in the district use contraception according to a logistic distribution. Adding an overall intercept would just create a new parameter that was colinear with the 60 alphas, one for each district. It would be redundant. It could create sampling problems due to misindetification, since a model where the overall intercept was 0.1 higher and all the $alpha_i$ were 0.1 lower would be the same. It's worth pointing out here that if the priors on the individual districts were grouped more tightly around zero, while the prior for the overall rate was wider, then this model specification might have some benefits.

**A2** Fit a multi-level "varying-effects" model with an overall intercept `alpha`, and district-specific intercepts `alpha_district`. Assume that the overall intercept has a Normal(0, 10) prior, while the district specific intercepts are all drawn from the **same** normal distribution with mean 0 and standard deviation $\sigma$. Let $\sigma$ be drawn from HalfCauchy(2). The setup of this model is similar to the per-chimanzee models in the prosocial chimanzee labs.

**Specify the Variable Effects Model and Draw Samples**

In [21]:
```
# Define the varying-effects model
with pm.Model() as model_ve:
    # Set the prior for the overall intercept
    alpha = pm.Normal(name='alpha', mu=0.0, sd=10.0)
    # Set the width sigma for the variability among districts
    sigma = pm.HalfCauchy(name='sigma', beta=2.0)
    # Set the district-specific alphas to have mean 0 and standard deviation sigma
    alpha_district = pm.Normal(name='alpha_district', mu=0.0, sd=sigma, shape=num_districts)
    # Set the probability that each woman uses contraception in this model
    # It depends only on the district she lives in
    p = pm.math.invlogit(alpha + alpha_district[df.district_id])
    # The response variable - whether this woman used contraception; modeled as Bernoulli
    # Bind this to the observed values
    use_contraception = pm.Bernoulli('use_contraception', p=p, observed=df['use_contraception'])

# Sample from the variable-effects model
try:
    trace_ve = vartbl['trace_ve']
    print(f'Loaded samples for the Variable Effects model in trace_ve.')
except:
    with model_ve:
        trace_ve = pm.sample(draws=num_samples, tune=num_tune, chains=2, cores=1)
    vartbl['trace_ve'] = trace_ve
    save_vartbl(vartbl, fname)
```

```
Loaded samples for the Variable Effects model in trace_ve.
```

**Tabular Summary of the 60 alpha Parameters for Variable Effects Model**

In [22]:
```
summary_ve = pm.summary(trace_ve)
display(summary_ve)
```

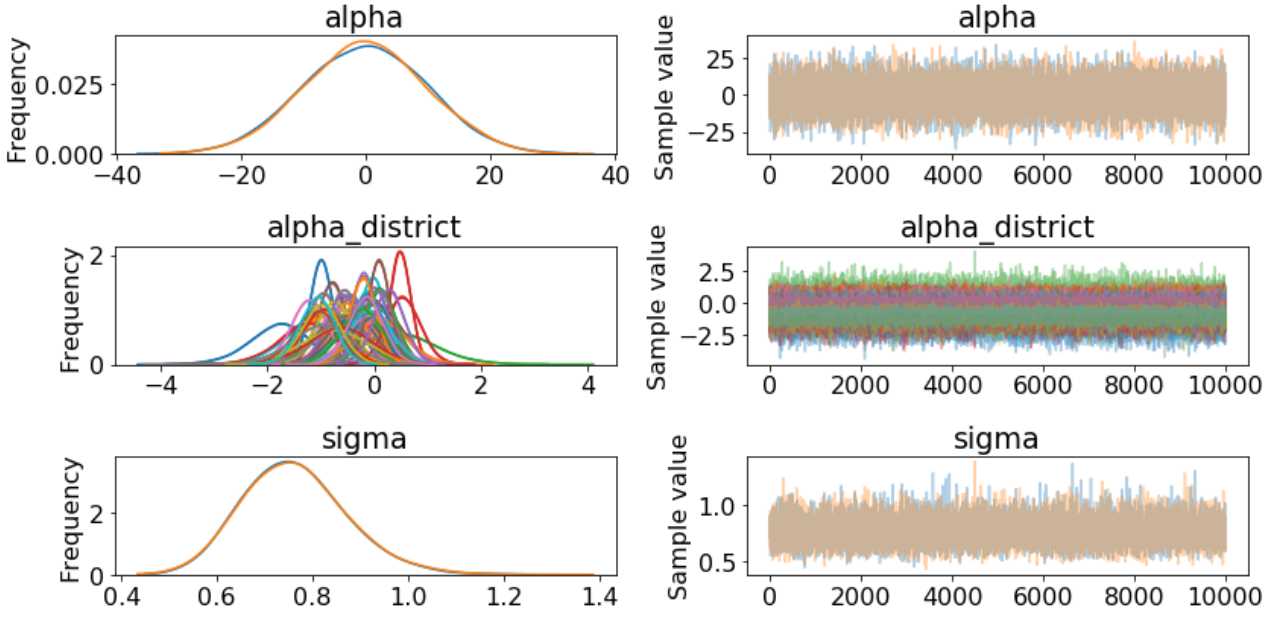
| | mean | sd | mc_error | hpd_2.5 | hpd_97.5 | n_eff | Rhat |
|---|---|---|---|---|---|---|---|
| **alpha** | -0.013008 | 9.872940 | 0.045847 | -19.250951 | 19.169345 | 40243.591760 | 0.999967 |
| **alpha_district__0** | -0.995859 | 0.202724 | 0.000964 | -1.399808 | -0.605373 | 34234.231612 | 0.999974 |
| **alpha_district__1** | -0.460104 | 0.400631 | 0.002094 | -1.245528 | 0.317751 | 36691.616480 | 0.999952 |
| **alpha_district__2** | 0.458601 | 0.699263 | 0.003785 | -0.915017 | 1.829906 | 28456.576307 | 0.999952 |
| **alpha_district__3** | -0.000413 | 0.334574 | 0.001954 | -0.657489 | 0.646902 | 29213.017921 | 0.999984 |
| **alpha_district__4** | -0.490959 | 0.308488 | 0.001472 | -1.102338 | 0.106186 | 33418.817333 | 0.999963 |
| **alpha_district__5** | -0.791990 | 0.258606 | 0.001385 | -1.308512 | -0.300280 | 28913.952031 | 0.999950 |
| **alpha_district__6** | -0.671257 | 0.425008 | 0.002076 | -1.504639 | 0.145123 | 34820.741801 | 0.999957 |
| **alpha_district__7** | -0.424509 | 0.311786 | 0.001876 | -1.025052 | 0.190891 | 33571.355573 | 0.999951 |
| **alpha_district__8** | -0.627072 | 0.389613 | 0.001998 | -1.403664 | 0.117607 | 36672.685295 | 0.999970 |
| **alpha_district__9** | -1.211693 | 0.522672 | 0.003134 | -2.282898 | -0.228779 | 27661.364230 | 0.999952 |
| **alpha_district__10** | -1.810997 | 0.532184 | 0.003603 | -2.887721 | -0.811196 | 18996.796170 | 0.999950 |
| **alpha_district__11** | -0.516405 | 0.348909 | 0.001606 | -1.185893 | 0.187040 | 37978.421475 | 0.999977 |
| **alpha_district__12** | -0.264061 | 0.364659 | 0.001894 | -0.989346 | 0.452644 | 43604.646136 | 0.999959 |
| **alpha_district__13** | 0.492100 | 0.187315 | 0.000922 | 0.127806 | 0.860685 | 38496.503805 | 0.999956 |
| **alpha_district__14** | -0.426260 | 0.389873 | 0.002220 | -1.188792 | 0.346780 | 33855.970800 | 1.000013 |
| **alpha_district__15** | 0.154725 | 0.392402 | 0.002125 | -0.623356 | 0.917347 | 36417.801484 | 0.999966 |
| **alpha_district__16** | -0.676254 | 0.383636 | 0.001929 | -1.407525 | 0.090589 | 35766.492929 | 0.999979 |
| **alpha_district__17** | -0.576369 | 0.285970 | 0.001322 | -1.140106 | -0.026561 | 32472.212659 | 0.999952 |
| **alpha_district__18** | -0.377680 | 0.362187 | 0.001793 | -1.075512 | 0.347678 | 35564.955879 | 0.999961 |
| **alpha_district__19** | -0.278198 | 0.434049 | 0.002562 | -1.108620 | 0.585324 | 30448.782382 | 0.999951 |
| **alpha_district__20** | -0.331641 | 0.407117 | 0.002101 | -1.129164 | 0.468578 | 35607.468056 | 1.000025 |
| **alpha_district__21** | -0.954426 | 0.432523 | 0.002463 | -1.764271 | -0.071759 | 32302.063943 | 0.999950 |
| **alpha_district__22** | -0.670007 | 0.447257 | 0.002240 | -1.577739 | 0.180708 | 36631.281770 | 0.999972 |
| **alpha_district__23** | -1.271709 | 0.522761 | 0.003427 | -2.327387 | -0.281375 | 30299.221712 | 0.999991 |
| **alpha_district__24** | -0.193962 | 0.235898 | 0.001165 | -0.656639 | 0.274792 | 39004.834319 | 0.999956 |
| **alpha_district__25** | -0.305833 | 0.460340 | 0.002290 | -1.210612 | 0.608669 | 37871.682815 | 0.999951 |
| **alpha_district__26** | -1.227418 | 0.330113 | 0.001731 | -1.877074 | -0.585072 | 33714.678848 | 0.999952 |
| **alpha_district__27** | -0.963022 | 0.299149 | 0.001577 | -1.554722 | -0.392403 | 36120.701260 | 0.999977 |
| **alpha_district__28** | -0.757881 | 0.340958 | 0.001863 | -1.452146 | -0.113642 | 34038.101425 | 0.999963 |
| **...** | ... | ... | ... | ... | ... | ... | ... |
| **alpha_district__31** | -0.976669 | 0.406058 | 0.002153 | -1.799018 | -0.222100 | 31704.699222 | 0.999961 |
| **alpha_district__32** | -0.200136 | 0.444076 | 0.002315 | -1.046213 | 0.705311 | 36710.864803 | 0.999968 |
| **alpha_district__33** | 0.544897 | 0.320400 | 0.001848 | -0.094215 | 1.168404 | 32718.151422 | 0.999970 |
| **alpha_district__34** | 0.000747 | 0.272759 | 0.001173 | -0.522022 | 0.545003 | 41129.671852 | 0.999950 |
| **alpha_district__35** | -0.431789 | 0.418762 | 0.002008 | -1.225337 | 0.409218 | 36396.351451 | 0.999985 |
| **alpha_district__36** | 0.100508 | 0.449867 | 0.002498 | -0.800554 | 0.962661 | 38862.824363 | 0.999957 |
| **alpha_district__37** | -0.596476 | 0.461522 | 0.002636 | -1.526369 | 0.285592 | 32298.420446 | 0.999988 |
| **alpha_district__38** | 0.001629 | 0.349487 | 0.001491 | -0.666923 | 0.698767 | 41899.109216 | 0.999970 |
| **alpha_district__39** | -0.124893 | 0.294651 | 0.001426 | -0.703515 | 0.447086 | 48668.201158 | 0.999953 |
| **alpha_district__40** | -0.001325 | 0.352741 | 0.001782 | -0.685500 | 0.698786 | 38980.242048 | 0.999957 |
| **alpha_district__41** | 0.115852 | 0.478585 | 0.002525 | -0.823661 | 1.050638 | 35935.441491 | 0.999964 |
| **alpha_district__42** | 0.119539 | 0.284853 | 0.001680 | -0.423860 | 0.693838 | 33330.964306 | 0.999957 |

| | mean | sd | mc_error | hpd_2.5 | hpd_97.5 | n_eff | Rhat |
|---|---|---|---|---|---|---|---|
| **alpha_district__43** | -0.949205 | 0.382961 | 0.002227 | -1.708485 | -0.208855 | 33049.893940 | 0.999951 |
| **alpha_district__44** | -0.585820 | 0.307506 | 0.001617 | -1.191694 | 0.004671 | 40146.670552 | 0.999974 |
| **alpha_district__45** | 0.088737 | 0.207356 | 0.000955 | -0.315485 | 0.495706 | 42394.759884 | 1.000012 |
| **alpha_district__46** | -0.087045 | 0.434842 | 0.002245 | -0.977093 | 0.741899 | 31689.674001 | 0.999966 |
| **alpha_district__47** | 0.085381 | 0.286674 | 0.001649 | -0.494807 | 0.634961 | 34762.977861 | 0.999995 |
| **alpha_district__48** | -0.767939 | 0.648454 | 0.003754 | -2.063064 | 0.490230 | 33027.371944 | 0.999960 |
| **alpha_district__49** | -0.077039 | 0.400108 | 0.001814 | -0.827295 | 0.736078 | 41667.872859 | 0.999976 |
| **alpha_district__50** | -0.137299 | 0.306978 | 0.001814 | -0.743466 | 0.458023 | 37365.513748 | 0.999962 |
| **alpha_district__51** | -0.206061 | 0.245690 | 0.001187 | -0.668202 | 0.296731 | 39558.749565 | 0.999957 |
| **alpha_district__52** | -0.234066 | 0.400111 | 0.001717 | -1.025082 | 0.546661 | 35740.407665 | 0.999952 |
| **alpha_district__53** | -0.642421 | 0.585532 | 0.003230 | -1.785063 | 0.504619 | 33379.230865 | 0.999960 |
| **alpha_district__54** | 0.275826 | 0.284950 | 0.001381 | -0.280436 | 0.845610 | 40129.381499 | 1.000002 |
| **alpha_district__55** | -1.095332 | 0.394145 | 0.002619 | -1.889586 | -0.351500 | 28558.537480 | 0.999952 |
| **alpha_district__56** | -0.152697 | 0.319590 | 0.001653 | -0.783120 | 0.469460 | 35394.482234 | 0.999956 |
| **alpha_district__57** | -1.020608 | 0.541647 | 0.003558 | -2.091876 | 0.024304 | 23701.374135 | 0.999950 |
| **alpha_district__58** | -1.001592 | 0.358599 | 0.002191 | -1.714683 | -0.300242 | 31250.678322 | 1.000016 |
| **alpha_district__59** | -1.073485 | 0.322751 | 0.001764 | -1.731599 | -0.459906 | 34195.219883 | 0.999950 |
| **sigma** | 0.763029 | 0.107300 | 0.001075 | 0.566081 | 0.982118 | 11150.713402 | 0.999951 |

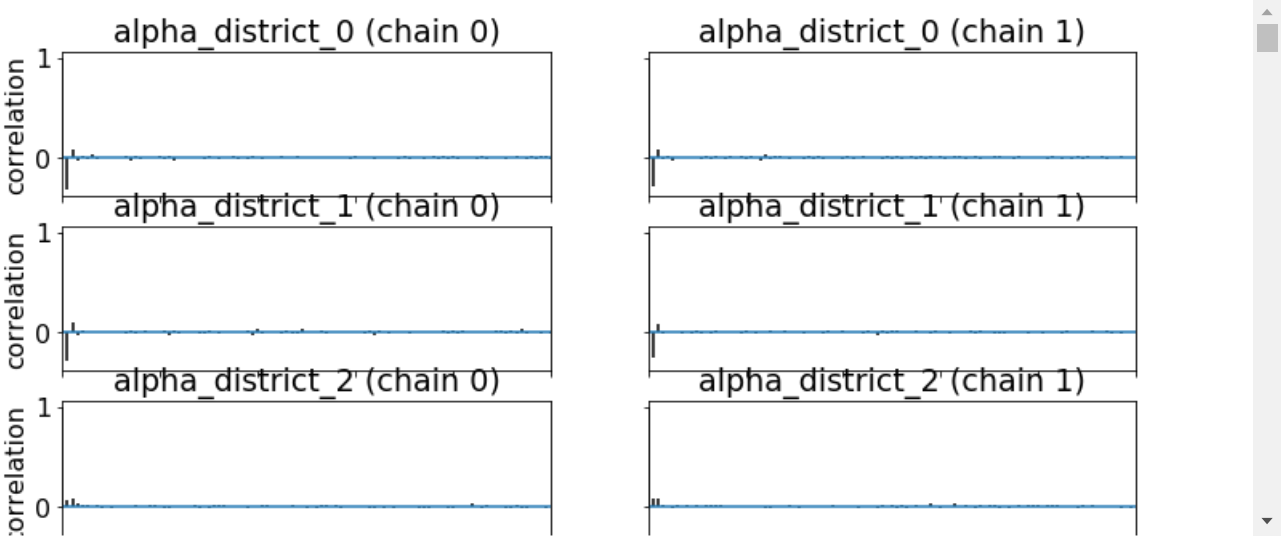
62 rows × 7 columns

**Plot Graphs for Fixed Effects Model: Traceplot, Autocorrplot**

In [23]:
```
fig = pm.traceplot(trace_ve)
```



In [24]:
```
fig = pm.autocorrplot(trace_fe)
```

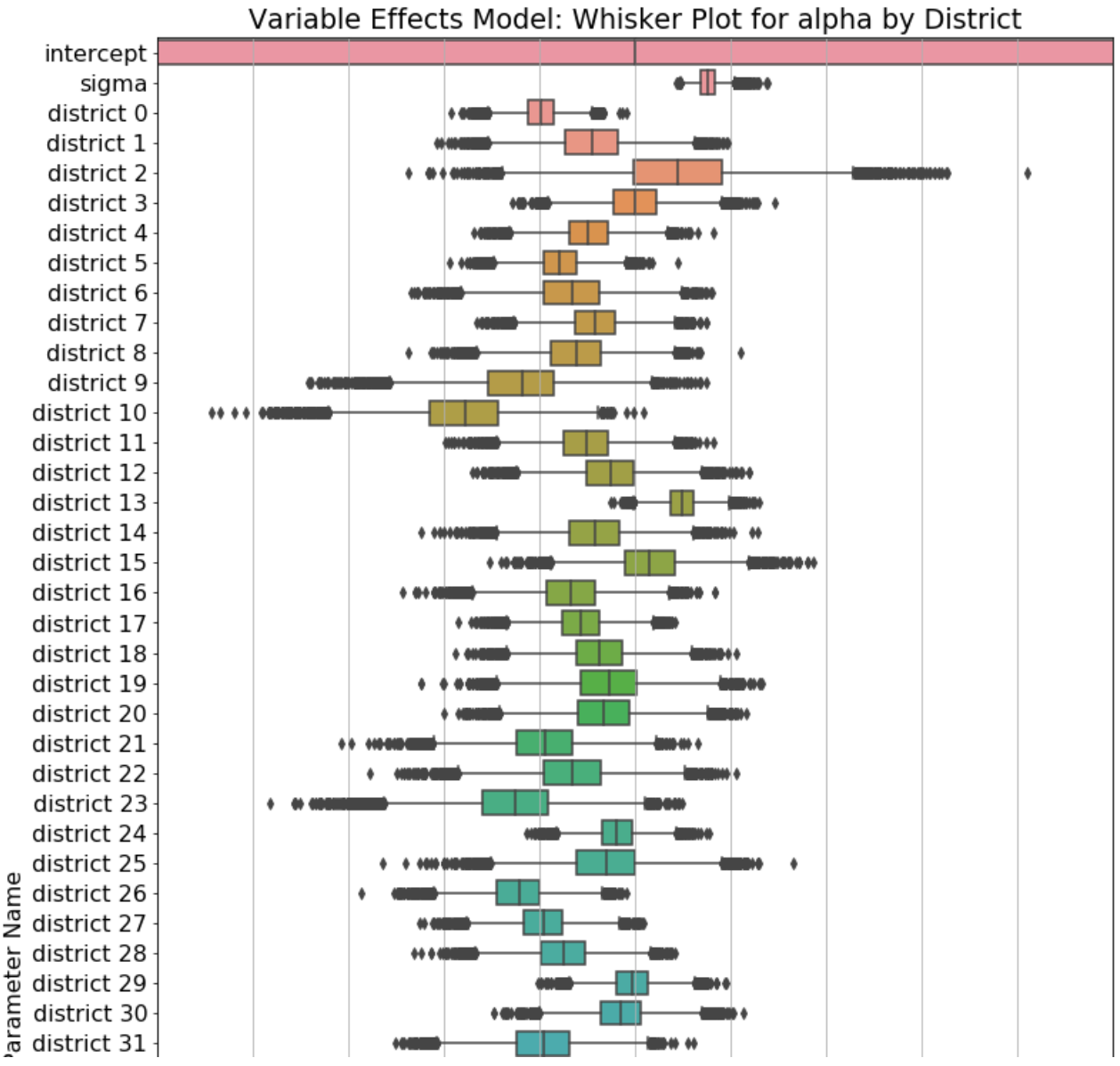


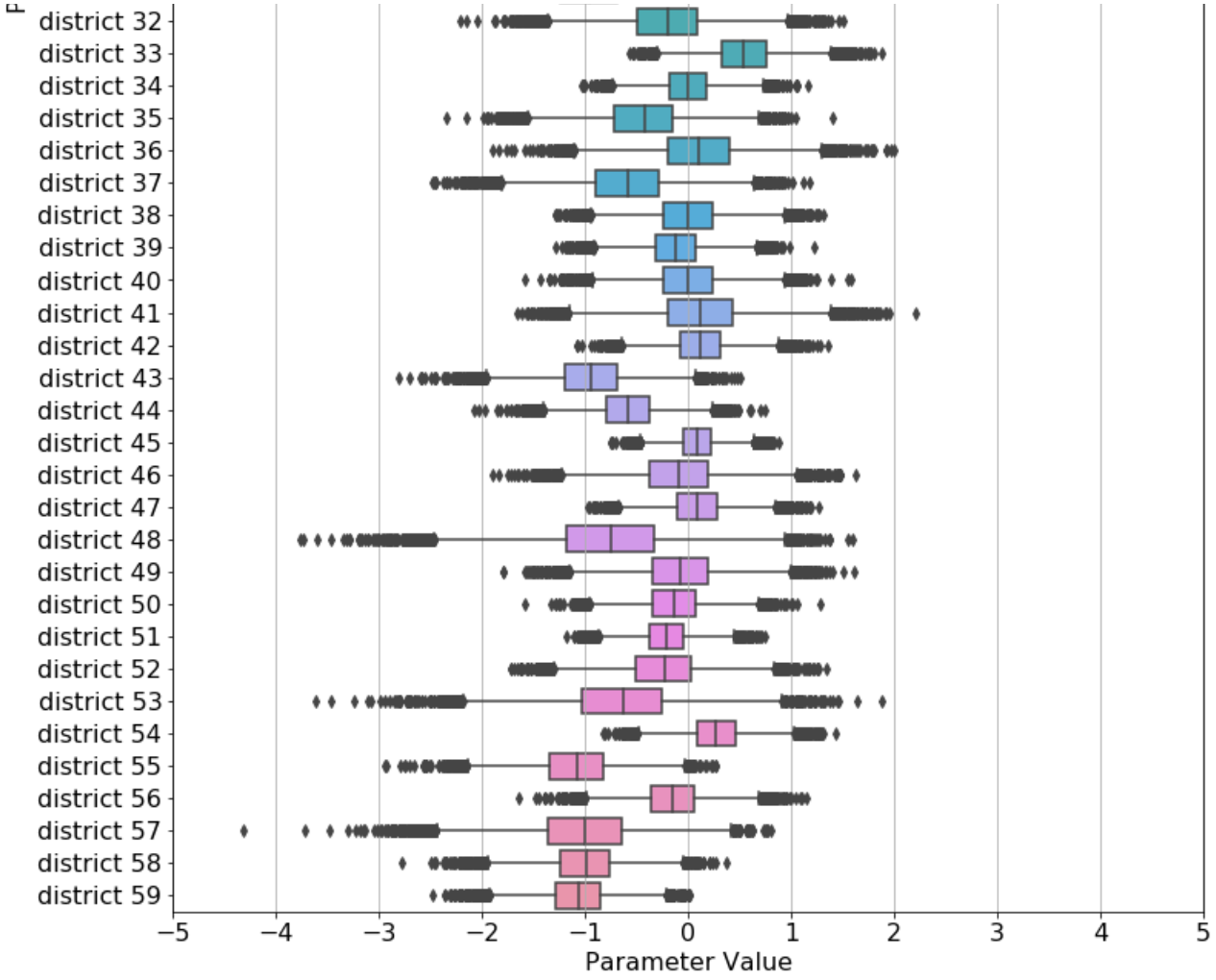

The additional structure of the variable effects model makes it easier to visualize. We can see that the alpha and sigma parameters have reasonable looking distributions that are consistent on the two chains. The traceplots look like pure white noise. We can see more of a pattern in the individual districts, where the center of each district reflects overall contraception usage in the district and the width reflects the number of women polled.

**Whisker Plot of Posterior Intercepts for Each District in Variable Effects Model**

In [25]:
```
# Samples of alpha as an Nx60 array
alpha_overall_ve = trace_ve.get_values('alpha')
sigma_ve = trace_ve.get_values('sigma')
alpha_district_samples_ve = trace_ve.get_values('alpha_district')
alpha_samples_ve = np.hstack([alpha_overall_ve.reshape(-1,1),
                              sigma_ve.reshape(-1,1),
                              alpha_district_samples_ve])
y_labels = ['intercept', 'sigma'] + [f'district {i}' for i in range(num_districts)]

# Arrange the alpha samples into a dataframe for plotting
col_names_ve = ['alpha', 'sigma'] + [f'alpha_{i}' for i in range(num_districts)]
df_alpha_samples_ve = pd.DataFrame(data=alpha_samples_ve, columns = col_names_ve)

# Generate a whisker plot
mpl.rcParams.update({'font.size': 16})
fig, ax = plt.subplots(figsize=[12,24])
ax.set_title('Variable Effects Model: Whisker Plot for alpha by District')
ax.set_xlabel('Parameter Value')
ax.set_ylabel('Parameter Name')
ax.set_xlim(-5, 5)
ax.set_xticks(np.arange(-5,6))
ax.grid()
sns.boxplot(data=alpha_samples_ve, orient='h', ax=ax)
ax.set_yticklabels(y_labels)
display(ax)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x1553e95d518>
```






One thing that jumps out here is that the pooled alpha (overall intercept of contraceptive usage) is quite tight compared to most of the districts. That's good! It shows that the information pooling is working and will help estimate the districts with small sample sizes. The estimate on $\sigma$, the standard deviation of the global intercept $\alpha$, is also quite tight.

**Forest Plot for Variable Effects Model**

(omitted in the interest of brevity; essentially the same as the plot above. All RHat close to 1.)

In [26]:
```
fig = plt.figure(figsize=(12,20))
gs = pm.forestplot(trace_ve)
gs.figure = fig
ax1, ax2 = fig.axes
ax1.set_xlim(0.8, 1.2)
plt.close(fig)
```

**Tests for Variable Effects Model**

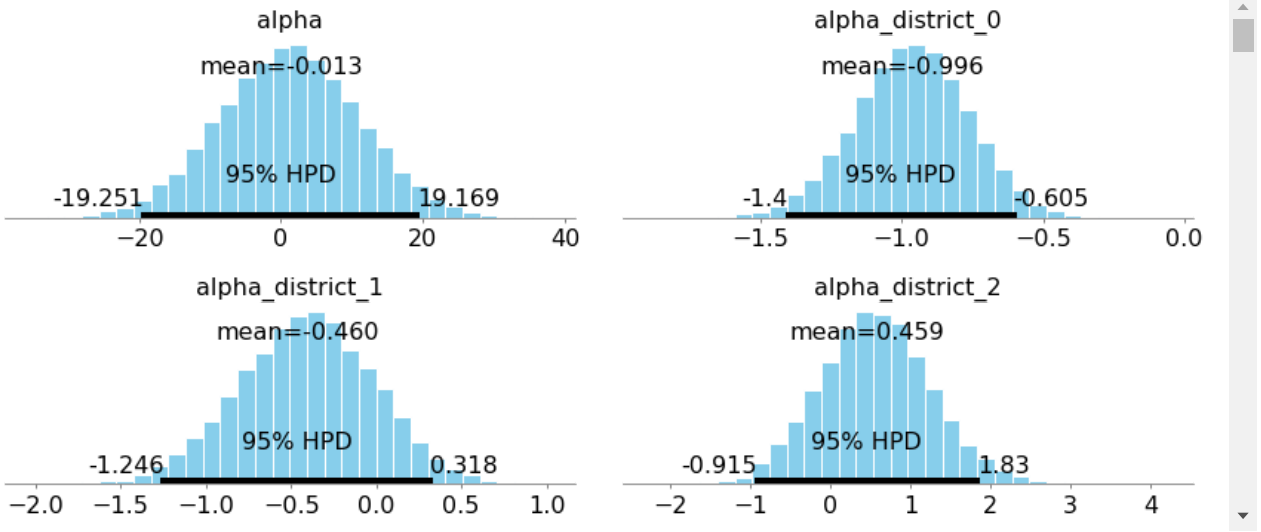
In [27]:
```
# Check n_eff, the number of effective draws
n_eff = summary_ve['n_eff']
n_eff_alpha = n_eff[0]
n_eff_sigma = n_eff[-1]
n_eff_alphas_mean = np.mean(n_eff[1:-1])
n_eff_alphas_min = np.min(n_eff[1:-1])
print(f'n_eff for overall intercept alpha: {n_eff_alpha:0.0f}')
print(f'n_eff for sigma (stdev of alpha):  {n_eff_sigma:0.0f}')
print(f'Mean of n_eff for {num_districts} alpha_district parameters is {n_eff_mean:0.0f}')
print(f'Min of n_eff is {n_eff_alphas_min:0.0f}')

# Check Rhat, the correlation between of parameter estimates between chains
rhat = summary_ve['Rhat']
rhat_alpha = rhat[0]
rhat_sigma = rhat[-1]
rhat_alphas_mean = np.mean(rhat[1:-1])
rhat_alphas_min = np.min(rhat[1:-1])
print(f'\nRHat for overall intercept alpha: {rhat_alpha:0.3f}')
print(f'RHat for sigma (stdev ov alpha):  {rhat_sigma:0.3f}')
print(f'Mean of RHat for {num_districts} alpha_district parameters is {rhat_mean:0.3f}')
print(f'Min of Rhat is {rhat_alphas_min:0.3f}')
```

```
n_eff for overall intercept alpha: 40244
n_eff for sigma (stdev of alpha):  11151
Mean of n_eff for 60 alpha_district parameters is 30700
Min of n_eff is 18997

RHat for overall intercept alpha: 1.000
RHat for sigma (stdev ov alpha):  1.000
Mean of RHat for 60 alpha_district parameters is 1.000
Min of Rhat is 1.000
```

The number of effective parameters for $\alpha$ and $\sigma$, describing the overall mean, are a bit smaller than for the district level parameters. But they are still quite high and completely adequate. The minimum for the districts is also OK. The RHat parameters are all very close to 1.0. These tests are all consistent with good sampling behavior.

**Plot Posterior for Variable Effects Model: alpha for Each District**

In [28]:
```
fig = pm.plot_posterior(trace_ve)
```




**Summarize the Posteriors for Variable Effects Model**

In [29]:
```
# Compute the weighted average of the mean and sd of alpha
# weighting is by the number of women in each district
alpha_mean = summary_ve['mean']['alpha']
alpha_std = summary_ve['sd']['alpha']
sigma_mean = summary_ve['mean']['sigma']
sigma_std = summary_ve['sd']['sigma']

print(f'alpha (overall intercept)')
print(f'mean: {alpha_mean:0.3f}')
print(f'std : {sigma_std:0.3f}')
print()
print(f'sigma (variability of districts)')
print(f'mean: {sigma_mean:0.3f}')
print(f'std : {sigma_std:0.3f}')

alphas_mean_wtd = np.average(a=summary_ve['mean'][1:-1], weights=district_count)
alphas_sd_wtd = np.average(a=summary_ve['sd'][1:-1], weights=district_count)

print(f'\nWeighted average of mean and std alpha by district, weighted by number of women in each distric
print(f'mean: {alphas_mean_wtd:0.4f}')
print(f'std:  {alphas_sd_wtd:0.4f}')
```

```
alpha (overall intercept)
mean: -0.013
std : 0.107

sigma (variability of districts)
mean: 0.763
std : 0.107

Weighted average of mean and std alpha by district, weighted by number of women in each district:
mean: -0.3718
std:  0.3156
```

The new model is easier to interpret. The global intercept $\alpha$ has a mean of -0.54 and standard deviation of -0.085. It reflects the overall prevalence of contraceptive use across districts.

$\sigma$ has a mean of 0.523 and a standard deviation of 0.085. It reflects how much individual district vary from the global intercept $\alpha$.

The estimates on both $\alpha$ and $\sigma$ are fairly tight, reflecting that the model has successfully pooled information across the 60 districts.

The mean of $\alpha_i$ over the districts is 0.073. It is close to zero because most of the overall departure from 50/50 contraceptive use is being absorbed in the global intercept $\alpha$.

**A3** What does a posterior-predictive sample in this model look like? What is the difference between district specific posterior predictives and woman specific posterior predictives. In other words, how might you model the posterior predictive for a new woman being from a particular district vs that of a new woman in the entire sample? This is a word answer; no programming required.

One posterior sample in this model assigns a global intercept $\alpha$, a global variability among districts $\sigma$, and draws $\alpha_i$ for the 60 districts. (This is the posterior over parameter space, not the posterior-predictive over y-space). The probability $p_i$ that women in a given district use contraceptive is the inverse logistic function of $\alpha + \alpha_i$.

We can then generate a posterior predictive sample by specifying the number of women in each of the 60 districts. The probability that one woman who lives in district $i$ uses contraception is $p_i$ as described above. The probability that $k_i$ of the $n_i$ women living in district $i$ will use contraception will be a binomial distribution with parameters $n_i$, $k_i$ and $p_i$, since the sum of Bernoulli trials is distributed binomially.

On the other hand, suppose we don't know anything about a woman and we want to predict the probability she uses contraception. We can follow a simple ancestral sampling strategy. First we model the probability that she lives in each of the districts, which we can do based on the overall population breakdown. Then we take a weighted average over the probabilities for women living in each district.

**A4** Plot the predicted proportions of women in each district using contraception against the id of the district, in both models. How

do these models disagree? Look at the extreme values of predicted contraceptive use in the fixed effects model. How is the disagreement in these cases?

In [30]:
```
# Generate posterior predictive samples in both models
num_samples_ppc: int = 10000
try:
    post_pred_fe = vartbl['post_pred_fe']
    post_pred_ve = vartbl['post_pred_ve']
    print(f'Loaded posterior predictive samples for Fixed Effects and Varying Effects models.')
except:
    post_pred_fe = pm.sample_ppc(trace_fe, samples=num_samples_ppc, model=model_fe)
    post_pred_ve = pm.sample_ppc(trace_ve, samples=num_samples_ppc, model=model_ve)
    vartbl['post_pred_fe'] = post_pred_fe
    vartbl['post_pred_ve'] = post_pred_ve
    save_vartbl(vartbl, fname)

# Compute the mean contraception use in posterior samples of each model
df['use_contraception_fe'] = np.mean(post_pred_fe['use_contraception'], axis=0)
df['use_contraception_ve'] = np.mean(post_pred_ve['use_contraception'], axis=0)

# Update the aggregated contraception use in each district
agg_tbl = {
        'woman': ['count'],
        'use_contraception': ['mean'],
        'use_contraception_fe': ['mean'],
        'use_contraception_ve': ['mean'],
        }
df_district = df.groupby(by=df.district_id).agg(agg_tbl)
df_district.columns = ["_".join(x) for x in df_district.columns.ravel()]
# Change column names to make model suffix at the end of the name
df_district.rename(axis='columns', inplace=True, mapper=
    {'use_contraception_fe_mean':'use_contraception_mean_fe',
     'use_contraception_ve_mean':'use_contraception_mean_ve',
     })
```

```
Loaded posterior predictive samples for Fixed Effects and Varying Effects models.
```

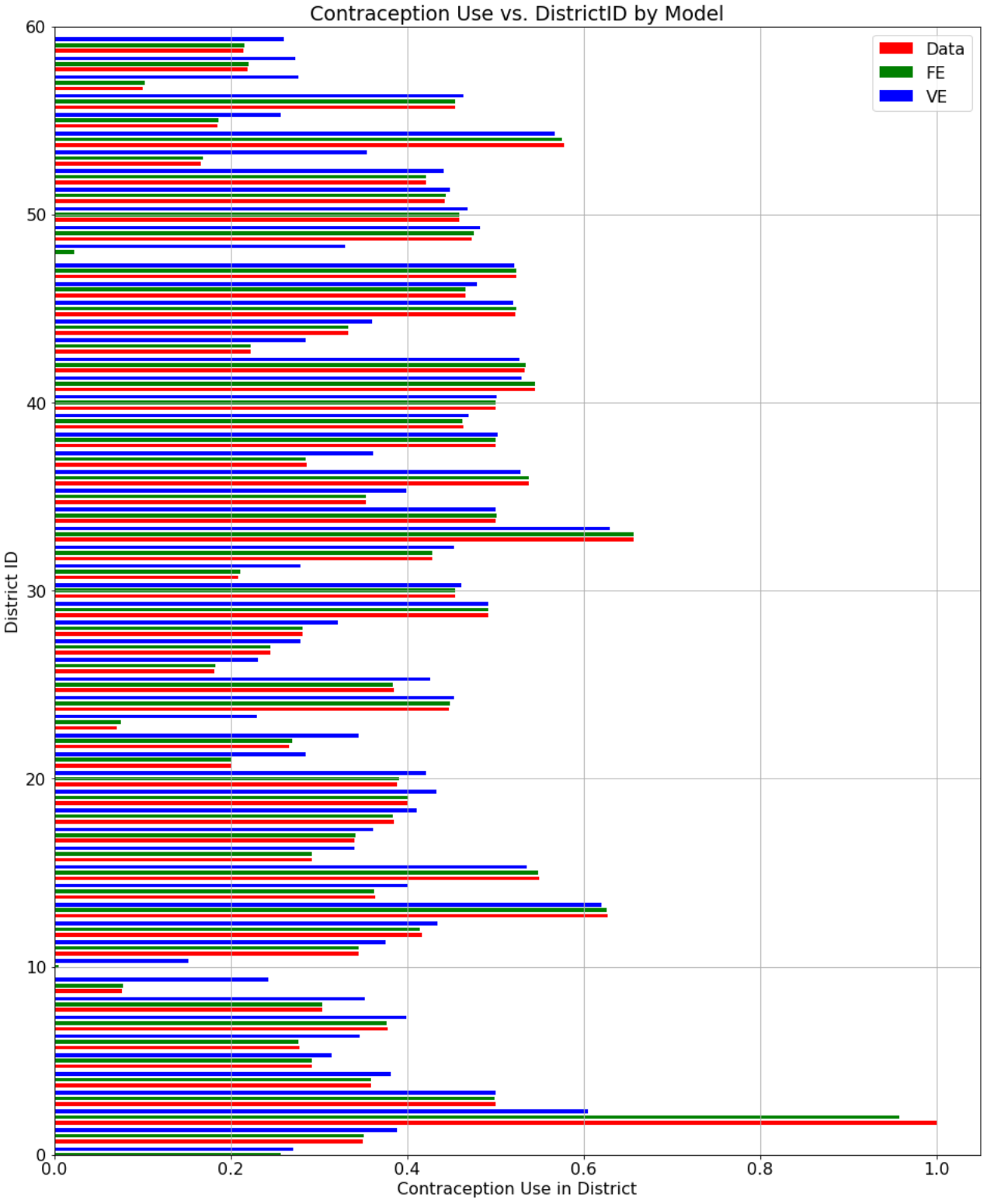
In [31]:
```
display(df_district)
```

| district_id | woman_count | use_contraception_mean | use_contraception_mean_fe | use_contraception_mean_ve |
|---|---|---|---|---|
| **0** | 117 | 0.256410 | 0.256603 | 0.271274 |
| **1** | 20 | 0.350000 | 0.350610 | 0.388650 |
| **2** | 2 | 1.000000 | 0.957450 | 0.604800 |
| **3** | 30 | 0.500000 | 0.498680 | 0.500870 |
| **4** | 39 | 0.358974 | 0.358795 | 0.382062 |
| **5** | 65 | 0.292308 | 0.292211 | 0.314469 |
| **6** | 18 | 0.277778 | 0.277306 | 0.345922 |
| **7** | 37 | 0.378378 | 0.377024 | 0.398819 |
| **8** | 23 | 0.304348 | 0.304013 | 0.352061 |
| **9** | 13 | 0.076923 | 0.078215 | 0.242192 |
| **10** | 21 | 0.000000 | 0.005157 | 0.152414 |
| **11** | 29 | 0.344828 | 0.345331 | 0.376066 |
| **12** | 24 | 0.416667 | 0.414862 | 0.434854 |
| **13** | 118 | 0.627119 | 0.626684 | 0.619803 |
| **14** | 22 | 0.363636 | 0.362486 | 0.399845 |
| **15** | 20 | 0.550000 | 0.548915 | 0.536000 |
| **16** | 24 | 0.291667 | 0.291879 | 0.340854 |
| **17** | 47 | 0.340426 | 0.341526 | 0.361632 |
| **18** | 26 | 0.384615 | 0.384027 | 0.411181 |
| **19** | 15 | 0.400000 | 0.401107 | 0.433873 |
| **20** | 18 | 0.388889 | 0.390728 | 0.421561 |
| **21** | 20 | 0.200000 | 0.200685 | 0.285665 |
| **22** | 15 | 0.266667 | 0.269360 | 0.344920 |
| **23** | 14 | 0.071429 | 0.075321 | 0.230036 |
| **24** | 67 | 0.447761 | 0.448328 | 0.453043 |
| **25** | 13 | 0.384615 | 0.383969 | 0.426738 |
| **26** | 44 | 0.181818 | 0.182530 | 0.231216 |
| **27** | 49 | 0.244898 | 0.244545 | 0.279210 |
| **28** | 32 | 0.281250 | 0.282037 | 0.321675 |
| **29** | 61 | 0.491803 | 0.492003 | 0.492344 |
| **30** | 33 | 0.454545 | 0.454770 | 0.461564 |
| **31** | 24 | 0.208333 | 0.210838 | 0.279283 |
| **32** | 14 | 0.428571 | 0.428457 | 0.453457 |
| **33** | 35 | 0.657143 | 0.657046 | 0.630226 |
| **34** | 48 | 0.500000 | 0.501165 | 0.499815 |
| **35** | 17 | 0.352941 | 0.353635 | 0.398906 |
| **36** | 13 | 0.538462 | 0.538538 | 0.528108 |
| **37** | 14 | 0.285714 | 0.285014 | 0.361900 |
| **38** | 26 | 0.500000 | 0.500800 | 0.502265 |
| **39** | 41 | 0.463415 | 0.463139 | 0.469215 |
| **40** | 26 | 0.500000 | 0.500392 | 0.501262 |
| **41** | 11 | 0.545455 | 0.544682 | 0.530109 |

| district_id | woman_count | use_contraception_mean | use_contraception_mean_fe | use_contraception_mean_ve |
|---|---|---|---|---|
| **42** | 45 | 0.533333 | 0.534696 | 0.527907 |
| **43** | 27 | 0.222222 | 0.222319 | 0.284741 |
| **44** | 39 | 0.333333 | 0.333882 | 0.360872 |
| **45** | 86 | 0.523256 | 0.524088 | 0.520448 |
| **46** | 15 | 0.466667 | 0.466180 | 0.478860 |
| **47** | 42 | 0.523810 | 0.524029 | 0.521467 |
| **48** | 4 | 0.000000 | 0.022275 | 0.330200 |
| **49** | 19 | 0.473684 | 0.475816 | 0.482363 |
| **50** | 37 | 0.459459 | 0.458803 | 0.468873 |
| **51** | 61 | 0.442623 | 0.444092 | 0.449075 |
| **52** | 19 | 0.421053 | 0.421353 | 0.442037 |
| **53** | 6 | 0.166667 | 0.169233 | 0.354767 |
| **54** | 45 | 0.577778 | 0.576091 | 0.566956 |
| **55** | 27 | 0.185185 | 0.186411 | 0.257126 |
| **56** | 33 | 0.454545 | 0.454330 | 0.463385 |
| **57** | 10 | 0.100000 | 0.103370 | 0.276540 |
| **58** | 32 | 0.218750 | 0.220003 | 0.273875 |
| **59** | 42 | 0.214286 | 0.215245 | 0.260029 |

In [32]:

```
# Set up horizontal bar chart
district_id_agg = df_district.index.values
# Spacing between models in each district
space = 0.3
plot_y1 = district_id_agg - space
plot_y2 = district_id_agg
plot_y3 = district_id_agg + space
# Height of each horizontal bar
height = 0.2

# Plot contraception use for each district
fig, ax = plt.subplots(figsize=[16,20])
ax.set_title('Contraception Use vs. DistrictID by Model')
ax.set_xlabel('Contraception Use in District')
ax.set_ylabel('District ID')
ax.set_ylim(0, 60)
ax.barh(y=plot_y1, width=df_district.use_contraception_mean, height=height, label='Data', color='r')
ax.barh(y=plot_y2, width=df_district.use_contraception_mean_fe, height=height, label='FE', color='g')
ax.barh(y=plot_y3, width=df_district.use_contraception_mean_ve, height=height, label='VE', color='b')
ax.legend()
ax.grid()
```




**How do these models disagree?**

**Look at the extreme values of predicted contraceptive use in the fixed effects model. How is the disagreement in these cases?**

In [33]:
```
# Get districts with min and max contraception use in the FE model
district_min_fe = np.argmin(df_district.use_contraception_mean_fe)
district_max_fe = np.argmax(df_district.use_contraception_mean_fe)
# Display results
print(f'District ID {district_min_fe} has Minimum Contraception in Fixed Effect Model')
display(df_district.loc[district_min_fe])
print(f'District ID {district_max_fe} has Maximum Contraception in Fixed Effect Model')
display(df_district.loc[district_max_fe])
```

```
District ID 10 has Minimum Contraception in Fixed Effect Model

woman_count                  21.000000
use_contraception_mean        0.000000
use_contraception_mean_fe     0.005157
use_contraception_mean_ve     0.152414
Name: 10, dtype: float64

District ID 2 has Maximum Contraception in Fixed Effect Model

woman_count                  2.00000
use_contraception_mean       1.00000
use_contraception_mean_fe    0.95745
use_contraception_mean_ve    0.60480
Name: 2, dtype: float64
```

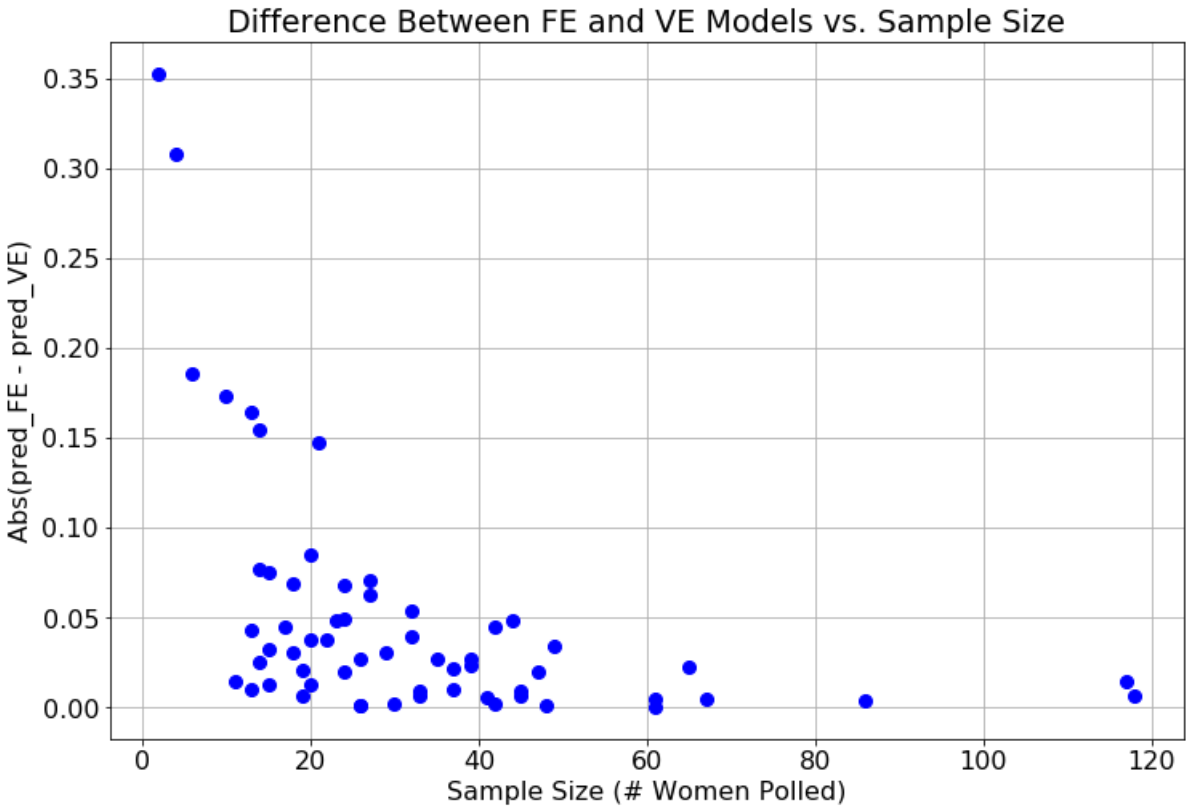
With 60 districts, it can hard to read off the district numbers, so we can cross reference the chart to to the table above. It's easiest to spot the districts where contraception use is high. District ID 2 has 100% contraceptive use in the data, but based on a tiny sample of 2 responedents. The fixed effects model has a much weaker prior and predicts that women in this district will use contraceptives 95.7% of the time. The varying effects model benefits from information pooling across all the districts and does far more shrinkage to the population mean for a district with such a small sample. It predicts 45.7% contraceptive use in this district.

District ID 10 has the lowest predicted contraception use among all 60 districts in the fixed effect model. The FE model predicts 0.5% contraception use, whereas the varying effect model predicts 18.2% contraception use. This case is more interesting than the last one. With 21 women in the sample count, and 0 of them reporting contraception use, it appears at first glance that in this case, the FE model is more accurate, and the VE model may be overcorrecting for the population mean.

In general, these two extreme cases give us a good idea for how the two models disagree. The FE model comes quite close to simply predicting that each district has a contraception use rate close to the observed rate in the data. This makes it vulnerable to making extreme predictions in districts that have a small number of responses. The VE model is pooling across all the districts, and performing shrikage to the population mean. It does more shrinkage in districts with a small response rate. The two models with vary the most in districts with a small response count that is far from the population mean.

**A5** Plot the absolute value of the difference in probability of contraceptive use against the number of women sampled in each district. What do you see?

In [34]:
```
# Assemble series for this plot
# the x-axis is the number of women in the district
plot_x = df_district.woman_count.values
# the y-axis is the absolute value of the difference between the FE and VE models
plot_y = np.abs(df_district.use_contraception_mean_fe - df_district.use_contraception_mean_ve).values

# Generate the plot
fig, ax = plt.subplots(figsize=[12,8])
ax.set_title('Difference Between FE and VE Models vs. Sample Size')
ax.set_xlabel('Sample Size (# Women Polled)')
ax.set_ylabel('Abs(pred_FE - pred_VE)')
ax.plot(plot_x, plot_y, color='b', marker='o', markersize=8, linewidth=0)
ax.grid()
```




As expected, the smaller the sample size, the larger the difference between the FE and VE models. This is because the VE model is performing more shrinkage on districts with a smaller sample size. We could have gotten a stronger regression fit if we added a second explanatory variable, the absolute value of the difference between the sample contraception use rate in each district and the overall population contraception use rate.

## Part B.

Let us now fit a model with both varying intercepts by `district_id` (like we did in the varying effects model above) and varying slopes of `urban` by `district_id`. To do this, we will

- (a) have an overall intercept, call it `alpha`
- (b) have an overall slope of `urban`, call it `beta`.
- (c) have district specific intercepts `alpha_district`
- (d) district specific slopes for `urban`, `beta_district`
- (e) model the co-relation between these slopes and intercepts.

We have not modelled covariance and correlation before, so look at http://am207.info/wiki/corr.html (http://am207.info/wiki/corr.html) for notes on how this is done.

To see the ideas behind this, see section 13.2.2 on the income data from your textbook (included as a pdf in this zip). Feel free to use code with attribution from Osvaldo Martin (https://github.com/aloctavodia/Statistical-Rethinking-with-Python-and-PyMC3/blob/master/Chp_13.ipynb)..with attribution and understanding...there is some sweet pymc3 technical wrangling in there.

**B1** Write down the model as a pymc3 specification and look at its graph. Note that this model builds a 60 by 2 matrix with `alpha_district` values in the first column and `beta_district` values in the second. By assumption, the first column and the second column have correlation structure given by an LKJ prior, but there is no explicit correlation among the rows. In other words, the correlation matrix is 2x2 (not 60x60). Make sure to obtain the value of the off-diagonal correlation as a `pm.Deterministic` . (See Osvaldo Martin's code above)

**B2**: Sample from the posterior of the model above *with a target acceptance rate of .9 or more*. (Sampling takes me 7 minutes 30 seconds on my 2013 Macbook Air). Comment on the quality of the samples obtained.

**B3** Propose a method based on the reparametrization trick for multi-variate gaussians) of improving the quality of the samples obtained and implement it. (A hint can be obtained from here: https://docs.pymc.io/api/distributions/multivariate.html#pymc3.distributions.multivariate.MvNormal (https://docs.pymc.io/api/distributions/multivariate.html#pymc3.distributions.multivariate.MvNormal) . Using that hint lowered the sampling time to 2.5 minutes on my laptop).

**B4** Inspect the trace of the correlation between the intercepts and slopes, plotting the correlation marginal. What does this correlation tell you about the pattern of contraceptive use in the sample? It might help to plot the mean (or median) varying effect estimates for both the intercepts and slopes, by district. Then you can visualize the correlation and maybe more easily think through what it means to have a particular correlation. Also plot the predicted proportion of women using contraception, with urban women on one axis and rural on the other. Finally, also plot the difference between urban and rural probabilities against rural probabilities. All of these will help you interpret your findings. (Hint: think in terms of low or high rural contraceptive use)

**B5** Add additional "slope" terms (one-by-one) into the model for

- (a) the centered-age of the women and
- (b) an indicator for whether the women have a small number or large number of existing kids in the house (you can treat 1-2 kids as low, 3-4 as high, but you might want to experiment with this split).

Are any of these effects significant? Are any significant effects similar over the urban/rural divide?

**B6** Use WAIC to compare your models. What are your conclusions?

**B1** Write down the model as a pymc3 specification and look at its graph. Note that this model builds a 60 by 2 matrix with `alpha_district` values in the first column and `beta_district` values in the second. By assumption, the first column and the second column have correlation structure given by an LKJ prior, but there is no explicit correlation among the rows. In other words, the correlation matrix is 2x2 (not 60x60). Make sure to obtain the value of the off-diagonal correlation as a `pm.Deterministic` . (See Osvaldo Martin's code above)

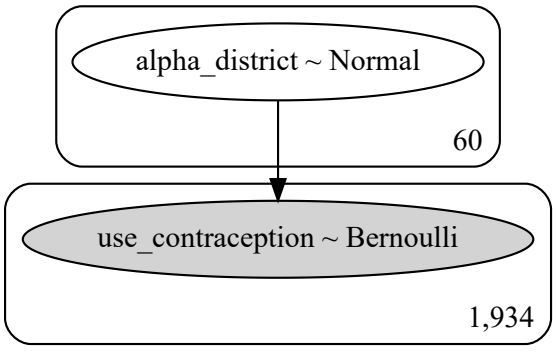
In [35]:
```
def pm_make_cov(sigma_priors, corr_coeffs, ndim):
    """Assemble a covariance matrix single variable standard deviations and correlation coefficients"""
    # Citation: AM 207 lecture notes: http://am207.info/wiki/corr.html
    # Diagonal matrix of standard deviation for each varialbes
    sigma_matrix = tt.nlinalg.diag(sigma_priors)
    # A symmetric nxn matrix has n choose 2 = n(n-1)/2 distinct elements
    n_elem = int(ndim * (ndim - 1) / 2)
    # Convert between array indexing and [i, j) indexing
    tri_index = np.zeros([ndim, ndim], dtype=int)
    tri_index[np.triu_indices(ndim, k=1)] = np.arange(n_elem)
    tri_index[np.triu_indices(ndim, k=1)[::-1]] = np.arange(n_elem)
    # Assemble the covariance matrix using the equation
    # CovMat = DiagMat * CorrMat * DiagMat
    corr_matrix = corr_coeffs[tri_index]
    corr_matrix = tt.fill_diagonal(corr_matrix, 1)
    return tt.nlinalg.matrix_dot(sigma_matrix, corr_matrix, sigma_matrix)
```

In [36]:
```
# Define a varying slopes model incorporating a beta_urban term
with pm.Model() as model_vs:
    # Set the prior for the overall intercept
    alpha = pm.Normal(name='alpha', mu=0.0, sd=10.0)
    # Set the prior for the overall intercept on urban, beta
    beta = pm.Normal(name='beta', mu=0.0, sd=10.0)

    # Citation: http://am207.info/wiki/corr.html for code controlling correlation structure
    # The parameter nu is the prior on correlation; 0 is uniform, infinity is no corelation
    nu = pm.Uniform('nu', 1.0, 5.0)
    # The number of dimensions here is 2: correlation structure is bewteen alpha and beta by district
    num_factors: int = 2
    # Sample the correlation coefficients using the LKJ distribution
    corr_coeffs = pm.LKJCorr('corr_coeffs', nu, num_factors)

    # Sample the variances of the single factors
    sigma_priors = tt.stack([pm.Lognormal('sigma_prior_alpha', mu=0.0, tau=1.0),
                             pm.Lognormal('sigma_prior_beta', mu=0.0, tau=1.0)])

    # Make the covariance matrix as a Theano tensor
    cov = pm.Deterministic('cov', pm_make_cov(sigma_priors, corr_coeffs, num_factors))
    # The multivariate Gaussian of (alpha, beta) by district
    theta_district = pm.MvNormal('theta_district', mu=[0.0, 0.0], cov=cov, shape=(num_districts, num_fact

    # The vector of standard deviations for each variable; size num_factors x num_factors
    # Citation: efficient generation of sigmas and rhos from cov
    # https://github.com/aloctavodia/Statistical-Rethinking-with-Python-and-PyMC3/blob/master/Chp_13.ipyn
    sigmas = pm.Deterministic('sigmas', tt.sqrt(tt.diag(cov)))
    # correlation matrix (num_factors x num_factors)
    rhos = pm.Deterministic('rhos', tt.diag(sigmas**-1).dot(cov.dot(tt.diag(sigmas**-1))))

    # Extract the standard deviations of alpha and beta, and the correlation coefficient rho
    sigma_alpha = pm.Deterministic('sigma_alpha', sigmas[0])
    sigma_beta = pm.Deterministic('sigma_beta', sigmas[1])
    rho = pm.Deterministic('rho', rhos[0, 0])

    # Extract alpha_district and beta_district from theta_district
    alpha_district = pm.Deterministic('alpha_district', theta_district[:,0])
    beta_district = pm.Deterministic('beta_district', theta_district[:, 1])

    # Set the probability that each woman uses contraception in this model
    # It depends on the district she lives in and whether the district is urban
    # p = pm.math.invlogit(alpha + alpha_district[df.district_id] +
    #                      (beta + beta_district[df.district_id]) * df.urban)
    p = pm.math.invlogit(alpha + theta_district[df.district_id, 0] +
                         (beta + theta_district[df.district_id, 1]) * df.urban)

    # The response variable - whether this woman used contraception; modeled as Bernoulli
    # Bind this to the observed values
    use_contraception = pm.Bernoulli('use_contraception', p=p, observed=df['use_contraception'])
```

In [37]:
```
# Generate graphs for each model
graph_fe = pm.model_to_graphviz(model_fe)
graph_ve = pm.model_to_graphviz(model_ve)
graph_vs = pm.model_to_graphviz(model_vs)
```

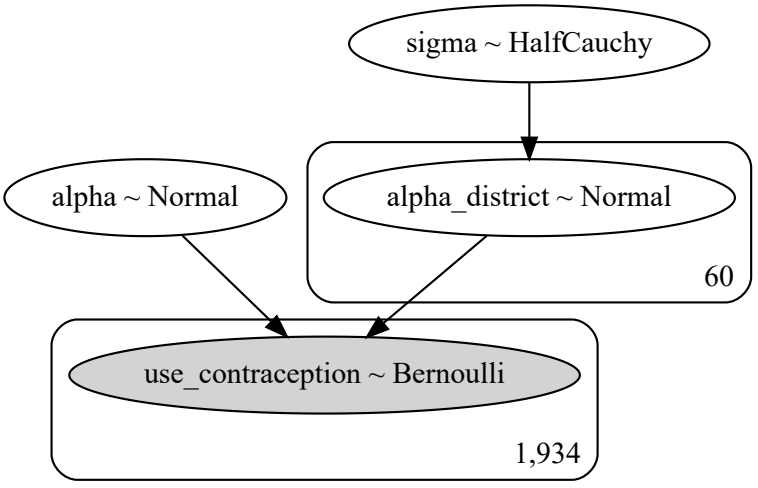
In [38]:
```
# The Fixed Effect model
print('Fixed Effect Model Graph')
display(graph_fe)
```

```
Fixed Effect Model Graph
```



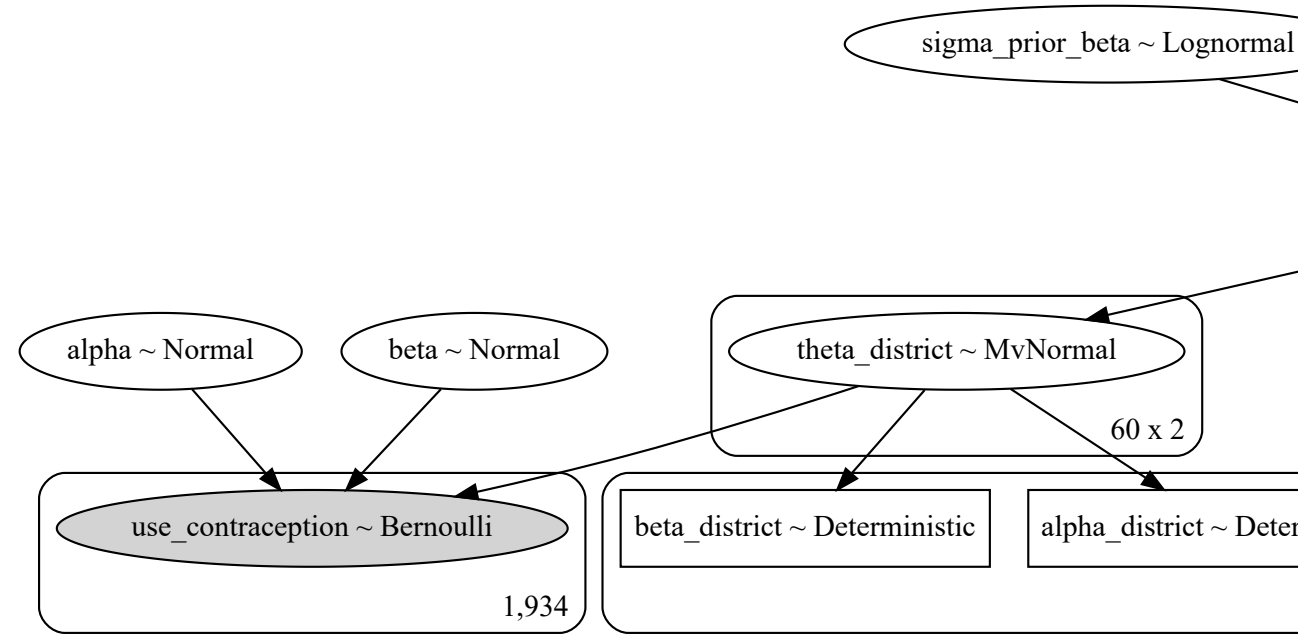

In [39]:
```
# The Variable Effect model
print('Variable Effect Model Graph')
display(graph_ve)
```

```
Variable Effect Model Graph
```



In [40]:
```
# The Variable Slope model
print('Variable Slope Model Graph')
display(graph_vs)
```

```
Variable Slope Model Graph
```




**B2**: Sample from the posterior of the model above *with a target acceptance rate of .9 or more*. (Sampling takes me 7 minutes 30 seconds on my 2013 Macbook Air). Comment on the quality of the samples obtained.

In [41]:
```
# Sample from the varying-slope model
try:
    trace_vs = vartbl['trace_vs']
    print(f'Loaded samples for the Variable Slopes model in trace_vs.')
except:
    with model_vs:
        nuts_kwargs = {'target_accept': 0.90}
        trace_vs = pm.sample(draws=num_samples, tune=num_tune, nuts_kwargs=nuts_kwargs, chains=2, cores=1
    vartbl['trace_vs'] = trace_vs
    save_vartbl(vartbl, fname)

# Summary of the varying-slope model
summary_vs = pm.summary(trace_vs)
# List of parameters to report
params_all = summary_vs.index.values
num_pairs = num_factors * (num_factors-1) // 2
# The "key" parameters
params_key = \
    ['alpha', 'beta', 'nu', 'sigma_prior_alpha', 'sigma_prior_beta'] + \
    [f'sigmas__{i}' for i in range(num_factors)] + \
    [f'corr_coeffs__{i}' for i in range(num_pairs)] + \
    [f'cov__{i}_{j}' for i in range(num_factors) for j in range(num_factors)]
# The parameters pertaining to different districts
params_district = \
        [f'alpha_district__{i}' for i in range(num_districts)] + \
        [f'beta_district__{i}' for i in range(num_districts)]
# The parameters to display
params = params_key + params_district
display(summary_vs.loc[params])
```

```
Loaded samples for the Variable Slopes model in trace_vs.
```

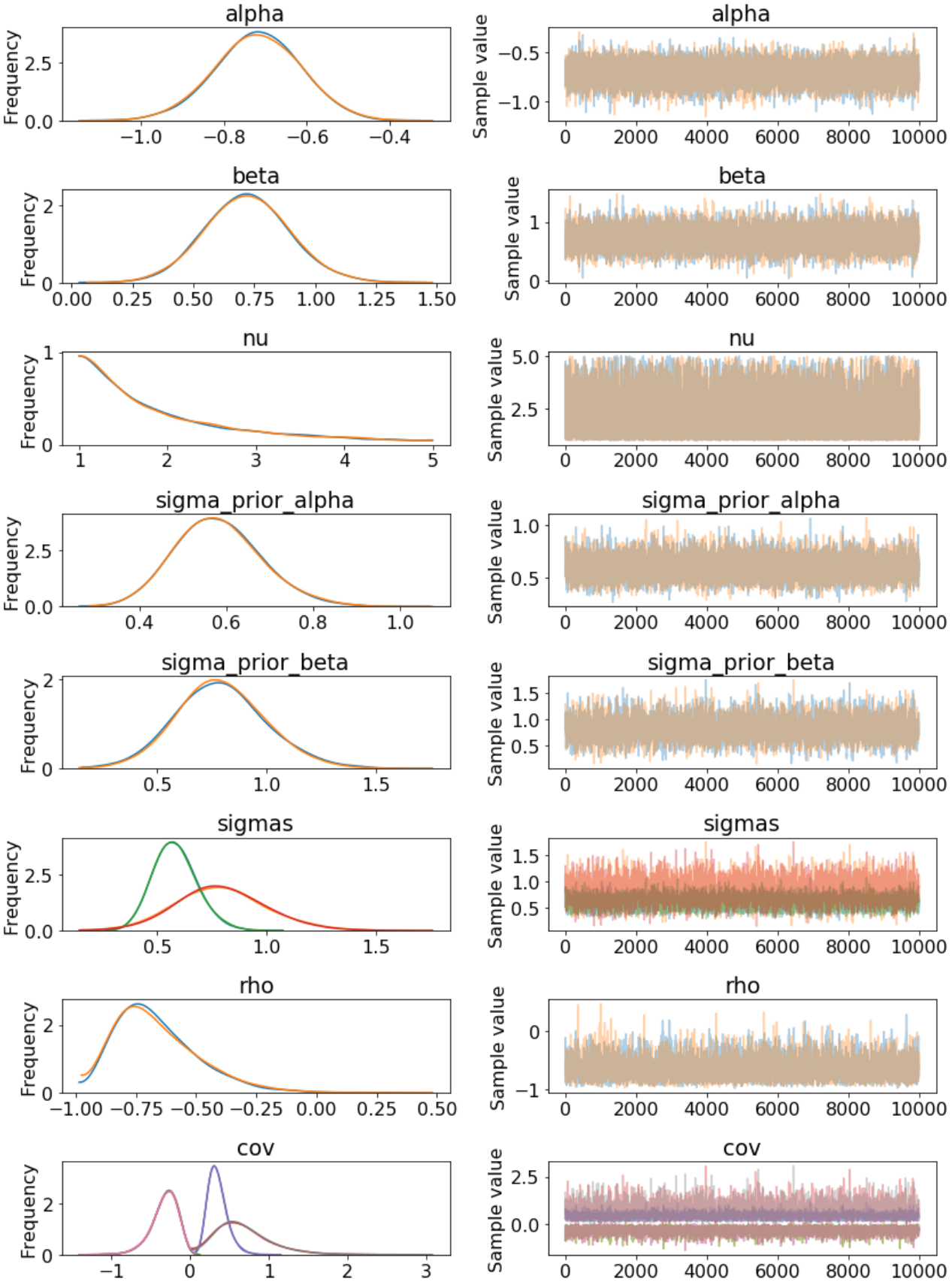
| | mean | sd | mc_error | hpd_2.5 | hpd_97.5 | n_eff | Rhat |
|---|---|---|---|---|---|---|---|
| **alpha** | -0.717558 | 0.102911 | 0.001004 | -0.919558 | -0.514400 | 11070.830381 | 0.999974 |
| **beta** | 0.722576 | 0.172503 | 0.001602 | 0.376777 | 1.057797 | 12368.340875 | 0.999967 |
| **nu** | 2.010216 | 0.951964 | 0.008052 | 1.000024 | 4.087116 | 15564.554455 | 0.999955 |
| **sigma_prior_alpha** | 0.582088 | 0.099569 | 0.001328 | 0.395724 | 0.784062 | 5962.864974 | 0.999959 |
| **sigma_prior_beta** | 0.793183 | 0.205469 | 0.003759 | 0.391791 | 1.205187 | 2617.369731 | 1.000001 |
| **sigmas__0** | 0.582088 | 0.099569 | 0.001328 | 0.395724 | 0.784062 | 5962.864974 | 0.999959 |
| **sigmas__1** | 0.793183 | 0.205469 | 0.003759 | 0.391791 | 1.205187 | 2617.369731 | 1.000001 |
| **corr_coeffs__0** | -0.663937 | 0.170751 | 0.002972 | -0.941034 | -0.326773 | 3635.092552 | 1.000013 |
| **cov__0_0** | 0.348740 | 0.120421 | 0.001532 | 0.139771 | 0.589955 | 6369.782184 | 0.999969 |
| **cov__0_1** | -0.324187 | 0.167470 | 0.002700 | -0.656332 | -0.033705 | 3872.849468 | 0.999966 |
| **cov__1_0** | -0.324187 | 0.167470 | 0.002700 | -0.656332 | -0.033705 | 3872.849468 | 0.999966 |
| **cov__1_1** | 0.671356 | 0.342191 | 0.005752 | 0.087396 | 1.345849 | 3022.422090 | 0.999955 |
| **alpha_district__0** | -0.829041 | 0.315595 | 0.002295 | -1.494327 | -0.253472 | 20592.375650 | 0.999953 |
| **alpha_district__1** | 0.045918 | 0.372946 | 0.002288 | -0.687816 | 0.772423 | 28473.249662 | 0.999951 |
| **alpha_district__2** | 0.021726 | 0.586856 | 0.003623 | -1.125060 | 1.181636 | 24677.923840 | 0.999952 |
| **alpha_district__3** | -0.151585 | 0.404644 | 0.003275 | -0.966043 | 0.624490 | 15983.702050 | 0.999957 |
| **alpha_district__4** | 0.066747 | 0.306789 | 0.001875 | -0.541859 | 0.657783 | 28399.490637 | 0.999970 |
| **alpha_district__5** | -0.335427 | 0.277162 | 0.001562 | -0.869563 | 0.213173 | 24583.712839 | 1.000003 |
| **alpha_district__6** | -0.140765 | 0.393180 | 0.002056 | -0.913508 | 0.621336 | 28956.787408 | 0.999960 |
| **alpha_district__7** | 0.045714 | 0.311733 | 0.001962 | -0.565566 | 0.660832 | 28292.313320 | 0.999993 |
| **alpha_district__8** | -0.225373 | 0.386979 | 0.002226 | -0.984506 | 0.519599 | 26763.759198 | 0.999950 |
| **alpha_district__9** | -0.614211 | 0.465289 | 0.003530 | -1.539103 | 0.276922 | 20052.125556 | 0.999951 |
| **alpha_district__10** | -1.063187 | 0.468584 | 0.004082 | -1.979045 | -0.163206 | 14821.665887 | 0.999990 |
| **alpha_district__11** | 0.034956 | 0.358723 | 0.001949 | -0.675092 | 0.724591 | 28159.510064 | 0.999952 |

| | mean | sd | mc_error | hpd_2.5 | hpd_97.5 | n_eff | Rhat |
|---|---|---|---|---|---|---|---|
| **alpha_district__12** | 0.241019 | 0.388220 | 0.002160 | -0.499316 | 1.022766 | 25481.116584 | 0.999963 |
| **alpha_district__13** | 0.085663 | 0.392035 | 0.002851 | -0.693301 | 0.840862 | 18295.333524 | 0.999959 |
| **alpha_district__14** | 0.042385 | 0.401386 | 0.002609 | -0.765192 | 0.817849 | 26050.170008 | 0.999950 |
| **alpha_district__15** | 0.424020 | 0.383613 | 0.002577 | -0.347618 | 1.157139 | 22994.259747 | 0.999950 |
| **alpha_district__16** | -0.123871 | 0.355896 | 0.001861 | -0.816897 | 0.588030 | 28789.612332 | 0.999960 |
| **alpha_district__17** | -0.189098 | 0.325981 | 0.001948 | -0.842595 | 0.432270 | 26430.032850 | 0.999954 |
| **...** | ... | ... | ... | ... | ... | ... | ... |
| **beta_district__30** | -0.306722 | 0.572410 | 0.003457 | -1.421859 | 0.826292 | 25365.025826 | 0.999950 |
| **beta_district__31** | 0.349871 | 0.691820 | 0.004324 | -1.004681 | 1.731574 | 23501.367659 | 0.999955 |
| **beta_district__32** | 0.587172 | 0.680825 | 0.005289 | -0.682491 | 1.996573 | 16279.067200 | 0.999966 |
| **beta_district__33** | -1.427483 | 0.655416 | 0.008092 | -2.686123 | -0.155577 | 6881.219584 | 0.999954 |
| **beta_district__34** | -0.256312 | 0.485043 | 0.003470 | -1.243398 | 0.661140 | 23653.134527 | 1.000006 |
| **beta_district__35** | -0.182676 | 0.650872 | 0.003680 | -1.507306 | 1.078411 | 25582.669806 | 0.999955 |
| **beta_district__36** | -0.408505 | 0.719070 | 0.004802 | -1.896078 | 0.965563 | 20473.478785 | 0.999950 |
| **beta_district__37** | 0.599071 | 0.682056 | 0.005619 | -0.699268 | 1.977989 | 16686.935519 | 0.999984 |
| **beta_district__38** | -0.435956 | 0.638142 | 0.004929 | -1.748181 | 0.787361 | 20054.033673 | 0.999960 |
| **beta_district__39** | -0.218963 | 0.520779 | 0.003348 | -1.234709 | 0.820250 | 25050.536043 | 0.999952 |
| **beta_district__40** | -0.986576 | 0.689192 | 0.006963 | -2.390931 | 0.267871 | 8963.284001 | 0.999974 |
| **beta_district__41** | -0.981412 | 0.737458 | 0.007510 | -2.467743 | 0.378582 | 8847.289239 | 0.999952 |
| **beta_district__42** | -0.271453 | 0.499459 | 0.003482 | -1.254523 | 0.703955 | 21658.582525 | 0.999953 |
| **beta_district__43** | 0.321581 | 0.679397 | 0.004209 | -1.012283 | 1.688417 | 22039.403007 | 0.999951 |
| **beta_district__44** | 0.542842 | 0.602085 | 0.004457 | -0.611906 | 1.771712 | 17093.116092 | 0.999961 |
| **beta_district__45** | -0.262549 | 0.486685 | 0.003629 | -1.199076 | 0.715471 | 17679.383126 | 0.999961 |
| **beta_district__46** | -0.193068 | 0.638363 | 0.004423 | -1.440478 | 1.071838 | 23530.958618 | 1.000078 |
| **beta_district__47** | -0.324183 | 0.505627 | 0.003740 | -1.328159 | 0.660130 | 20442.581967 | 0.999962 |
| **beta_district__48** | 0.329871 | 0.792102 | 0.005611 | -1.186346 | 1.974628 | 25986.807262 | 0.999951 |
| **beta_district__49** | 0.489463 | 0.672033 | 0.005328 | -0.774787 | 1.831199 | 13234.637097 | 1.000002 |
| **beta_district__50** | 0.145308 | 0.517007 | 0.003405 | -0.841514 | 1.198220 | 23954.825278 | 1.000035 |
| **beta_district__51** | -1.152976 | 0.523342 | 0.005712 | -2.161317 | -0.131136 | 7740.106650 | 0.999980 |
| **beta_district__52** | -0.187664 | 0.678423 | 0.004931 | -1.563616 | 1.129690 | 23652.379300 | 0.999950 |
| **beta_district__53** | -0.430889 | 0.772262 | 0.006373 | -1.959578 | 1.067396 | 14514.282287 | 1.000020 |
| **beta_district__54** | -0.268634 | 0.494556 | 0.003254 | -1.213387 | 0.728102 | 20303.140188 | 0.999955 |
| **beta_district__55** | 0.201603 | 0.632630 | 0.004086 | -1.075150 | 1.432489 | 22693.046945 | 1.000052 |
| **beta_district__56** | -0.912239 | 0.582738 | 0.006265 | -2.050376 | 0.184528 | 10012.598657 | 0.999970 |
| **beta_district__57** | 0.440616 | 0.755316 | 0.005773 | -1.021642 | 1.964097 | 15968.677461 | 0.999953 |
| **beta_district__58** | 0.070164 | 0.569774 | 0.004249 | -1.050752 | 1.208296 | 21346.973950 | 0.999968 |
| **beta_district__59** | 0.017355 | 0.554796 | 0.003905 | -1.114818 | 1.080227 | 20373.550681 | 0.999950 |

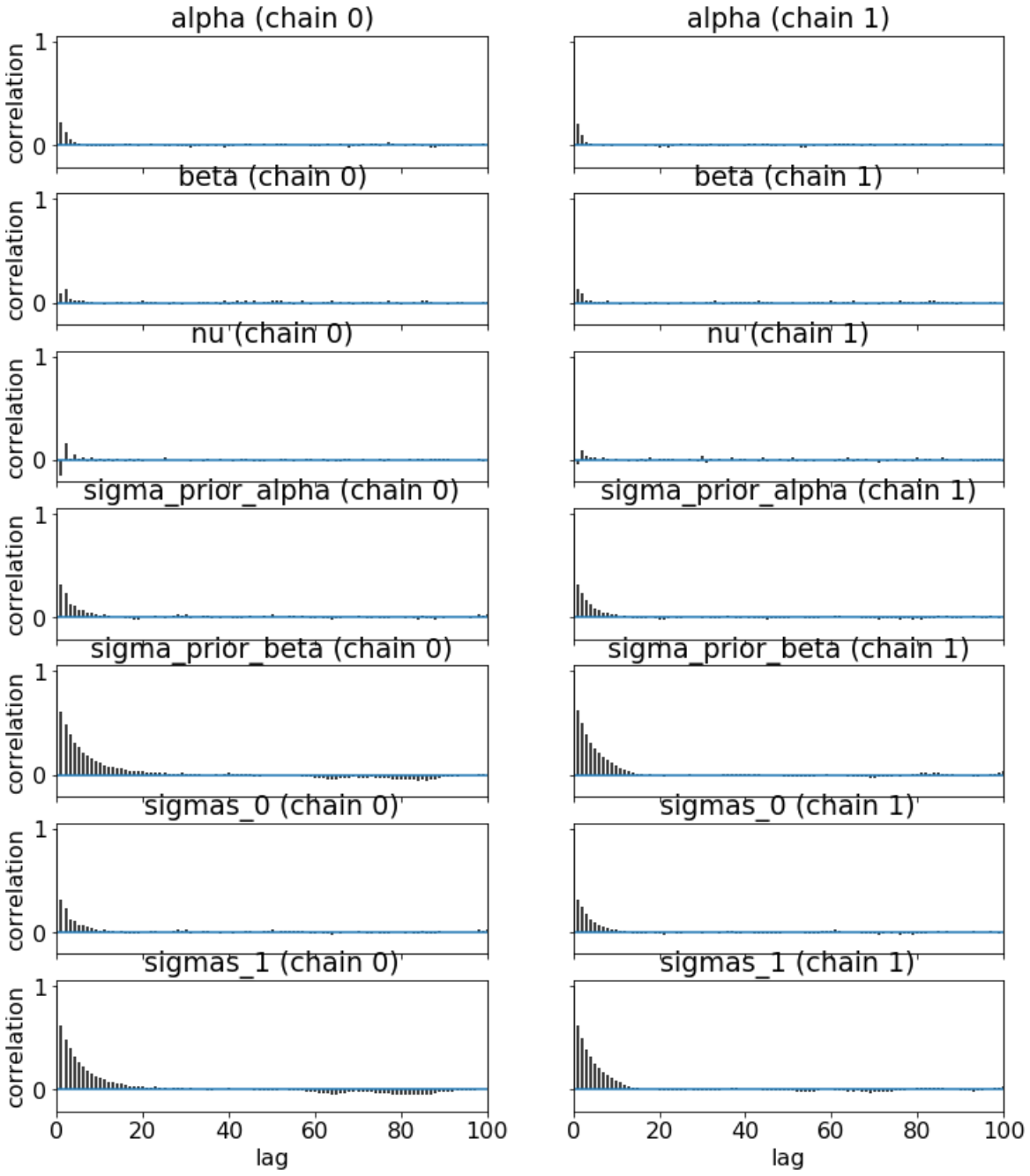
132 rows × 7 columns

**Comment on the quality of the samples obtained above**

In [42]:
```
# Variable name for the traceplot
varnames = ['alpha', 'beta', 'nu', 'sigma_prior_alpha', 'sigma_prior_beta', 'sigmas', 'rho', 'cov']
# Generate traceplot
fig = pm.traceplot(trace_vs, varnames=varnames)
```



In [43]:
```
# Generate autocorr plot
fig = pm.autocorrplot(trace_vs, varnames=varnames[0:6])
```




The quality of these samples is perhaps adequate, but not great. The first warning sign came as, well, a warning from the sampler. Each chain warned of 1 divergence during sampling. (And this was with a 90% acceptance rate). It also warned that the number of effective parameters was below 25% for some parameters.

The table above puts the parameters into a more "strategic" order, with the long vectors of alpha and beta by district at the bottom so they don't overwhelm the first few. We can focus on these parameters for the plots as well, because the district level parameters seem OK.

The traceplots above don't present any obvious problems. The autocorr plots are more informative. We can see that there is quite a bit of correlation for the priors on the sigma of alpha and beta. The covariance entries couldn't be plotted because pymc3 complained that the the array was too deep.

The number of effective parameters for sigma_prior_alpha, sigma_prior_beta, the sigmas, the correlation coefficients, and the covariances are all too small relative to the number of samples drawn. Indeed this deficiency triggered a runtime warning from the sampler, complaining that n_eff was below 25% for some of the samples.

**B3** Propose a method based on the reparametrization trick for multi-variate gaussians) of improving the quality of the samples obtained and implement it. (A hint can be obtained from here: https://docs.pymc.io/api/distributions/multivariate.html#pymc3.distributions.multivariate.MvNormal (https://docs.pymc.io/api/distributions/multivariate.html#pymc3.distributions.multivariate.MvNormal) . Using that hint lowered the sampling time to 2.5 minutes on my laptop).

There are two key ideas and they are both included in the hint cited above. The first idea is that instead of sampling correlation coefficients and standard deviations directly, it is more efficient to sample the entries of a Cholesky factor matrix. The second idea, which is more important for sampling performance, is the core of the reparameterization trick. In one dimension, the reparameterization trick tells us that if we want to draw samples for a variable x with mean mu and standard deviation sigma, it is often faster and numerically easier on our sampling back end to instead draw samples for a variable z that is distributed as the standard normal with mean = 0 and standard deviation = 1. Then we can construct the samples for x by applying the derministic rule

$$x_i = \sigma \cdot z_i + mu$$

The reparameterization trick for multivariate Guassians is completely analogous. The one difference is that we need to use the "matrix square root" of the covariance matrix, which is its Cholesky factor L. So the analogous equation in the multidimensional case is

$$x_i = L \cdot z_i \cdot L^T + \mu$$

This is exactly the strategy carried out in the reparameterized model below. The strategy subtantially sped up sampling. On a Windows platform unfortunately parallel processing doesn't work. Runtime on the first sampler for 2 chains with 12,000 samples (2,000 tuning, 10,000 retained) was 10:06 minutes per chain or 22 minutes total. With the improved version, sampling one chain took about 3:41, a speed-up of 2.74 times. Exact hardware configuration is a custom built PC with an AMD Ryzen Threadripper 2990WX 32 Core processor running at 3.00 GHz. Sadly having 32 cores isn't much help when pymc3 on Windows is stuck running on one core at a time... This notebook is set up to load the samples from a persisted data file if possible, so samples are not run redundantly.

In [44]:
```
# Define a varying slopes model incorporating a beta_urban term
with pm.Model() as model_vsr:
    # Citation: ideas to efficiently reparameterize samples from a MV Gaussian
    # https://docs.pymc.io/api/distributions/multivariate.html#pymc3.distributions.multivariate.MvNormal
    # Set the prior for the overall intercept
    alpha = pm.Normal(name='alpha', mu=0.0, sd=10.0)
    # Set the prior for the overall intercept on urban, beta
    beta = pm.Normal(name='beta', mu=0.0, sd=10.0)

    # Sample the variances of the single factors
    # sd_dist = pm.HalfCauchy.dist(beta=2.5, shape=num_factors)
    sd_dist = pm.Lognormal.dist(mu=0.0, tau=1.0, shape=num_factors)
    # The parameter nu is the prior on correlation; 0 is uniform, infinity is no corelation
    eta = pm.Uniform('nu', 1.0, 5.0)
    # The number of dimensions here is 2: correlation structure is bewteen alpha and beta by district
    num_factors: int = 2
    # Sample the correlation coefficients using the LKJ distribution
    chol_packed = pm.LKJCholeskyCov('chol_packed', n=num_factors, eta=eta, sd_dist = sd_dist)
    # Expand the packed Cholesky matrix to full size
    chol = pm.Deterministic('chol', pm.expand_packed_triangular(num_factors, chol_packed))
    # Make the covariance matrix by multiplying out the cholesky factor by its transpose
    cov = pm.Deterministic('cov', tt.dot(chol, chol.T))
    # The multivariate Gaussian of (alpha, beta) by district
    # Decompose this into a "raw" part and then scale it
    theta_raw = pm.Normal(name='theta_raw', mu=0.0, sd=1.0, shape=(num_districts, num_factors))
    # Now scale these to have the desired covariance structure
    theta_district = pm.Deterministic(name='theta_district', var=tt.dot(chol, theta_raw.T).T)

    # The vector of standard deviations for each variable; size num_factors x num_factors
    # Citation: efficient generation of sigmas and rhos from cov
    # https://github.com/aloctavodia/Statistical-Rethinking-with-Python-and-PyMC3/blob/master/Chp_13.ipyn
    sigmas = pm.Deterministic('sigmas', tt.sqrt(tt.diag(cov)))
    # correlation matrix (num_factors x num_factors)
    rhos = pm.Deterministic('rhos', tt.diag(sigmas**-1).dot(cov.dot(tt.diag(sigmas**-1))))

    # Extract the standard deviations of alpha and beta, and the correlation coefficient rho
    sigma_alpha = pm.Deterministic('sigma_alpha', sigmas[0])
    sigma_beta = pm.Deterministic('sigma_beta', sigmas[1])
    rho = pm.Deterministic('rho', rhos[0, 0])

    # Extract alpha_district and beta_district from theta_district
    alpha_district = pm.Deterministic('alpha_district', theta_district[:,0])
    beta_district = pm.Deterministic('beta_district', theta_district[:, 1])

    # Set the probability that each woman uses contraception in this model
    # It depends on the district she lives in and whether the district is urban
    # p = pm.math.invlogit(alpha + alpha_district[df.district_id] +
    #                      (beta + beta_district[df.district_id]) * df.urban)
    p = pm.math.invlogit(alpha + theta_district[df.district_id, 0] +
                         (beta + theta_district[df.district_id, 1]) * df.urban)

    # The response variable - whether this woman used contraception; modeled as Bernoulli
    # Bind this to the observed values
    use_contraception = pm.Bernoulli('use_contraception', p=p, observed=df['use_contraception'])
```

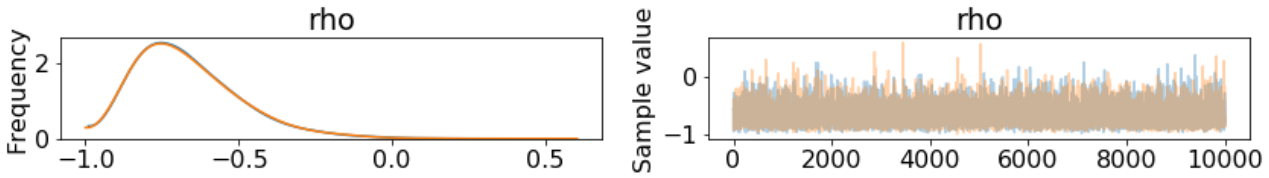
In [45]:
```
# Sample from the reparameterized varying-slope model
try:
    trace_vsr = vartbl['trace_vsr']
    print(f'Loaded samples for the Variable Slopes Reparameterized model in trace_vsr.')
except:
    with model_vsr:
        nuts_kwargs = {'target_accept': 0.90}
        trace_vsr = pm.sample(draws=num_samples, tune=num_tune, nuts_kwargs=nuts_kwargs, chains=2, cores=
    vartbl['trace_vsr'] = trace_vsr
    save_vartbl(vartbl, fname)

# Summary of the variable-effects model
summary_vsr = pm.summary(trace_vsr)
```

```
Loaded samples for the Variable Slopes Reparameterized model in trace_vsr.
```

**B4** Inspect the trace of the correlation between the intercepts and slopes, plotting the correlation marginal. What does this correlation tell you about the pattern of contraceptive use in the sample? It might help to plot the mean (or median) varying effect estimates for both the intercepts and slopes, by district. Then you can visualize the correlation and maybe more easily think through what it means to have a particular correlation. Also plot the predicted proportion of women using contraception, with urban women on one axis and rural on the other. Finally, also plot the difference between urban and rural probabilities against rural probabilities. All of these will help you interpret your findings. (Hint: think in terms of low or high rural contraceptive use)

In [46]:
```
# Run the traceplot of the correlation rho
pm.traceplot(trace_vsr, varnames=['rho'])
```

Out[46]:
```
array([[<matplotlib.axes._subplots.AxesSubplot object at 0x00000155423D1F28>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x000001553E9E0438>]],
      dtype=object)
```




**Summarize Contraception Use by District for Urban vs. Rural, and Mean Parameter Values**

In [47]:
```
# Add columns for urban and rural contraceptive users
df['use_contraception_urban'] = df.use_contraception * df.urban
df['use_contraception_rural'] = df.use_contraception * (1-df.urban)

# Update the aggregated contraception use in each district
agg_tbl = {
        'woman': ['count'],
        'urban': ['sum', 'mean'],
        'use_contraception': ['mean'],
        'use_contraception_urban':['sum'],
        'use_contraception_rural':['sum'],
        }
df_district = df.groupby(by=df.district_id).agg(agg_tbl)
df_district.columns = ["_".join(x) for x in df_district.columns.ravel()]
# Compute number of rural women
df_district['rural_sum'] = df_district['woman_count'] - df_district['urban_sum']

# Change column names to make model suffix at the end of the name
df_district.rename(axis='columns', inplace=True, mapper=
    {'use_contraception_mean': 'contraception',
     'urban_sum':'urban_count',
     'rural_sum':'rural_count',
     'use_contraception_urban_sum': 'contraception_urban',
     'use_contraception_rural_sum': 'contraception_rural',
     })

# Change use_contraception_urban and use_contraception_rural to rates
df_district.contraception_urban = df_district.contraception_urban / df_district.urban_count
df_district.contraception_rural = df_district.contraception_rural / df_district.rural_count
```

In [48]:
```
# List of parameter names for alpha_district and beta_district for each district
district_suffix = [f'district__{i}' for i in range(num_districts)]
params_alpha_district = [f'alpha_{suffix}' for suffix in district_suffix]
params_beta_district = [f'beta_{suffix}' for suffix in district_suffix]

# Get the mean of alpha_district and beta_district for all the districts; mean taken over samples
alpha_district_mean = summary_vsr.loc[params_alpha_district]['mean'].values
beta_district_mean = summary_vsr.loc[params_beta_district]['mean'].values

# Mean of "global" parameters alpha, beta, and rho
alpha_mean = summary_vsr.loc['alpha']['mean']
beta_mean = summary_vsr.loc['beta']['mean']
rho_mean = summary_vsr.loc['rho']['mean']
print(f'Mean values of the global parameters alpha, beta and rho:')
print(f'alpha: {alpha_mean:0.4f}')
print(f'beta:  {beta_mean:0.4f}')
print(f'rno:   {rho_mean:0.4f}')

# Add alpha_district, beta_district, alpha_eff, beta_eff
df_district['alpha_district'] = alpha_district_mean
df_district['beta_district'] = beta_district_mean
df_district['alpha_eff'] = alpha_mean + alpha_district_mean
df_district['beta_eff'] = beta_mean + beta_district_mean

# Compute predicted probability for urban and rural contraceptive use
df_district['pred_urban'] = logistic.cdf(df_district.alpha_eff + df_district.beta_eff)
df_district['pred_rural'] = logistic.cdf(df_district.alpha_eff)

display_cols = ['woman_count', 'urban_count', 'rural_count', 'contraception_urban', 'contraception_rural
                'alpha_eff', 'beta_eff', 'pred_urban', 'pred_rural']
display(df_district[display_cols])
```

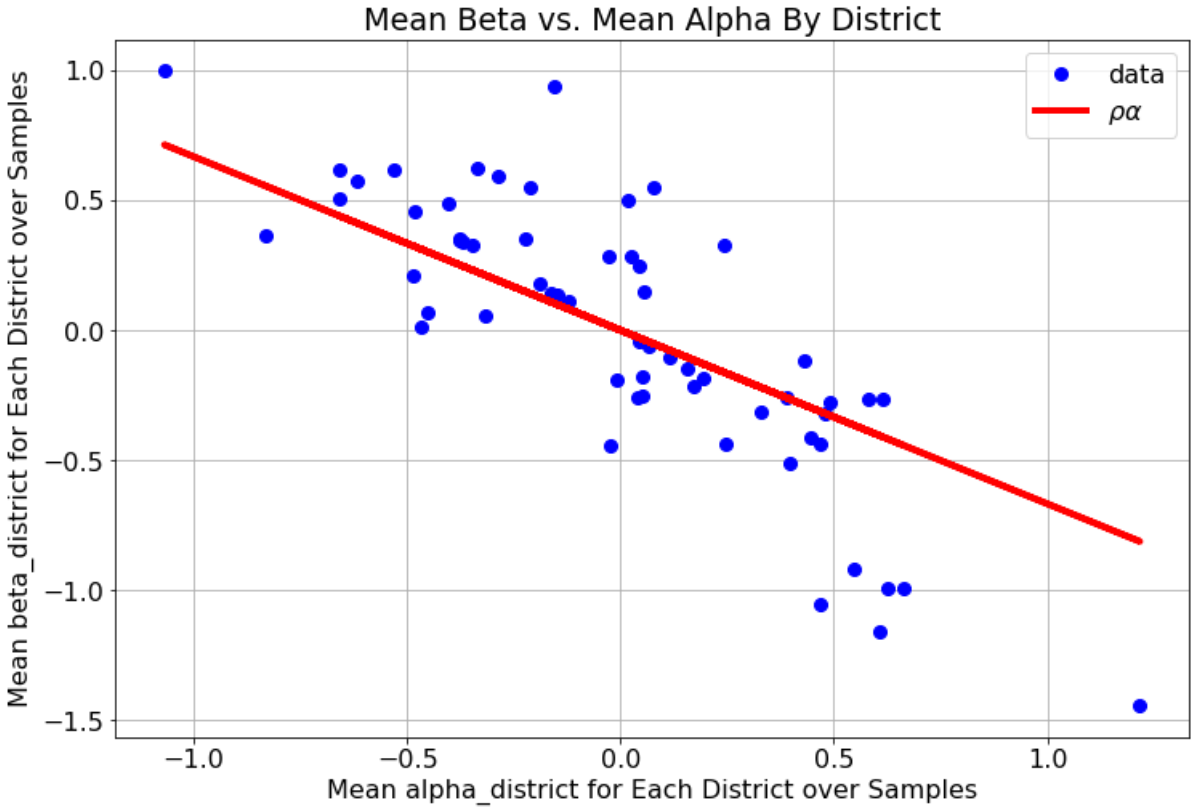
```
Mean values of the global parameters alpha, beta and rho:
alpha: -0.7179
beta:  0.7219
rno:   -0.6677
```

| district_id | woman_count | urban_count | rural_count | contraception_urban | contraception_rural | contraception | alpha_eff | beta_eff |
|---|---|---|---|---|---|---|---|---|
| **0** | 117 | 63 | 54 | 0.365079 | 0.129630 | 0.256410 | -1.548480 | 1.083927 |
| **1** | 20 | 0 | 20 | NaN | 0.350000 | 0.350000 | -0.673147 | 0.678670 |
| **2** | 2 | 2 | 0 | 1.000000 | NaN | 1.000000 | -0.693785 | 1.006593 |
| **3** | 30 | 11 | 19 | 0.909091 | 0.263158 | 0.500000 | -0.873068 | 1.660437 |
| **4** | 39 | 2 | 37 | 0.500000 | 0.351351 | 0.358974 | -0.651046 | 0.660441 |
| **5** | 65 | 7 | 58 | 0.714286 | 0.241379 | 0.292308 | -1.054769 | 1.347993 |
| **6** | 18 | 0 | 18 | NaN | 0.277778 | 0.277778 | -0.865629 | 0.858192 |
| **7** | 37 | 2 | 35 | 1.000000 | 0.342857 | 0.378378 | -0.672723 | 0.970429 |
| **8** | 23 | 3 | 20 | 0.666667 | 0.250000 | 0.304348 | -0.942209 | 1.072180 |
| **9** | 13 | 0 | 13 | NaN | 0.076923 | 0.076923 | -1.336203 | 1.298575 |
| **10** | 21 | 0 | 21 | NaN | 0.000000 | 0.000000 | -1.786568 | 1.718305 |
| **11** | 29 | 6 | 23 | 0.333333 | 0.347826 | 0.344828 | -0.676903 | 0.461962 |
| **12** | 24 | 8 | 16 | 0.375000 | 0.437500 | 0.416667 | -0.472306 | 0.285114 |
| **13** | 118 | 101 | 17 | 0.673267 | 0.352941 | 0.627119 | -0.639625 | 1.268642 |
| **14** | 22 | 8 | 14 | 0.375000 | 0.357143 | 0.363636 | -0.667062 | 0.470784 |
| **15** | 20 | 2 | 18 | 1.000000 | 0.500000 | 0.550000 | -0.286222 | 0.602991 |
| **16** | 24 | 0 | 24 | NaN | 0.291667 | 0.291667 | -0.840023 | 0.834610 |
| **17** | 47 | 14 | 33 | 0.500000 | 0.272727 | 0.340426 | -0.906067 | 0.901603 |
| **18** | 26 | 4 | 22 | 0.750000 | 0.318182 | 0.384615 | -0.747095 | 1.004446 |
| **19** | 15 | 0 | 15 | NaN | 0.400000 | 0.400000 | -0.561221 | 0.574054 |

| district_id | woman_count | urban_count | rural_count | contraception_urban | contraception_rural | contraception | alpha_eff | beta_eff |
|---|---|---|---|---|---|---|---|---|
| **20** | 18 | 8 | 10 | 0.125000 | 0.600000 | 0.388889 | -0.248994 | -0.333702 |
| **21** | 20 | 0 | 20 | NaN | 0.200000 | 0.200000 | -1.093976 | 1.069502 |
| **22** | 15 | 0 | 15 | NaN | 0.266667 | 0.266667 | -0.880055 | 0.866785 |
| **23** | 14 | 0 | 14 | NaN | 0.071429 | 0.071429 | -1.375597 | 1.337353 |
| **24** | 67 | 18 | 49 | 0.444444 | 0.448980 | 0.447761 | -0.322609 | 0.207679 |
| **25** | 13 | 0 | 13 | NaN | 0.384615 | 0.384615 | -0.603865 | 0.617608 |
| **26** | 44 | 5 | 39 | 0.400000 | 0.153846 | 0.181818 | -1.377169 | 1.225416 |
| **27** | 49 | 4 | 45 | 0.250000 | 0.244444 | 0.244898 | -1.035922 | 0.776482 |
| **28** | 32 | 7 | 25 | 0.571429 | 0.200000 | 0.281250 | -1.121393 | 1.211939 |
| **29** | 61 | 16 | 45 | 0.750000 | 0.400000 | 0.491803 | -0.473504 | 1.047650 |
| **30** | 33 | 6 | 27 | 0.500000 | 0.444444 | 0.454545 | -0.390463 | 0.410136 |
| **31** | 24 | 0 | 24 | NaN | 0.208333 | 0.208333 | -1.094500 | 1.073022 |
| **32** | 14 | 7 | 7 | 0.714286 | 0.142857 | 0.428571 | -1.004202 | 1.311659 |
| **33** | 35 | 9 | 26 | 0.333333 | 0.769231 | 0.657143 | 0.499151 | -0.719967 |
| **34** | 48 | 20 | 28 | 0.550000 | 0.464286 | 0.500000 | -0.326782 | 0.462619 |
| **35** | 17 | 3 | 14 | 0.333333 | 0.357143 | 0.352941 | -0.665556 | 0.540481 |
| **36** | 13 | 0 | 13 | NaN | 0.538462 | 0.538462 | -0.273853 | 0.309348 |
| **37** | 14 | 7 | 7 | 0.571429 | 0.000000 | 0.285714 | -1.247484 | 1.336489 |
| **38** | 26 | 2 | 24 | 0.500000 | 0.500000 | 0.500000 | -0.249085 | 0.282639 |
| **39** | 41 | 29 | 12 | 0.482759 | 0.416667 | 0.463415 | -0.545571 | 0.504713 |
| **40** | 26 | 3 | 23 | 0.000000 | 0.565217 | 0.500000 | -0.093485 | -0.268650 |
| **41** | 11 | 5 | 6 | 0.200000 | 0.833333 | 0.545455 | -0.054740 | -0.268384 |
| **42** | 45 | 17 | 28 | 0.588235 | 0.500000 | 0.533333 | -0.227826 | 0.446134 |
| **43** | 27 | 0 | 27 | NaN | 0.222222 | 0.222222 | -1.065328 | 1.048288 |
| **44** | 39 | 5 | 34 | 0.800000 | 0.264706 | 0.333333 | -0.930353 | 1.273683 |
| **45** | 86 | 12 | 74 | 0.666667 | 0.500000 | 0.523256 | -0.102255 | 0.456680 |
| **46** | 15 | 6 | 9 | 0.500000 | 0.444444 | 0.466667 | -0.522074 | 0.536146 |
| **47** | 42 | 16 | 26 | 0.562500 | 0.500000 | 0.523810 | -0.237487 | 0.398909 |
| **48** | 4 | 0 | 4 | NaN | 0.000000 | 0.000000 | -1.085392 | 1.063040 |
| **49** | 19 | 4 | 15 | 1.000000 | 0.333333 | 0.473684 | -0.701202 | 1.219995 |
| **50** | 37 | 17 | 20 | 0.588235 | 0.350000 | 0.459459 | -0.661411 | 0.868206 |
| **51** | 61 | 19 | 42 | 0.263158 | 0.523810 | 0.442623 | -0.111357 | -0.437469 |
| **52** | 19 | 19 | 0 | 0.421053 | NaN | 0.421053 | -0.725079 | 0.532794 |
| **53** | 6 | 6 | 0 | 0.166667 | NaN | 0.166667 | -0.743318 | 0.280713 |
| **54** | 45 | 21 | 24 | 0.619048 | 0.541667 | 0.577778 | -0.136353 | 0.454032 |
| **55** | 27 | 4 | 23 | 0.250000 | 0.173913 | 0.185185 | -1.203710 | 0.928789 |
| **56** | 33 | 13 | 20 | 0.307692 | 0.550000 | 0.454545 | -0.172522 | -0.198271 |
| **57** | 10 | 0 | 10 | NaN | 0.100000 | 0.100000 | -1.200224 | 1.179782 |
| **58** | 32 | 10 | 22 | 0.300000 | 0.181818 | 0.218750 | -1.171458 | 0.788243 |
| **59** | 42 | 11 | 31 | 0.272727 | 0.193548 | 0.214286 | -1.185709 | 0.735601 |

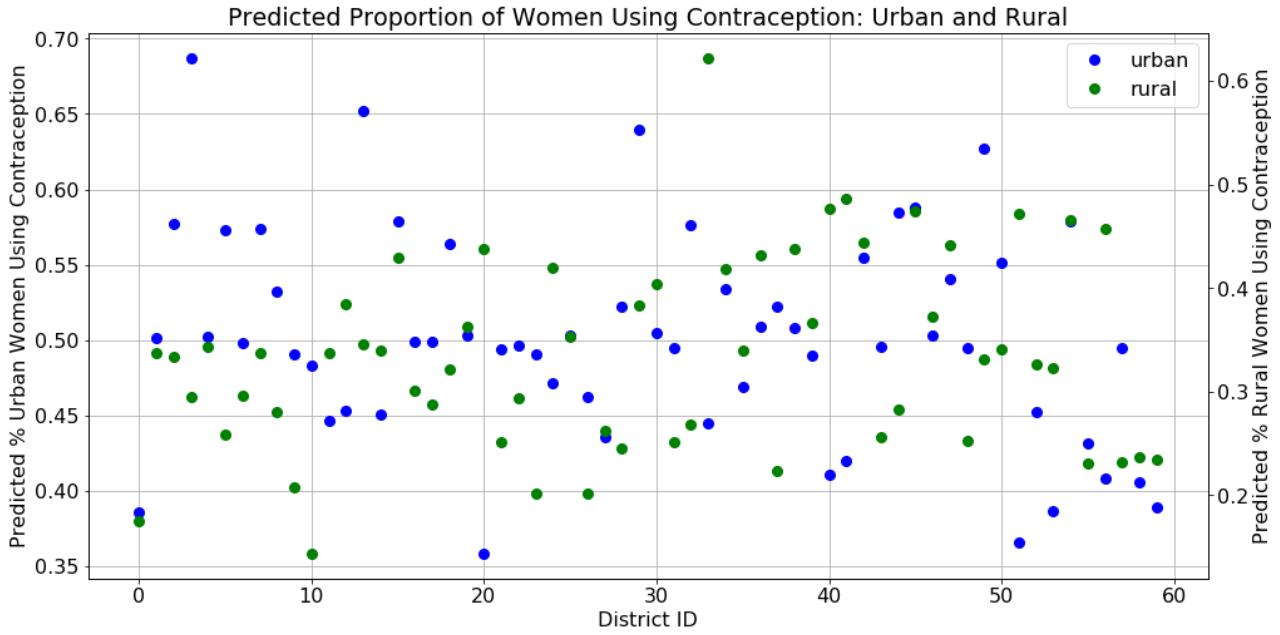
**Plot the mean varying effect estimates for both the incercepts and slopes, by district**

In [49]:
```
# Plot beta vs. alpha
fig, ax = plt.subplots(figsize=[12,8])
ax.set_title('Mean Beta vs. Mean Alpha By District')
ax.set_xlabel('Mean alpha_district for Each District over Samples')
ax.set_ylabel('Mean beta_district for Each District over Samples')
ax.plot(alpha_district_mean, beta_district_mean, label='data', color='b', linewidth=0, marker='o', marke
ax.plot(alpha_district_mean, alpha_district_mean * rho_mean, label=r'$\rho \alpha$', linewidth=4, color=
ax.legend()
ax.grid()
```

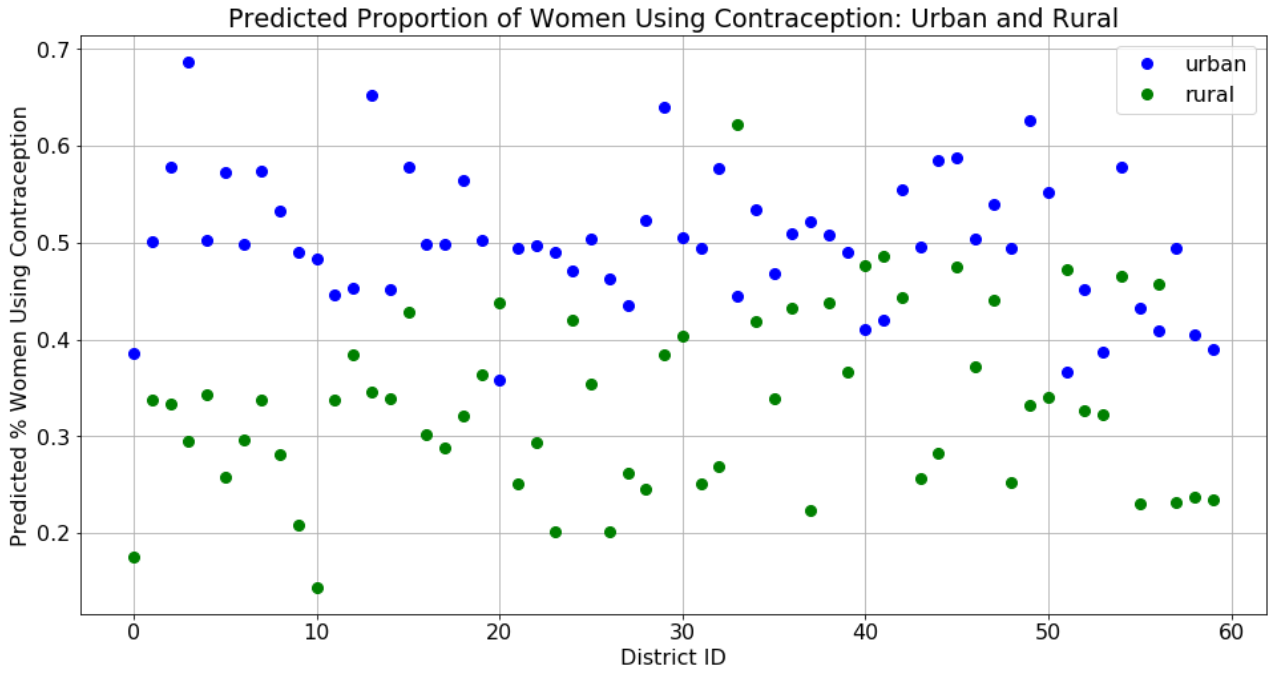



**Plot the predicted proportion of women using contraception, with urban women on one axis and rural on the other**

In [50]:
```
fig, ax1 = plt.subplots(figsize=[16,8])
ax2 = ax1.twinx()
ax1.set_title('Predicted Proportion of Women Using Contraception: Urban and Rural')
ax1.set_xlabel('District ID')
ax1.set_ylabel('Predicted % Urban Women Using Contraception')
ax2.set_ylabel('Predicted % Rural Women Using Contraception')
district_ids = df_district.index.values
h1 = ax1.plot(district_ids, df_district.pred_urban, label='urban', color='b', linewidth=0, marker='o', ma
h2 = ax2.plot(district_ids, df_district.pred_rural, label='rural', color='g', linewidth=0, marker='o', ma
ax1.legend(handles=[h1[0], h2[0]], labels=['urban', 'rural'])
ax1.grid()
```

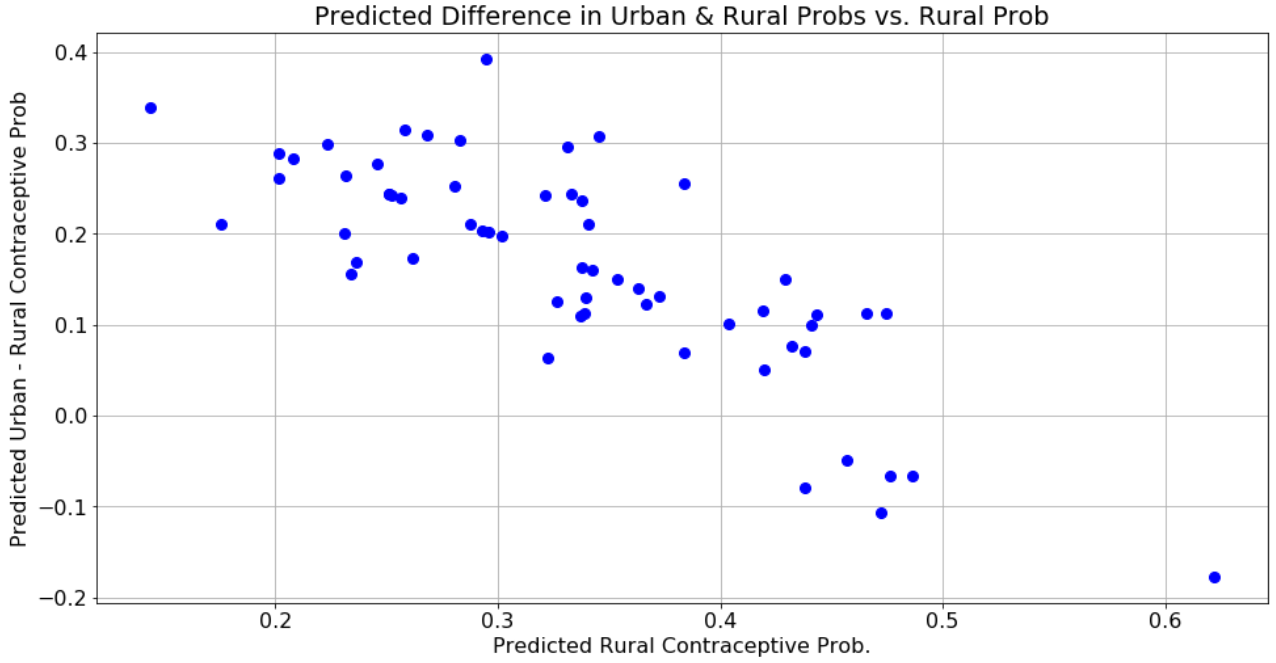


In [51]:
```
fig, ax = plt.subplots(figsize=[16,8])
ax.set_title('Predicted Proportion of Women Using Contraception: Urban and Rural')
ax.set_xlabel('District ID')
ax.set_ylabel('Predicted % Women Using Contraception')

ax.plot(district_ids, df_district.pred_urban, label='urban', color='b', linewidth=0, marker='o', markers
ax.plot(district_ids, df_district.pred_rural, label='rural', color='g', linewidth=0, marker='o', markers
ax.legend()
ax.grid()
```




**Plot the difference between urban and rural probabilities against rural probabilities**

In [52]:
```
fig, ax = plt.subplots(figsize=[16,8])
ax.set_title('Predicted Difference in Urban & Rural Probs vs. Rural Prob')
ax.set_xlabel('Predicted Rural Contraceptive Prob.')
ax.set_ylabel('Predicted Urban - Rural Contraceptive Prob')
plot_x = df_district.pred_rural
plot_y = df_district.pred_urban - df_district.pred_rural
ax.plot(plot_x, plot_y, color='b', linewidth=0, marker='o', markersize=8)
ax.grid()
```




The first chart above is a scatter plot with data points for each of the 60 districts. One dot represents model parameters for one district. The x-axis is the mean value of alpha for that district (i.e. the intercept for that district). The y-axis is the mean value of beta for that district (i.e. the change in predicted z-score for contraceptive use for urban women).

Let's put the two corners of this plot into words to develop an intuition for what's going on. At the top left of the graph, we see one district (district 10) where alpha_district is close to -1 and beta_district is close to +1. Let's also remember that the overall values are alpha = -0.72 and beta = 0.72. So, in this district, alpha = -0.72 -1.0 = -1.72 and beta = 0.72 + 1.0 = +1.72. This is saying that rural women who live in this district have a z-score for contraceptive use of -1.72, i.e. they have a 14% probability to use contraception. The model thinks urban women have a z-score of -1.72 + 1.72 = 0.0, i.e. about half of them use contraception. It turns out that this district has 21 women, all of whom live in rural areas and none of whom use contraception. So the parameters describing urban women in this district are essentially noise.

At the bottom right of the chart, we see a district with an alpha_district of +1.20 and a beta_district of -1.42 (district id 33). Adding these district level offsets to the global means, alpha in this district is -0.72 + 1.20 = 0.48 and beta is +0.72 - 1.42 = -0.70. In this district, rural women use contraceptives with a 62% probatility while urban women use contraceptives with a 45% probability.

The next charts allow us to see that

- Overall, urban contraception use is higher than rural contraception use
- Rural contraceptive use is quite a bit more variable than urban contraception use, which is clustered more tightly
- Therefore, in a district that has high overall contraceptive use (large alpha_district), the urban contraceptive use needs to be dialed back with a lower setting on beta_district

This is exactly the pattern that we see in the first scatter plot.

**B5** Add additional "slope" terms (one-by-one) into the model for

- (a) the centered-age of the women and
- (b) an indicator for whether the women have a small number or large number of existing kids in the house (you can treat 1-2 kids as low, 3-4 as high, but you might want to experiment with this split).

In [53]:
```
# Model with slope for the age of the woman;
# model_DUA stands for District, Urban, Age
with pm.Model() as model_DUA:
    # Set the prior for the overall intercept
    alpha = pm.Normal(name='alpha', mu=0.0, sd=10.0)
    # Set the prior for the overall intercept on urban, beta_age
    beta_urban = pm.Normal(name='beta_urban', mu=0.0, sd=10.0)

    # Set the prior for the intercept on age, beta_age; this is fixed only, no by district version
    # Set the sd of beta_age to 1.0 not 10.0, because the standard deviation of age_centered is 9.0
    beta_age = pm.Normal(name='beta_age', mu=0.0, sd=1.0)

    # Sample the variances of the single factors by district
    sd_dist = pm.Lognormal.dist(mu=0.0, tau=1.0, shape=num_factors)
    # The parameter nu is the prior on correlation; 0 is uniform, infinity is no corelation
    eta = pm.Uniform('nu', 1.0, 5.0)
    # The number of dimensions here is 2: correlation structure is bewteen alpha and beta by district
    num_factors: int = 2
    # Sample the correlation coefficients using the LKJ distribution
    chol_packed = pm.LKJCholeskyCov('chol_packed', n=num_factors, eta=eta, sd_dist = sd_dist)
    # Expand the packed Cholesky matrix to full size
    chol = pm.Deterministic('chol', pm.expand_packed_triangular(num_factors, chol_packed))
    # Make the covariance matrix by multiplying out the cholesky factor by its transpose
    cov = pm.Deterministic('cov', tt.dot(chol, chol.T))
    # The multivariate Gaussian of (alpha, beta) by district
    # Decompose this into a "raw" part and then scale it
    theta_raw = pm.Normal(name='theta_raw', mu=0.0, sd=1.0, shape=(num_districts, num_factors))
    # Now scale these to have the desired covariance structure
    theta_district = pm.Deterministic(name='theta_district', var=tt.dot(chol, theta_raw.T).T)

    # Set the probability that each woman uses contraception in this model
    # It depends on the district she lives in and whether the district is urban
    p = pm.math.invlogit(alpha + theta_district[df.district_id, 0] +
                         (beta_urban + theta_district[df.district_id, 1]) * df.urban +
                         beta_age * df.age_centered)

    # The response variable - whether this woman used contraception; modeled as Bernoulli
    # Bind this to the observed values
    use_contraception = pm.Bernoulli('use_contraception', p=p, observed=df['use_contraception'])
```

In [54]:
```
# Sample from the DUA model
try:
    trace_DUA = vartbl['trace_DUA']
    print(f'Loaded samples for the District-Urban-Age model in trace_DUA.')
except:
    print(f'Sampling from District-Urban-Age model...')
    with model_DUA:
        nuts_kwargs = {'target_accept': 0.90}
        trace_DUA = pm.sample(draws=num_samples, tune=num_tune, nuts_kwargs=nuts_kwargs, chains=chains, c
    vartbl['trace_DUA'] = trace_DUA
    save_vartbl(vartbl, fname)
```

```
Loaded samples for the District-Urban-Age model in trace_DUA.
```

In [55]:
```
# Add a new column to the dataframe, many_kids, indicating whether the woman has 3 or more kids at home
df['many_kids'] = (df.living_children > 3) * 1

# Model with slopes for the age of the woman and whether she has a large number of children;
# model_DUAK stands for District, Urban, Age, Kids
with pm.Model() as model_DUAK:
    # Set the prior for the overall intercept
    alpha = pm.Normal(name='alpha', mu=0.0, sd=10.0)
    # Set the prior for the overall intercept on urban, beta_age
    beta_urban = pm.Normal(name='beta_urban', mu=0.0, sd=10.0)

    # Set the prior for the slope on age, beta_age; this is fixed only, no by district version
    # Set the sd of beta_age to 1.0 not 10.0, because the standard deviation of age_centered is 9.0
    beta_age = pm.Normal(name='beta_age', mu=0.0, sd=1.0)

    # Set the prior for the slope on many_kids, beta_kids
    beta_kids = pm.Normal(name='beta_kids', mu=0.0, sd=10.0)

    # Sample the variances of the single factors by district
    sd_dist = pm.Lognormal.dist(mu=0.0, tau=1.0, shape=num_factors)
    # The parameter nu is the prior on correlation; 0 is uniform, infinity is no corelation
    eta = pm.Uniform('nu', 1.0, 5.0)
    # The number of dimensions here is 2: correlation structure is bewteen alpha and beta by district
    num_factors: int = 2
    # Sample the correlation coefficients using the LKJ distribution
    chol_packed = pm.LKJCholeskyCov('chol_packed', n=num_factors, eta=eta, sd_dist = sd_dist)
    # Expand the packed Cholesky matrix to full size
    chol = pm.Deterministic('chol', pm.expand_packed_triangular(num_factors, chol_packed))
    # Make the covariance matrix by multiplying out the cholesky factor by its transpose
    cov = pm.Deterministic('cov', tt.dot(chol, chol.T))
    # The multivariate Gaussian of (alpha, beta) by district
    # Decompose this into a "raw" part and then scale it
    theta_raw = pm.Normal(name='theta_raw', mu=0.0, sd=1.0, shape=(num_districts, num_factors))
    # Now scale these to have the desired covariance structure
    theta_district = pm.Deterministic(name='theta_district', var=tt.dot(chol, theta_raw.T).T)

    # Set the probability that each woman uses contraception in this model
    # It depends on (1) district where she lives (2) whether she lives in an urban area
    # (3) her age (4) whether she has a lot of kids (3+) living at home
    p = pm.math.invlogit(alpha + theta_district[df.district_id, 0] +
                         (beta_urban + theta_district[df.district_id, 1]) * df.urban +
                         beta_age * df.age_centered +
                         beta_kids * df.many_kids)

    # The response variable - whether this woman used contraception; modeled as Bernoulli
    # Bind this to the observed values
    use_contraception = pm.Bernoulli('use_contraception', p=p, observed=df['use_contraception'])
```

In [56]:
```
# Sample from the DUAK model
try:
    trace_DUAK = vartbl['trace_DUAK']
    print(f'Loaded samples for the District-Urban-Age-Kids model in trace_DUAK.')
except:
    print(f'Sampling from District-Urban-Age-Kids model...')
    with model_DUAK:
        nuts_kwargs = {'target_accept': 0.90}
        trace_DUAK = pm.sample(draws=num_samples, tune=num_tune, nuts_kwargs=nuts_kwargs, chains=chains,
    vartbl['trace_DUAK'] = trace_DUAK
    save_vartbl(vartbl, fname)
```

```
Loaded samples for the District-Urban-Age-Kids model in trace_DUAK.
```

**B6** Use WAIC to compare your models. What are your conclusions?

In [57]:
```
# Compute WAIC for each model under consideration
waic_fe = pm.waic(trace_fe, model_fe)
waic_ve = pm.waic(trace_ve, model_ve)
waic_DUA = pm.waic(trace_DUA, model_DUA)
waic_DUAK = pm.waic(trace_DUAK, model_DUAK)
```

```
C:\Python\Anaconda3\lib\site-packages\pymc3\stats.py:211: UserWarning: For one or more samples the post
erior variance of the
        log predictive densities exceeds 0.4. This could be indication of
        WAIC starting to fail see http://arxiv.org/abs/1507.04544 (http://arxiv.org/abs/1507.04544) for
 details

  """)
C:\Python\Anaconda3\lib\site-packages\pymc3\stats.py:211: UserWarning: For one or more samples the post
erior variance of the
        log predictive densities exceeds 0.4. This could be indication of
        WAIC starting to fail see http://arxiv.org/abs/1507.04544 (http://arxiv.org/abs/1507.04544) for
 details

  """)
```

In [58]:
```
# Set the names of these models
model_fe.name = 'FixedEffect'
model_ve.name = 'VariableEffect'
model_DUA.name = 'DistrictUrbanAge'
model_DUAK.name = 'DistrictUrbanAgeKids'

# Compare the models
df_model_comp = pm.compare({model_fe: trace_fe,
                            model_ve: trace_ve,
                            model_DUA: trace_DUA,
                            model_DUAK: trace_DUAK})

display(df_model_comp)
```

```
C:\Python\Anaconda3\lib\site-packages\pymc3\stats.py:211: UserWarning: For one or more samples the post
erior variance of the
        log predictive densities exceeds 0.4. This could be indication of
        WAIC starting to fail see http://arxiv.org/abs/1507.04544 (http://arxiv.org/abs/1507.04544) for
 details

  """)
C:\Python\Anaconda3\lib\site-packages\pymc3\stats.py:211: UserWarning: For one or more samples the post
erior variance of the
        log predictive densities exceeds 0.4. This could be indication of
        WAIC starting to fail see http://arxiv.org/abs/1507.04544 (http://arxiv.org/abs/1507.04544) for
 details

  """)
```

| | WAIC | pWAIC | dWAIC | weight | SE | dSE | var_warn |
|---|---|---|---|---|---|---|---|
| **DistrictUrbanAgeKids** | 2461.45 | 55.17 | 0 | 0.9 | 28.75 | 0 | 0 |
| **DistrictUrbanAge** | 2467.26 | 53.48 | 5.82 | 0.01 | 28.32 | 5.28 | 0 |
| **FixedEffect** | 2533.34 | 62.6 | 71.9 | 0.08 | 33 | 19.5 | 1 |
| **VariableEffect** | 126266 | 61824.6 | 123805 | 0 | 207.33 | 184.38 | 1 |

In [59]:
```
pm.summary(trace_DUAK).loc[['alpha', 'beta_urban', 'beta_age', 'beta_kids']]
```

Out[59]:

| | mean | sd | mc_error | hpd_2.5 | hpd_97.5 | n_eff | Rhat |
|---|---|---|---|---|---|---|---|
| **alpha** | -0.860077 | 0.117857 | 0.001279 | -1.089235 | -0.627896 | 8920.846959 | 1.000772 |
| **beta_urban** | 0.732639 | 0.176113 | 0.001674 | 0.393978 | 1.084928 | 9976.836846 | 1.000562 |
| **beta_age** | -0.002406 | 0.007226 | 0.000050 | -0.016557 | 0.011694 | 18702.920297 | 0.999986 |
| **beta_kids** | 0.346777 | 0.133670 | 0.000989 | 0.083761 | 0.608178 | 16370.459790 | 1.000005 |

In [60]:
```
pm.summary(trace_DUA).loc[['alpha', 'beta_urban', 'beta_age']]
```

Out[60]:

| | mean | sd | mc_error | hpd_2.5 | hpd_97.5 | n_eff | Rhat |
|---|---|---|---|---|---|---|---|
| **alpha** | -0.720181 | 0.102221 | 0.001154 | -0.928149 | -0.525025 | 8257.341821 | 1.000246 |
| **beta_urban** | 0.721885 | 0.171277 | 0.001859 | 0.382755 | 1.054725 | 9169.762094 | 1.000456 |
| **beta_age** | 0.009383 | 0.005532 | 0.000040 | -0.001328 | 0.020257 | 22009.292814 | 0.999959 |

Before I ran this test, I made a prediction that the model including the woman's age and whether she had a lot of children living at home would be the best one. I also predicted that being older and having a lot of kids at home would be associated with using contraception more. These predictions were made based on life experience. This batch of predictions was almost correct. It turns out that the sign of beta_age was a tiny bit negative. Perhaps women who are old enough that they are no longer in their childbearing years stopped using contraception because they didn't need it anymore.

I also predicted that the model including age would outperform the other two, with the same sign in the regression as above. This prediction was completely accurate.

Finally, I wasn't sure whether the Fixed Effect model would be preferable to the Variable Effect model. There was something that felt wrong to me about fitting a different slope to each district, but I couldn't quite pin it down. The WAIC clearly dislikes the variable effect model though. Before we delve in too deeply to conclusions based on the WAIC, let's say why the WAIC is a suitable metric in this case. Here we have a "nested" series of model that share the same likelihood function. The only difference is the set of features we are using. This is an ideal scenario for the WAIC.

Conclusions based on the WAIC:

- The model including the district, urban, age, and whether a woman has a lot of kids is the best one
- The model including the district, urban, and age is only slightly worse
- The fixed effect model with one overall rate per district weaker, but would get some weight in a model ensemble because it adds diversity
- The variable effect model is very poor; it is probably overfit

Conclusions based on the regression coefficients, pertaining to women polled in Bangladesh on this survey

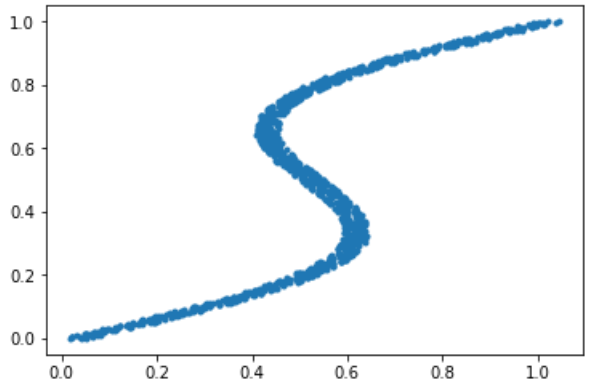
- There is quite a bit of variation in the rate of contraceptive use in different districts in the country
- Women who live in urban areas are more likely to use contraception
- Women who have a lot of children living at home with them (3 or more) are more likely to use contraception

## Q2: Mixture of experts and mixture density networks to solve inverse problems

What if you had to predict a one-to-many function? The data provided below comes from a dataset generated by Chris Bishop (yes that Bishop) to explain the models mentioned in the title above. We have included pdfs from his book which describe these models in some detail. We saw this model earlier in HW where we did an EM like algorithm to obtain a mixture of regressions.

The data is in `one-to-many.csv`

When we plot the data it looks like this. Notice both the uneven sampling (more towards the center), and the "more than one y" for a given x.




Normal regression approaches to modeling such a function wont work, as they expect the function to be a proper mathematical function, that is, single valued.

These kind of problems are called **inverse problems**, where more than one input state leads to an output state, and we have to try and model these multiple input states.

A mixture of gaussians (or other distributions) might is a sensible way to do this.

You choose one of the gaussians with some probability. The nean of the gaussian is then given by some regression function, say for example a straight line. We could additionally fix the standard deviation or model it as well.

Thus, for each component Gaussian, we choose a functional form for the mean and standard deviation. So our model looks something like this:

$$f(x) = \sum_i \lambda_i g_i(x)$$

Say we fit a model with 3 gaussians to this data. Such a model cannot fit the function above. Notice for example that at $x = 0.2$ only one of the gaussians will dominate, different from the situation at $x = 0.5$. This means that the probabilities of "belonging" to one or the other gaussians is also changing with $x$.

If we allow the mixing probabilities to depend on $x$, we can model this situation.

$$f(x) = \sum_i \lambda_i(x) g_i(x)$$

Such a model is called a "mixture of experts" model. The idea is that one "expert" gaussian is responsible in one sector of the feature space, while another expert is responsible in another sector.

You can think of this model as implementing a "standard" gaussian mixture at each "point" x, with the added complexity that all of the means, standard deviations, and mixture probabilities change from one x to another.

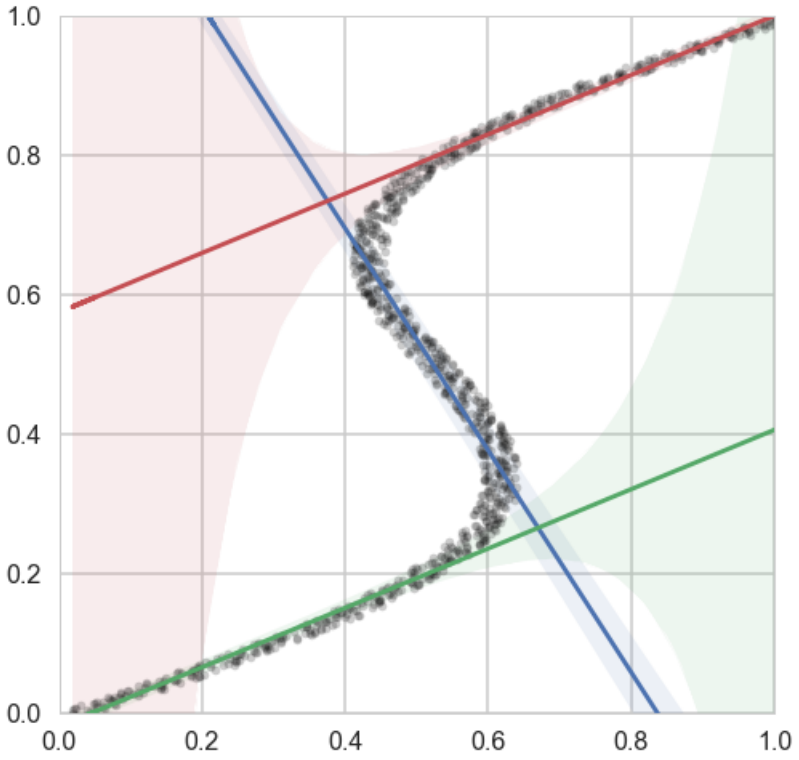
See https://www.cs.toronto.edu/~hinton/absps/hme.pdf (https://www.cs.toronto.edu/~hinton/absps/hme.pdf) and http://www.ee.hacettepe.edu.tr/~eyuksel/Publications/2012_TwentyYearsofMixtureofExperts.pdf (http://www.ee.hacettepe.edu.tr/~eyuksel/Publications/2012_TwentyYearsofMixtureofExperts.pdf) for more details. I found the latter clearer and easier to understand.

For this entire question you might find diagram code from here (https://github.com/hardmaru/pytorch_notebooks/blob/master/mixture_density_networks.ipynb) useful. Take with attribution.

We will assume we have **3 gaussians**.

## Part A: Variational Mixture of experts

We'll construct a gaussian mixture model of 3 "expert" linear regressions. The idea is to create a fit which looks like this:




Here the three regression lines work in different regions of $f$. We want a pricipled way to sample from this model and to be able to produce posteriors and posterior-predictives.

There are 3 parts to this model. First the means of the gaussians in the mixture are modeled with linear regression as shown in the picture above. We will also model $log(\sigma)$ for each gaussian in the mixture as a linear regression as well ($\sigma$ needs to be positive).

We now need to model the mixture probabilities, i.e., the probabilities required to choose one or the other gaussian. These mixing probabilities, the $\lambda$s will be modeled as a softmax regression (ie do a linear regression and softmax it to get 3 probabilities).

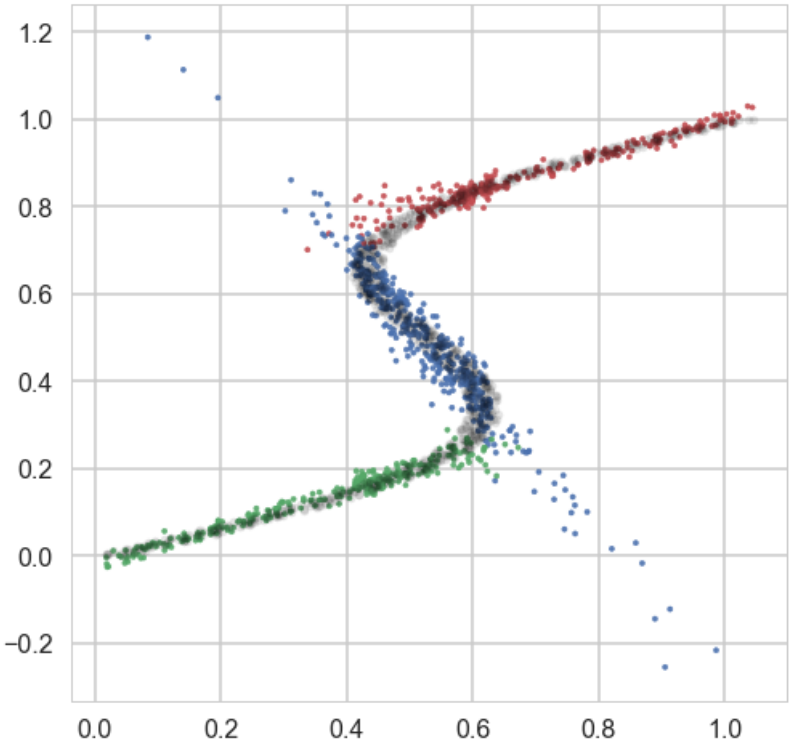
**A1** Write a pymc3 model for this problem. For all biases and weights in your regressions, assume N(0,5) priors. Add noise 0.01 to each of the three $\sigma$s to make sure you dont have a collapsed 0 width gaussian, ie we want some data in every cluster. (Thus to get the final $\sigma$, you will exponentiate your regression for $log(\sigma)$ and add 0.01.)

**A2** Fit this model variationally for about 50,000 iterations using the adam optimizer. ( `obj_optimizer=pm.adam()` ) Plot the ELBO to make sure you have converged. Print summaries and traceplots for the means, $\sigma$s and probabilities.

**A3** Plot the mean posteriors with standard deviations against x. Also produce a diagram like the one above to show the mean"s with standard deviations showing their uncertainty overlaid on the data.

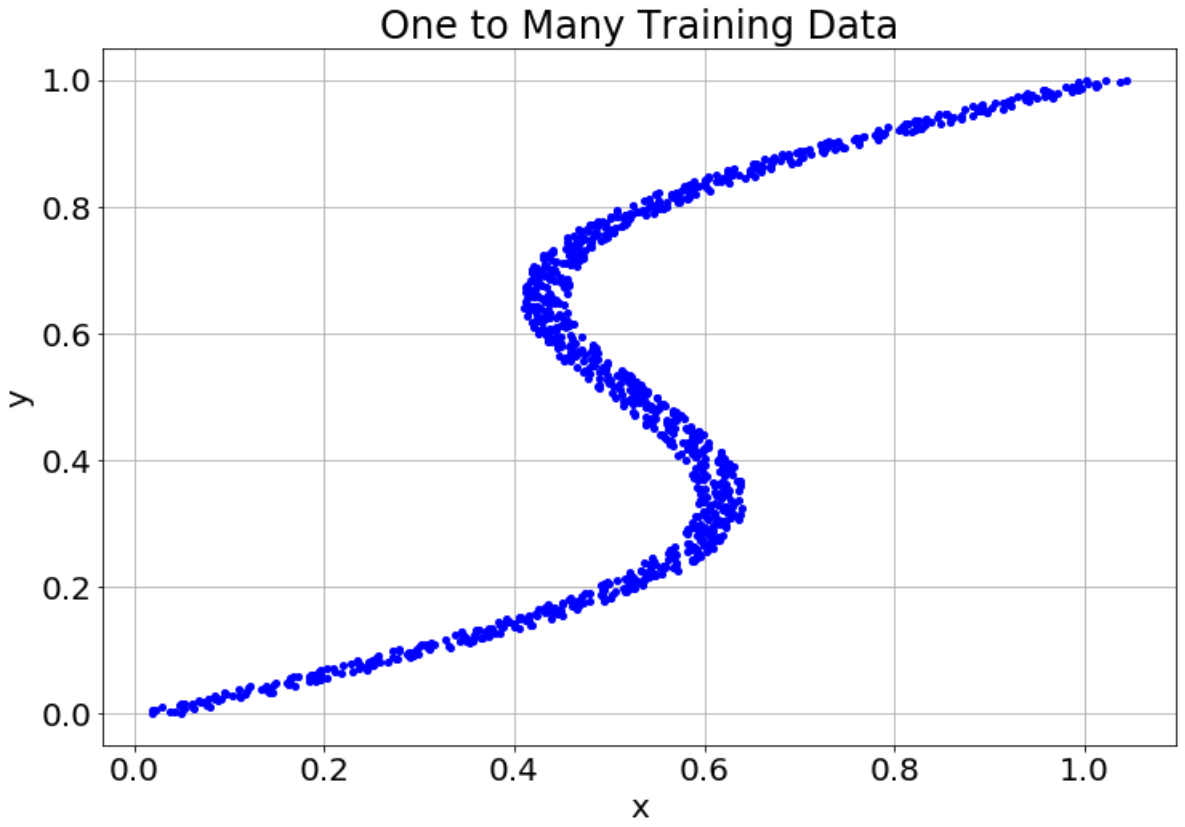
**A4** Plot the posterior predictive (mean and variance) as a function of x) for this model (using `sample_ppc` for example). Why does the posterior predictive look nothing like the data?

**A5** Make a "correct" posterior predictive diagram by taking into account which "cluster" or "regression line" the data is coming from. To do this you will need to sample using the softmax probabilities. A nice way to do this is "Gumbel softmax sampling". See http://timvieira.github.io/blog/post/2014/07/31/gumbel-max-trick/ (http://timvieira.github.io/blog/post/2014/07/31/gumbel-max-trick/) for details. Color-code the predictive samples with the gaussian they came from. Superimpose the predictive on the original data. You may want to contrast a prediction from a point estimate at the mean values of the $\mu$ and $\sigma$ traces at a given x (given the picked gaussian) to the "full" posterior predictive obtained from sampling from the entire trace of $\mu$ and $\sigma$ and $\lambda$. The former diagram may look something like this:




In [61]:
```
# Load persisted table of variables
fname: str = 'mixture.pickle'
vartbl: Dict = load_vartbl(fname)

# Set plot style
mpl.rcParams.update({'font.size': 20})

# Set random seed for reproducibility
np.random.seed(42)
```

In [62]:
```
# Load the data
df = pd.read_csv('one-to-many.csv')
x = df.x.values
y = df.target.values

# Exploratory plot of the data
fig, ax = plt.subplots(figsize=[12,8])
ax.set_title('One to Many Training Data')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.plot(x, y, color='b', linewidth=0, marker='o', markersize=4)
ax.grid()
```




**A1** Write a pymc3 model for this problem. For all biases and weights in your regressions, assume N(0,5) priors. Add noise 0.01 to each of the three $\sigma$s to make sure you dont have a collapsed 0 width gaussian, ie we want some data in every cluster. (Thus to get the final $\sigma$, you will exponentiate your regression for $log(\sigma)$ and add 0.01.)

In [63]:
```
# Shift applied to sigmas
sigma_shift = 0.01

with pm.Model() as model:
    # The number of data points
    N: int = len(x)
    # The number of functions
    K: int = 3

    # Setting for mu and sigma in normal priors
    normal_prior_mu = 0.0
    normal_prior_sd = 5.0

    # Reshape x to an Nx1 vector (makes matrix multiplications easier to follow)
    xt = tt.reshape(x, (N,1))

    # The parameters for the three regression lines; mean(x) = alpha + x*beta; shape (K)
    alpha = pm.Normal(name='alpha', mu=normal_prior_mu, sd=normal_prior_sd, shape=(K,))
    beta = pm.Normal(name='beta', mu=normal_prior_mu, sd=normal_prior_sd, shape=(K,))

    # Reshape into row vectors
    alpha_row = tt.reshape(alpha, (1,K))
    beta_row = tt.reshape(beta, (1,K))

    # The mean of the regression line, mu, is a deterministic function of x = alpha + x * beta; shape (N,
    mu = pm.Deterministic('mu', tt.add(alpha_row, tt.dot(xt, beta_row)))

    # The parameters for the regression line of log-sigma for each gaussian
    log_sigma_alpha = pm.Normal(name='log_sigma_alpha', mu=normal_prior_mu, sd=normal_prior_sd, shape=(K
    log_sigma_beta = pm.Normal(name='log_sigma_beta', mu=normal_prior_mu, sd=normal_prior_sd, shape=(K,)
    # Reshape into row vectors
    log_sigma_alpha_row = tt.reshape(log_sigma_alpha, (1,K))
    log_sigma_beta_row = tt.reshape(log_sigma_beta, (1,K))
    # log_sigma for each gaussian is deterministic
    log_sigma = tt.add(log_sigma_alpha_row, tt.dot(xt, log_sigma_beta_row))

    # Sigma for each gaussian is deterministic; include shift of 0.01; shape (N,K)
    sigma = pm.Deterministic('sigma', tt.exp(log_sigma) + sigma_shift)

    # Weighting factors weight_i are modeled as linear regressions also
    weight_alpha = pm.Normal(name='weight_alpha', mu=normal_prior_mu, sd=normal_prior_sd, shape=(K,))
    weight_beta = pm.Normal(name='weight_beta', mu=normal_prior_mu, sd=normal_prior_sd, shape=(K,))
    # Reshape into row vectors
    weight_alpha_row = tt.reshape(weight_alpha, (1,K))
    weight_beta_row = tt.reshape(weight_beta, (1,K))

    # The weighting factors are a softmax of weight_alpha + x*weight_beta; shape (N,K)
    weight = pm.Deterministic('weight', softmax(weight_alpha_row + tt.dot(xt, weight_beta_row)))

    # Sample points using a pymc3 NormalMixture
    # See lecture notes 25, p. 44
    y_obs = pm.NormalMixture('y_obs', w=weight, mu=mu, sd=sigma, observed=y)
```

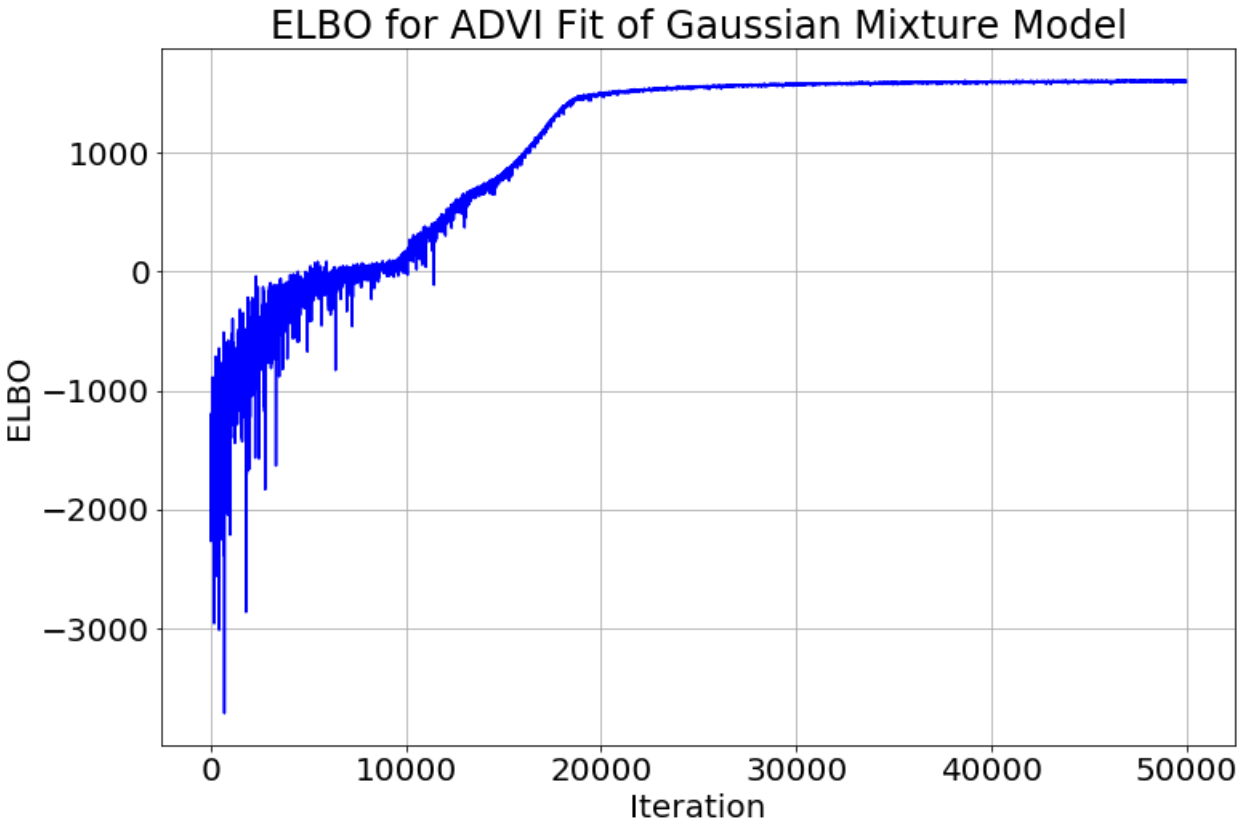
**A2** Fit this model variationally for about 50,000 iterations using the adam optimizer. ( `obj_optimizer=pm.adam()` ) Plot the ELBO to make sure you have converged. Print summaries and traceplots for the means, $\sigma$s and probabilities.

In [64]:
```
# Number of iterations for ADVI fit
num_iters: int = 50000

# Fit the model using ADVI
# Tried to fit using FullRankADVI as well; results were horrible
try:
    advi = vartbl['advi']
    print(f'Loaded ADVI fit for Gaussian Mixture Model.')
except:
    print(f'Running ADVI fit for Gaussian Mixture Model...')
    advi = pm.ADVI(model=model)
    advi.fit(n=num_iters, obj_optimizer=pm.adam(),
             callbacks=[CheckParametersConvergence()])
    vartbl['advi'] = advi
    save_vartbl(vartbl, fname)
```

```
Loaded ADVI fit for Gaussian Mixture Model.
```

**Plot the ELBO**

In [65]:
```
def plot_elbo(elbo, plot_step, title):
    """Generate the ELBO plot"""
    fig, ax = plt.subplots(figsize=[12,8])
    ax.set_title(title)
    ax.set_xlabel('Iteration')
    ax.set_ylabel('ELBO')
    n = len(elbo)
    plot_x = np.arange(0,n,plot_step)
    plot_y = elbo[::plot_step]
    ax.plot(plot_x, plot_y, color='b')
    ax.grid()
    return fig


def plot_elbo_log(elbo, plot_step, title):
    """Generate the ELBO plot"""
    fig, ax = plt.subplots(figsize=[12,8])
    ax.set_title(title)
    ax.set_xlabel('Iteration')
    ax.set_ylabel('Log ELBO')
    n = len(elbo)
    plot_x = np.arange(0,n,plot_step)
    # Un-normalized data points
    y = elbo[::plot_step]
    y_max = np.max(y)
    # Shift to maximum value of y is -1
    y = y - y_max - 1.0
    # Plot - log(-y)
    plot_y = -np.log(-y)
    ax.plot(plot_x, plot_y, color='b')
    ax.grid()
    return fig

# Plot the ELBO
fig = plot_elbo(-advi.hist, 10, 'ELBO for ADVI Fit of Gaussian Mixture Model')
```




We can wee that the ELBO has converged some time around 25,000 iterations.

### Draw Samples from the ADVI Fit

In [66]:
```
# Number of samples to draw
num_samples: int = 2000

# Draw parameter samples (trace)
# See lecture 24, p. 33 for example
try:
    trace = vartbl['trace']
    print(f'Loaded trace from ADVI fit of Gaussian Mixture Model.')
except:
    print(f'Drawing posterior samples (parameters) from ADVI fit of Gaussian Mixture Model...')
    trace = advi.approx.sample(num_samples)
    vartbl['trace'] = trace
    save_vartbl(vartbl, fname)
```

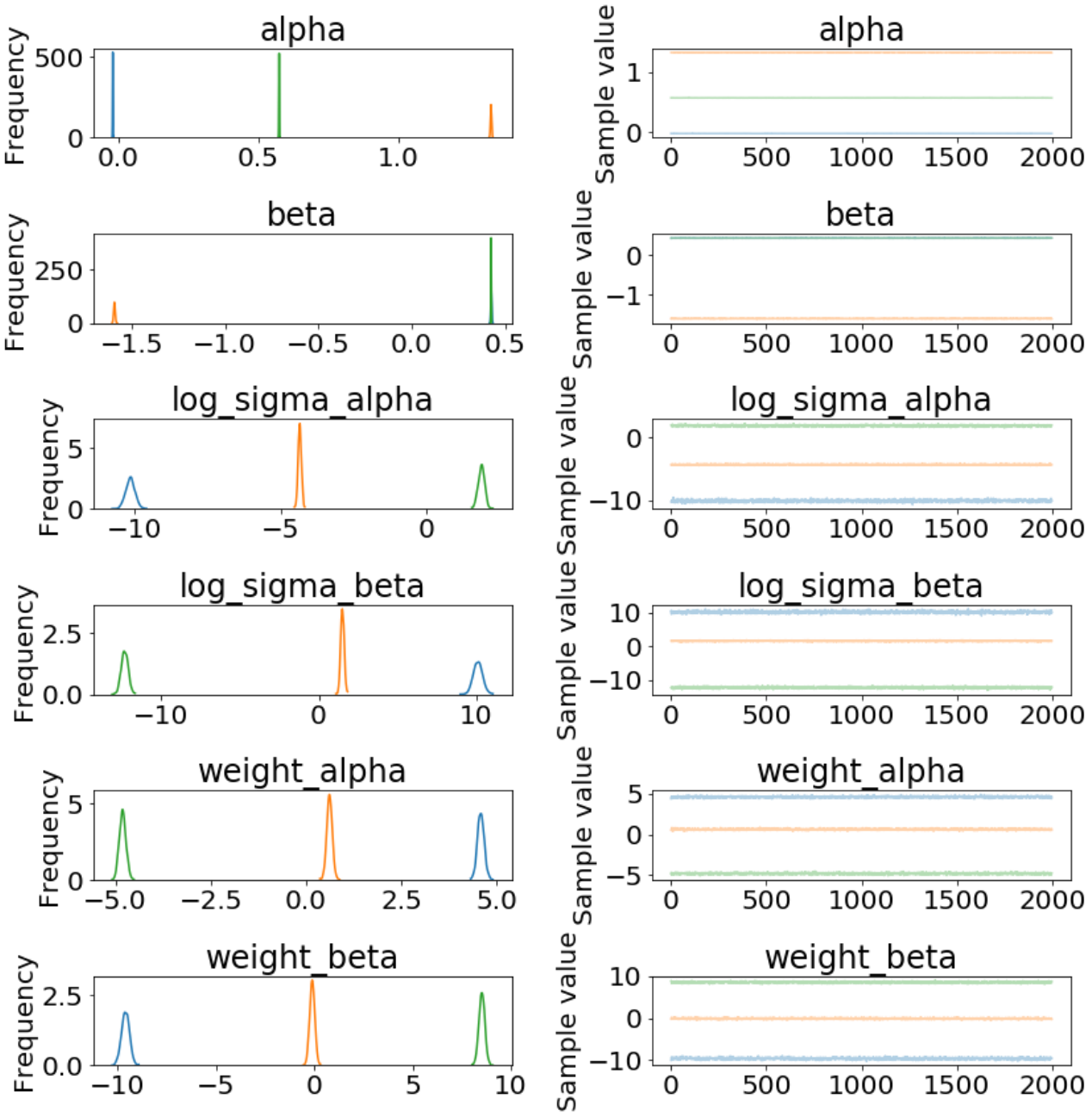
```
Loaded trace from ADVI fit of Gaussian Mixture Model.
```

### Print summaries and traceplots for the means, σ s and probabilities

In [67]:
```
varnames = ['alpha', 'beta', 'log_sigma_alpha', 'log_sigma_beta', 'weight_alpha', 'weight_beta']
display(pm.summary(trace, varnames=varnames))
```

| | mean | sd | mc_error | hpd_2.5 | hpd_97.5 |
|---|---|---|---|---|---|
| **alpha__0** | -0.020499 | 0.000728 | 0.000015 | -0.021913 | -0.019066 |
| **alpha__1** | 1.331399 | 0.001977 | 0.000045 | 1.327660 | 1.335351 |
| **alpha__2** | 0.573867 | 0.000747 | 0.000017 | 0.572407 | 0.575278 |
| **beta__0** | 0.425347 | 0.002474 | 0.000047 | 0.420366 | 0.430069 |
| **beta__1** | -1.592394 | 0.003926 | 0.000090 | -1.599842 | -1.584581 |
| **beta__2** | 0.424672 | 0.000975 | 0.000023 | 0.422842 | 0.426514 |
| **log_sigma_alpha__0** | -10.136210 | 0.153093 | 0.003445 | -10.433391 | -9.834685 |
| **log_sigma_alpha__1** | -4.342857 | 0.056152 | 0.001292 | -4.444397 | -4.226735 |
| **log_sigma_alpha__2** | 1.866666 | 0.107685 | 0.002590 | 1.650042 | 2.059463 |
| **log_sigma_beta__0** | 10.092550 | 0.287816 | 0.005712 | 9.544062 | 10.667523 |
| **log_sigma_beta__1** | 1.503633 | 0.108805 | 0.002503 | 1.299761 | 1.724808 |
| **log_sigma_beta__2** | -12.302023 | 0.213077 | 0.004837 | -12.718058 | -11.893271 |
| **weight_alpha__0** | 4.581370 | 0.086584 | 0.001744 | 4.404012 | 4.750421 |
| **weight_alpha__1** | 0.613306 | 0.070020 | 0.001611 | 0.471633 | 0.749088 |
| **weight_alpha__2** | -4.813862 | 0.085731 | 0.001965 | -4.965011 | -4.635323 |
| **weight_beta__0** | -9.600561 | 0.198017 | 0.004690 | -10.008297 | -9.232757 |
| **weight_beta__1** | -0.104839 | 0.126634 | 0.002626 | -0.344822 | 0.146629 |
| **weight_beta__2** | 8.520242 | 0.146722 | 0.003321 | 8.224076 | 8.795666 |

In [68]:
```
fig = pm.traceplot(trace, varnames=varnames)
```




These results are intuitive. Let's start with alpha and beta, the slopes of the three lines. Looking at the graph we can see that the there are three lines:

- red line: slope ~0.4, intercept ~0.6
- green line: slope ~0.4, intercept ~0.0
- blue line: slope ~-1.6, intercept ~1.3

Let's compare these lines to the mean parameter estimates. We can see

- beta_i = 0.43, alpha_i = 0.58; this is the red line
- beta_j = -1.59, alpha_j = 1.33; this is the blue line
- beta_k = 0.43, alpha_k = 0.02; this is the green line

The coefficients are indexed with i, j, and k because the order of three clusters is not defined across runs; their location is stable though. (I experimented with using a potential or ordering but it wasn't working and was not necessary to solve the problem). All the traceplot show pretty tight bands, indicating that the samples have found stable and consistent values for the lines (means of the gaussians vs. x), the sigmas (width of the gaussians vs. x), and weights vs x.

**A3** Plot the mean posteriors with standard deviations against x. Also produce a diagram like the one above to show the mean"s with standard deviations showing their uncertainty overlaid on the data.

**Produce a diagram like the one above to show the means with standard deviations showing their uncertainty overlaid on the data**

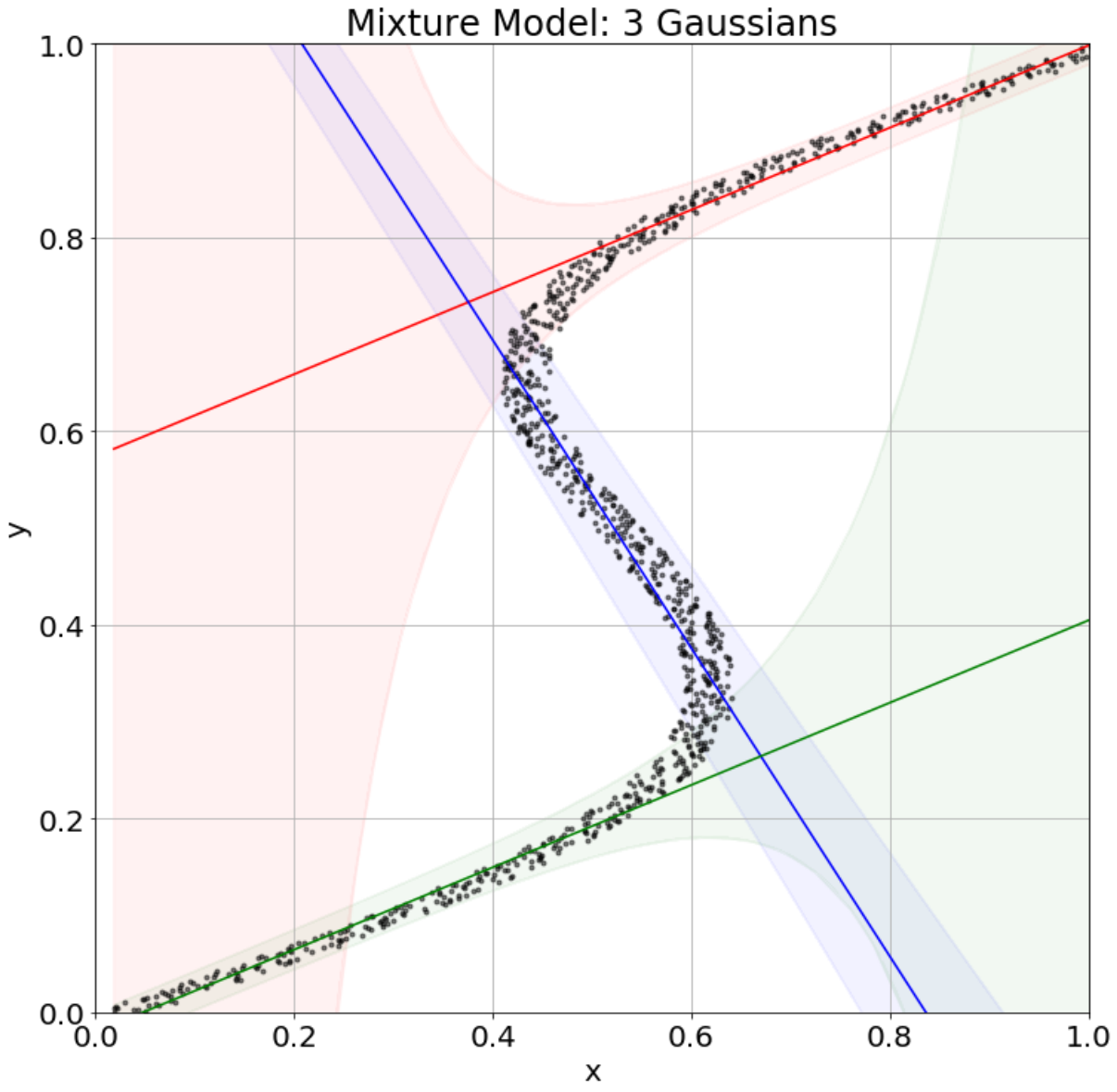
In [69]:
```
def standardize_gaussians(mu, sigma, weight):
    """Standardize the identities of the three gaussians"""
    # Get index to sort the three series for consistency
    mu_med = np.median(mu, axis=0)
    idx = np.argsort(mu_med, axis=0)[::-1]

    # Permute columns of all three arrays
    mu = mu.copy()[:, idx]
    sigma = sigma.copy()[:, idx]
    weight = weight.copy()[:, idx]

    return (mu, sigma, weight, idx)
```

In [70]:
```
def plot_gaussians(xx, mu, sigma, x, y, title):
    """Generate a plot with the envelope around each of the three gaussians in the mixture"""
    # Compute mu_lo and mu_hi for the envelopes
    mu_lo = mu - 2.0 * sigma
    mu_hi = mu + 2.0 * sigma

    # Frame for the plot
    fig, ax = plt.subplots(figsize=[12,12])
    ax.set_title(title)
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.set_xlim(0.0, 1.0)
    ax.set_ylim(0.0, 1.0)
    ax.grid()

    # Plot the data in black
    ax.plot(x, y, color='k', linewidth=0, marker='o', markersize=3, alpha=0.5)
    # Plot the three lines; match colors to problem
    ax.plot(xx, mu[:,0], color='r')
    ax.plot(xx, mu[:,1], color='b')
    ax.plot(xx, mu[:,2], color='g')
    # Fill in the color between the lower and upper bounds of each line
    ax.fill_between(xx, mu_lo[:,0], mu_hi[:,0], color='r', alpha=0.05)
    ax.fill_between(xx, mu_lo[:,1], mu_hi[:,1], color='b', alpha=0.05)
    ax.fill_between(xx, mu_lo[:,2], mu_hi[:,2], color='g', alpha=0.05)
    return fig
```

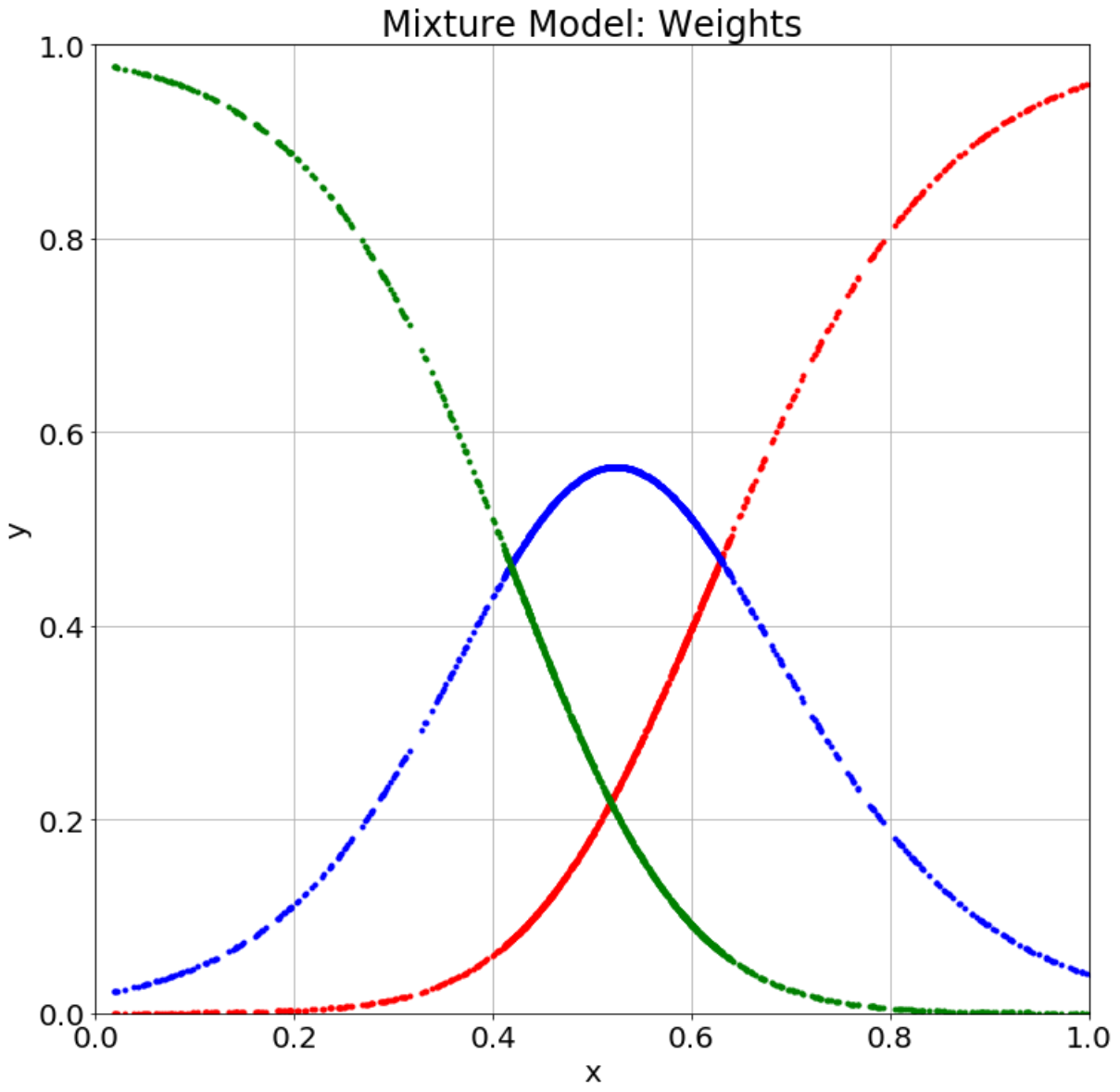
In [71]:
```
def plot_weights(x, weight, title):
    """Generate plot with the weights on the three gaussians"""
    fig, ax = plt.subplots(figsize=[12,12])
    ax.set_title(title)
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.set_xlim(0.0, 1.0)
    ax.set_ylim(0.0, 1.0)
    ax.grid()
    # Plot the weights
    markersize=3
    ax.plot(x, weight[:,0], color='r', linewidth=0, marker='o', markersize=markersize)
    ax.plot(x, weight[:,1], color='b', linewidth=0, marker='o', markersize=markersize)
    ax.plot(x, weight[:,2], color='g', linewidth=0, marker='o', markersize=markersize)
    return fig
```

In [72]:
```
# Posterior means for the six parameters
post_means = dict()
post_means['alpha'] = np.mean(trace['alpha'], axis=0)
post_means['beta'] = np.mean(trace['beta'], axis=0)
post_means['log_sigma_alpha'] = np.mean(trace['log_sigma_alpha'], axis=0)
post_means['log_sigma_beta'] = np.mean(trace['log_sigma_beta'], axis=0)
post_means['weight_alpha'] = np.mean(trace['weight_alpha'], axis=0)
post_means['weight_beta'] = np.mean(trace['weight_beta'], axis=0)

# Compute posterior means from the trace, which includes the intermediate results for mu, sigma, and weig
mu = np.mean(trace['mu'], axis=0)
sigma = np.mean(trace['sigma'], axis=0)
weight = np.mean(trace['weight'], axis=0)

# Standardize the identities of the three Guassians for consistent colors
mu, sigma, weight, idx = standardize_gaussians(mu, sigma, weight)
```

In [73]:
```
# Generate plot similar to one in the problem
fig = plot_gaussians(x, mu, sigma, x, y, 'Mixture Model: 3 Gaussians')
```



In [74]:
```
# Plot the weights
fig = plot_weights(x, weight, 'Mixture Model: Weights')
```

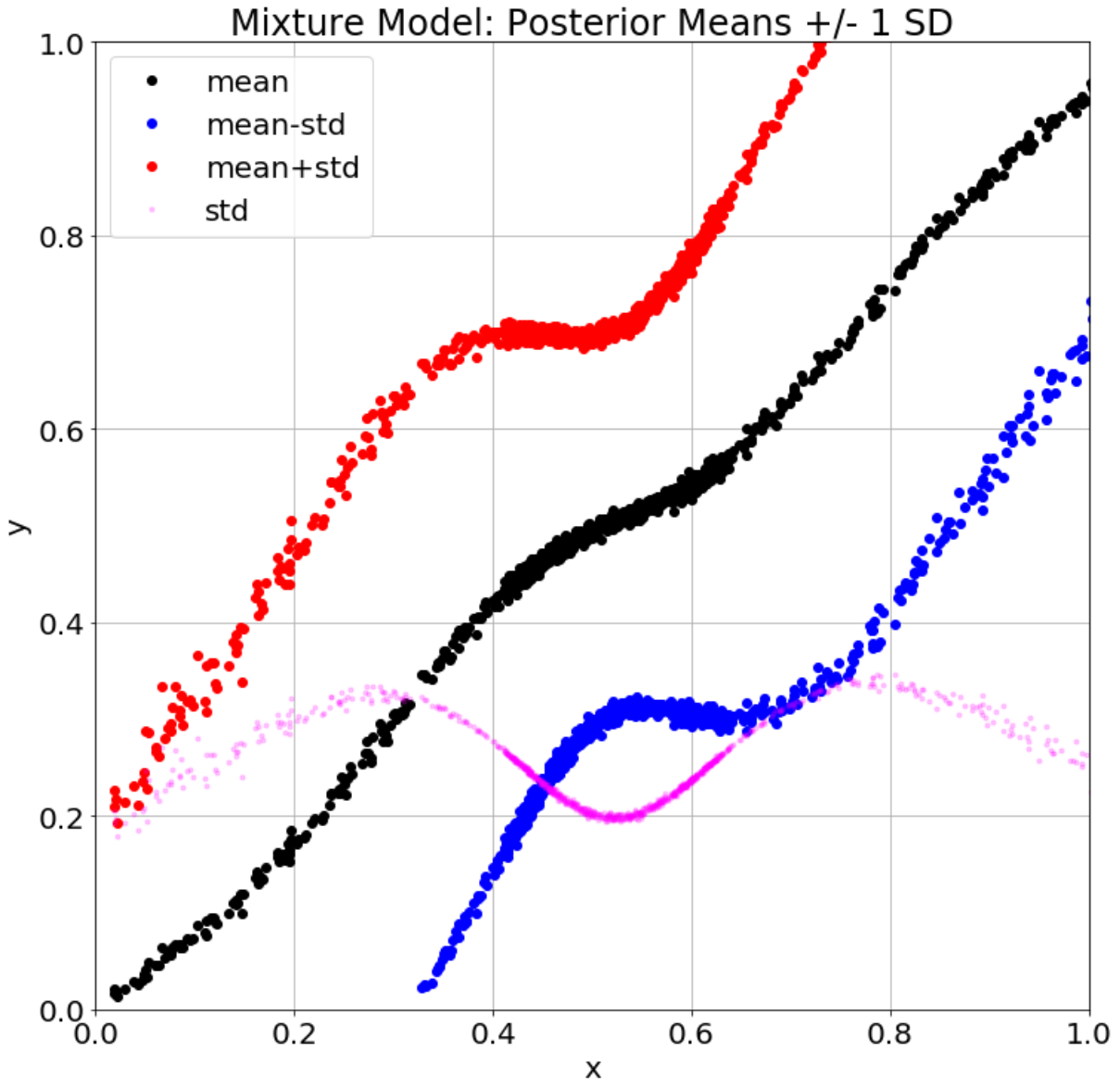



**A4** Plot the posterior predictive (mean and variance) as a function of x) for this model (using `sample_ppc` for example). Why does the posterior predictive look nothing like the data?

In [75]:
```
def plot_post_mean_std(x, y_mean, y_std, title):
    """Plot posterior mean and std for a set of samples"""
    fig, ax = plt.subplots(figsize=[12,12])
    ax.set_title(title)
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.set_xlim(0.0, 1.0)
    ax.set_ylim(0.0, 1.0)
    ax.grid()

    # Plot posterior mean of y, Shifted +/- 1 SD
    ax.plot(x, y_mean, color='k', linewidth=0, marker='o', label='mean')
    ax.plot(x, y_mean-y_std, color='b', linewidth=0, marker='o', label='mean-std')
    ax.plot(x, y_mean+y_std, color='r', linewidth=0, marker='o', label='mean+std')
    # The standard deviation
    ax.plot(x, y_std, color='magenta', linewidth=0, marker='o', label='std', markersize=3, alpha=0.2)
    ax.legend()
```

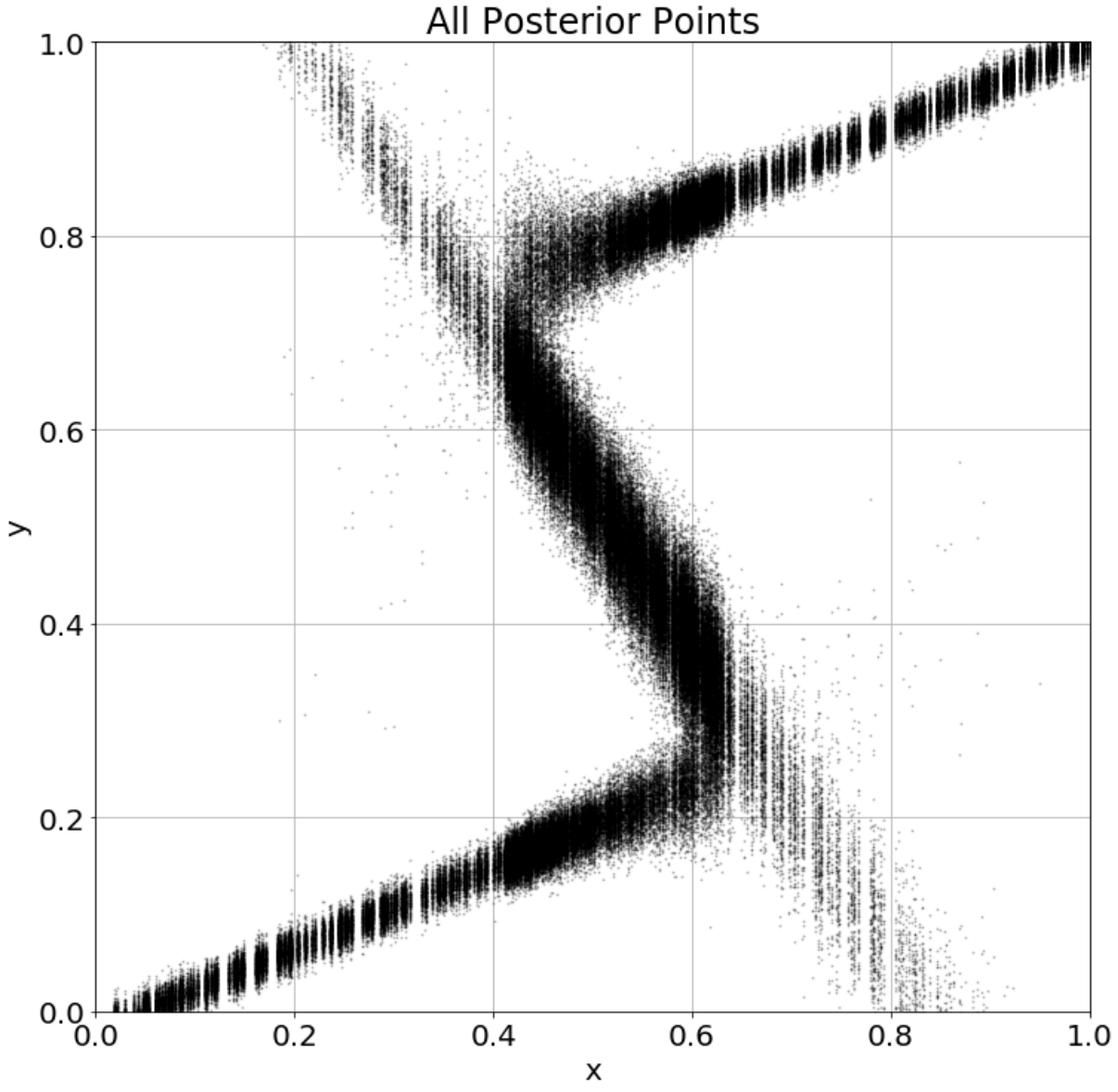
In [76]:
```
# Draw posterior predictive
# See lecture 24, p. 33 for example
try:
    pred = vartbl['pred']
    print(f'Loaded posterior predictive from ADVI fit of Gaussian Mixture Model.')
except:
    print(f'Drawing posterior predictive from ADVI fit of Gaussian Mixture Model...')
    pred = pm.sample_ppc(trace, model=model)
    vartbl['pred'] = pred
    save_vartbl(vartbl, fname)

# Extract y_pp as an array; shape (num_samples, N)
y_pp = pred['y_obs']
# Mean and standard deviation of y
y_mean = np.mean(y_pp, axis=0)
y_std = np.std(y_pp, axis=0)

# Plot the posterior means and stds
plot_post_mean_std(x, y_mean, y_std, 'Mixture Model: Posterior Means +/- 1 SD')
```

```
Loaded posterior predictive from ADVI fit of Gaussian Mixture Model.
```




**Bonus Chart: All Posterior Points - Understanding Why the Posterior Mean Doesn't Match the Data**

In [77]:
```
def plot_post_all(x, y_pp, title):
    """Plot all the posterior points in one big cloud"""
    fig, ax = plt.subplots(figsize=[12,12])
    ax.set_title(title)
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.set_xlim(0.0, 1.0)
    ax.set_ylim(0.0, 1.0)
    ax.grid()

    # Plot all the posterior points
    num_samples = y_pp.shape[0]
    x_plot = np.tile(x, (num_samples, 1))
    ax.plot(x_plot, y_pp, color='k', linewidth=0, marker='o', markersize=1, alpha=0.2)
    return fig

# Plot the full cloud of posterior points to show how the posterior mean fails
fig = plot_post_all(x, y_pp[0:200, :], 'All Posterior Points')
```




**Why does the posterior predictive look nothing like the data?**

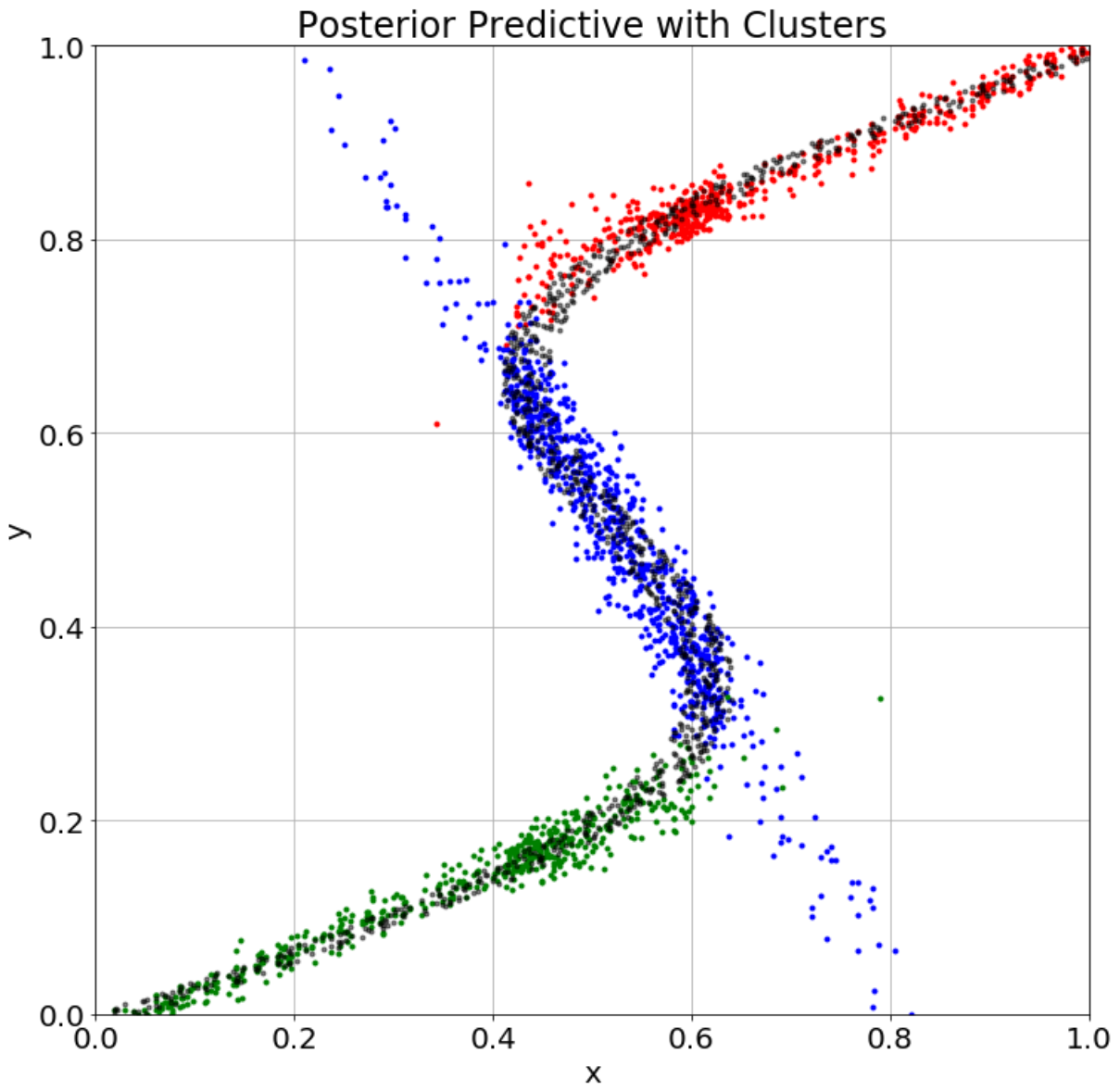
Because this data set has a one to many relationship, the posterior predictive drawn this is trying to average out over different y values that share the same x value. The sampling process will only recover the shape of the data if it also knows which cluster the data was drawn from. The x-value by itself is insufficient.

The chart directly above makes it even more clear what is going wrong. It shows a cloud of all the posterior points generated by the PPC. While there is indeed a fairly heavy cloud of points overlapping with the data, there are quite a few spurious points on the extrapolations of the lines where there is no data. This is especially true for the blue line with a large negative slope. The presense of these "phantom points" in the data doesn't hurt the likelihood too much, because there are still plenty of points where the real data is located. But they have a large enough effect on the posterior mean to completely throw it off. To quote Yoda, "that is why you fail."


**A5** Make a "correct" posterior predictive diagram by taking into account which "cluster" or "regression line" the data is coming from. To do this you will need to sample using the softmax probabilities. A nice way to do this is "Gumbel softmax sampling". See http://timvieira.github.io/blog/post/2014/07/31/gumbel-max-trick/ (http://timvieira.github.io/blog/post/2014/07/31/gumbel-max-trick/) for details. Color-code the predictive samples with the gaussian they came from. Superimpose the predictive on the original data. You may want to contrast a prediction from a point estimate at the mean values of the $\mu$ and $\sigma$ traces at a given x (given the picked gaussian) to the "full" posterior predictive obtained from sampling from the entire trace of $\mu$ and $\sigma$ and $\lambda$. The former diagram may look something like this:

In [78]:
```
def plot_post_cluster(x_pp, y_pp, cluster, x, y, title):
    """Generate plot with the corrected posterior predictive"""
    fig, ax = plt.subplots(figsize=[12,12])
    ax.set_title(title)
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.set_xlim(0.0, 1.0)
    ax.set_ylim(0.0, 1.0)
    ax.grid()
    # Plot the three clusters in different colors
    markersize=3
    ax.plot(x_pp[cluster==0], y_pp[cluster==0], color='r', linewidth=0, marker='o', markersize=markersize
    ax.plot(x_pp[cluster==1], y_pp[cluster==1], color='b', linewidth=0, marker='o', markersize=markersize
    ax.plot(x_pp[cluster==2], y_pp[cluster==2], color='g', linewidth=0, marker='o', markersize=markersize
    # Plot the original data too
    ax.plot(x, y, color='k', linewidth=0, marker='o', markersize=markersize, alpha=0.5)
    return fig
```

In [79]:
```
# Extract the parameters mu, sigma, and weight from the posterior samples
# These all have shape (num_samples, N, 3)
# Use idx to standardize the identities of the clusters!
mu = trace['mu'][:,:,idx]
sigma = trace['sigma'][:,:,idx]
weight = trace['weight'][:,:,idx]

# Perform a simple "ancestral sampling" style strategy
# (I don't see the need for a fancy Gumbel sampling approach here!)
cluster = np.zeros(num_samples, dtype=np.int8)
x_pred_cl = np.zeros(num_samples)
y_pred_cl = np.zeros(num_samples)
for i in range(num_samples):
    # Draw a random row of the sample data
    row_num = np.random.choice(num_samples)
    # Cycle throught the x's in order
    col_num = i % N
    # Sample the cluster assignments according to the weights here
    cluster_i = np.random.choice(a=K, p=weight[row_num, col_num])
    # Draw a sample from this normal
    x_pred_cl[i] = x[col_num]
    y_pred_cl[i] = np.random.normal(loc = mu[row_num, col_num, cluster_i],
                                    scale = sigma[row_num, col_num, cluster_i])
    # Save the cluster assignment
    cluster[i] = cluster_i

# Plot the corrected posterior predictor based on sampled clusters
fig = plot_post_cluster(x_pred_cl, y_pred_cl, cluster, x, y, 'Posterior Predictive with Clusters')
```




## Part B. Mixture Density Network

A mixture density network (see the enclosed Chapter 5 excerpt from Bishop or https://publications.aston.ac.uk/373/1/NCRG_94_004.pdf (https://publications.aston.ac.uk/373/1/NCRG_94_004.pdf)) is very closely related to the mixture of experts model. The difference is that we fit the regressions using a neural network where hidden layers are shared amongst the mean, sigma, and mixing probability regressions. (We could have fit 3 separate neural networks in Part A but opted to fit linear regressions for simplicity)

(More explanation here (https://github.com/hardmaru/pytorch_notebooks/blob/master/mixture_density_networks.ipynb). You are welcome to take code from here with attribution.)

You job here is to construct a multi-layer perceptron model with a linear hidden layer with 20 units followed by a `Tanh` activation. After the activation layer, 3 separate linear layers with `n_hidden` inputs and `n_gaussian=3` outputs will complete the network. The probabilities part of the network is then passed through a softmax. The means part is left as is. The sigma part is exponentiated and 0.01 added, as in part A

Thus the structure looks like:

```
input:1 --linear-> n_hidden -> Tanh --linear-->n_gaussians      ...mu
                            --linear-->n_gaussians->softmax     ...lambda
                            --linear-->n_gaussians->exp + 0.01  ...sigma
```

We then need to use a loss function for the last layer of the network.

Using the mean-squared-error loss is not appropriate as the expected value of samples drawn from the sampling distribution of the network will not reflect the 3-gaussian structure (this is the essence of the difference between A4 and A5 above). Thus we'll use the negative loss likelihood of the gaussian mixture model explicitly.

**B1**: Write the network as a class `MixtureDensityNetwork` which inherits from pytorch `nn.Module`. Implement a constructor which allows at-least the number of hidden layers to be varied. Also implement the `forward` method.

**B2**: Train the network using the Adam or similiar optimizer and gradient descent/SGD. Make sure your loss converges and plot this convergence.

**B3**: Plot the MLE parameters against x. Make a plot similar to A3 above where you overlay the "means" of the gaussians against the data. Plot traces of the mu/sigma/lambda as an aid in debugging.

**B4**: Sample from the sampling distributions at the estimated point values of $\mu$ and $\sigma$ (given cluster) to make a plot similar to A5 above

**To think but not to hand in** What are the differences between a mixture density network and the mixture of experts. How do these differences translate to feature space? What would happen if we took the shared hidden layer nonlinearity (Tanh) out?

**B1**: Write the network as a class `MixtureDensityNetwork` which inherits from pytorch `nn.Module`. Implement a constructor which allows at-least the number of hidden layers to be varied. Also implement the `forward` method.

In [80]:
```
class MixtureDensityNetwork(nn.Module):
    """Implement a mixture density network as a torch.nn Module subclass."""
    # Citation: the following example was consulted in developing this class
    # No code was copy / pasted into what is presented here
    # (a reference implementation was temporarily copied for testing purposes)
    # https://github.com/hardmaru/pytorch_notebooks/blob/master/mixture_density_networks.ipynb
    def __init__(self, n_hidden: int, n_gaussian: int):
        # Initialize the parent instance of nn.Module
        super(MixtureDensityNetwork, self).__init__()

        # The first layer is linear -> n_hidden followed by a tanh activation
        # z should be viewed as the hidden activations
        self.z = nn.Sequential(nn.Linear(1, n_hidden), nn.Tanh())

        # A linear layer to predict mu from z_h
        self.mu = nn.Linear(n_hidden, n_gaussian)

        # A linear layer followed by exp with an offset to predict sigma
        self.log_sigma = nn.Linear(n_hidden, n_gaussian)

        # A linear layer followed by a softmax to predict the mixture weights
        self.weight_z = nn.Linear(n_hidden, n_gaussian)

    def forward(self, x):
        """Forward mode for this network"""
        # Compute the hidden activations z_h
        z = self.z(x)
        # Compute the mean mu and standard deviation sigma
        mu = self.mu(z)
        sigma = torch.exp(self.log_sigma(z)) + sigma_shift
        # Compute the weights
        weight = nn.functional.softmax(self.weight_z(z), dim=-1)
        # Return a tuple with weight, sigma, and mu
        return weight, sigma, mu

    def set_x(self, x: np.ndarray):
        """Bind x to the network; passed as a numpy array and converted to an autograd Variable"""
        # Length of data, N
        N: int = len(x)
        # Create torch tensor from x
        x_tensor = torch.from_numpy(np.float32(x).reshape((N,1)))
        # Create torch autograd variable from x_tensor and bind it to the network
        self.x = Variable(x_tensor)

    def set_y(self, y: np.ndarray):
        """Bind x to the network; passed as a numpy array and converted to an autograd Variable"""
        # Length of data, N
        N: int = len(y)
        # Create torch tensor from x
        y_tensor = torch.from_numpy(np.float32(y).reshape((N,1)))
        # Create torch autograd variable from y_tensor and bind it to the network
        self.y = Variable(y_tensor)

    def bind_data(self, x: np.ndarray, y: np.ndarray):
        """Bind the x and y data to the network as autograd variables."""
        # Dispatch calls to set_x and set_y
        self.set_x(x)
        self.set_y(y)

    def gaussian(self, mu, sigma, y_variable):
        """
        Evaluate the probability density of y on the gaussian distribution
        Used to compute the likelihood and the loss function
        """
        # Normalization factor for the gaussian distribution
        norm = 1.0 / np.sqrt(2.0*np.pi)
        # Compute inverse of sigma
        sigma_inv = torch.reciprocal(sigma)
        # The error in standardized z units is (y - mu) / sigma
        # Difference between predicted y and mu has shape (N,num_gaussian)
        # call to y.expand_as(mu) is like numpy broadcasting, expands from (N,) to (N,num_gauassian)
```

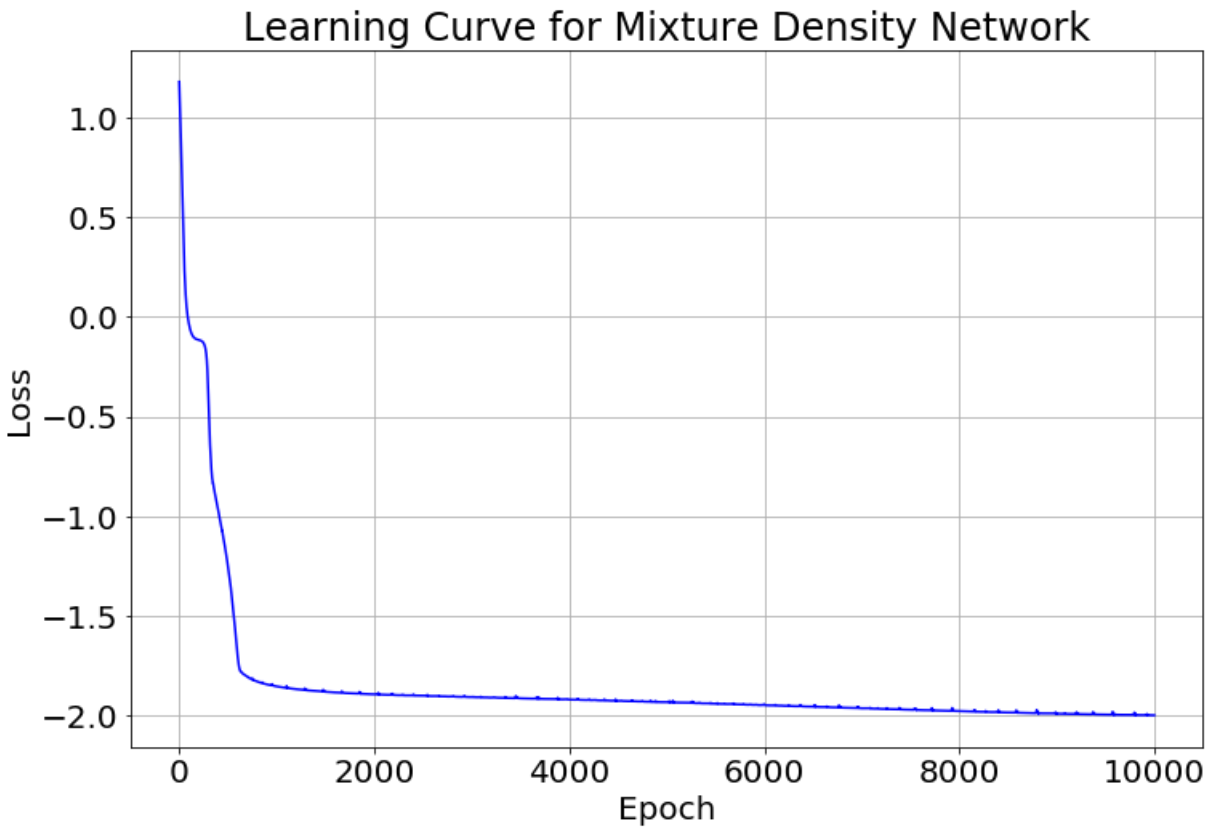
```
        z = sigma_inv * (y_variable.expand_as(mu) - mu)
        # The probability density is f(z) = norm / sigma * exp(-1/2 z^2)
        return (sigma_inv * torch.exp(-0.5 * z * z)) * norm

    def loss_func(self, weight, mu, sigma):
        """Compute the loss function with these weights and parameter values."""
        # Compute the likelihood of the predictions
        like = self.gaussian(mu, sigma, self.y) * weight
        # Add this up over all num_gaussian classes
        like = torch.sum(like, dim=1)
        # The loss on each sample is the negative log likelihood
        loss = -torch.log(like)
        # Return the mean loss across the full data set (all N samples)
        return torch.mean(loss)

    def train(self, num_epochs: int):
        """Train this network with an Adam optimizer"""
        # Initialize array of training losses
        self.loss_history = np.zeros(num_epochs)
        # Use Adam optimizer
        self.optimizer = torch.optim.Adam(self.parameters())
        # Train over num_epochs epochs
        for epoch in range(num_epochs):
            # Evaluate the network on the full set of x data
            # weight_variable, sigma_variable, mu_variable = network(x_variable)
            weight_variable, sigma_variable, mu_variable = self(self.x)
            # Compute the loss function; this is differentiable thanks to autograd
            loss = self.loss_func(weight_variable,  mu_variable, sigma_variable)
            # Take one step with the optimizer to minimize the loss
            self.optimizer.zero_grad()
            loss.backward()
            self.optimizer.step()
            # Save the loss
            loss_current = loss.item()
            self.loss_history[epoch] = loss_current
            # Status update
            if epoch % 500 == 0:
                print(f'Epoch {epoch:5d}; loss {loss_current:0.3f}')
```

**B2**: Train the network using the Adam or similiar optimizer and gradient descent/SGD. Make sure your loss converges and plot this convergence.

In [81]:
```
# Number of epochs to train the network
num_epochs: int = 10000
# File name for saving the trained network
network_fname = 'mixture_network.pt'

# Load the trained network if available; otherwise train it.
try:
    network = torch.load(network_fname)
    print(f'Loaded trained gaussian mixture network.')
except:
    # Instantiate a mixture density network with 20 hidden units and 3 gaussians
    network = MixtureDensityNetwork(n_hidden=20, n_gaussian=3)
    # Bind x and y data to the network
    network.bind_data(x, y)
    # Train the network
    network.train(num_epochs)
    # Save the model
    torch.save(network, network_fname)
```

```
C:\Python\Anaconda3\lib\site-packages\torch\serialization.py:400: UserWarning: Couldn't retrieve source
code for container of type MixtureDensityNetwork. It won't be checked for correctness upon loading.
  "type " + container_type.__name__ + ". It won't be checked "
C:\Python\Anaconda3\lib\site-packages\torch\serialization.py:434: SourceChangeWarning: source code of c
lass 'torch.nn.modules.container.Sequential' has changed. you can retrieve the original source code by
accessing the object's source attribute or set `torch.nn.Module.dump_patches = True` and use the patch
tool to revert the changes.
  warnings.warn(msg, SourceChangeWarning)
C:\Python\Anaconda3\lib\site-packages\torch\serialization.py:434: SourceChangeWarning: source code of c
lass 'torch.nn.modules.linear.Linear' has changed. you can retrieve the original source code by accessi
ng the object's source attribute or set `torch.nn.Module.dump_patches = True` and use the patch tool to
revert the changes.
  warnings.warn(msg, SourceChangeWarning)
C:\Python\Anaconda3\lib\site-packages\torch\serialization.py:434: SourceChangeWarning: source code of c
lass 'torch.nn.modules.activation.Tanh' has changed. you can retrieve the original source code by acces
sing the object's source attribute or set `torch.nn.Module.dump_patches = True` and use the patch tool
to revert the changes.
  warnings.warn(msg, SourceChangeWarning)

Loaded trained gaussian mixture network.
```

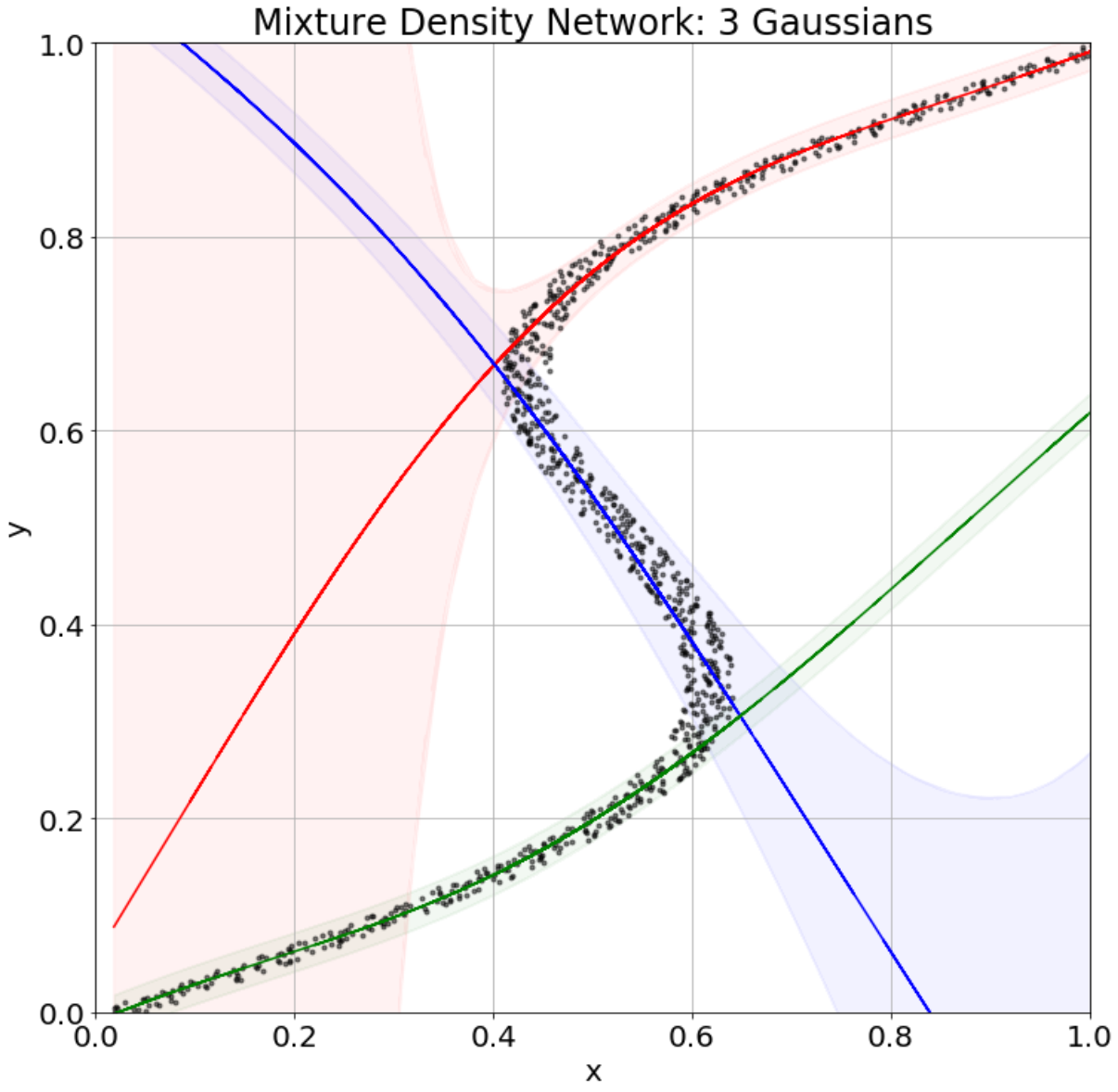
In [82]:
```
# Plot the loss history for the mixture density network training
fig, ax = plt.subplots(figsize=[12,8])
ax.set_title('Learning Curve for Mixture Density Network')
ax.set_xlabel('Epoch')
ax.set_ylabel('Loss')
ax.plot(np.arange(num_epochs), network.loss_history, color='b')
ax.grid()
```



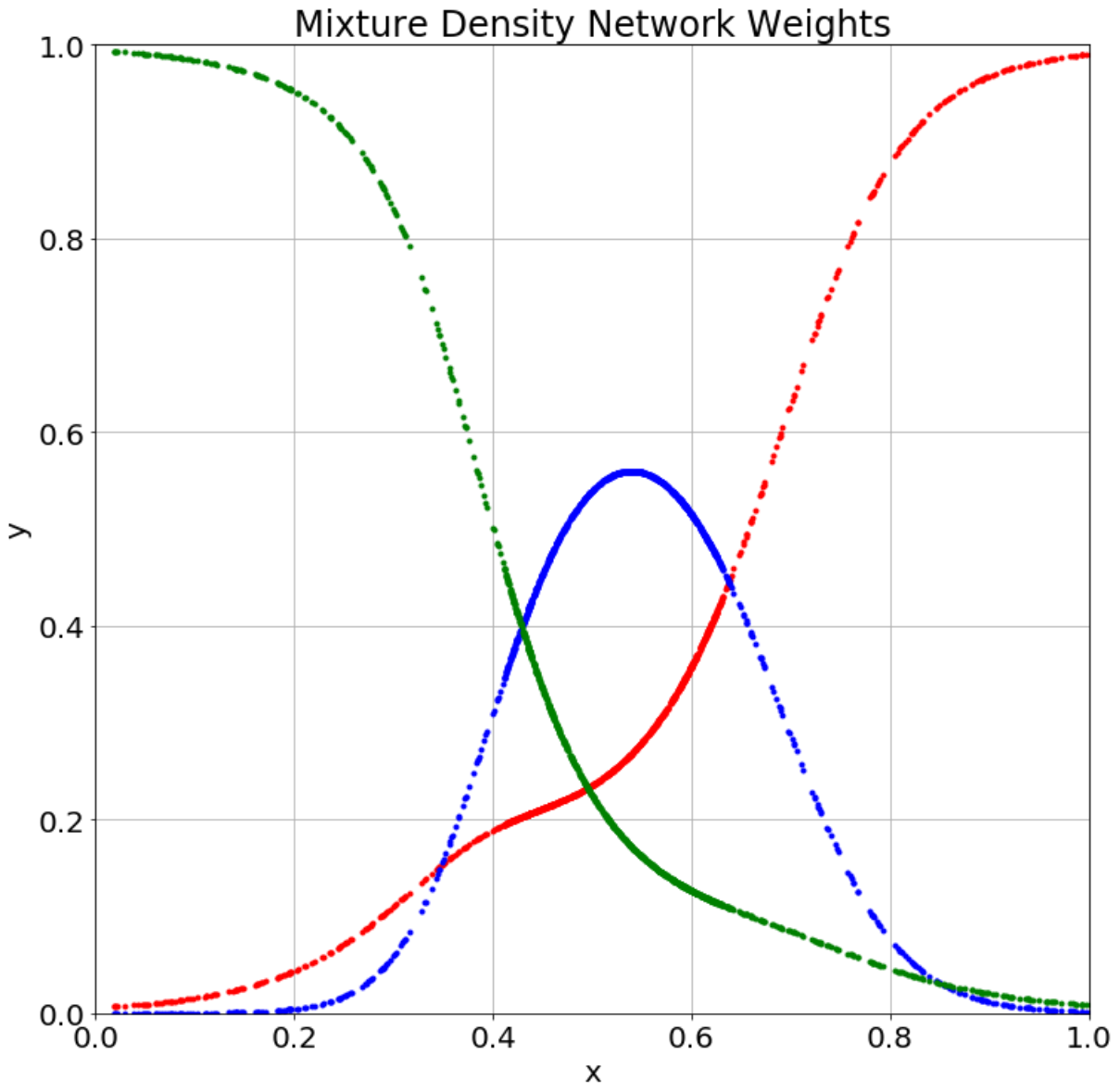

We can see good convergence to a loss around -2.0.

**B3**: Plot the MLE parameters against x. Make a plot similar to A3 above where you overlay the "means" of the gaussians against the data. Plot traces of the mu/sigma/lambda as an aid in debugging.

In [83]:
```
def network_params(network, x: np.ndarray):
    """Run the network on new x values; return weight, mu and sigma for these parameters"""
    # Build a tensor with this data x
    N: int = len(x)
    x_tensor = torch.from_numpy(np.float32(x).reshape((N,1)))
    # Extract tensors for weight, sigma, and mu by running the network on these points (not the training
    weight_tensor, sigma_tensor, mu_tensor = network.forward(x_tensor)
    # Convert these to "plain old data" in numpy arrays for plotting
    weight = weight_tensor.data.numpy()
    sigma = sigma_tensor.data.numpy()
    mu = mu_tensor.data.numpy()
    # Sort these so order is constent with previous treatment and red, blue, and green line up
    idx = np.argsort(np.median(mu, axis=0))[::-1]
    weight = weight[:, idx]
    sigma = sigma[:, idx]
    mu = mu[:, idx]
    # Return weight, sigma, mu for these x values
    return weight, sigma, mu

# Plot the three gaussians in the mixture network model
weight, sigma, mu = network_params(network, x)
fig = plot_gaussians(x, mu, sigma, x, y, 'Mixture Density Network: 3 Gaussians')
```



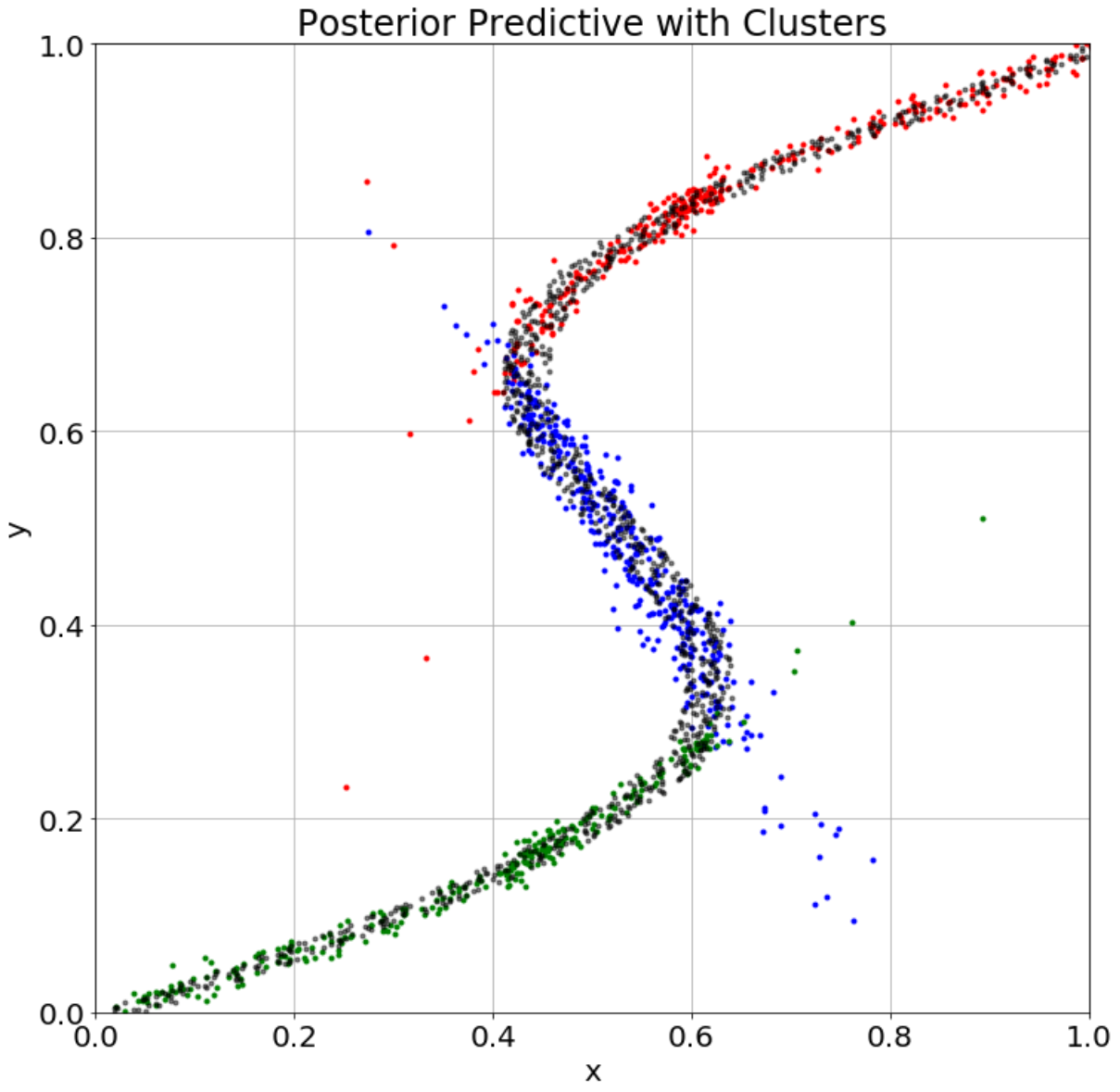
In [84]:
```
# Plot the weights
fig = plot_weights(x, weight, 'Mixture Density Network Weights')
```




Cool! The mixture density network with 20 hidden units does a significantly better job than the simple linear regression model. It's also easier to fit than pymc3 model (IMO).

**B4**: Sample from the sampling distributions at the estimated point values of $\mu$ and $\sigma$ (given cluster) to make a plot similar to A5 above

In [85]:
```
# Sample x with the same data as the original sample
weight, sigma, mu = network_params(network, x)

# Initialize arrays for the clusters and predictions
cluster = np.zeros(N, dtype=np.int8)
x_pred = x
y_pred = np.zeros(N)
# Sample N posterior predictive points
for i in range(N):
    # Sample the cluster based on the probabilities above
    cluster_i = np.random.choice(a=3, p=weight[i])
    # The mean and sigma are in row i and the cluster_i column
    mu_i = mu[i, cluster_i]
    sigma_i = sigma[i, cluster_i]
    # Draw a sample from this normal distribution
    y_pred[i] = np.random.normal(loc=mu_i, scale=sigma_i)
    # Save the cluster that produced this sample
    cluster[i] = cluster_i

# Plot the corrected posterior predictor based on sampled clusters
fig = plot_post_cluster(x_pred, y_pred, cluster, x, y, 'Posterior Predictive with Clusters')
```




Cool! The mixture density network produces pretty nice samples. There are still some stragglers around the two "shoulders" between blue and the other two, but on the whole it's a noticeable improvement from the simple linear model.

# Part C Variational Mixture Density Network

We want to implement the Mixture Density Metwork model that we constructed in Part B directly in pymc3 and use variational inference to sample from it. We may need more iterations in order to get convergence as this model will likely not converge as fast as the pytorch equivalent.

**C1**: Write out the equivalent pymc3 version of the MDN and generate posterior samples with ADVI.

**C2**: Sample from the posterior predictive and produce a diagram like B4 and A5 for this model. Plot traces of the mu/sigma/lambda as an aid in debugging your sampler.

**C3**: Plot the "mean" regression curves (similar to B3 and A3). Do the "mean" regression curves in this model look the same from those in Part B? If they differ why so?

**C1**: Write out the equivalent pymc3 version of the MDN and generate posterior samples with ADVI.

**Get Good Settings for the Neural Net so we have a good starting point for sampling**

In [86]:
```
# current parameter estimates
sd = network.state_dict()
network_wts = dict()
# hidden inputs from data input x
network_wts['z_w'] = sd['z.0.weight'].numpy().squeeze()
network_wts['z_b'] = sd['z.0.bias'].numpy().squeeze()
# mean mu from hidden activation z
network_wts['mu_w'] = sd['mu.weight'].numpy().squeeze()
network_wts['mu_b'] = sd['mu.bias'].numpy().squeeze()
# log_sigma from hidden activation z
network_wts['log_sigma_w'] = sd['log_sigma.weight'].numpy().squeeze()
network_wts['log_sigma_b'] = sd['log_sigma.bias'].numpy().squeeze()
# weight_z from hidden activation z
network_wts['weight_z_w'] = sd['weight_z.weight'].numpy().squeeze()
network_wts['weight_z_b'] = sd['weight_z.bias'].numpy().squeeze()
```

**Warmup: Fit a Trivial Model that Replicated the Neural Net with Minimal Stochasity**
Why do this? Because this model is a LOT easier to debug than the real model.

In [87]:
```
# "warm up": fit a completely trivial deteministic model that replicates the neural network
# and then adds random noise to it.  Why? It's much easier to debug this than the real model!
with pm.Model() as model_det:
    """Deterministic model to test the neural network; for diagnostic purposes only!"""
    # The number of gaussians
    K: int = 3
    # The number of hidden units
    num_hidden: int = 20
    # reshape x to (1,N)
    xr = x.reshape((1,N))

    # reshape input weights to (20, 1)
    w_in_a1 = network_wts['z_w'].reshape((num_hidden,1))
    b_in_a1 = network_wts['z_b'].reshape((num_hidden,1))
    # z1 = wx+b; sizes (20,1)*(1,N) = (20,N)
    z1 = tt.add(pm.math.dot(w_in_a1, xr), b_in_a1)
    a1 = pm.math.tanh(z1)

    # weights for mu have shape (3,20); (3,20) * (20,N) = (3, N)
    w_a1_mu = network_wts['mu_w']
    # bias for mu have shape (3,1)
    b_a1_mu = network_wts['mu_b'].reshape((K,1))
    # mu = w*a1 + b
    mu_val = tt.add(pm.math.dot(w_a1_mu, a1), b_a1_mu)
    # Save mu in the "normal" orientation as a column vector
    mu = pm.Deterministic('mu', mu_val.T)

    # log_sigma analogous to mu
    w_a1_log_sigma = network_wts['log_sigma_w']
    b_a1_log_sigma = network_wts['log_sigma_b'].reshape((K,1))
    log_sigma = tt.add(pm.math.dot(w_a1_log_sigma, a1), b_a1_log_sigma)

    # sigma from log_sigma
    sigma_val = tt.add(pm.math.exp(log_sigma), sigma_shift)
    # Save sigma as a column vector
    sigma = pm.Deterministic('sigma', sigma_val.T)

    # weight_z analogous to mu and log_sigma
    w_a1_weight_z = network_wts['weight_z_w']
    b_a1_weight_z = network_wts['weight_z_b'].reshape((K,1))
    weight_z = tt.add(pm.math.dot(w_a1_weight_z, a1), b_a1_weight_z).T

    # weight from weight_z
    weight_val = softmax(weight_z)
    # Save weight as a column vector
    weight = pm.Deterministic('weight', weight_val)
    # "tune" mu so the model has something to train
    mu_tune = pm.Normal('noise', mu=0.0, sd=1.0, shape=(N,K))

    y_obs = pm.NormalMixture('y_obs', w=weight, mu=tt.add(mu, mu_tune), sd=sigma, observed=y)
```

In [88]:
```
# Number of iterations for test model
num_iters_det = 100000

try:
    advi_det = vartbl['advi_det']
    print(f'Loaded ADVI fit for Deterministic NN Model (for testing / debugging).')
except:
    print(f'Running ADVI fit for Deterministic NN Model...')
    advi_det = pm.ADVI(model=model_det)
    advi_det.fit(n=num_iters_det, obj_optimizer=pm.adam(),
             callbacks=[CheckParametersConvergence()])
    vartbl['advi_det'] = advi_det
    save_vartbl(vartbl, fname)
```

```
Loaded ADVI fit for Deterministic NN Model (for testing / debugging).
```

**Fit the Mixture Density Network for Real with ADVI**

**First Attempt: Use Local Environment Around ML Estimate from Neural Network a.k.a. "Remember Your Failure at the Cave"**

In [89]:
```
# after the warm-up, ready for the main event...
with pm.Model() as model_mdn:
    """Mixture Density Network model"""
    # The number of data points
    N: int = len(x)
    # The number of gaussians
    K: int = 3
    # The number of hidden units
    num_hidden: int = 20
    # reshape x to (1,N)
    xr = x.reshape((1,N))

    # Priors for standard deviations of weights and biases in the network
    sd_w: float = 2.0E-3
    sd_b: float = 2.0E-3

    # Input weights; shape (20, 1)
    # w_in_a1 = network_wts['z_w'].reshape((num_hidden,1))
    w_in_a1_mu = network_wts['z_w'].reshape((num_hidden,1))
    w_in_a1 = pm.Normal('w_in_a1', mu=w_in_a1_mu, sd=sd_w, shape=w_in_a1_mu.shape)

    # Input bias; shape (20, 1)
    # b_in_a1 = network_wts['z_b'].reshape((num_hidden,1))
    b_in_a1_mu = network_wts['z_b'].reshape((num_hidden,1))
    b_in_a1 = pm.Normal('b_in_a1', mu=b_in_a1_mu, sd=sd_b, shape=b_in_a1_mu.shape)

    # z1 = wx+b; sizes (20,1)*(1,N) = (20,N)
    z1 = tt.add(pm.math.dot(w_in_a1, xr), b_in_a1)
    a1 = pm.math.tanh(z1)

    # weights for mu have shape (3,20); (3,20) * (20,N) = (3, N)
    # w_a1_mu = network_wts['mu_w']
    w_a1_mu_mu = network_wts['mu_w']
    w_a1_mu = pm.Normal('w_a1_mu', mu=w_a1_mu_mu, sd=sd_w, shape=w_a1_mu_mu.shape)

    # bias for mu have shape (3,1)
    # b_a1_mu = network_wts['mu_b'].reshape((K,1))
    b_a1_mu_mu = network_wts['mu_b'].reshape((K,1))
    b_a1_mu = pm.Normal('b_a1_mu', mu=b_a1_mu_mu, sd=sd_b, shape=b_a1_mu_mu.shape)

    # mu = w*a1 + b
    mu_val = tt.add(pm.math.dot(w_a1_mu, a1), b_a1_mu)
    # tune mu; helps training by absorbing the noise that was added to the data
    # mu_tune = pm.Normal('mu_tune', mu=0.0, sd=0.02, shape=(K,N))
    # Save mu in the "normal" orientation as a column vector
    # mu = pm.Deterministic('mu', tt.add(mu_val.T, mu_tune.T))
    mu = pm.Deterministic('mu', mu_val.T)

    # weight for log_sigma
    # w_a1_log_sigma = network_wts['log_sigma_w']
    w_a1_log_sigma_mu = network_wts['log_sigma_w']
    w_a1_log_sigma = pm.Normal('w_a1_log_sigma', mu=w_a1_log_sigma_mu, sd=sd_w, shape=w_a1_log_sigma_mu.s

    # bias for log_sigma
    # b_a1_log_sigma = network_wts['log_sigma_b'].reshape((K,1))
    b_a1_log_sigma_mu = network_wts['log_sigma_b'].reshape((K,1))
    b_a1_log_sigma = pm.Normal('b_a1_log_sigma', mu=b_a1_log_sigma_mu, sd=sd_b, shape=b_a1_log_sigma_mu.s

    # log_sigma = w*a1 + b
    log_sigma = tt.add(pm.math.dot(w_a1_log_sigma, a1), b_a1_log_sigma)

    # sigma from log_sigma
    sigma_val = tt.add(pm.math.exp(log_sigma), sigma_shift)
    # save sigma as a column vector
    sigma = pm.Deterministic('sigma', sigma_val.T)

    # weight_z is the intermediate results that is fed into the softmax to get the cluster weights
    # weight for weight_z
    # w_a1_weight_z = network_wts['weight_z_w']
    w_a1_weight_z_mu = network_wts['weight_z_w']
    w_a1_weight_z = pm.Normal('w_a1_weight_z', mu=w_a1_weight_z_mu, sd=sd_w, shape=w_a1_weight_z_mu.shape
```

```
    # bias for weight_z
    # b_a1_weight_z = network_wts['weight_z_b'].reshape((K,1))
    b_a1_weight_z_mu = network_wts['weight_z_b'].reshape((K,1))
    b_a1_weight_z = pm.Normal('b_a1_weight_z', mu=b_a1_weight_z_mu, sd=sd_b, shape=b_a1_weight_z_mu.shape

    # mixing weight = w*a1 + b
    weight_z = tt.add(pm.math.dot(w_a1_weight_z, a1), b_a1_weight_z).T

    # weight from weight_z
    weight_val = softmax(weight_z)
    # Save weight as a column vector
    weight = pm.Deterministic('weight', weight_val)
    # Sample from a normal mixture model and compare to observed data points
    y_obs = pm.NormalMixture('y_obs', w=weight, mu=mu, sd=sigma, observed=y)
```

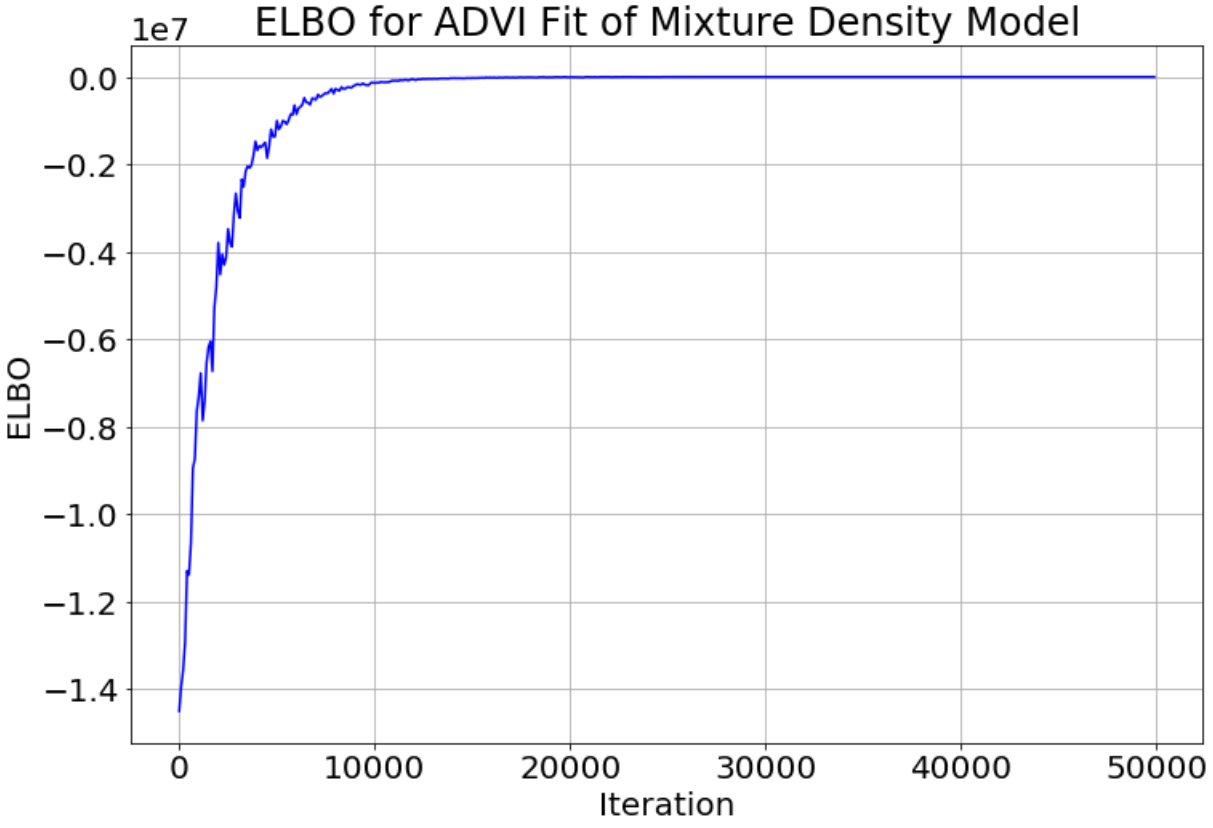
**Explanation of Approach**

I spent a really long time testing this to try to get it to a point where it produced good samples. When I ran it with "neutral" parameters it repeatedly produced garabage output. A common pattern was that it would assign all the weight to the middle segment (green) and produce a very diffuse cloud of points with almost no resemblance to the data.

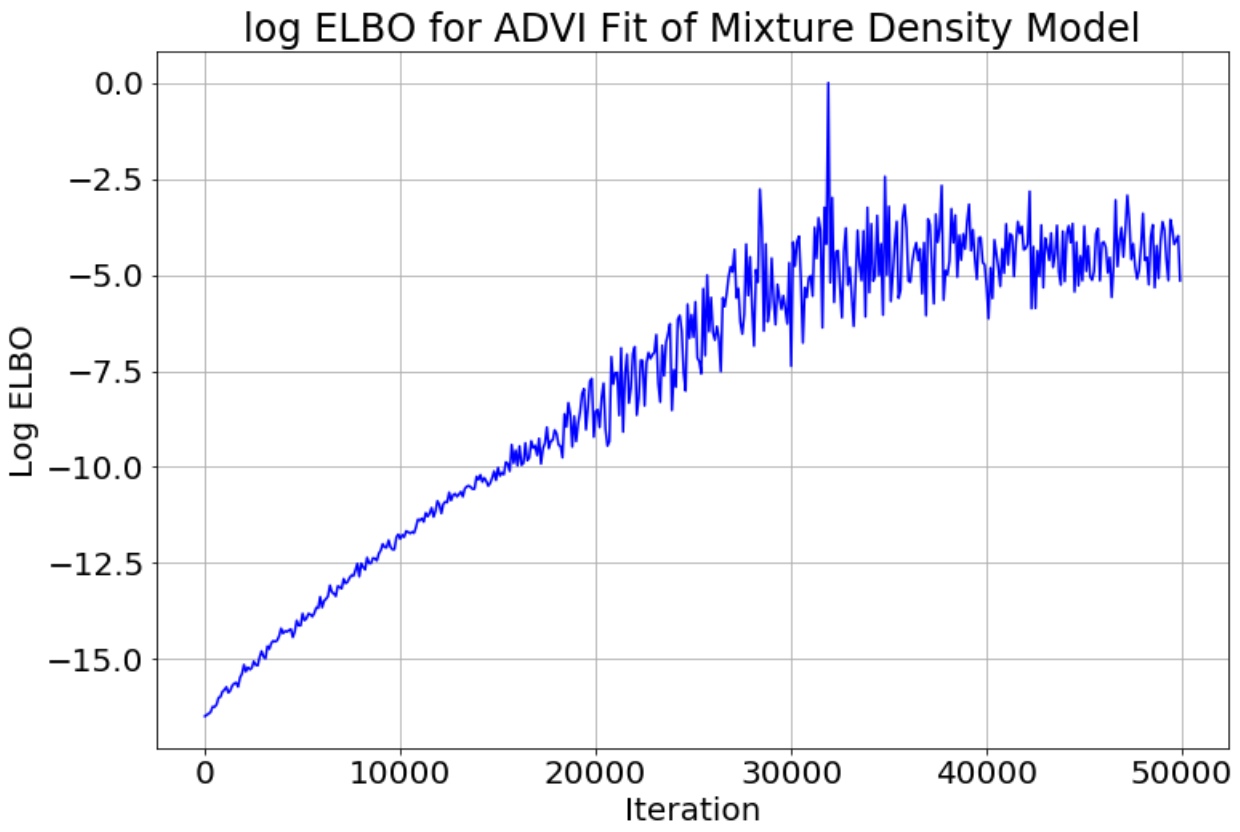
After much painful experimentation, I was able to get decent samples by choosing means for the network parameters equal to the ML estimate found in training, and tuning a reasonable SD around that. In the end, I found that a SD of 2E-3 around the network parameters allowed it to move around the space somewhat while still producing good quality samples, which I will display below.

In [90]:
```
try:
    advi_mdn = vartbl['advi_mdn']
    print(f'Loaded ADVI fit for Mixture Density Model.')
except:
    print(f'Running ADVI fit for Mixture Density Model...')
    advi_mdn = pm.ADVI(model=model_mdn)
    advi_mdn.fit(n=num_iters_mdn, obj_optimizer=pm.adam(),
             callbacks=[CheckParametersConvergence()])
    vartbl['advi_mdn'] = advi_mdn
    save_vartbl(vartbl, fname)

# Plot the ELBO
fig = plot_elbo(-advi_mdn.hist, 100, 'ELBO for ADVI Fit of Mixture Density Model')
```

```
Loaded ADVI fit for Mixture Density Model.
```



In [91]:
```
# Plot the ELBO on a log scale so it's easier to visualize
fig = plot_elbo_log(-advi_mdn.hist, 100, 'log ELBO for ADVI Fit of Mixture Density Model')
```




We can see that the training here plateaud around 30,000 iterations.

**Utility Functions for Sampling and Visualizing**

These functions will be used on the real Mixture Density Network (MDN) model, but they work equally well on the "Deterministic" toy model above. In the development process, they were used to visualize the output of the Deterministic model when debugging the MDN model.

In [92]:
```
def draw_samples(weight, mu, sigma):
    """Draw samples with an ancestral sampling approach"""
    # The number of samples
    num_samples: int = len(weight)
    # Initialize arrays with the cluster and predicted points
    cluster = np.zeros(num_samples, dtype=np.int8)
    x_pp = np.zeros(num_samples)
    y_pp = np.zeros(num_samples)
    # Arrays for the sampled parameters
    weight_pp = np.zeros((num_samples, K))
    mu_pp = np.zeros((num_samples, K))
    sigma_pp = np.zeros((num_samples, K))
    # Iterate over samples
    for i in range(num_samples):
        # Draw a random row of the sample data
        row_num = np.random.choice(num_samples)
        # Cycle throught the x's in order
        col_num = i % N
        # Sample the cluster assignments according to the weights here
        cluster_i = np.random.choice(a=K, p=weight[row_num, col_num])
        # Draw a sample from this normal
        x_pp[i] = x[col_num]
        y_pp[i] = np.random.normal(loc = mu[row_num, col_num, cluster_i],
                                   scale = sigma[row_num, col_num, cluster_i])
        # Save the cluster assignment
        cluster[i] = cluster_i
        # Save diagnostic
        weight_pp[i] = weight[row_num, col_num]
        mu_pp[i] = mu[row_num, col_num]
        sigma_pp[i] = sigma[row_num, col_num]
    return x_pp, y_pp, cluster, weight_pp, mu_pp, sigma_pp


def plot_post_mu(x_pp, mu, cluster, x, y, title):
    """Plot the mean of each cluster at the sample points"""
    fig, ax = plt.subplots(figsize=[12,12])
    ax.set_title(title)
    ax.set_xlim(0,1)
    ax.set_ylim(0,1)
    ax.grid()
    # Plot the mean of each gaussian
    ax.plot(x_pp, mu[:,0], color='r', linewidth=0, marker='o')
    ax.plot(x_pp, mu[:,1], color='b', linewidth=0, marker='o')
    ax.plot(x_pp, mu[:,2], color='g', linewidth=0, marker='o')
    # Plot the original data too
    ax.plot(x, y, color='k', linewidth=0, marker='o', markersize=3, alpha=0.5)
```

This function was used in devlopment to diagnose why a trained model was failling. By comparing the profile of mu, sigma, the class weights, and the posterior points drawn, it was easier to figure out what was going wrong in the sampling.

In [93]:
```
def debug_posteriors(trace_det, trace_mdn):
    """Generate series of charts to debug posterior """

    # Extract mu, sigma, and weight for posterior sampling with clusters
    mu_det = trace_det['mu']
    sigma_det = trace_det['sigma']
    weight_det = trace_det['weight']

    # Draw samples for the deterministic model
    x_pp_det, y_pp_det, cluster_det, weight_pp_det, mu_pp_det, sigma_pp_det = \
        draw_samples(weight_det, mu_det, sigma_det)

    # Extract mu, sigma, and weight for posterior sampling with clusters
    weight_mdn = trace_mdn['weight']
    mu_mdn = trace_mdn['mu']
    sigma_mdn = trace_mdn['sigma']

    # Draw samples for the Mixture Density Network model
    x_pp_mdn, y_pp_mdn, cluster_mdn, weight_pp_mdn, mu_pp_mdn, sigma_pp_mdn = \
        draw_samples(weight_mdn, mu_mdn, sigma_mdn)

    mean_weight_det = np.mean(weight_det, axis=0)
    mean_weight_mdn = np.mean(weight_mdn, axis=0)

    plot_weights(x, mean_weight_det, 'Mean Weights: DET')
    plot_weights(x, mean_weight_mdn, 'Mean Weights: MDN')

    # Check cluster means
    mean_mu_det = np.mean(mu_det, axis=0)
    mean_mu_mdn = np.mean(mu_mdn, axis=0)

    plot_post_mu(x, mean_mu_det, cluster_det, x, y, 'Mean mu: DET')
    plot_post_mu(x, mean_mu_mdn, cluster_mdn, x, y, 'Mean mu: MDN')

    # Check sigmas means
    sigma_mu_det = np.mean(sigma_det, axis=0)
    sigma_mu_mdn = np.mean(sigma_mdn, axis=0)

    plot_weights(x, sigma_mu_det, 'Mean sigma: DET')
    plot_weights(x, sigma_mu_mdn, 'Mean sigma: MDN')

    # The posterior predictive with clusters
    plot_post_cluster(x_pp_det, y_pp_det, cluster_det, x, y, 'Posterior Predictive: Deterministic Model')
    plot_post_cluster(x_pp_mdn, y_pp_mdn, cluster_mdn, x, y, 'Posterior Predictive: Mixture Density Model
```

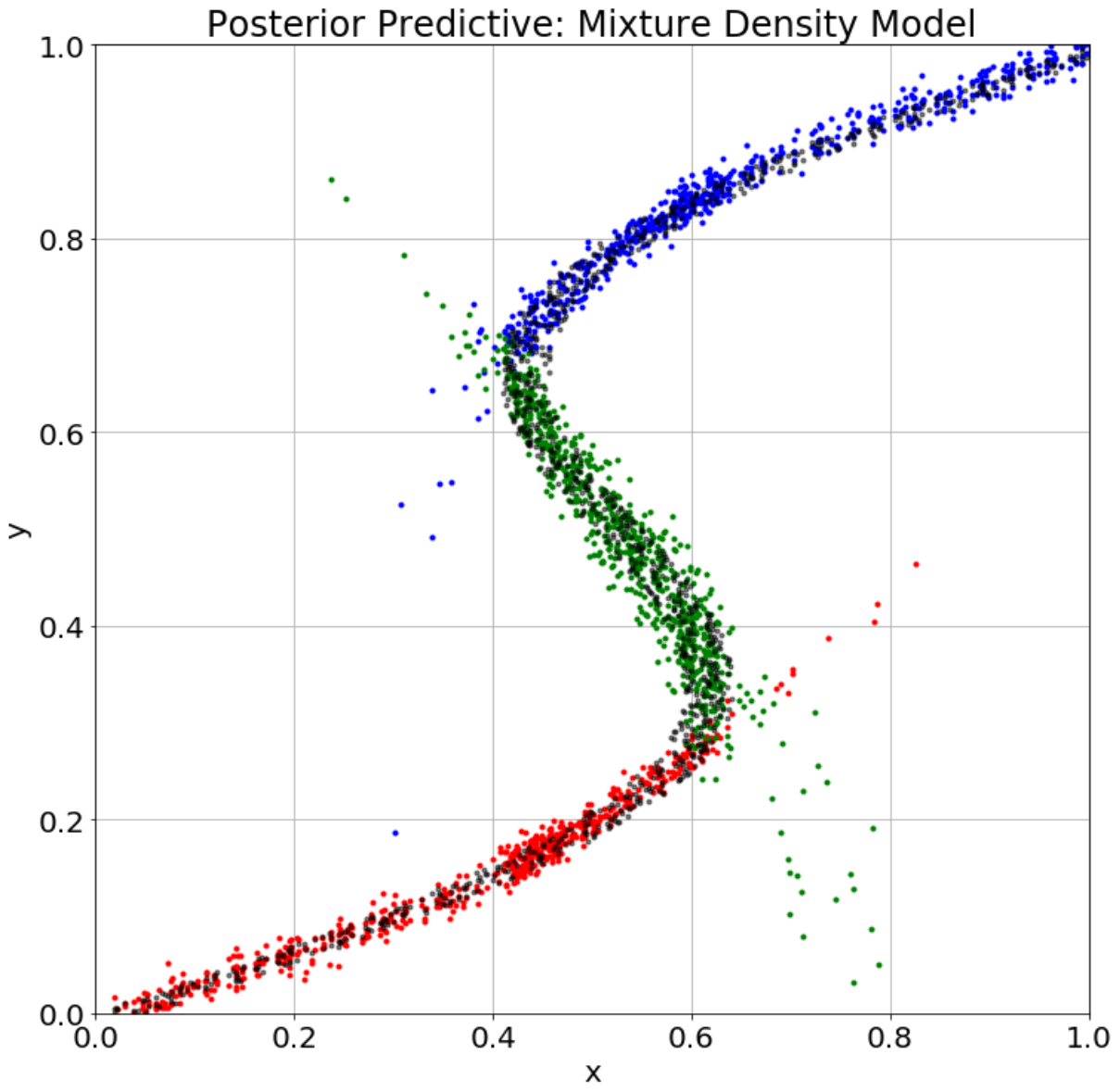
**C2: Sample from the posterior predictive and produce a diagram like B4 and A5 for this model**

In [260]:
```
# Draw parameter samples (trace)
try:
    trace_mdn = vartbl['trace_mdn']
    print(f'Loaded trace from ADVI fit of Mixture Density Model.')
except:
    print(f'Drawing posterior samples (parameters) from ADVI fit of Mixture Density Model...')
    trace_mdn = advi_mdn.approx.sample(num_samples)
    vartbl['trace_mdn'] = trace_mdn
    save_vartbl(vartbl, fname)

# Extract mu, sigma, and weight for posterior sampling with clusters
weight_mdn = trace_mdn['weight']
mu_mdn = trace_mdn['mu']
sigma_mdn = trace_mdn['sigma']

# Draw samples for the Mixture Density Network model
x_pp_mdn, y_pp_mdn, cluster_mdn, weight_pp_mdn, mu_pp_mdn, sigma_pp_mdn = \
    draw_samples(weight_mdn, mu_mdn, sigma_mdn)
```

```
Loaded trace from ADVI fit of Mixture Density Model.
```

**Plot the Posterior Sample with Cluster Assignments**

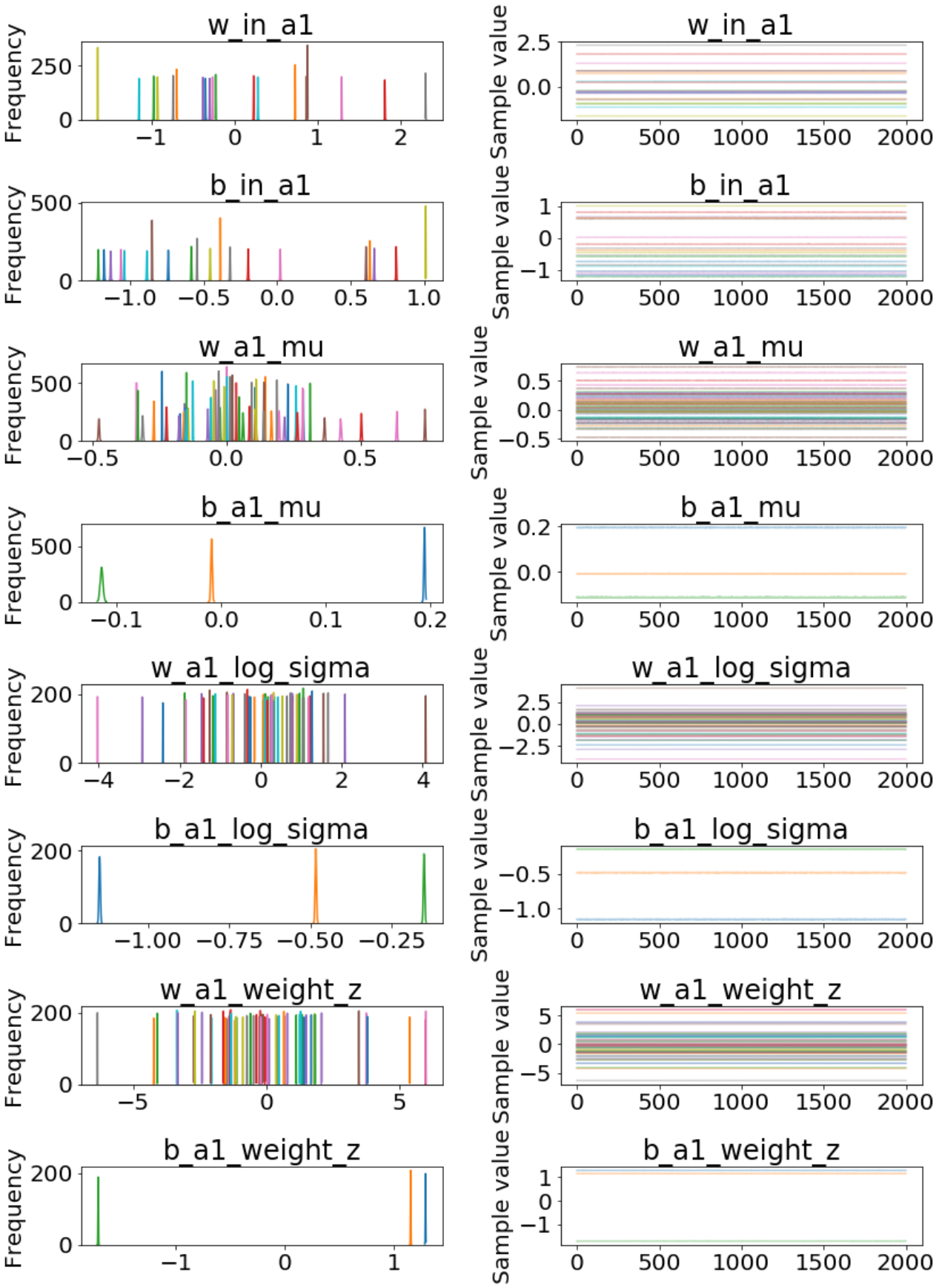
In [261]:
```
# Plot the posterior sample
fig = plot_post_cluster(x_pp_mdn, y_pp_mdn, cluster_mdn, x, y, 'Posterior Predictive: Mixture Density Mo
```




Cool! We have good quality samples with clearly identified clusters. Now they're being drawn from a probabilistic model tuned with ADVI. Is it really better than the neural network it was based on?

**Plot traces of the mu/sigma/lambda as an aid in debugging your sampler**

In [266]:
```
varnames_mdn =['w_in_a1', 'b_in_a1', 'w_a1_mu', 'b_a1_mu',
               'w_a1_log_sigma', 'b_a1_log_sigma', 'w_a1_weight_z', 'b_a1_weight_z']
figs = pm.traceplot(trace_mdn, varnames=varnames_mdn)
```




**Why was this a failure?** After viewing the traceplot I realized this was unfortunately a failure. In what sense? The posterior isn't

really exploring much of the space. It's just regurgitating back

**Second Attempt: Fit MDN Model "For Real" with ADVI with Uninformative Priors**
**a.k.a. Confrontation with Vader at Cloud City** The version presented is identical to the one above, except it uses neutral (uninformative priors). All weights and biases in the model are initialized with a standard normal distribution. I had to train this overnight from 2:45 AM to 7:01 AM...

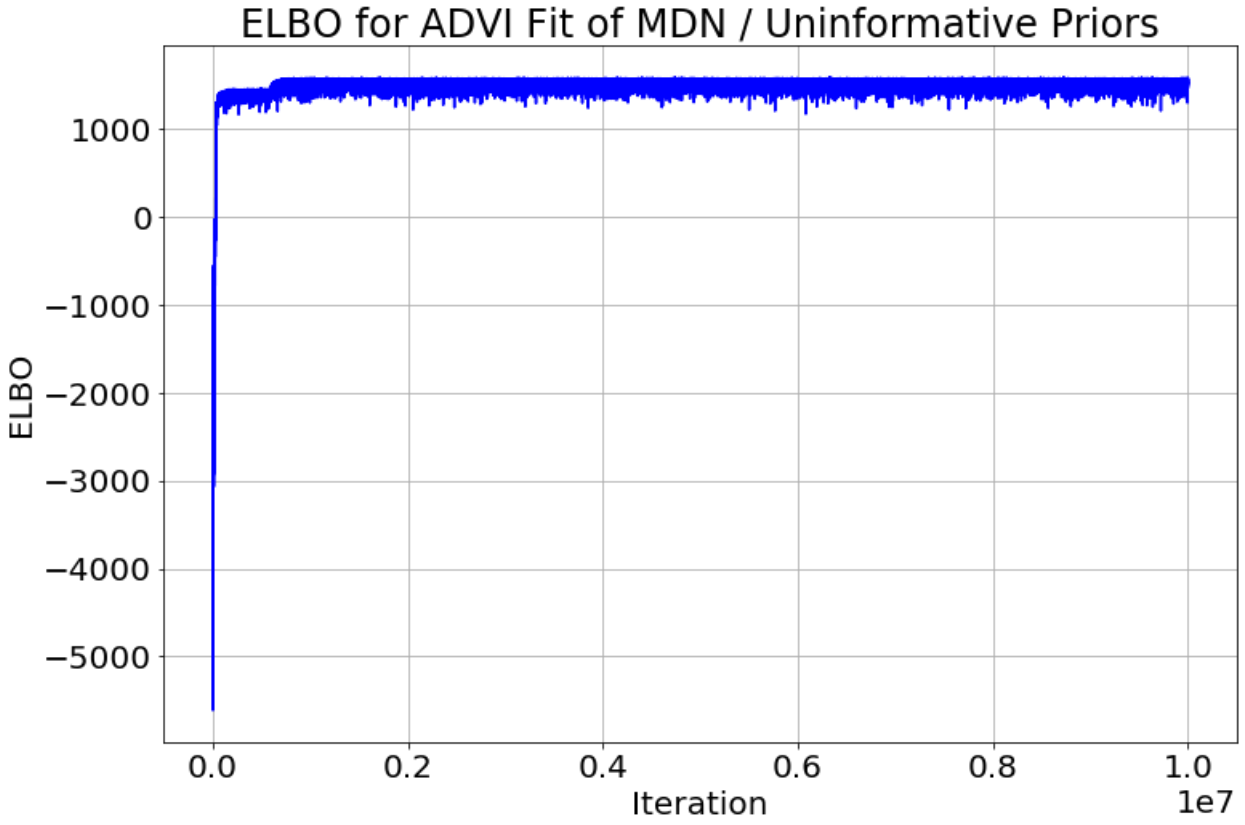
In [94]:
```
with pm.Model() as model_mdn2:
    """Mixture Density Network model"""
    # The number of data points
    N: int = len(x)
    # The number of gaussians
    K: int = 3
    # The number of hidden units
    num_hidden: int = 20
    # reshape x to (1,N)
    xr = x.reshape((1,N))

    # Priors for standard deviations of weights and biases in the network
    sd_w: float = 1.0
    sd_b: float = 1.0

    # Input weights; shape (20, 1)
    # w_in_a1 = network_wts['z_w'].reshape((num_hidden,1))
    w_in_a1_mu = network_wts['z_w'].reshape((num_hidden,1))*0.0
    w_in_a1 = pm.Normal('w_in_a1', mu=w_in_a1_mu, sd=sd_w, shape=w_in_a1_mu.shape)

    # Input bias; shape (20, 1)
    # b_in_a1 = network_wts['z_b'].reshape((num_hidden,1))
    b_in_a1_mu = network_wts['z_b'].reshape((num_hidden,1))*0.0
    b_in_a1 = pm.Normal('b_in_a1', mu=b_in_a1_mu, sd=sd_b, shape=b_in_a1_mu.shape)

    # z1 = wx+b; sizes (20,1)*(1,N) = (20,N)
    z1 = tt.add(pm.math.dot(w_in_a1, xr), b_in_a1)
    a1 = pm.math.tanh(z1)

    # weights for mu have shape (3,20); (3,20) * (20,N) = (3, N)
    # w_a1_mu = network_wts['mu_w']
    w_a1_mu_mu = network_wts['mu_w']*0.0
    w_a1_mu = pm.Normal('w_a1_mu', mu=w_a1_mu_mu, sd=sd_w, shape=w_a1_mu_mu.shape)

    # bias for mu have shape (3,1)
    # b_a1_mu = network_wts['mu_b'].reshape((K,1))
    b_a1_mu_mu = network_wts['mu_b'].reshape((K,1))*0.0
    b_a1_mu = pm.Normal('b_a1_mu', mu=b_a1_mu_mu, sd=sd_b, shape=b_a1_mu_mu.shape)

    # mu = w*a1 + b
    mu_val = tt.add(pm.math.dot(w_a1_mu, a1), b_a1_mu)
    # tune mu; helps training by absorbing the noise that was added to the data
    # mu_tune = pm.Normal('mu_tune', mu=0.0, sd=0.02, shape=(K,N))
    # Save mu in the "normal" orientation as a column vector
    # mu = pm.Deterministic('mu', tt.add(mu_val.T, mu_tune.T))
    mu = pm.Deterministic('mu', mu_val.T)

    # weight for log_sigma
    # w_a1_log_sigma = network_wts['log_sigma_w']
    w_a1_log_sigma_mu = network_wts['log_sigma_w']*0.0
    w_a1_log_sigma = pm.Normal('w_a1_log_sigma', mu=w_a1_log_sigma_mu, sd=sd_w, shape=w_a1_log_sigma_mu.s

    # bias for log_sigma
    # b_a1_log_sigma = network_wts['log_sigma_b'].reshape((K,1))
    b_a1_log_sigma_mu = network_wts['log_sigma_b'].reshape((K,1))*0.0
    b_a1_log_sigma = pm.Normal('b_a1_log_sigma', mu=b_a1_log_sigma_mu, sd=sd_b, shape=b_a1_log_sigma_mu.s

    # log_sigma = w*a1 + b
    log_sigma = tt.add(pm.math.dot(w_a1_log_sigma, a1), b_a1_log_sigma)

    # sigma from log_sigma
    sigma_val = tt.add(pm.math.exp(log_sigma), sigma_shift)
    # save sigma as a column vector
    sigma = pm.Deterministic('sigma', sigma_val.T)

    # weight_z is the intermediate results that is fed into the softmax to get the cluster weights
    # weight for weight_z
    # w_a1_weight_z = network_wts['weight_z_w']
    w_a1_weight_z_mu = network_wts['weight_z_w']*0.0
    w_a1_weight_z = pm.Normal('w_a1_weight_z', mu=w_a1_weight_z_mu, sd=sd_w, shape=w_a1_weight_z_mu.shape
```

```
    # bias for weight_z
    # b_a1_weight_z = network_wts['weight_z_b'].reshape((K,1))
    b_a1_weight_z_mu = network_wts['weight_z_b'].reshape((K,1))*0.0
    b_a1_weight_z = pm.Normal('b_a1_weight_z', mu=b_a1_weight_z_mu, sd=sd_b, shape=b_a1_weight_z_mu.shape

    # mixing weight = w*a1 + b
    weight_z = tt.add(pm.math.dot(w_a1_weight_z, a1), b_a1_weight_z).T

    # weight from weight_z
    weight_val = softmax(weight_z)
    # Save weight as a column vector
    weight = pm.Deterministic('weight', weight_val)
    # Sample from a normal mixture model and compare to observed data points
    y_obs = pm.NormalMixture('y_obs', w=weight, mu=mu, sd=sigma, observed=y)
```

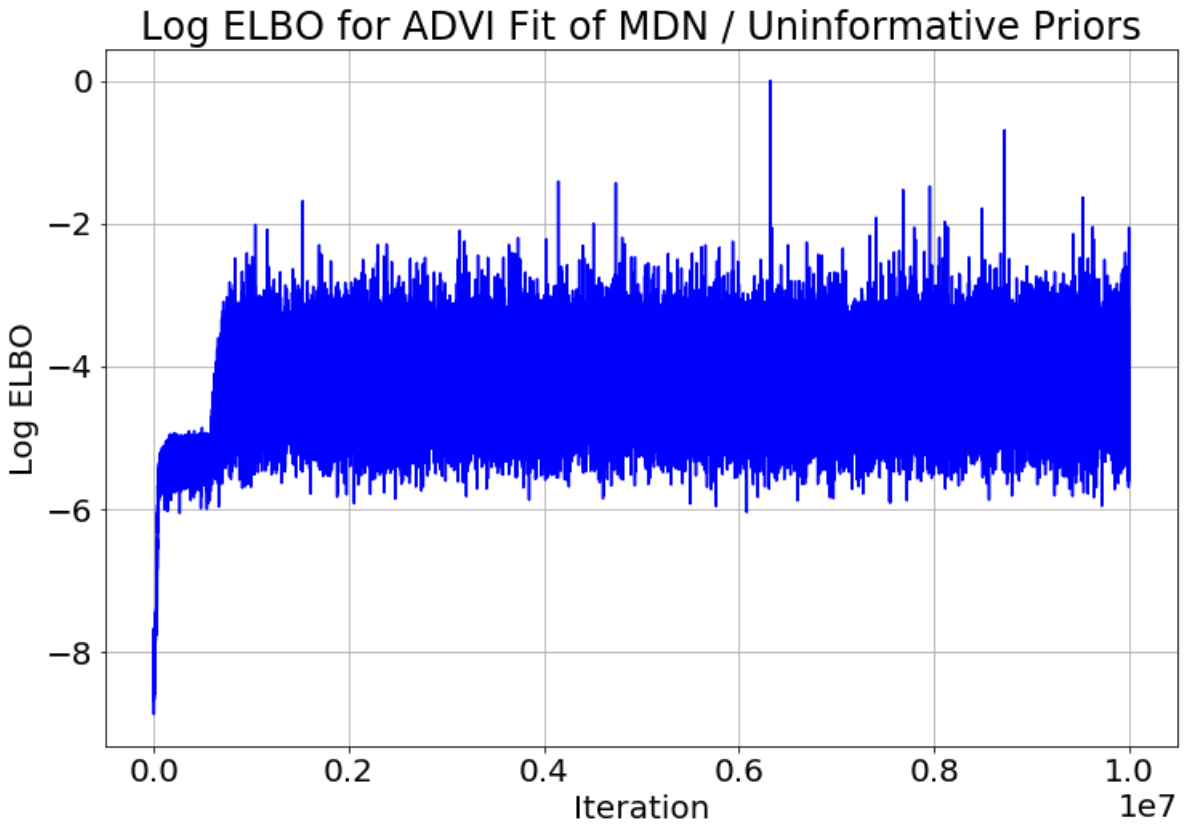
In [97]:
```
# This model needs a *lot* of iterations to train because of its uninformative priors!
num_iters_mdn2 = 10**7

try:
    advi_mdn2 = vartbl['advi_mdn2']
    print(f'Loaded ADVI fit for Mixture Density Model with uninformative priors.')
except:
    print(f'Running ADVI fit for Mixture Density Model with uninformative priors...')
    print(f'Warning: This is going to take a long time...')
    advi_mdn = pm.ADVI(model=model_mdn)
    advi_mdn.fit(n=num_iters_mdn, obj_optimizer=pm.adam(),
             callbacks=[CheckParametersConvergence()])
    vartbl['advi_mdn'] = advi_mdn
    save_vartbl(vartbl, fname)
```

```
Loaded ADVI fit for Mixture Density Model with uninformative priors.
```

In [102]:
```
# Plot the ELBO
fig = plot_elbo(-advi_mdn2.hist, 100, 'ELBO for ADVI Fit of MDN / Uninformative Priors')
```



In [103]:
```
# Plot the log ELBO
fig = plot_elbo_log(-advi_mdn2.hist, 100, 'Log ELBO for ADVI Fit of MDN / Uninformative Priors')
```




**C2: Sample from the posterior predictive and produce a diagram like B4 and A5 for this model**

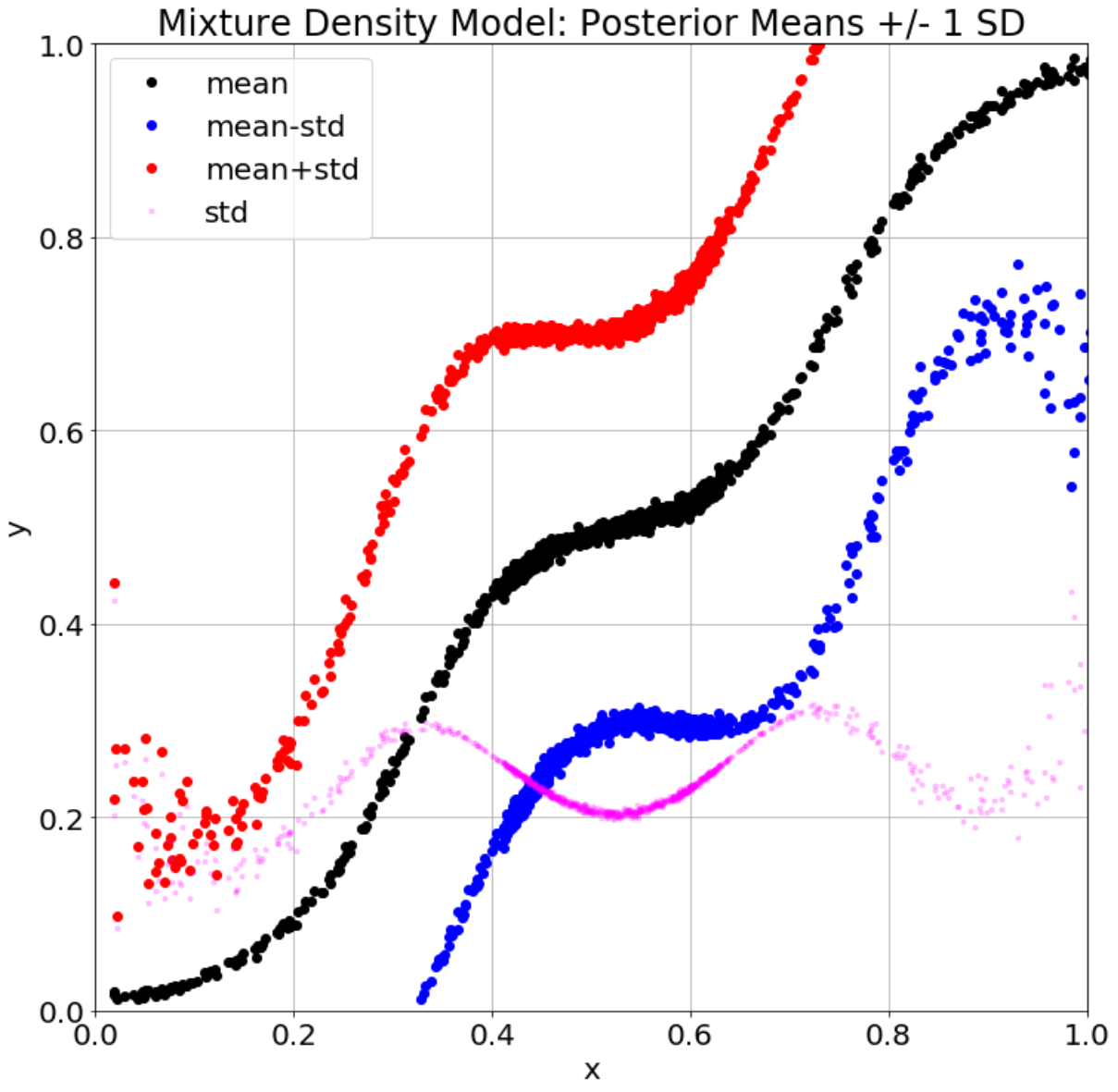
In [107]:
```
# Draw parameter samples (trace)
try:
    trace_mdn2 = vartbl['trace_mdn2']
    print(f'Loaded trace from ADVI fit of Mixture Density Model with uninformative priors.')
except:
    print(f'Drawing posterior samples (parameters) from ADVI fit of MDN with uninformative priors...')
    trace_mdn2 = advi_mdn2.approx.sample(num_samples)
    vartbl['trace_mdn2'] = trace_mdn2
    save_vartbl(vartbl, fname)

# Extract mu, sigma, and weight for posterior sampling with clusters
weight_mdn2 = trace_mdn2['weight']
mu_mdn2 = trace_mdn2['mu']
sigma_mdn2 = trace_mdn2['sigma']

# Draw samples for the Mixture Density Network model
x_pp_mdn2, y_pp_mdn2, cluster_mdn2, weight_pp_mdn2, mu_pp_mdn2, sigma_pp_mdn2 = \
    draw_samples(weight_mdn2, mu_mdn2, sigma_mdn2)
```

```
Loaded trace from ADVI fit of Mixture Density Model with uninformative priors.
```

**C3**: Plot the "mean" regression curves (similar to B3 and A3). Do the "mean" regression curves in this model look the same from those in Part B? If they differ why so?

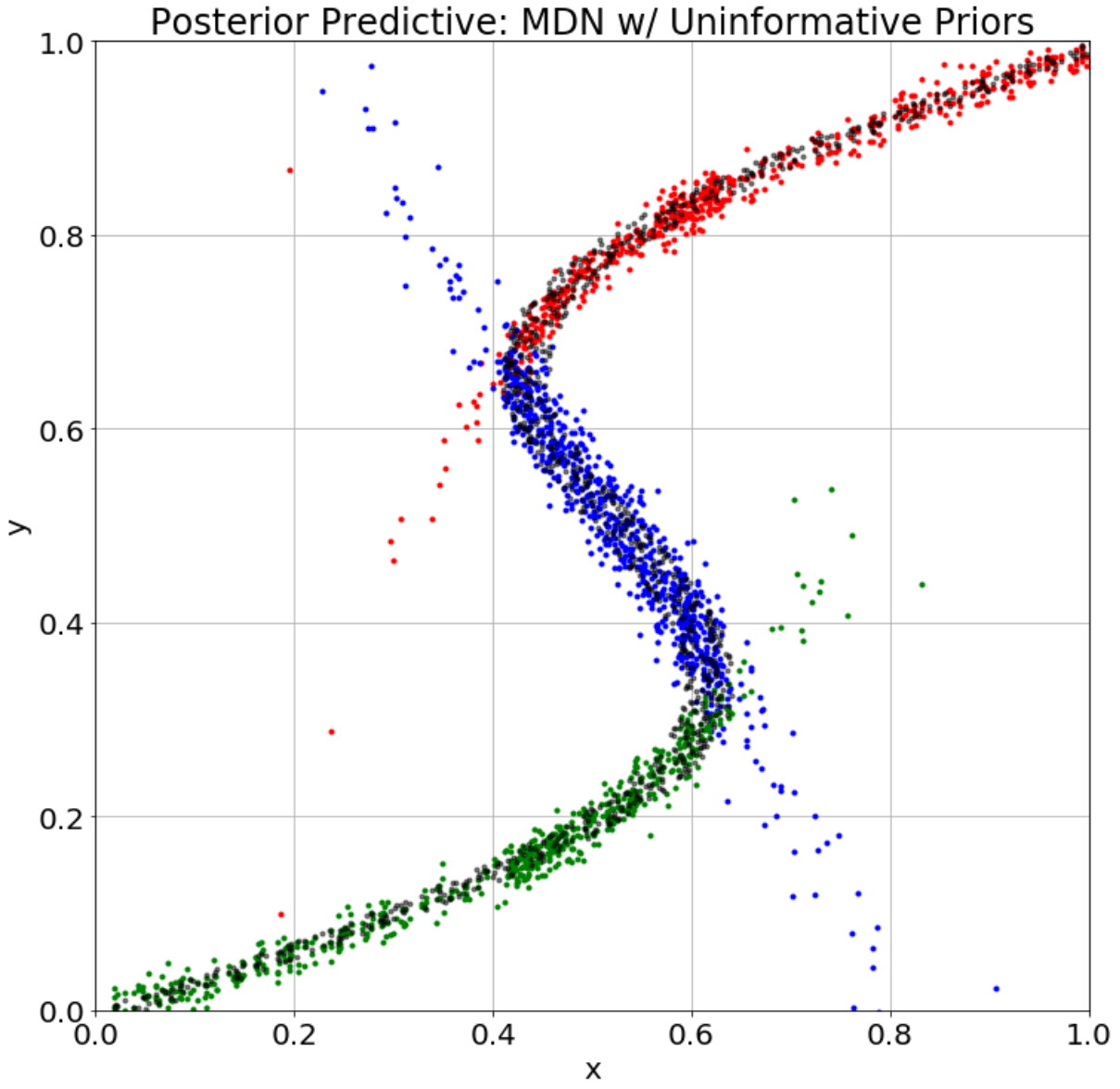
In [109]:
```
try:
    pred_mdn2 = vartbl['pred_mdn2']
    print(f'Loaded posterior predictive from ADVI fit of Mixture Density Model with uninformative priors
except:
    print(f'Drawing posterior predictive from ADVI fit of Mixture Density Model with uninformative priors
    pred_mdn2 = pm.sample_ppc(trace_mdn2, model=model_mdn2)
    vartbl['pred_mdn2'] = pred_mdn2
    save_vartbl(vartbl, fname)

# Extract y_pred as an array; shape (num_samples, N)
y_pred = pred_mdn2['y_obs']
# Mean and standard deviation of y
y_mean = np.mean(y_pred, axis=0)
y_std = np.std(y_pred, axis=0)

# Plot the posterior mean with +/- 1 SD envolope
fig = plot_post_mean_std(x, y_mean, y_std, 'Mixture Density Model: Posterior Means +/- 1 SD')
```

```
Loaded posterior predictive from ADVI fit of Mixture Density Model with uninformative priors.
```



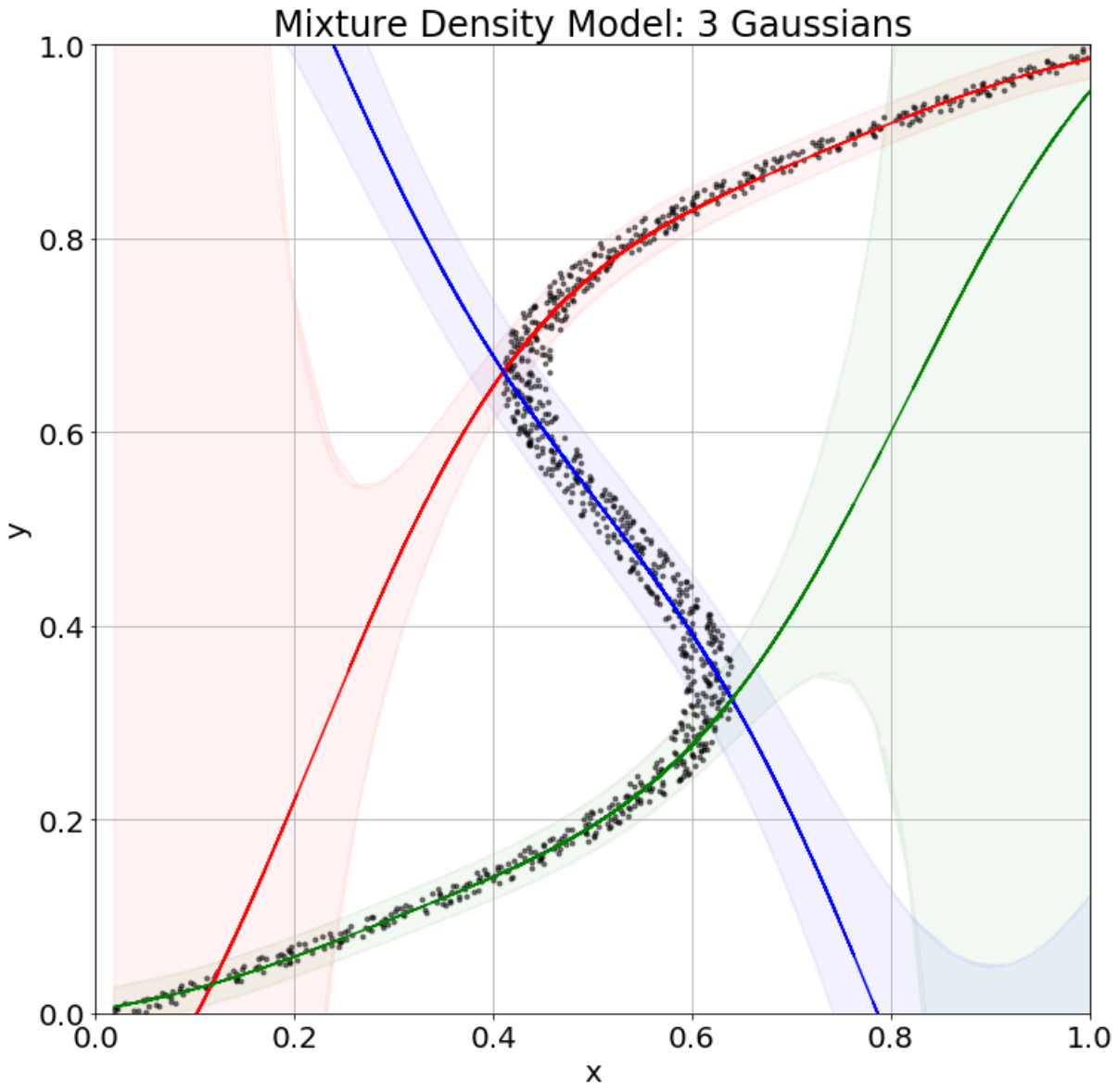

**Plot the Posterior Sample with Cluster Assignments**

In [110]:
```
# Plot the posterior sample
fig = plot_post_cluster(x_pp_mdn2, y_pp_mdn2, cluster_mdn2, x, y, 'Posterior Predictive: MDN w/ Uninforma
```




Much cooler! We managed to a good quality fit again, using uninformative priors. Admittedly this fit is actually not as good the ML estimate obtained by the neural network. But it was legitimately obtained by a Bayesian approach.

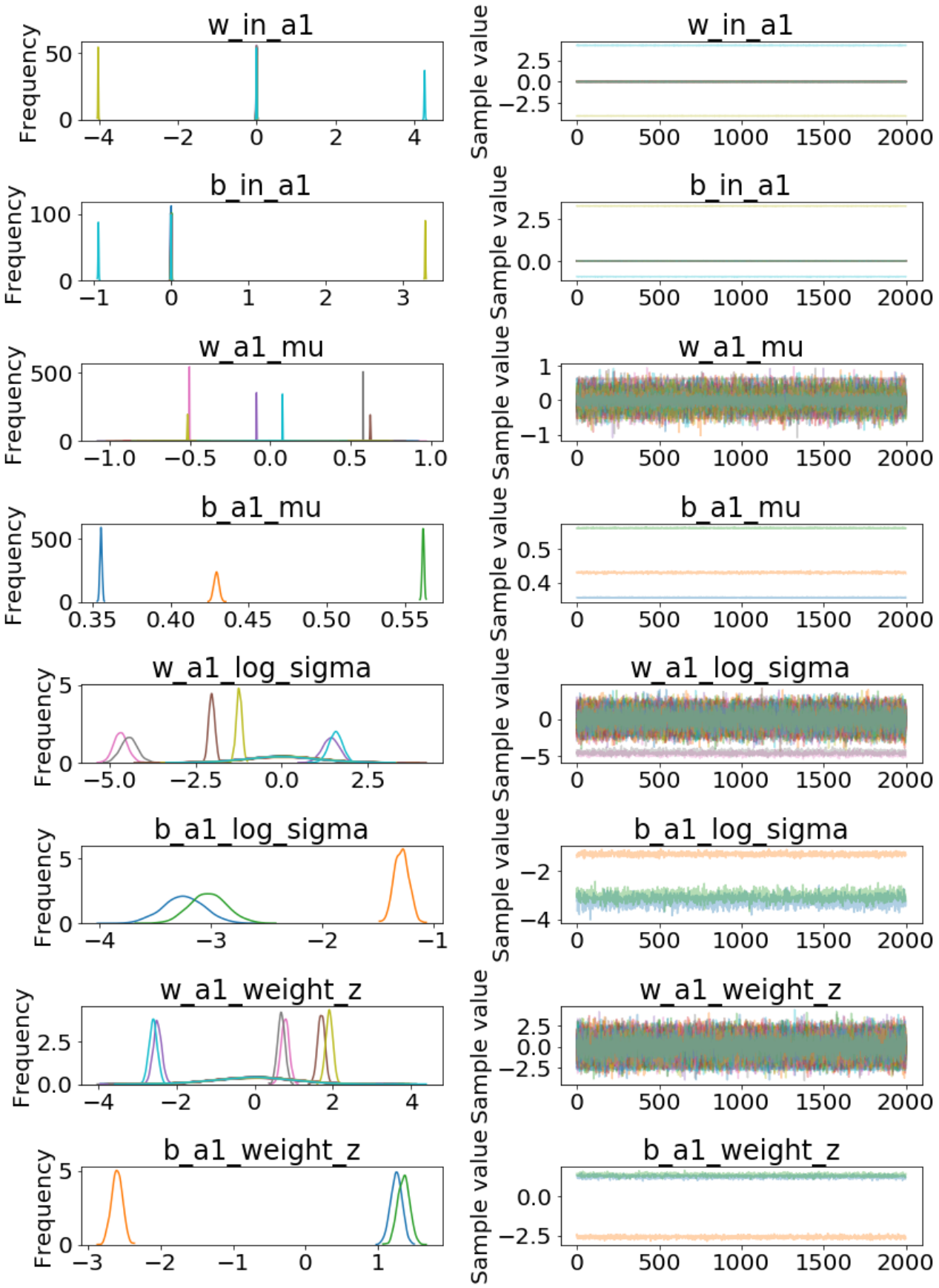
In [111]:
```
# Compute posterior means from the trace, which includes the intermediate results for mu, sigma, and weig
mu = np.mean(trace_mdn2['mu'], axis=0)
sigma = np.mean(trace_mdn2['sigma'], axis=0)
weight = np.mean(trace_mdn2['weight'], axis=0)

# Standardize the identities of the three Guassians for consistent colors
mu, sigma, weight, idx = standardize_gaussians(mu, sigma, weight)

# Generate plot similar to one in the problem
fig = plot_gaussians(x, mu, sigma, x, y, 'Mixture Density Model: 3 Gaussians')
```




We can see that the ADVI fit has learned something qualitatively similar to ML fit found by the optimizer in torch.

**Plot traces of the mu/sigma/lambda as an aid in debugging your sampler**

In [113]:
```
varnames_mdn2 =['w_in_a1', 'b_in_a1', 'w_a1_mu', 'b_a1_mu',
                'w_a1_log_sigma', 'b_a1_log_sigma', 'w_a1_weight_z', 'b_a1_weight_z']
figs = pm.traceplot(trace_mdn2, varnames=varnames_mdn2)
```




These traceplots are very interesting. We can see that some parameters stayed essentially frozen once reaonable values had

been found. These include the weights and biases defining the activations on the first layer. Once those were learned, the model explored different settings on the weights and biases that determined the location mu, the dispersion sigma, and the mixing weights. While it didn't explor the "whole" space, it explored a lot more of the space than the ML estimation did.

**So if this was a success, why the reference to a battle where Luke got his right hand chopped off by his dad?**

Because I spent three days of my life making every mistake possible until I stumbled onto a decent solution.

## Q3: Exploring Temperature in Sampling and Optimiztion

At various times in class we've discussed in very vague terms the relation between "temperature" and sampling from or finding optima of distributions. Promises would invariably be made that at some later point we'd discuss the concept of temperature and sampling/optima finding in more detail. Let's take this problem as an opportunity to keep our promise.

Let's start by considering the function $f(x, y)$ defined in the following code cell. $f(x, y)$ is a mixture of three well separated Gaussian probability densities.

In [125]:
```
make_cov = lambda  theta: np.array([[np.cos(theta), -np.sin(theta)], [np.sin(theta), np.cos(theta)]])

theta_vec = (5.847707364986893, 5.696776968254305, 1.908095937315489)
theta1, theta2, theta3 = theta_vec

# define gaussian mixture 1
cov1 = make_cov(theta1)
sigma1 = np.array([[2, 0],[0, 1]])
mvn1 = scipy.stats.multivariate_normal([12, 7], cov=cov1@sigma1@cov1.T)

# define gaussian mixture 2
cov2 = make_cov(theta2)
sigma2 = np.array([[1, 0],[0, 3]])
mvn2 = scipy.stats.multivariate_normal([-1, 6], cov=cov2@sigma2@cov2.T)

cov3 = make_cov(theta3)
sigma3 = np.array([[.4, 0],[0, 1.3]])
mvn3 = scipy.stats.multivariate_normal([3,-2], cov=cov3@sigma3@cov3.T)

f = lambda xvec: mvn1.pdf(xvec) + mvn2.pdf(xvec) + .5*mvn3.pdf(xvec)

p = lambda x, y: f([x,y])
```

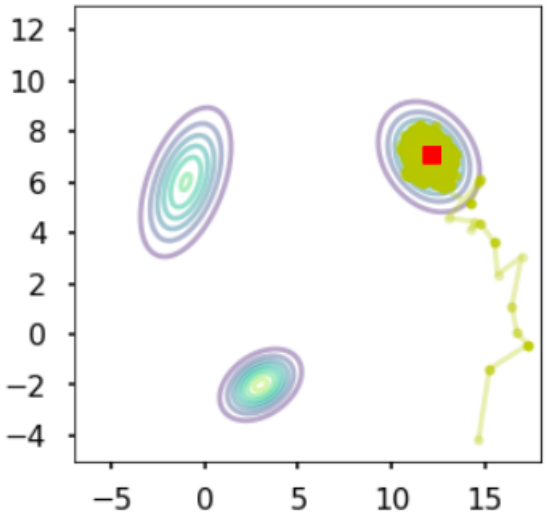
In [140]:
```
# Load persisted table of variables
fname: str = 'temperature.pickle'
vartbl: Dict = load_vartbl(fname)

# Set plot style
mpl.rcParams.update({'font.size': 20})

def arange_inc(x: float, y: float = None, z: float = None) -> np.ndarray:
    """Return a numpy arange inclusive of the end point, i.e. range(start, stop + 1, step)"""
    if y is None:
        (start, stop, step) = (1, x + 1, 1)
    elif z is None:
        (start, stop, step) = (x, y + 1, 1)
    elif z > 0:
        (start, stop, step) = (x, y + z, z)
    elif z < 0:
        (start, stop, step) = (x, y - z, z)
    return np.arange(start, stop, step)
```

## Part A Visualization and Metropolis

**A1**. Visualize $p(x, y)$ with a contour or surface plot. Make sure to title your plot and label all axes. What do you notice about $p(x, y)$? Do you think it will be an easy function to sample?

**A2**. Generate 20000 samples from $p(x, y)$ using the Metropolis algorithm. Pick individual gaussian proposals in $x$ and $y$ with $\sigma = 1$, initial values, burnin parameters, and thinning parameter. Plot traceplots of the $x$ and $y$ marginals as well as autocorrelation plots. Plot a pathplot of your samples. Based on your visualizations, has your Metropolis sampler generated an appropriate representation of the distribution $p(x, y)$?

A pathplot is just your samples trace overlaid on your pdf, so that you can see how the sampler traversed. It looks something like this:

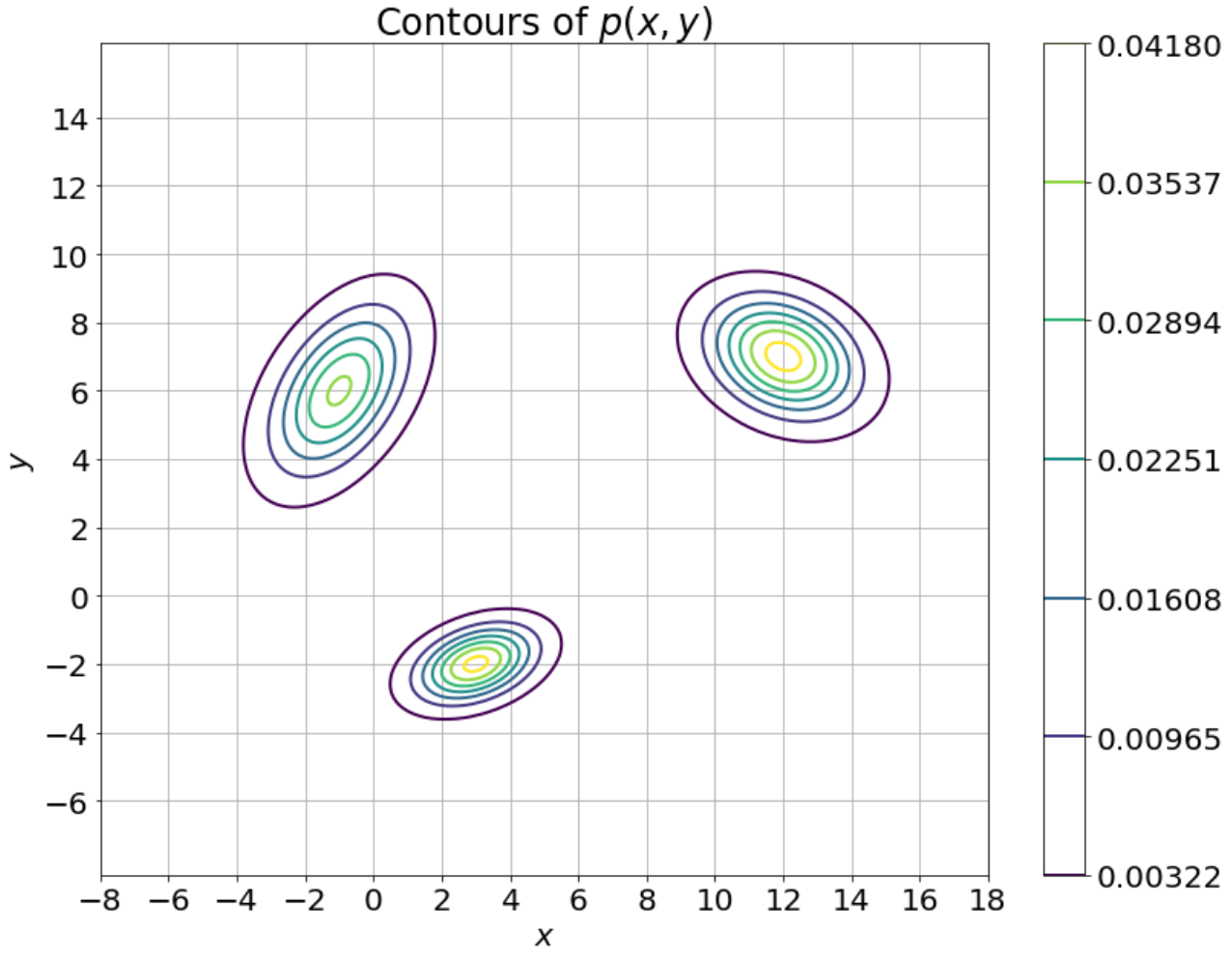



**A1**. Visualize $p(x, y)$ with a contour or surface plot. Make sure to title your plot and label all axes. What do you notice about $p(x, y)$? Do you think it will be an easy function to sample?

In [141]:
```
# Range of x and y
x_min: float = -8.0
x_max: float = 18.0
y_min: float = -6.0
y_max: float = 14.0

# Step size for x and y
dx: float = 0.125
dy: float = 0.125
# Discrete sample points for x and y
xx = arange_inc(x_min, x_max, dx)
yy = arange_inc(y_min, y_max, dy)
# Grid of x and y
grid_size_x: int = len(xx)
grid_size_y: int = len(yy)
# Grid of sample points for contour plots
x_grid, y_grid = np.meshgrid(xx, yy)

# Grid of p(x, y)
try:
    # raise ValueError
    p_grid = vartbl['p_grid']
    print(f'Loaded grid of p(x, y))')
except:
    p_grid = np.zeros((grid_size_y, grid_size_x))
    for j, x in enumerate(xx):
        for i, y in enumerate(yy):
            p_grid[i, j] = p(x, y)
    # Approximate normalization constant
    norm: float = np.sum(p_grid)*dx*dy
    p_grid = p_grid / norm
    # Save normalized grid
    vartbl['p_grid'] = p_grid
    save_vartbl(vartbl, fname)
```

```
Loaded grid of p(x, y))
```

In [143]:
```
def plot_contour(x_grid, y_grid, p_grid, title):
    # Get min and max from grids
    x_min: float = np.min(x_grid[0,:])
    x_max: float = np.max(x_grid[0,:])
    y_min: float = np.min(y_grid[:,0])
    y_max: float = np.max(y_grid[:,0])

    # Plot the contours of f(x,y)
    fig, ax = plt.subplots(figsize=[13, 10])
    ax.set_title(title)
    ax.set_xlabel('$x$')
    ax.set_ylabel('$y$')
    ax.set_xlim(x_min, x_max)
    ax.set_ylim(y_min, y_max)
    ax.set_xticks(arange_inc(x_min, x_max, 2.0))
    ax.set_yticks(arange_inc(y_min, y_max, 2.0))
    ax.set_aspect('equal', 'datalim')
    # compute desired contour levels
    p_max = np.max(p_grid)
    num_levels = 8
    step_size = p_max / (num_levels-1)
    half_step = 0.5*step_size
    levels = arange_inc(half_step, p_max - half_step, step_size)
    # Generate contour plot
    cs = ax.contour(x_grid, y_grid, p_grid, levels=levels, linewidths=2)
    fig.colorbar(cs, ax=ax)
    ax.grid()
    return fig, ax

# Plot contours of p(x, y)
fig = plot_contour(x_grid, y_grid, p_grid, 'Contours of $p(x, y)$')
```




The function is highly multi-modal. It has three pockets of high density that are well separated. As a result of this structure,

sampling from this function with Metropolis is going to be very hard. If the step size is small enough to have a decent acceptance rate, once the sampler lands in one pocket, it is likely to never leave. If the step size is big enough to hop around, it will likely be so large that the acceptance rate will be very low and the sampler will be extremely inefficient.

**A2**. Generate 20000 samples from $p(x, y)$ using the Metropolis algorithm. Pick individual gaussian proposals in $x$ and $y$ with $\sigma = 1$, initial values, burnin parameters, and thinning parameter. Plot traceplots of the $x$ and $y$ marginals as well as autocorrelation plots. Plot a pathplot of your samples. Based on your visualizations, has your Metropolis sampler generated an appropriate representation of the distribution $p(x, y)$?

**Parameters describing each cluster**

In [186]:
```
# mu, sigma, and weight for Gaussian 1
mu_1 = np.array([12, 7])
R_1 = make_cov(theta1)
cov_1 = R_1 @ sigma1 @ R_1.T
weight_1 = 1.0 / 2.5
#
# mu, sigma, and weight for Gaussian 2
mu_2 = np.array([-1, 6])
R_2 = make_cov(theta2)
cov_2 = R_2 @ sigma2 @ R_2.T
weight_2 = 1.0 / 2.5

# mu, sigma, and weight for Gaussian 3
mu_3 = np.array([3, -2])
R_3 = make_cov(theta3)
cov_3 = R_3 @ sigma3 @ R_3.T
weight_3 = 0.5 / 2.5

# Assemble into one mixture distribution
weights = np.array([weight_1, weight_2, weight_3])
mus = np.array([mu_1, mu_2, mu_3])
```

**Metropolis Sampler & Friends**

In [235]:
```
# Citation: this sampler was presented in lab 9
def metropolis(logp, proposal, step_size, num_samples, X_init):
    """Metropolis sampler"""
    # Initialize array of samples
    samples=np.zeros((num_samples, K))
    # Initialize X_prev to the initial point that was passed in
    X_prev = X_init
    accepted = 0
    # Draw num_samples sample points
    iterator_i = tqdm.tqdm(range(num_samples)) if num_samples > 1000 else range(num_samples)
    for i in iterator_i:
        X_star = proposal(X_prev, step_size)
        logp_star = logp(X_star)
        logp_prev = logp(X_prev)
        logpdf_ratio = logp_star -logp_prev
        u = np.random.uniform()
        if np.log(u) <= logpdf_ratio:
            samples[i] = X_star
            X_prev = X_star
            accepted += 1
        # We always draw a sample; whether it was accepted just determines whether we take the step or no
        else:
            samples[i]= X_prev
    # Return the samples and the acceptance ratio
    return samples, accepted


# The log-probability function
def logp(X: np.array):
    """Log probability at this point"""
    # Dispatch call to vectorized pdf f(X) for efficiency
    return log(f(X))

# Crate a proposal distribution with sigma=1 as per problem statement
def proposal(X, step_size):
    """Normal proposal distribution; step_size is passed from metropolis"""
    return np.random.normal(loc=X, scale=step_size * np.ones(K))
```

**Draw Samples**

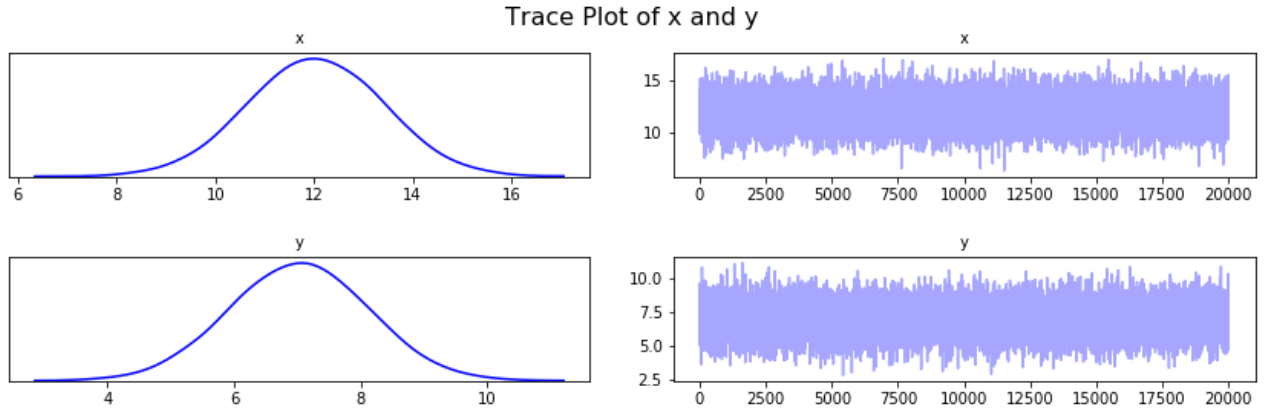
In [197]:
```python
# Number of samples
num_samples: int = 20000
# Number of dimensions
K: int = 2

# Set the starting point as the weighted average of the clusters
X_init = np.average(mus, axis=0, weights=weights)
# Set the stepsize to 1
step_size = 1.0
# Set tuning and thinning
tuning: int = 10000
thinning: int = 10
# Compute total number of raw samples required to generate num_samples
num_samples_raw: int = tuning + thinning * num_samples

# Load the samples if available
try:
    # raise ValueError
    raw_samples = vartbl['raw_samples']
    samples = vartbl['samples']
    acceptance_ratio = vartbl['acceptance_ratio']
    print(f'Loaded metropolis samples')
except:
    # Draw samples
    print(f'Drawing {num_samples_raw} samples with Metropolis algorithm...')
    raw_samples, accepted = metropolis(logp, proposal, step_size, num_samples_raw, X_init)
    acceptance_ratio = accepted / num_samples_raw
    # Thin the samples
    samples = raw_samples[tuning::thinning]
    # Save samples to vartbl
    vartbl['raw_samples'] = raw_samples
    vartbl['samples'] = samples
    vartbl['acceptance_ratio'] = acceptance_ratio
    save_vartbl(vartbl, fname)

# Report acceptance ratio
print(f'Acceptance ratio was {acceptance_ratio:0.4f}.')

# Make trace by hand
trace = dict()
trace['x'] = samples[:,0]
trace['y'] = samples[:,1]
```

```
Loaded metropolis samples
Acceptance ratio was 0.6037.
```

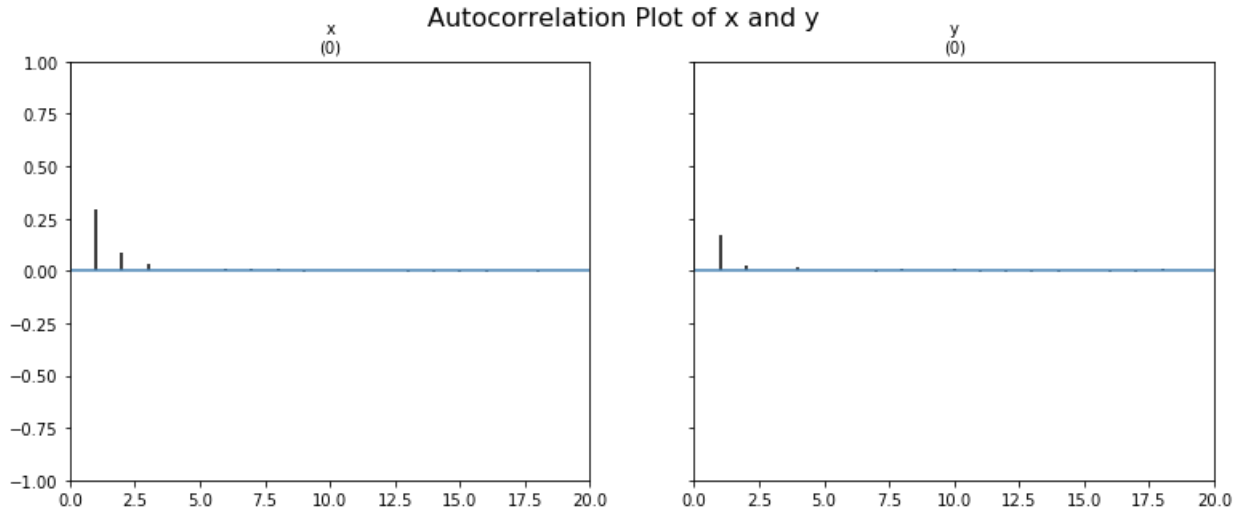
**Trace Plot & Autocorr Plot**

In [209]:
```
def plot_trace(trace, title):
    """Generate a trace plot with Arviz"""
    plot_kwargs = {'color':'b'}
    axs = az.plot_trace(trace, trace_kwargs=plot_kwargs)
    fig = plt.gcf()
    fig.suptitle(title, fontsize=16)
    # ax0, ax1 = axs[0,0], axs[0,1]
    return fig, axs


def plot_autocorr(trace, title):
    """Generate an autocorr plot with Arviz"""
    axs = az.plot_autocorr(trace, max_lag=20)
    fig = plt.gcf()
    fig.suptitle(title, fontsize=16)
    # ax0, ax1 = axs[0,0], axs[0,1]
    return axs


def plot_path(samples, x_grid, y_grid, p_grid, title):
    """Generate a cloud of sample points"""
    # Extract x and y from the samples
    x = samples[:,0]
    y = samples[:,1]
    # Thin out x and y for the plot
    thin = 10
    x = x[::thin]
    y = y[::thin]
    # Start with the contour plot
    fig, ax = plot_contour(x_grid, y_grid, p_grid, title)
    ax.set_title(title)
    # Add path of sample points
    ax.plot(x, y, color='r', linewidth=1, marker='o', markersize=2.5, alpha=0.25)
    return fig

def plot_cloud(samples, x_grid, y_grid, p_grid, title):
    """Generate a cloud of sample points"""
    # Extract x and y frmo the samples
    x = samples[:,0]
    y = samples[:,1]
    # Thin out x and y for the plot
    thin = 1
    x = x[::thin]
    y = y[::thin]
    # Start with the contour plot
    fig, ax = plot_contour(x_grid, y_grid, p_grid, title)
    # Add path of sample points
    ax.plot(x, y, color='r', linewidth=0, marker='o', markersize=1, alpha=0.15)
    return fig
```

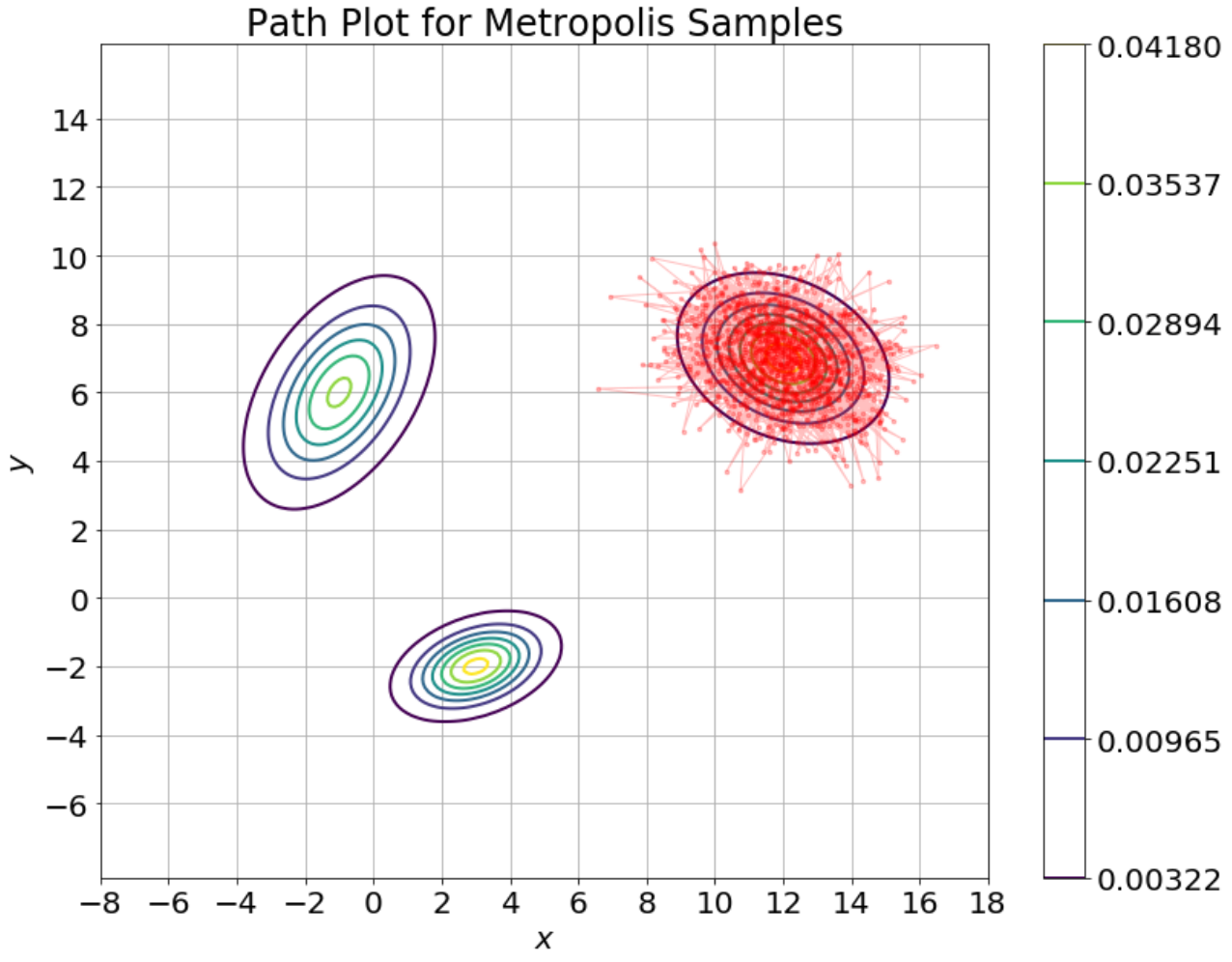
In [190]:
```
# Generate the traceplot
fig = plot_trace(trace, 'Trace Plot of x and y')
```



In [191]:
```
# Generate the autocorr plot
fig = plot_autocorr(trace, 'Autocorrelation Plot of x and y')
```




In [192]:
```
# Plot the path taken by the sampler
fig = plot_path(samples, x_grid, y_grid, p_grid, 'Path Plot for Metropolis Samples')
```




Ouch! The trace plot and autocorr plot both look fine. If we didn't know that this function really had three peaks, we would think we were doing a great job sampling it and have no idea that we were only touching 40% of it. This is a good illustration of why mixture distributions are difficult to sample from.

As an aside, I didn't include line segments in the plot because it would have made a huge red splotch. These are the tiniest red dots possible with an alpha of just 0.1 so we can see the contour lines behind them.

## Part B: Changing pdfs using temperature

Given a function $p(x)$ we can rewrite that function in following way:

$$p(x) = e^{-(-\log(p(x))}$$

So if define the energy density for a function as $E(x) \equiv -\log p(x)$

We can now aim to sample from the function parameratized by a Temperature $T$.

$$p(x|T) = e^{-\frac{1}{T}E(x)} = p(x)^{\frac{1}{T}}$$

If we set T=1 we're sampling from our original function $p(x)$.

**B1** In line with A1, visualize modified pdfs (dont worry about normalization) by setting the temperatures to $T = 10$ and $T = 0.1$.

**B2**. Modify your Metropolis algorithm above to take a temperature parameter `T` as well as to keep track of the number of rejected proposals. Generate 20000 samples from $p(x, y)$ at for each of the following temperatures: {0.1, 1, 3, 7, 10}. Construct histograms of the marginals, traceplots, autocorrelation plots, and a pathplot for your samples at each temperature. What happens to the number of rejections as temperature increases? In the limits $T \to 0$ and $T \to \infty$ what do you think your samplers will do?

**B3**. Approximate the $f(X)$ by the appropriate mixture of Gaussians as a way of generating samples from $f(X)$ to compare with other sampling methods. Use scipy.stats.multivariate_normal to generate 20000 samples. How do the histograms compare with the histograms for the samples from $f(X)$ at each temperature. At what temperature do the samples best represent the function?

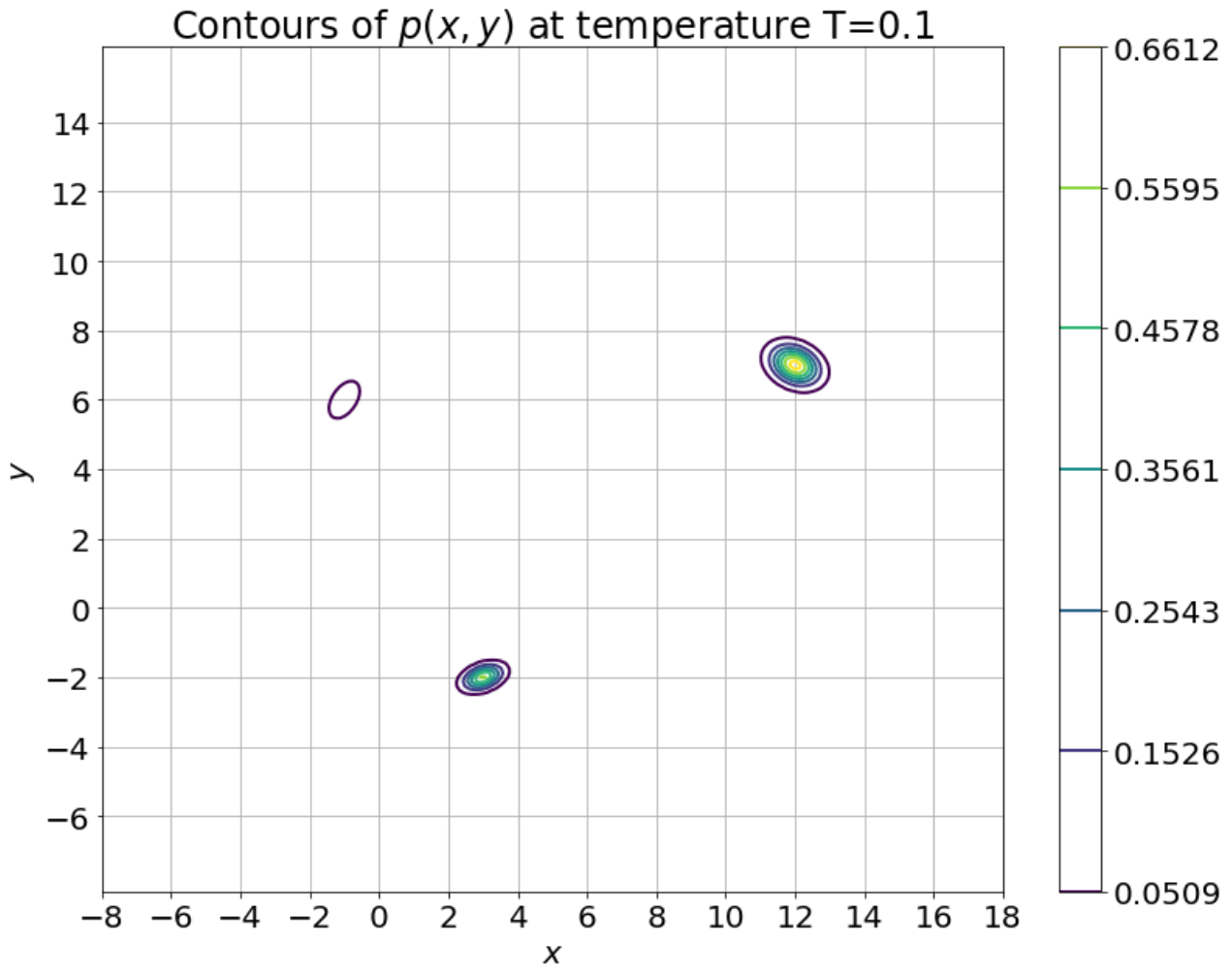
In [193]:
```python
def logp_temp(X: np.ndarray, T: float):
    """log probability function with a temperature parameter T"""
    return logp(X) / T

def p_temp(x: float, y: float, T: float):
    """Probability density including a temperature parameter T"""
    return np.power(p(x, y), 1.0/T)
```

**B1** In line with A1, visualize modified pdfs (dont worry about normalization) by setting the temperatures to $T = 10$ and $T = 0.1$.

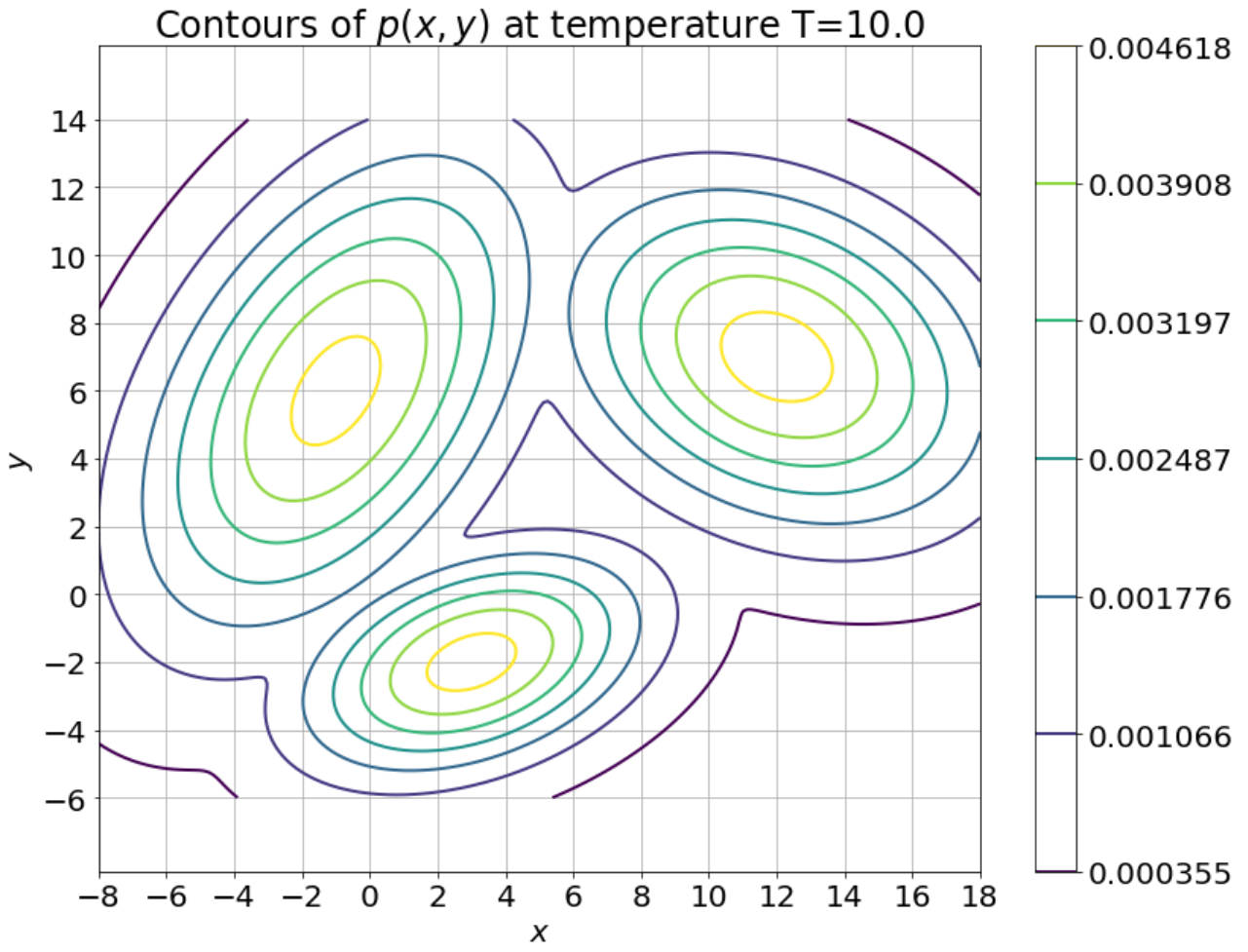
In [194]:

```
# Draw contour grid at selected temperatures
try:
    # raise ValueError
    p_grid_by_temp = vartbl['p_grid_by_temp']
except:
    p_grid_by_temp = dict()

# List of temperatures to sample from
temps = np.array([0.1, 1.0, 3.0, 7.0, 10.0])
# Build a probability grid at each temperature if it's not in the table
for T in temps:
    if T not in p_grid_by_temp:
        p_grid_T = np.zeros((grid_size_y, grid_size_x))
        for j, x in enumerate(xx):
            for i, y in enumerate(yy):
                # The pdf at this point with the given temperature
                p_grid_T[i, j] = p_temp(x, y, T)
        # Approximate normalization constant
        norm: float = np.sum(p_grid_T)*dx*dy
        p_grid_T = p_grid_T / norm
        # Save this grid to the table
        fig = p_grid_by_temp[T] = p_grid_T
    else:
        # Load this grid from the table
        p_grid_T = p_grid_by_temp[T]

# Save vartbl with updated entries
vartbl['p_grid_by_temp'] = p_grid_by_temp
save_vartbl(vartbl, fname)
```

In [195]:
```
# Build a contour plot at temperature 0.1
T = 0.1
fig = plot_contour(x_grid, y_grid, p_grid_by_temp[T], f'Contours of $p(x, y)$ at temperature T={T}')
```



In [196]:
```
# Build a contour plot at temperature 10.0
T = 10.0
fig = plot_contour(x_grid, y_grid, p_grid_by_temp[T], f'Contours of $p(x, y)$ at temperature T={T}')
```




**B2**. Modify your Metropolis algorithm above to take a temperature parameter `T` as well as to keep track of the number of rejected proposals. Generate 20000 samples from $p(x, y)$ at for each of the following temperatures: {0.1, 1, 3, 7, 10}. Construct histograms of the marginals, traceplots, autocorrelation plots, and a pathplot for your samples at each temperature. What happens to the number of rejections as temperature increases? In the limits $T \to 0$ and $T \to \infty$ what do you think your samplers will do?

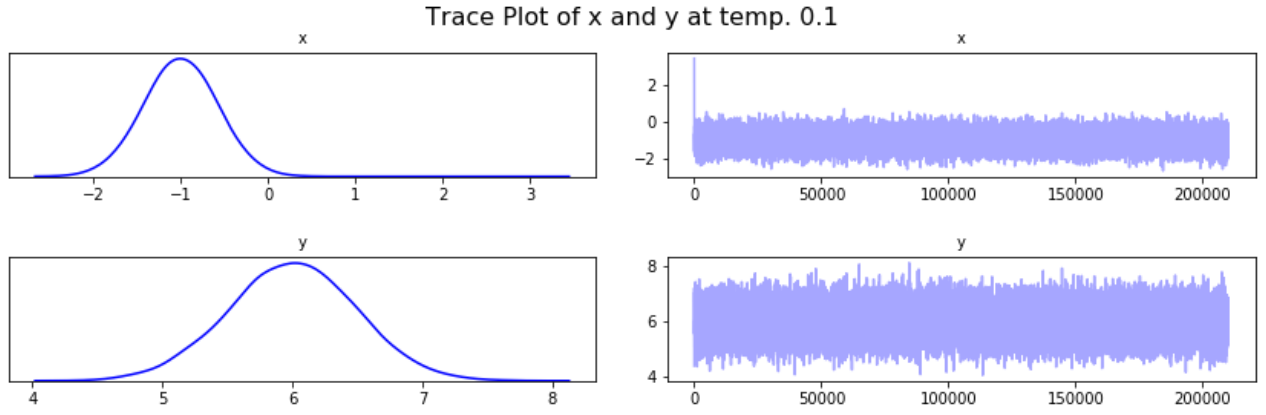
In [169]:
```
# Draw samples by temperature
try:
    samples_by_temp = vartbl['samples_by_temp']
    acceptance_by_temp = vartbl['acceptance_by_temp']
except:
    samples_by_temp = dict()
    acceptance_by_temp = dict()

# Compute a sample for each temperature if it's not in the table
for T in temps:
    if T not in samples_by_temp:
        # create p(X;T) in situ by binding T
        logp_T = lambda X : logp_temp(X, T)
        # Draw samples
        print(f'Drawing {num_samples_raw} samples at temperature T={T}')
        raw_samples_T, accepted_T = metropolis(logp_T, proposal, step_size, num_samples_raw, X_init)
        # Save samples and acceptance ratio
        samples_by_temp[T] = raw_samples_T
        acceptance_by_temp[T] = accepted_T / num_samples_raw
    else:
        print(f'Loaded {num_samples_raw} samples at temprerature T={T}')
    # Report acceptance ratio
    print(f'Acceptance ratio at T={T} was {acceptance_by_temp[T]:0.4f}.')

# Save vartbl with updated entries
vartbl['samples_by_temp'] = samples_by_temp
vartbl['acceptance_by_temp'] = acceptance_by_temp
save_vartbl(vartbl, fname)
```

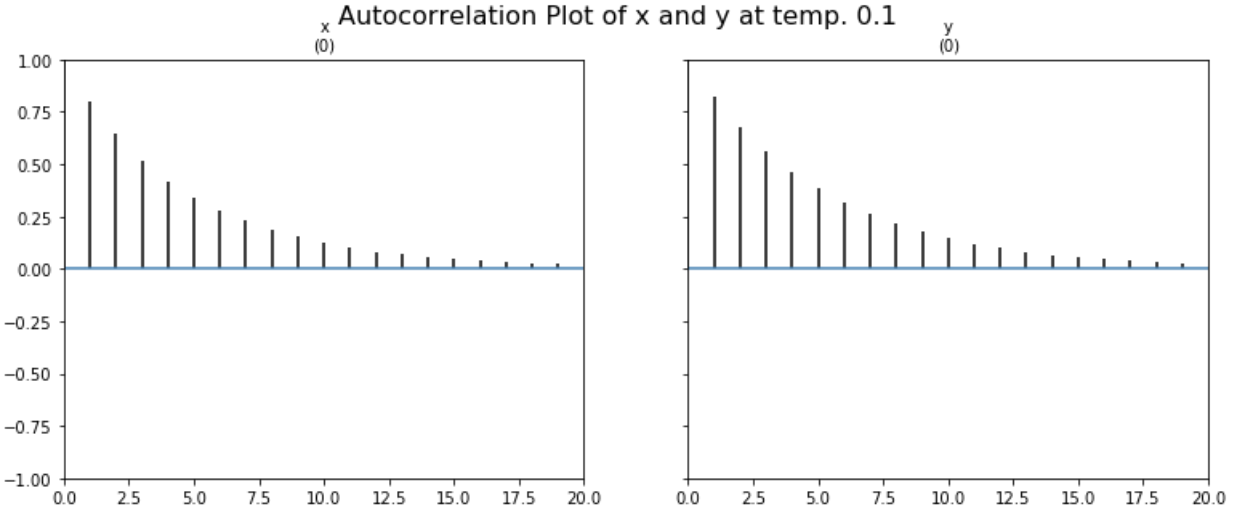
```
Loaded 210000 samples at temprerature T=0.1
Acceptance ratio at T=0.1 was 0.2258.
Loaded 210000 samples at temprerature T=1.0
Acceptance ratio at T=1.0 was 0.6228.
Loaded 210000 samples at temprerature T=3.0
Acceptance ratio at T=3.0 was 0.7483.
Loaded 210000 samples at temprerature T=7.0
Acceptance ratio at T=7.0 was 0.8457.
Loaded 210000 samples at temprerature T=10.0
Acceptance ratio at T=10.0 was 0.8739.
```

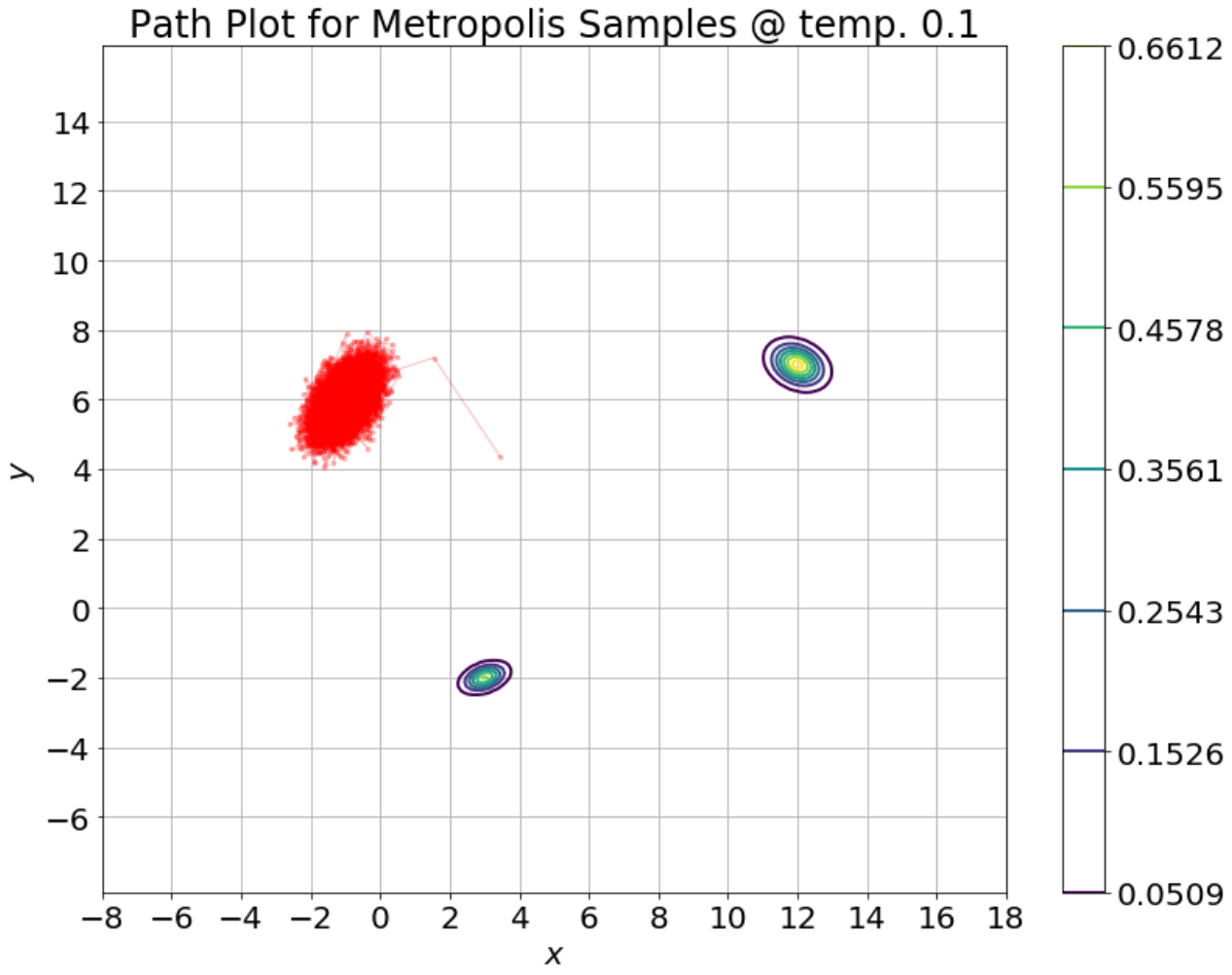
**B3**. Approximate the $f(X)$ by the appropriate mixture of Gaussians as a way of generating samples from $f(X)$ to compare with other sampling methods. Use scipy.stats.multivariate_normal to generate 20000 samples. How do the histograms compare with the histograms for the samples from $f(X)$ at each temperature. At what temperature do the samples best represent the function?

In [203]:
```
def plots_one_temp(T):
    """Genereate all three plots of interest at one temperature"""
    # Get the trace for this temperature
    trace_T = {'x': samples_by_temp[T][:,0],
               'y': samples_by_temp[T][:,1]}
    # Generate the traceplot
    fig1 = plot_trace(trace_T, f'Trace Plot of x and y at temp. {T}')
    # Generate the autocorr plot
    fig2 = plot_autocorr(trace_T, f'Autocorrelation Plot of x and y at temp. {T}')
    # Plot the path taken by the sampler
    fig3 = plot_path(samples_by_temp[T], x_grid, y_grid, p_grid_by_temp[T],
                     f'Path Plot for Metropolis Samples @ temp. {T}')
    return [fig1, fig2, fig3]
```

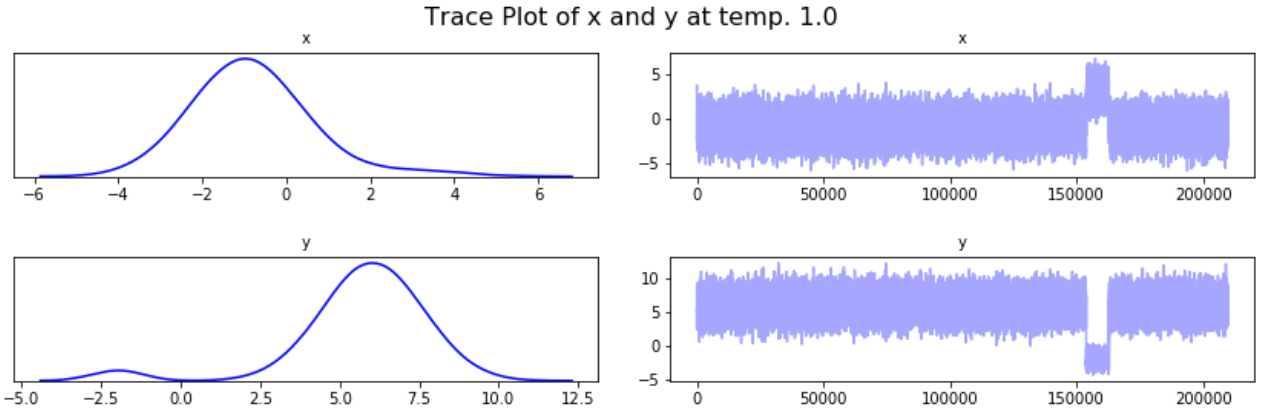
In [204]:
```
figs = plots_one_temp(0.1)
```

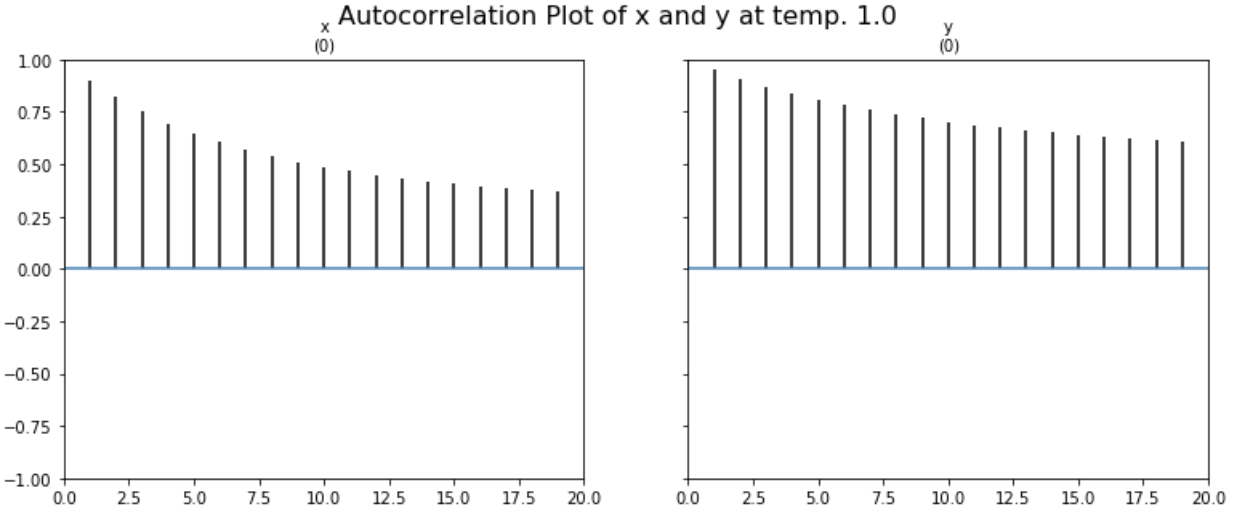


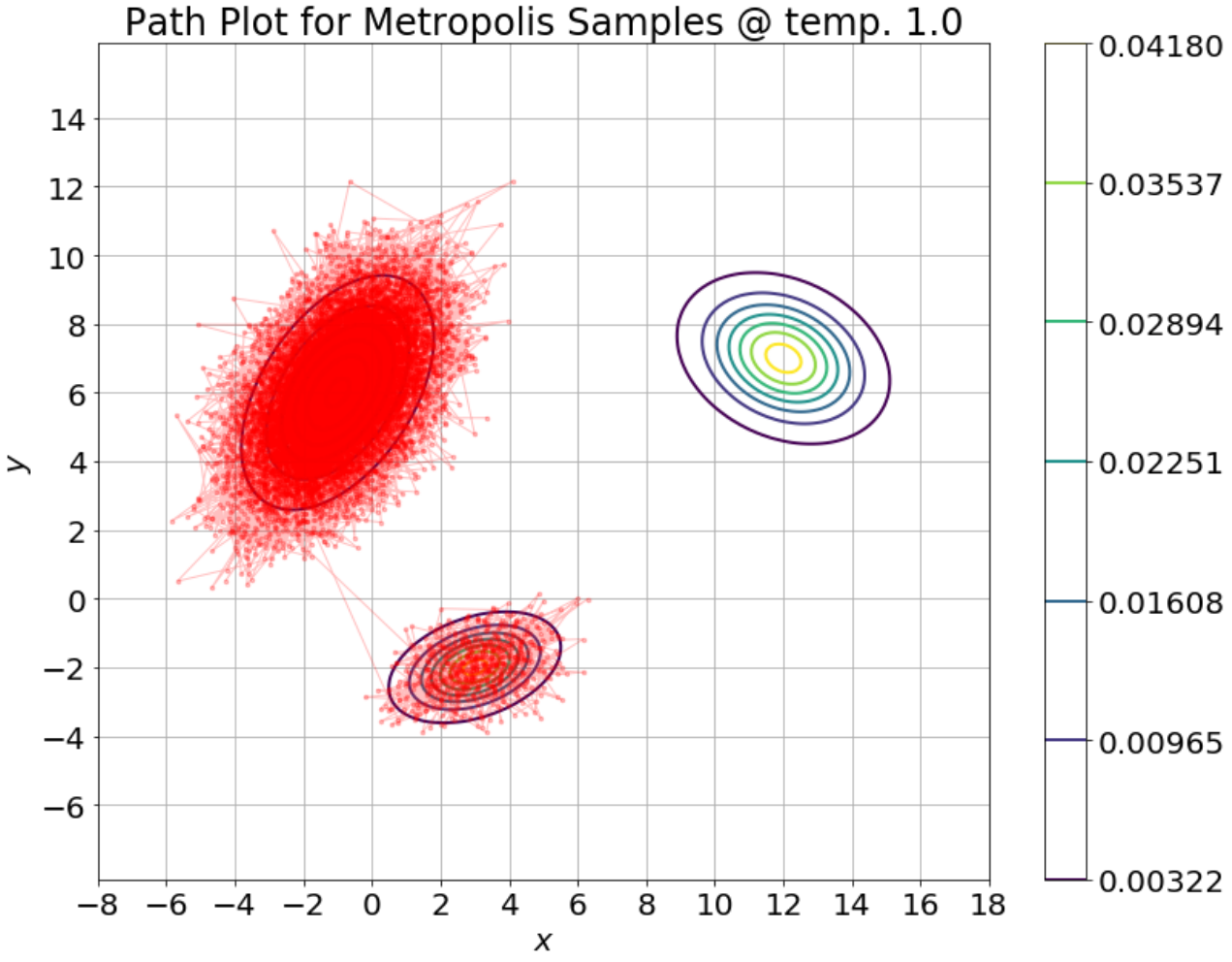




Path Plot for Metropolis Samples @ temp. 0.1
14
12
10
8
6
4
2
0
−2
−4
−6
y
−8 −6 −4 −2 0 2 4 6 8 10 12 14 16 18
x
0.6612
0.5595
0.4578
0.3561
0.2543
0.1526
0.0509

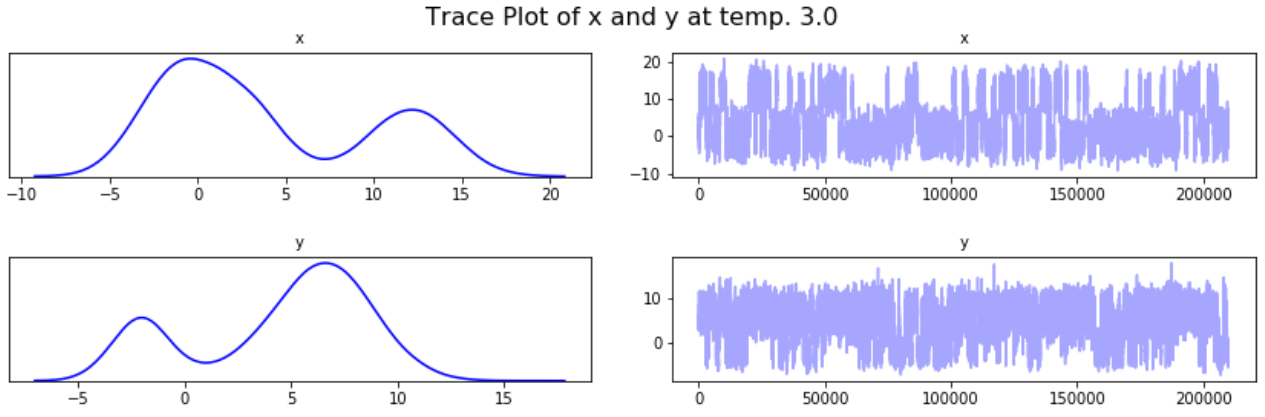
In [205]:
```
figs = plots_one_temp(1.0)
```

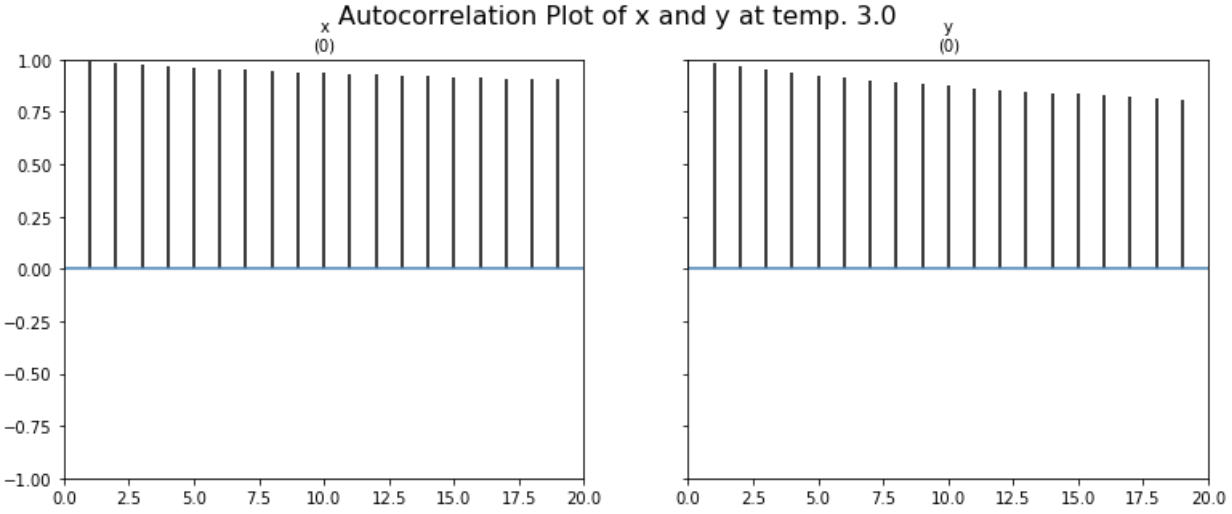


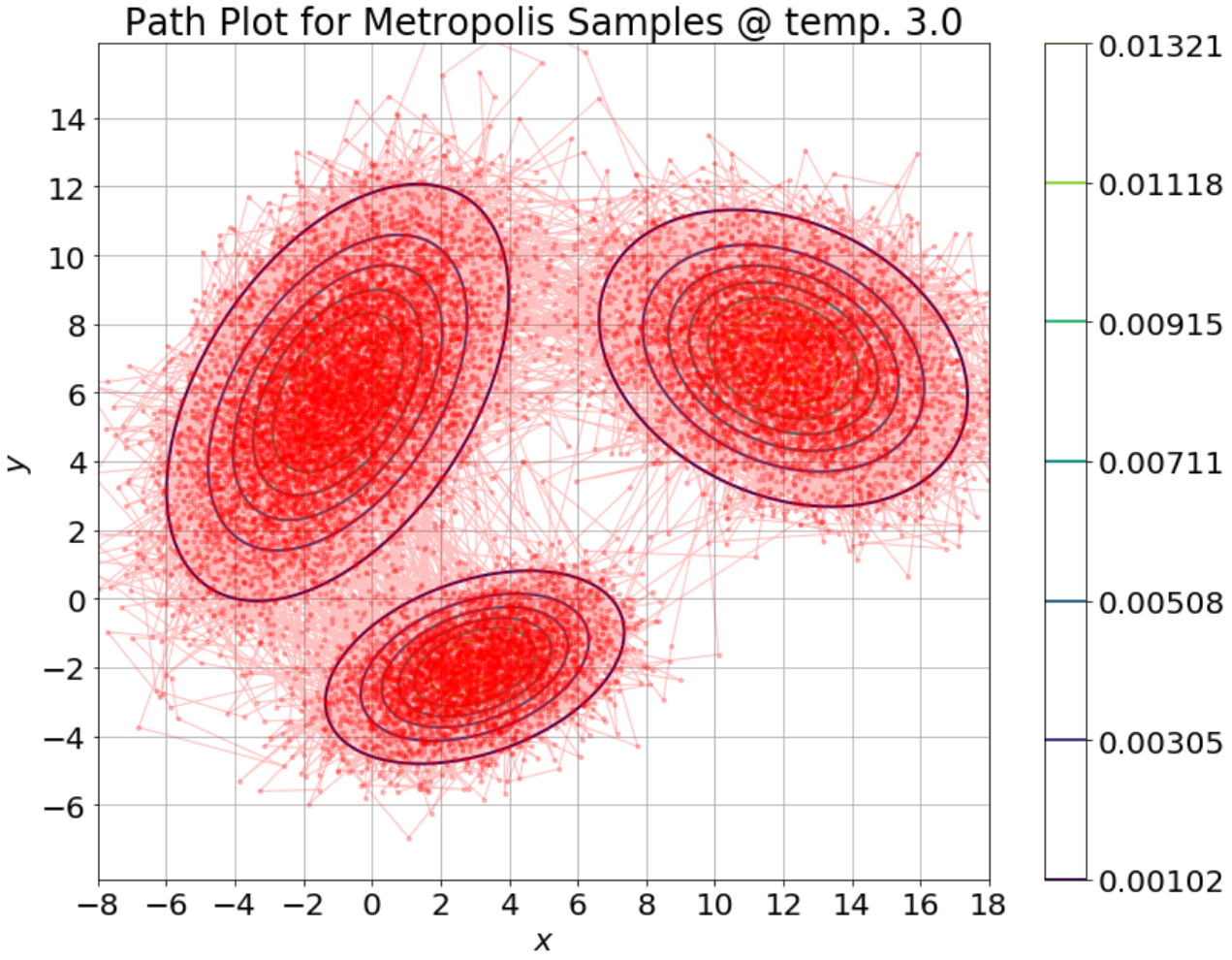




Path Plot for Metropolis Samples @ temp. 1.0
14
12
10
8
6
4
2
0
−2
−4
−6
y
−8 −6 −4 −2 0 2 4 6 8 10 12 14 16 18
x
0.04180
0.03537
0.02894
0.02251
0.01608
0.00965
0.00322

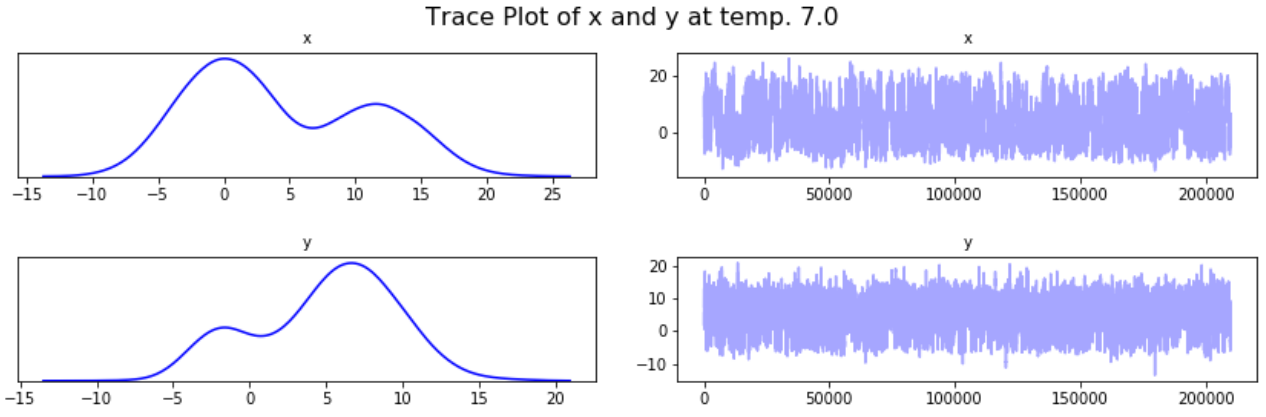
In [206]:
```
figs = plots_one_temp(3.0)
```

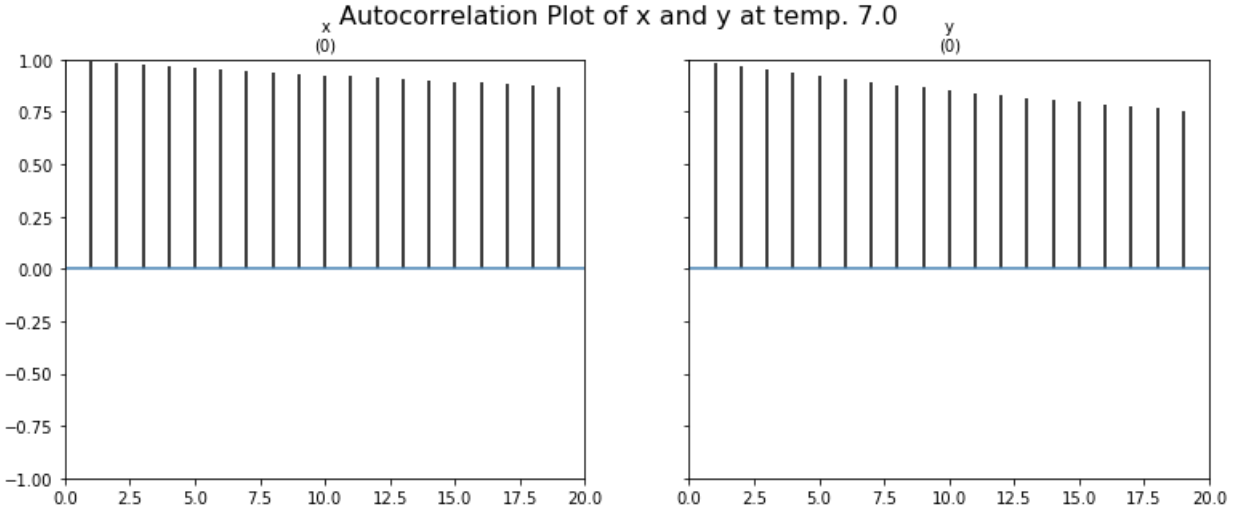


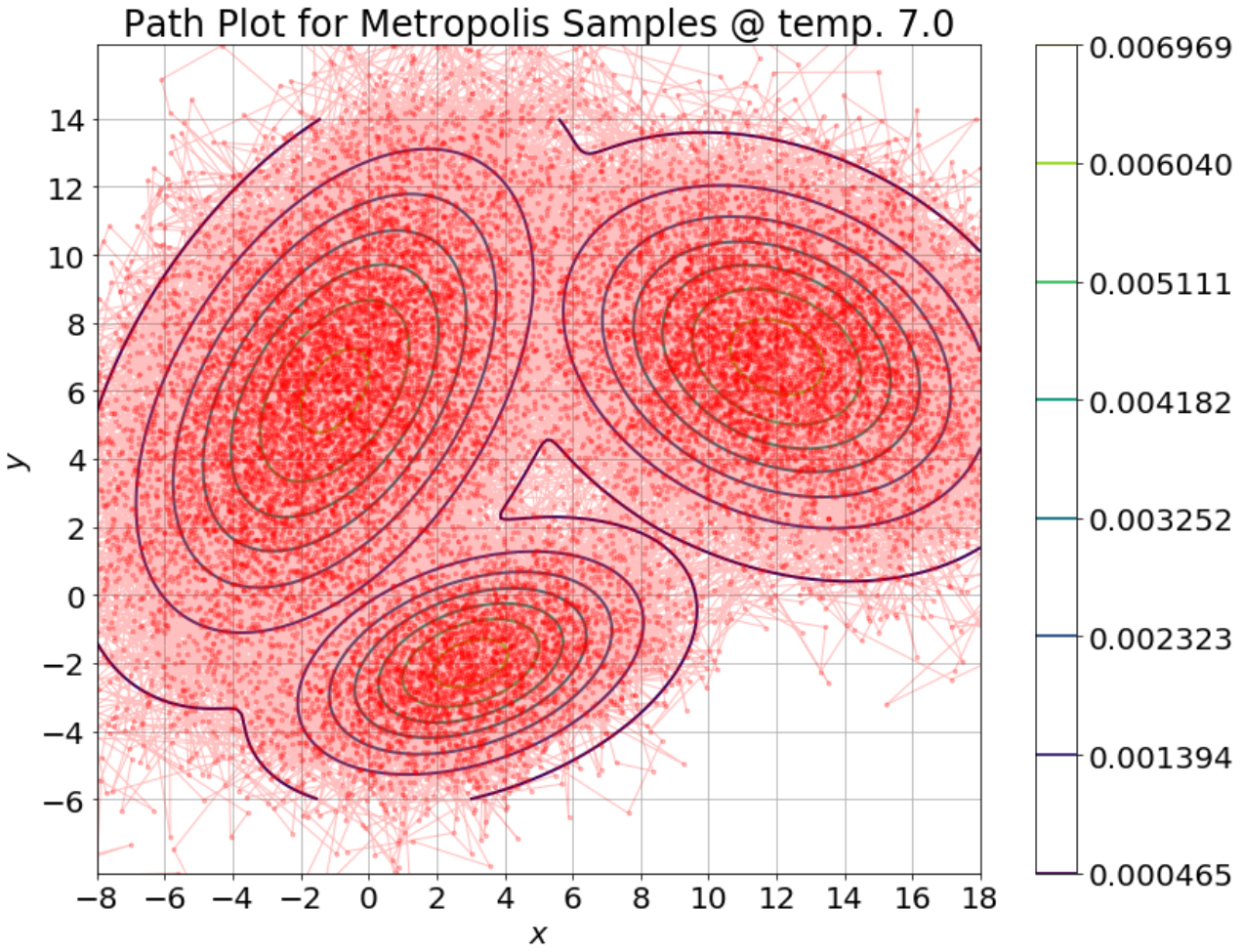




Path Plot for Metropolis Samples @ temp. 3.0
14
12
10
8
6
4
2
0
−2
−4
−6
y
−8
−6
−4
−2
0
2
4
6
8
10
12
14
16
18
x
0.01321
0.01118
0.00915
0.00711
0.00508
0.00305
0.00102

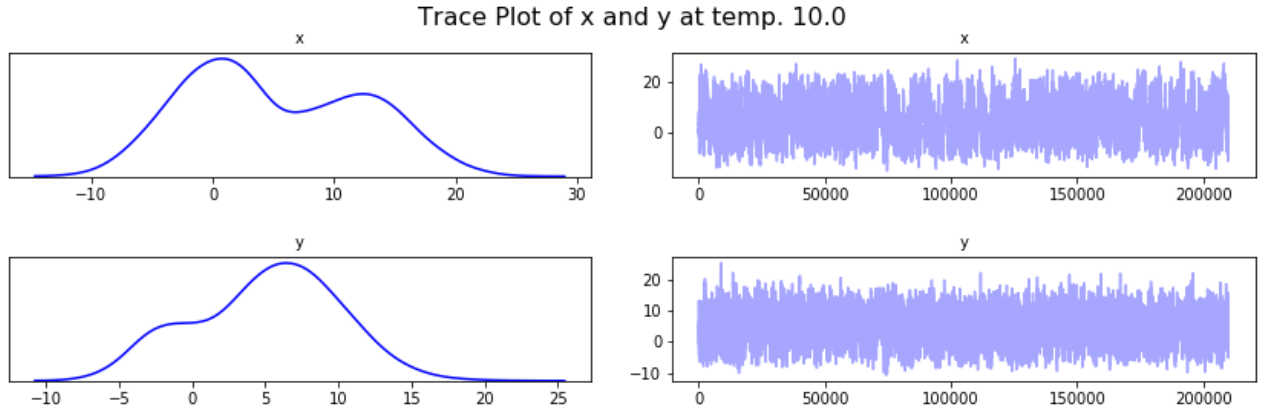
In [207]:
```
figs = plots_one_temp(7.0)
```

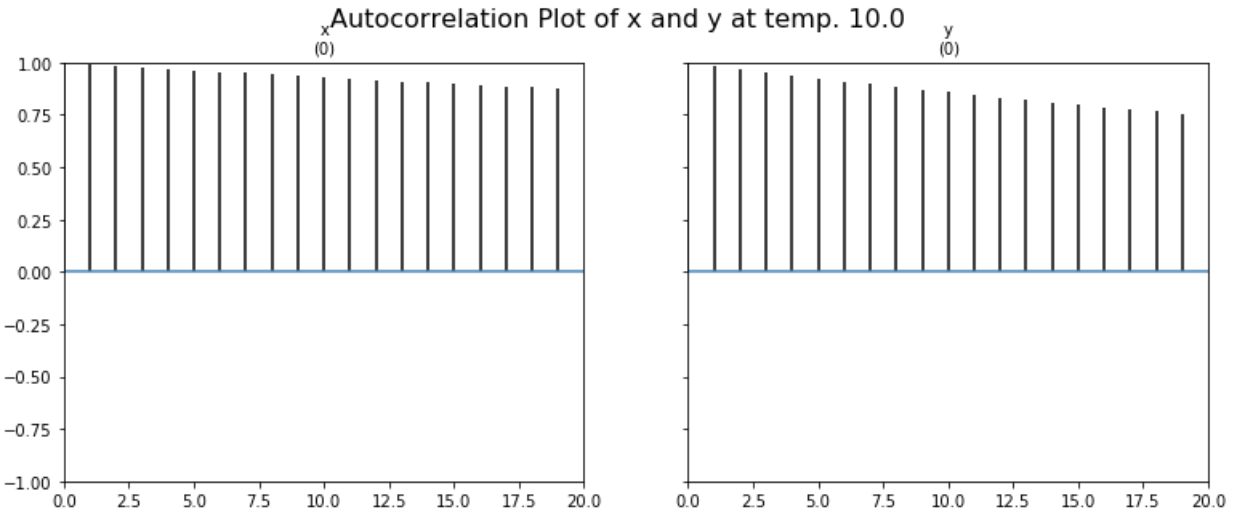


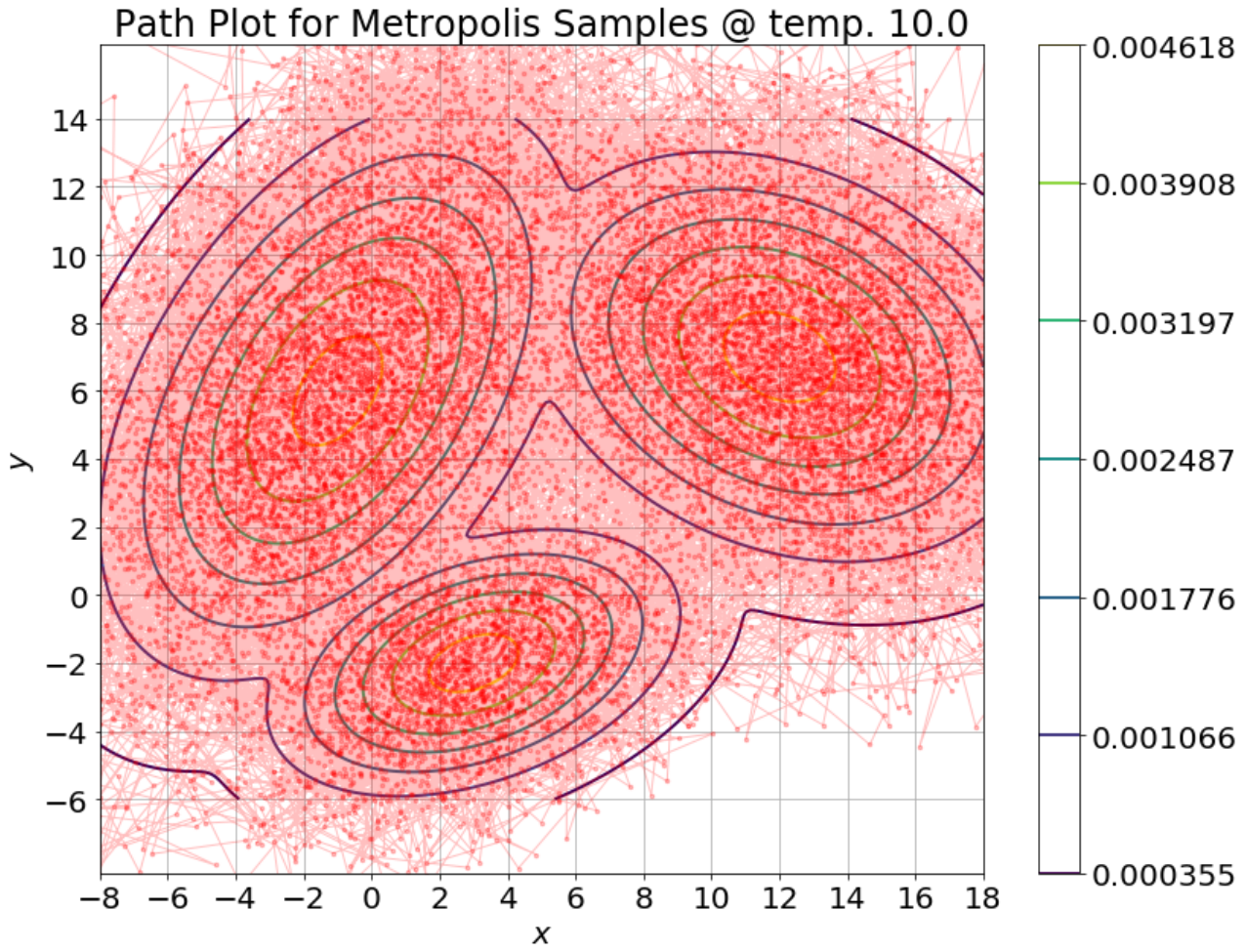




Path Plot for Metropolis Samples @ temp. 7.0
y
x
14
12
10
8
6
4
2
0
−2
−4
−6
−8
−6
−4
−2
0
2
4
6
8
10
12
14
16
18
0.006969
0.006040
0.005111
0.004182
0.003252
0.002323
0.001394
0.000465

In [208]:
```
figs = plots_one_temp(10.0)
```






**What happens to the number of rejections as temperature increases?**
**In the limits $T \to 0$ and $T \to \infty$ what do you think your samplers will do?**

This series of charts makes the pattern very clear. At low temperatures, the contours of the probabilty function are very tight (the function is steep with narrow peaks). The sampler has a low acceptance rate and is very unlikely to accept a step that does not move towards the peak. Once it finds whichever peak it stumbles across first, it is unlikely to leave.

As the temperature increases, the contours expand, and the peaks become broader. The acceptance rate increases, become more and more likely to take exploratory steps in a direction that does not improve the probability. Eventually when the temperature is high enough, the sampler will have enough energy that it can hop from one peak to another.

As the temperature continues to increase, it eventually swamps out the entire structure of the probability function. The acceptance probability approaches one, and the sampled distribution will be uniform (well, a random walk anyway).

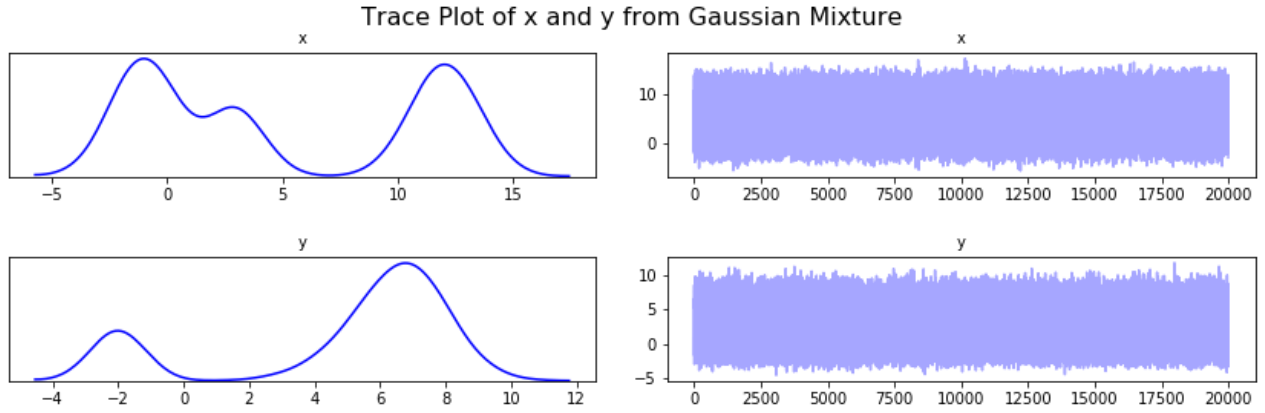
In the limit where T approaches 0, the sampler will *only* accept steps that improve the probability. Once it finds a local maximumm it will never leave it. In the limit where T approaches $\infty$, the sampler will accept all steps. The resulting set of samples will reflect a random walk. This will be a normal distribution with standard deviation proportional to the square root of the number of steps taken times the dimension. It will be a very dispersed distribution that locally looks like a uniform distribution with infinitesimally low height.

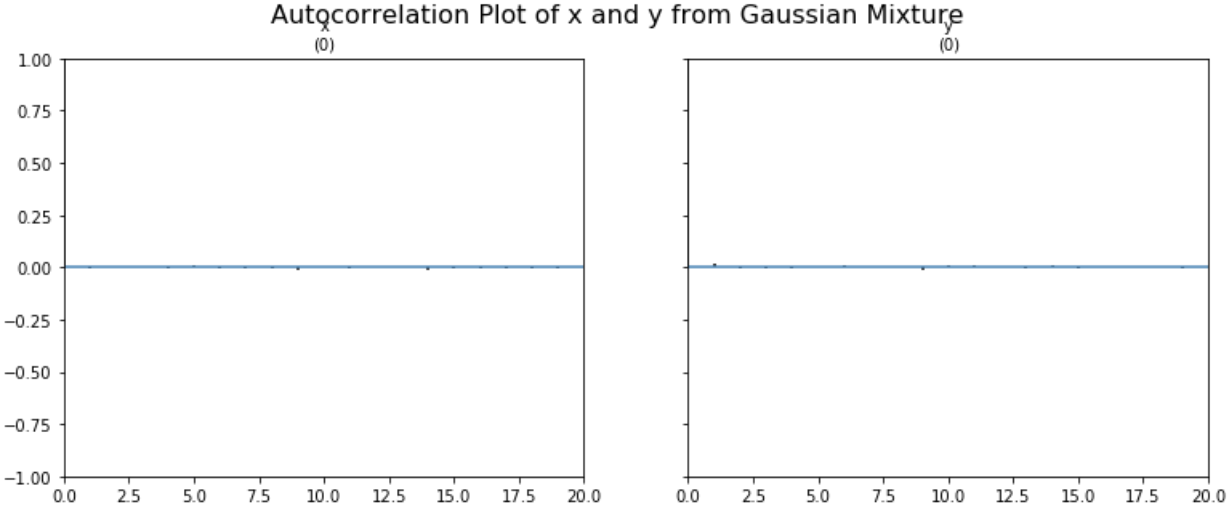
**B3**. Approximate the $f(X)$ by the appropriate mixture of Gaussians as a way of generating samples from $f(X)$ to compare with other sampling methods. Use scipy.stats.multivariate_normal to generate 20000 samples.

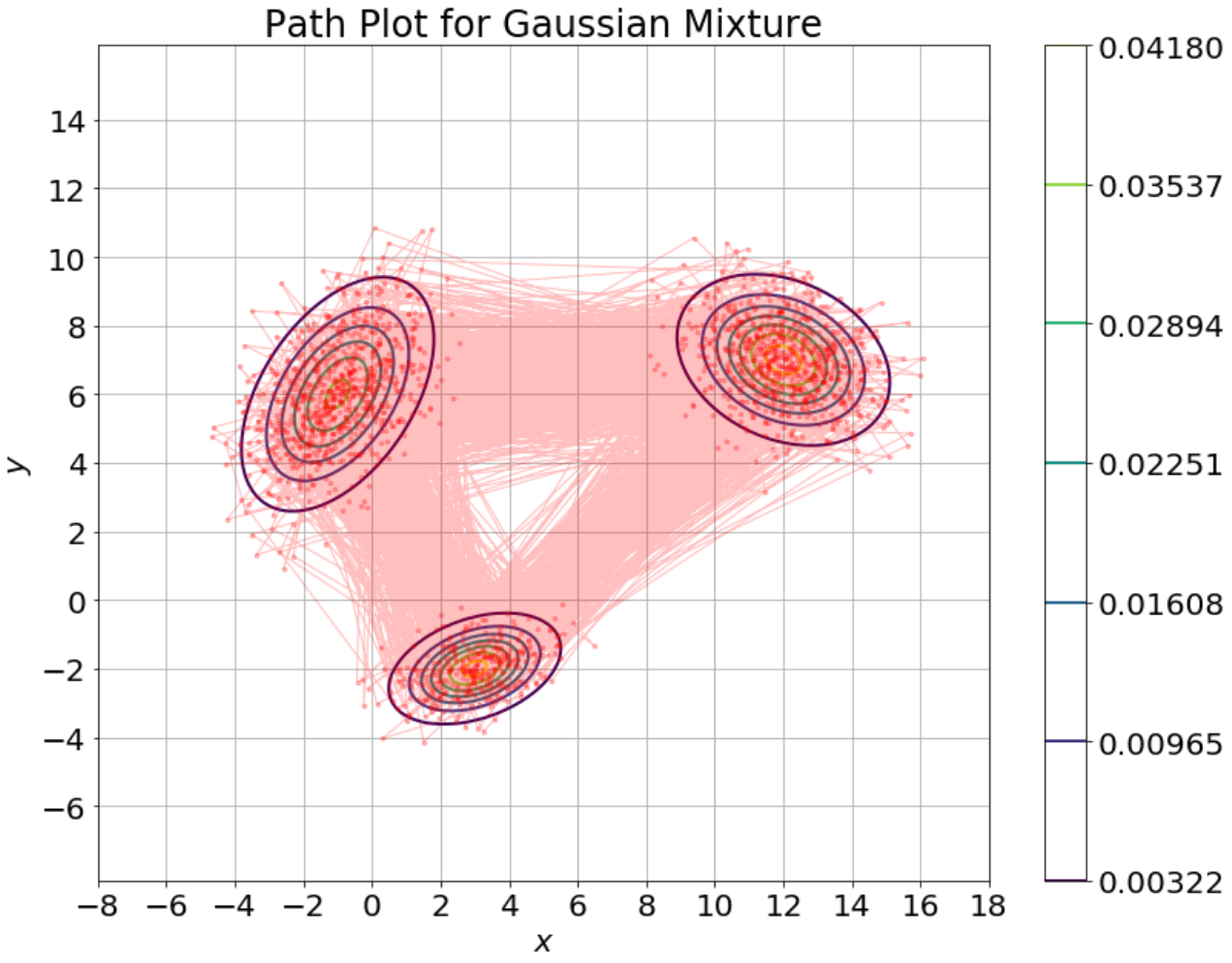
In [210]:

```
# Draw a matrix of samples from all three Guassians
samples_1 = multivariate_normal(mean=mu_1, cov=cov_1).rvs(num_samples)
samples_2 = multivariate_normal(mean=mu_2, cov=cov_2).rvs(num_samples)
samples_3 = multivariate_normal(mean=mu_3, cov=cov_3).rvs(num_samples)
# Stack the candidate samples;
samples_cand = np.stack([samples_1, samples_2, samples_3])

# Sample ancestrally (first pick a Guassian, then use that column)
cluster = np.random.choice(3, size=num_samples, p=weights)
samples_gm = samples_cand[cluster,range(num_samples),:]
```

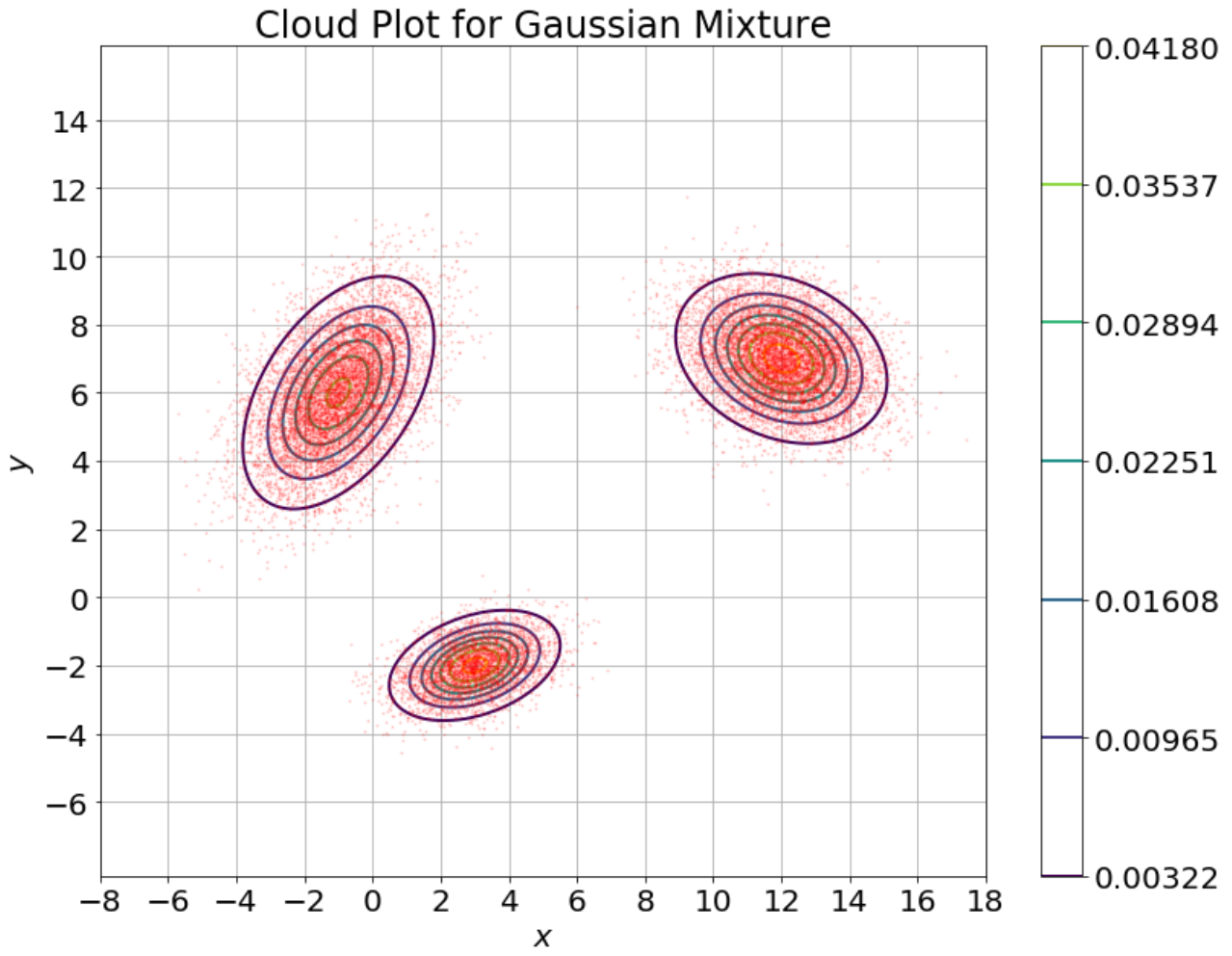
In [229]:
```
def plots_all(samples, model_name):
    """Generate all three plots of interest from the Gaussian Mixture"""
    # Get the trace for this temperature
    trace = {'x': samples[:,0],
             'y': samples[:,1]}
    # Generate the traceplot
    fig1 = plot_trace(trace, f'Trace Plot of x and y from {model_name}')
    # Generate the autocorr plot
    fig2 = plot_autocorr(trace, f'Autocorrelation Plot of x and y from {model_name}')
    # Plot the path taken by the sampler
    fig3 = plot_path(samples, x_grid, y_grid, p_grid,
                     f'Path Plot for {model_name}')
    return [fig1, fig2, fig3]

# Generate all plots for the Gaussian Mixture sampling
figs = plots_all(samples_gm, 'Gaussian Mixture')
```




Path Plot for Gaussian Mixture
14
12
10
8
6
4
2
0
−2
−4
−6
y
−8 −6 −4 −2 0 2 4 6 8 10 12 14 16 18
x
0.04180
0.03537
0.02894
0.02251
0.01608
0.00965
0.00322

In [216]:
```
# Bonus plot - cloud plot for GM
fig = plot_cloud(samples_gm, x_grid, y_grid, p_grid, f'Cloud Plot for Gaussian Mixture')
```




**How do the histograms compare with the histograms for the samples from $f(X)$ at each temperature.**

First off, let's say that these samples are far better. They're perfect because they were specified exactly in line with known probability distribution. There are no artifacts due to path based sampling with steps. the traceplots show perfect white noise with no autocorrelation. The marginal probabilities correctly match the marginals with three visible peaks along the x-axis and 2 along the y-axis. As an aside, the last plot above is a cloud not a path, because there are so many hops between clusters that it overwhelmed the plot a bit.

Comparing these "perfect" traceplots to the ones at each temperature:

- At T=0.1, the sampler spends all its time in one cluster, so the marginals are close to normal. Autocorrelation is high. The traces have noise within the normal where they reside. Not a good match for the true traceplot.
- At T=1.0, the sampler at least touches 2 of the three clusters. The marginal plot of x still looks unimodal, thought th plot of y is starting to show a little bump at low values corresponding to the cluster centered at (3,-2). The autocorrelation is still high and the traceplots are similar to above. Slight improvement from T=0.1, but still not a good match.
- At T=3.0, the sampler has enough energy (temperature) to visit all three clusters. The shape of the x and y marginals both show 2 peaks, matching the data better. The traces reveal that the sampler is spending time in one cluster, then hopping to another one, a pattern that wasn't apparent at T=1. The autocorrelation is very high.
- At T=10.0, the sampler has so much energy that it's able to move around well, but it's starting to flatten out and distort the shape of the distribution. Both marginals are bimodal, but the peaks are smooshed together (flattened) noticeably. The autocorrelation is again high.

**At what temperature do the samples best represent the function?** An unscientific visual inspection suggests that temperature **T=3.0** does the best job. This is the lowest temperature where the sampler visits all the clusters. Once the temperature is high enough not to miss clusters, continuing to increase it further just distorts the resulting samples for no real gain.

# Part C: Parallel Tempering

Now that we've seen some of the pproperties of sampling at higher temperatures, let's explore a way to incorporate the improved exploration of the *entire pdf* from sampling at higher temperatures while still getting samples that match our distribution. We'll use a technique called *parallel tempering*.

The general idea of parallel tempering is to simulate $N$ replicas of the original system of interest (in our case, a single Metropolis Hastings chain), each replica at a different temperature. The temperature of a Metropolis Hastings Markov Chain defines how likely it is to sample from a low-density part of the target distribution. The high temperature systems are generally able to sample large volumes of parameter space, whereas low temperature systems, while having precise sampling in a local region of parameter space, may become trapped around local energy minima/probability maxima. Parallel tempering achieves good sampling by allowing the chains at different temperatures to exchange complete configurations. Thus, the inclusion of higher temperature chains ensures that the lower temperature chains can access *all* the low-temperature regions of phase space: the higher temperatures help these chains make the jump-over.

Darren Wilkinson's blog post has a good description (https://darrenjw.wordpress.com/2013/09/29/parallel-tempering-and-metropolis-coupled-mcmc/) of whats going on.

Here is the idea that you must implement.

There are $N$ replicas each at different temperatures $T_i$ that produce $n$ samples each before possibly swapping states.

We simplify matters by only swapping states at adjacent temperatures. The probability of swapping any two instances of the replicas is given by

$$A = min\left(1, \frac{p_k(x_{k+1})p_{k+1}(x_k)}{p_k(x_k)p_{k+1}(x_{k+1})}\right)$$

One of the $T_i$'s in our set will always be 1 and this is the only replica that we use as output of the Parallel tempering algorithm.

An algorithm for Parallel Tempering is as follows:

1. Initialize the parameters $\{(x_{init}, y_{init})_i\}, \{T_i\}, L$ where
    - $L$ is the number of iterations between temperature swap proposals.
    - $\{T_i\}$ is a list of temperatures. You'll run one chain at each temperature.
    - $\{(x_{init}, y_{init})_i\}$ is a list of starting points, one for each chain
2. For each chain (one per temperature) use the simple Metropolis code you wrote earlier. Perform $L$ transitions on each chain.
3. Set the $\{(x_{init}, y_{init})_i\}$ for the next Metropolis run on each chain to the last sample for each chain i.
4. Randomly choose 2 chains at adjacent temperatures.
    A. Use the above formula to calculate the Acceptance probability $A$.
    B. With probability $A$, swap the positions between the 2 chains (that is swap the $x$s of the two chains, and separately swap the $y$s of the chains .
5. Go back to 2 above, and start the next L-step epoch
6. Continue until you finish $Num.Samples//L$ epochs.

**C1**. Explain why swapping states with the given acceptance probability is in keeping with detailed balance. The linked blog post might help.

**C2**. Create a parallel tempering sampler that uses 5 chains at the temperatures {0.1, 1, 3, 7, 10} to sample from $f(x, y)$. Choose a value of L around 10-20. Generate 10000 samples from $f(x, y)$. Construct histograms of the marginals, traceplots, autocorrelation plots, and a pathplot for your samples.

**C3**. How do your samples in **C2** compare to those of the Metropolis sampler? How do they compare to the samples generated from the Gaussian Mixture approximation of $f(x, y)$?

**C1**. Explain why swapping states with the given acceptance probability is in keeping with detailed balance. The linked blog post might help.

As we saw in lecture 15 (p. 37-38), detail balance is implied by a symmetrical proposal distribution. So it suffices to demonstrate that the expression A above is symmetrical, i.e. that exchanging $k$ and $k+1$ will not change it. Define

$$B(i,j) = \frac{p_i(x_j)p_j(x_i)}{p_i(x_i)p_j(x_j)}$$

since the min part of $A$ doesn't depend on $k$ and $k+1$, it suffices to show that $B$ is symmetric, i.e. that $B(i,j) = B(j,i)$.
To lighten the notation, let the energy $E_k = -\log(p(x_k))$. By the definition of the temperature induced probability distributions,

$$log(p_i(x_k)) = log(p(x_k)/T_i = -E_k/T_i$$

. Using this expression, we can simplify the logs of both the numerator and denominator of $B$:

$$\log(\text{num}) = log(p_i(x_j)p_j(x_i)) = -\left(\frac{E_j}{T_i} + \frac{E_i}{T_j}\right) = -\left(\frac{T_jE_j + T_iE_i}{T_iT_j}\right)$$

$$\log(\text{den}) = log(p_i(x_i)p_j(x_j)) = -\left(\frac{E_i}{T_i} + \frac{E_j}{T_j}\right) = -\left(\frac{T_jE_i + T_iE_j}{T_iT_j}\right)$$

$$\log(B) = \log(\text{num}) - \log(\text{den}) = \frac{(T_jE_i + T_iE_j) - (T_jE_j + E_iE_i)}{T_iT_j}$$

$$\log(B) = \frac{(T_j - T_i)(E_i - E_j)}{T_iT_j}$$

We can see immediately that this is a symmetric expression in i and j. Switching i and j will leave the denominator unchanged, and by flipping the signs of both differences that are multiplied in the numerator, that won't change either. So this transition probability is symmetric and maintains detail balance. Limiting transitions to adjacent entries still leaves with a symmetric matrix; the dense symmetric transition matrix becomes a symmetric tridiagonal matrix.

**C2**. Create a parallel tempering sampler that uses 5 chains at the temperatures {0.1, 1, 3, 7, 10} to sample from $f(x,y)$. Choose a value of L around 10-20. Generate 10000 samples from $f(x,y)$. Construct histograms of the marginals, traceplots, autocorrelation plots, and a pathplot for your samples.

**Helper functions for parallel tempering algorithm**

In [223]:
```
def metropolis_temp(pdf: Callable, T: float, num_samples: int, X_init: np.ndarray):
    """Modified metropolis sampler that accept a pdf and temperature"""
    # create p(X;T) in situ by binding T
    logp_T = lambda X : log(pdf(X)) / T
    # Draw samples
    samples, accepted = metropolis(logp_T, proposal, step_size, num_samples, X_init)
    # Only return the samples; discard number of accepted points
    return samples


def trans_prob(pdf, Xi: np.ndarray, Xj: np.ndarray, Ti: float, Tj: float):
    """
    Compute the transition probability A(i, j) between two states
    pdf: probability density function taking vectorized input
    Xi: the first point, corresponding to sampler with temperature Ti
    Xj: the second point, corresponding to sampler with temperature Tj
    Ti:  temperature of the first point, Ti
    Tj:  temperature of the second pont, Tj
    """
    # Evaluate the basic probability (without temperature) at both points
    pi = pdf(Xi)
    pj = pdf(Xj)
    # Compute the four probabilities appearing in the formula
    pi_xi = np.power(pi, 1.0 / Ti)
    pi_xj = np.power(pi, 1.0 / Tj)
    pj_xi = np.power(pj, 1.0 / Ti)
    pj_xj = np.power(pj, 1.0 / Tj)
    # Apply the formula for A
    return min(1.0, (pi_xj * pj_xi) / (pi_xi * pj_xj))


def swap_inits(X_inits: List[np.ndarray], i: int, j: int):
    """Swap the states (initializers) for two chains"""
    # Copy the starting states
    init_i, init_j = X_inits[i].copy(), X_inits[j].copy()
    # Swap them
    X_inits[i] = init_j
    X_inits[j] = init_i
```

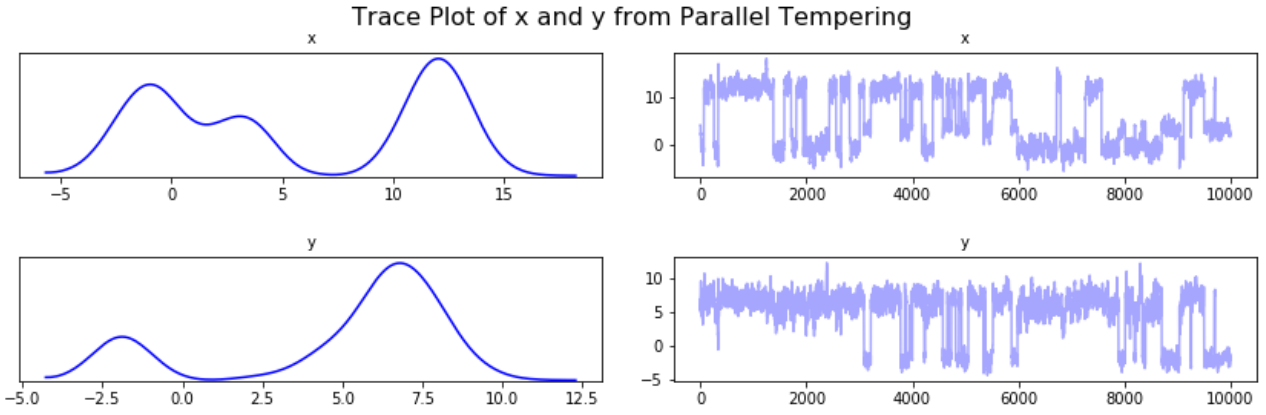
**Parallel Tempering**

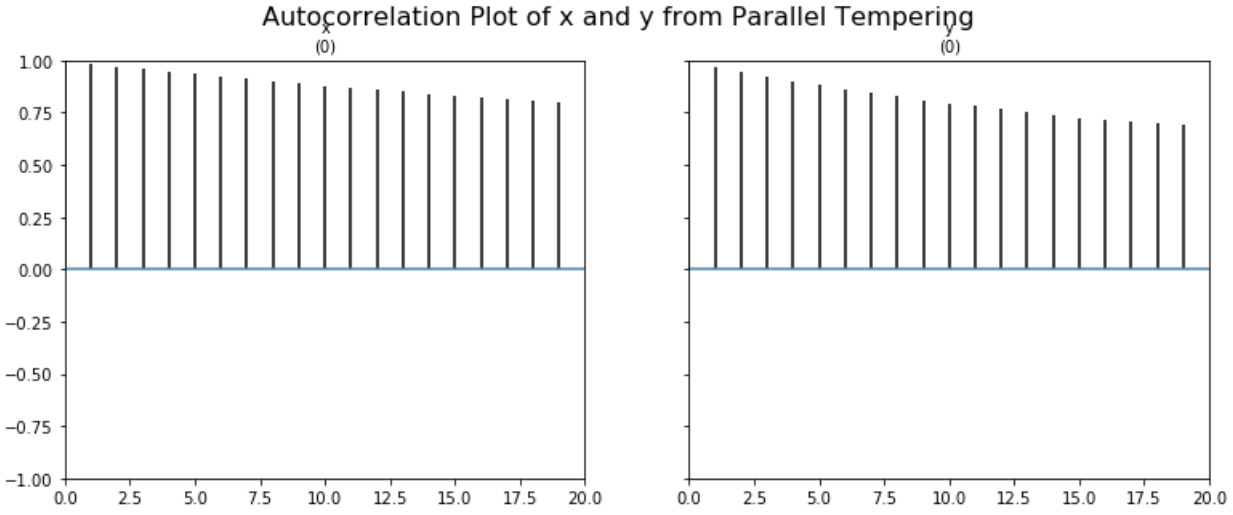
In [224]:
```python
def parallel_temper(pdf: Callable, temps: np.ndarray, X_init, L: int, epochs: int):
    """Parallel tempering algorithm"""
    # Get the number of parallel chains
    num_chains: int = len(temps)
    # Shape of data
    K: int = X_init.shape[0]
    # Compute the total number of samples and preallocate space
    # Will only save the results from the chain with temperature=1
    num_samples: int = L * epochs
    samples = np.zeros((num_samples, K))
    # Get index of the chain with temperature T=1
    idx_t0: int = np.searchsorted(temps, 1.0)
    assert temps[idx_t0] == 1.0, 'Error: one of the temps must be 1.0!'

    # Bind arguments with the PDF and L to metropolis_temp
    make_chain = lambda X_start, T : metropolis_temp(pdf, T, L, X_start)
    # Save range of i's for legibility
    ii = range(num_chains)
    # Initialize X_inits
    X_inits = [X_init.copy() for i in ii]
    # initialize over epochs
    for epoch in tqdm.tqdm(range(epochs)):
        # Run new chains
        chains = [make_chain(X_inits[i], temps[i]) for i in ii]
        # Copy output from chain at temp 0 to samples
        i0: int = L*(epoch+0)
        i1: int = L*(epoch+1)
        samples[i0:i1] = chains[idx_t0]

        # Copy current states of each chain to X_starts
        X_inits = [chain[-1] for chain in chains]
        # Randomly choose a pair of adjacent chains; equivalent to picking k in [0, num_chains-1)
        k = np.random.randint(num_chains-1)
        # Extract Xi, Xj, Ti, Tj for transition probability
        Xi = X_inits[k]
        Xj = X_inits[k+1]
        Ti = temps[k]
        Tj = temps[k+1]
        # Compute the transition probability
        tp: float = trans_prob(pdf, Xi, Xj, Ti, Tj)
        # Draw a random uniform to determine whether there is a transition
        if np.random.random() < tp:
            swap_inits(X_inits, k, k+1)

    # Return only the "good" samples from the chain with temp-1
    return samples
```

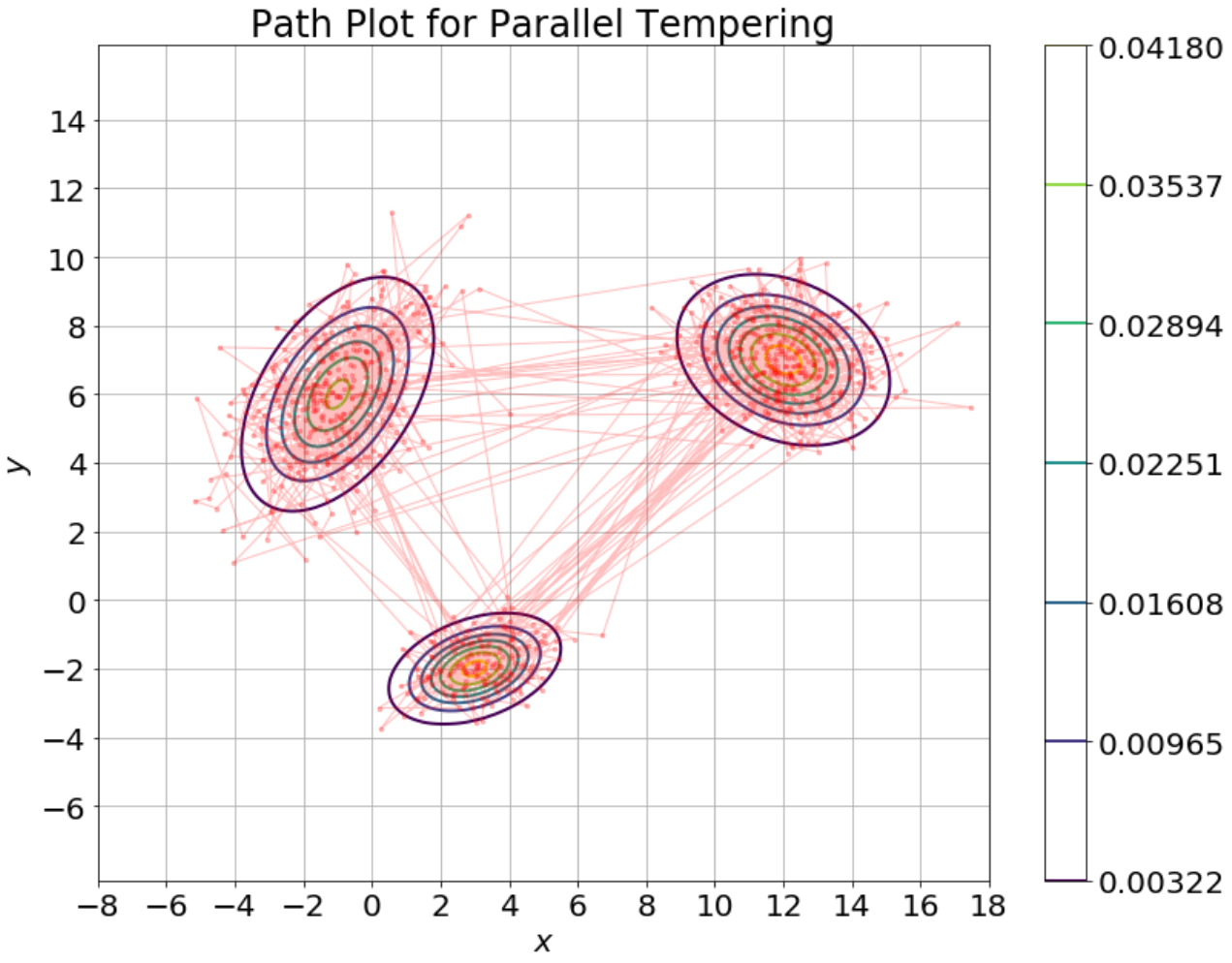
In [228]:
```python
# Alias the vectorized probability desnity function for legibility
pdf = f
# Set parameters for paralllel tempering
temps_pt = np.array([0.1, 1.0, 3.0, 7.0, 10.0])
num_samples_pt: int = 10000
L: int = 16
epochs = int(np.ceil(num_samples_pt / L))
# Run parallel tempering
try:
    samples_pt = vartbl['samples_pt']
    print(f'Loaded {len(samples_pt)} samples from parallel tempering.')
except:
    print(f'Generating {num_samples_pt} samples with parallel tempering...')
    samples_pt = parallel_temper(pdf, temps_pt, X_init, L, epochs)
    vartbl['samples_pt'] = samples_pt
    save_vartbl(vartbl, fname)
```

```
Generating 10000 samples with parallel tempering...

100%|██████████████████████████████████████████████████████████████████████████| 625/625 [00:07<0
0:00, 87.19it/s]
```

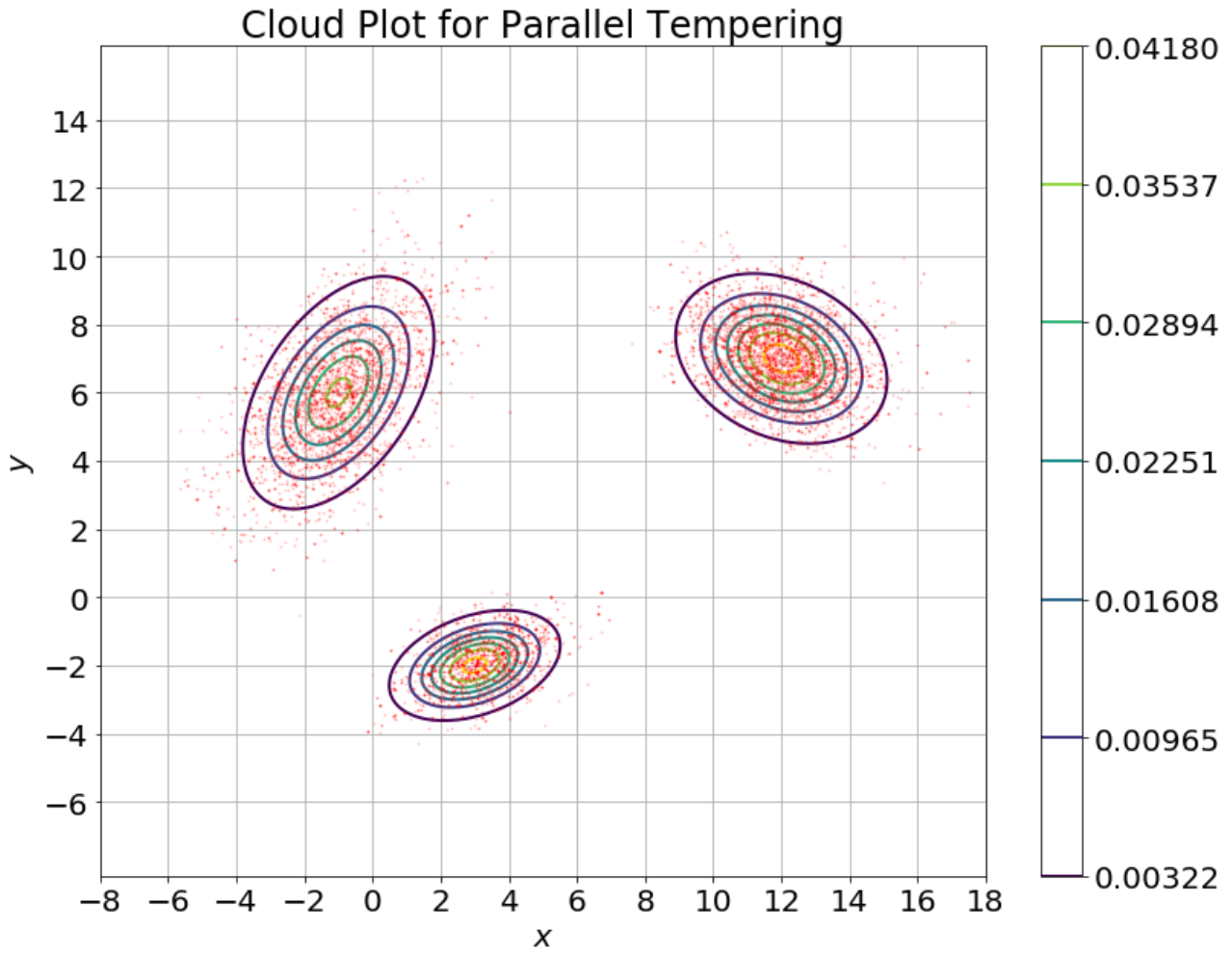
In [230]:
```
# Generate all plots for the Parallel Tempering sampling
figs = plots_all(samples_pt, 'Parallel Tempering')
```




Path Plot for Parallel Tempering
14
12
10
8
6
4
2
0
−2
−4
−6
y
−8 −6 −4 −2 0 2 4 6 8 10 12 14 16 18
x
0.04180
0.03537
0.02894
0.02251
0.01608
0.00965
0.00322

In [232]:
```
# Bonus plot - cloud plot for PT
fig = plot_cloud(samples_pt, x_grid, y_grid, p_grid, f'Cloud Plot for Parallel Tempering')
```




**C3**. How do your samples in **C2** compare to those of the Metropolis sampler?
How do they compare to the samples generated from the Gaussian Mixture approximation of $f(x, y)$?

These samples are **way** better than the ones from the Metropolis sampler!. They visit all the clusters and have the right marginal (and joint!) distributions. Admittedly the traceplots still don't look much like white noise, and there is a lot of autocorrelation.

I cannot see any major difference in the pattern of the marginal and joint probabilities between the Parallel Tempering and Gaussian mixture models. This to me is strong evidence that it has "worked". The one deficiency of the PT output is revealed in the traceplots and autocorrelation: there is clearly quite a bit of path dependence, and the order of the samples drawn is not at all random. But if you run this decently long and then randomly reorder the results, you should have an excellent sample from the misture distribution. Cool!

If you have ever competed in the Boston Marathon, you should recognize this image: it's the intersection of Hereford and Boylston street. A few blocks back you needed to take a "right on Hereford, left on Boylston" and you will reach the finish line!

# Part D. Global Optima using Simulated Annealing

We have new-found intuition about how to use temperature to improve our sampling. Lets now tackle the inverse idea: what happens if you sample at a lower temperature than 1. Our visualizations from Part B should indicate to us that the distributions become extremely tightly peaked arounnd their maxima.

If we initialized a metropolis-hastings sampler around an optimum at a really low temperature, it would find us a local minimum. But if we had a higher temperature at the beginning, we can use Metropolis-Hastings sampling at high temperatures to travel around the distribution and find all the peaks (valleys). Then we will slowly cool down the temperature (which will allow us to escape local optima at higher temperatures) and finally focus us into a particular optimum region and allow you to find the optimum. It can be shown that for ceratin *temperture schedules* this method is guaranteed to find us a global minimum in the limit of infinite iterations.

We'll use this methd to find the global minimum of our distribution. The algorithm is as follows. Now we have only one chain, but we very slowly dial down its temperature to below T=1.

1. Initialize $(x, y)_i, T, L(T)$ where $L$ is the number of iterations at a particular temperature.
2. Perform $L$ transitions thus(we will call this an epoch):
   A. Generate a new proposed position $(x, y)_*$ using 2 independent gaussians with $\sigma = 1$.
   B. If $(x, y)_*$ is accepted (according to probability $P = e^{(-\Delta E/T)}$, set $(x, y)_{i+1} = (x, y)_*$, else set $(x, y)_{i+1} = x_i$
3. Update T and L
4. Until some fixed number of epochs, or until some stop criterion is fulfilled, goto 2.

$\Delta E$ is the change in enery, or the change in the negative log of the probability function. That is, $E = -logp(x, y)$. For a given T and L, this is just Metropolis!

This algorithm is called *simulated annealing* and we'll use it to find the global maximum for $f(X)$

**D1**. Use simulated annealing with a cooling schedule of $T_{k+1} = 0.98T_k$ and a L(T) defined initially at 100 with $L_{k+1} = 1.2L_k$ to find the global optima for $p(x, y)$. Plot $E(x, y)$ vs iterations. Given how we constructed $p(x, y)$ it should be fairly straight-forward to observe the true optima by inspection. How does the optima found by SA compare to the true optima?

In [233]:

```python
def E(X: np.ndarray):
    """Energy function for this problem"""
    return -logp(X)


def anneal(f: Callable, X_init, T0 : float, L0: int, num_iters: int):
    """
    Simulated annealing algorithm:
    f:      function to optimize
    X_init: starting point
    T0:     initial temperature
    L0:     initial chain size
    epochs: number of epochs to run
    """
    # shape of data
    K: int = X_init.shape[0]
    # growth factor in iterations per epoch
    gL = 1.20
    # growth factor in termperature per epoch
    gT = 0.98
    # compute number of iterations in closed form using geometric series
    epochs = int(np.ceil(log(1.0 + (num_iters / L0) * (gL-1.0)) / log(gL)))
    # Length of each epoch
    epoch_lens = np.array([np.round(L0*(gL**k)) for k in range(epochs)], dtype=np.int32)
    # Cumulative iteration counter
    iter_counter = np.minimum(np.concatenate([np.zeros(1, dtype=np.int32), np.cumsum(epoch_lens)]), num_
    # Adjust last epoch length to reflect truncation
    epoch_lens[-1] = iter_counter[-1] - iter_counter[-2]
    # Check that epoch_lens calculation was done correctly
    assert iter_counter[-2] < num_iters and num_iters == iter_counter[-1]
    # Generate temperature schedule
    temps = np.array([T0 * (gT**k) for k in range(epochs)])
    # Bind arguments with the PDF and L to metropolis_temp
    make_segment = lambda X_init, T, L : metropolis_temp(f, T, L, X_init)

    # Initialize path and energy arrays
    path = np.zeros((num_iters,K))
    energy = np.zeros(num_iters)

    # iterate over epochs
    for k in range(epochs):
        # Get indices for this chain in the big arrays
        i0: int = iter_counter[k+0]
        i1: int = iter_counter[k+1]
        # Temperature and number of iterations for this epoch
        T = temps[k]
        L = epoch_lens[k]
        # Generate one segment on the path; same as a chain in sampling context
        segment = make_segment(X_init, T, L)
        # Add this segment to the full path
        path[i0:i1] = segment
        # Save energies along this segment
        energy[i0:i1] = E(segment)
        # Set X_init for the next segment at the last value on this segment
        X_init = segment[-1]

    # Return both the samples and the energy function
    return path, energy
```

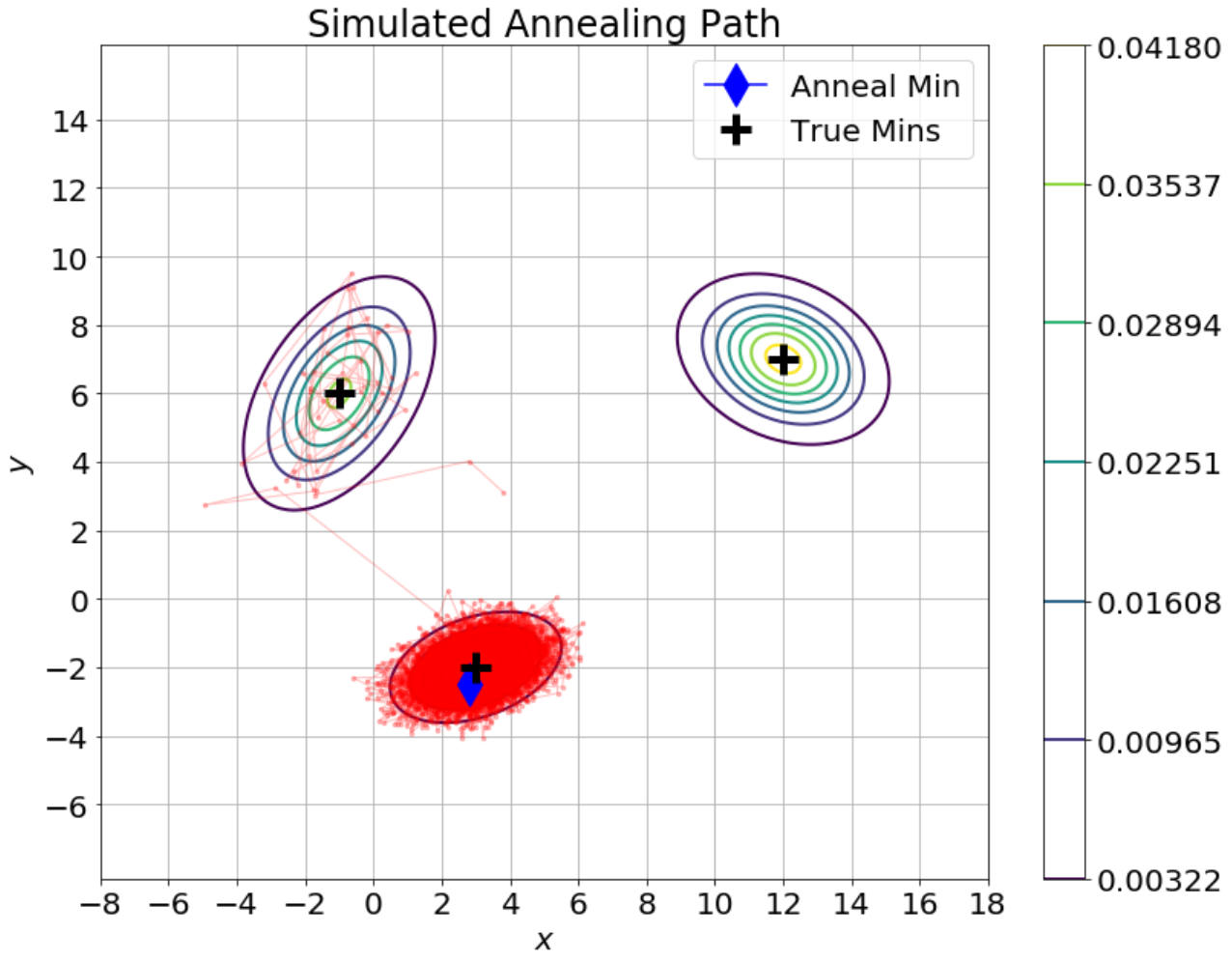
In [236]:
```
# Set parameters for simulated annealing
T0: float = 1.0
L0: int = 100
num_iters_anneal: int = 200000

# Find the global minimum with simulated annealing
try:
    # raise ValueError
    anneal_path = vartbl['anneal_path']
    anneal_energy = vartbl['anneal_energy']
    print(f'Loaded path and energy for simulated annealing.')
except:
    # Run simulated annealing for 100000 iterations
    print(f'Running simulated annealing for {num_iters_anneal} iterations...')
    anneal_path, anneal_energy = anneal(f, X_init, T0, L0, num_iters_anneal)
    vartbl['anneal_path'] = anneal_path
    vartbl['anneal_energy'] = anneal_energy
    save_vartbl(vartbl, fname)

# Compare to true local minimum
X_min_true = mu_1
E_min_true = E(X_min_true)
```

```
Running simulated annealing for 200000 iterations...

100%|████████████████████████████████████████████████████████████| 1070/1070 [00:00<00:
00, 5720.33it/s]
100%|████████████████████████████████████████████████████████████| 1284/1284 [00:00<00:
00, 5908.30it/s]
100%|████████████████████████████████████████████████████████████| 1541/1541 [00:00<00:
00, 5902.67it/s]
100%|████████████████████████████████████████████████████████████| 1849/1849 [00:00<00:
00, 6182.34it/s]
100%|████████████████████████████████████████████████████████████| 2219/2219 [00:00<00:
00, 6078.01it/s]
100%|████████████████████████████████████████████████████████████| 2662/2662 [00:00<00:
00, 6203.61it/s]
100%|████████████████████████████████████████████████████████████| 3195/3195 [00:00<00:
00, 6479.27it/s]
100%|████████████████████████████████████████████████████████████| 3834/3834 [00:00<00:
00, 6294.21it/s]
100%|████████████████████████████████████████████████████████████| 4601/4601 [00:00<00:
00, 6353.57it/s]
100%|████████████████████████████████████████████████████████████| 5521/5521 [00:00<00:
00, 6579.09it/s]
100%|████████████████████████████████████████████████████████████| 6625/6625 [00:01<00:
00, 6597.12it/s]
100%|████████████████████████████████████████████████████████████| 7950/7950 [00:01<00:
00, 6656.83it/s]
100%|████████████████████████████████████████████████████████████| 9540/9540 [00:01<00:
00, 6637.37it/s]
100%|███████████████████████████████████████████████████████████| 11448/11448 [00:01<00:
00, 6788.48it/s]
100%|███████████████████████████████████████████████████████████| 13737/13737 [00:02<00:
00, 6778.85it/s]
100%|███████████████████████████████████████████████████████████| 16484/16484 [00:02<00:
00, 6734.93it/s]
100%|███████████████████████████████████████████████████████████| 19781/19781 [00:02<00:
00, 6983.23it/s]
100%|███████████████████████████████████████████████████████████| 23738/23738 [00:03<00:
00, 6980.18it/s]
100%|███████████████████████████████████████████████████████████| 28485/28485 [00:04<00:
00, 6934.16it/s]
100%|███████████████████████████████████████████████████████████| 29586/29586 [00:04<00:
00, 7053.98it/s]
```

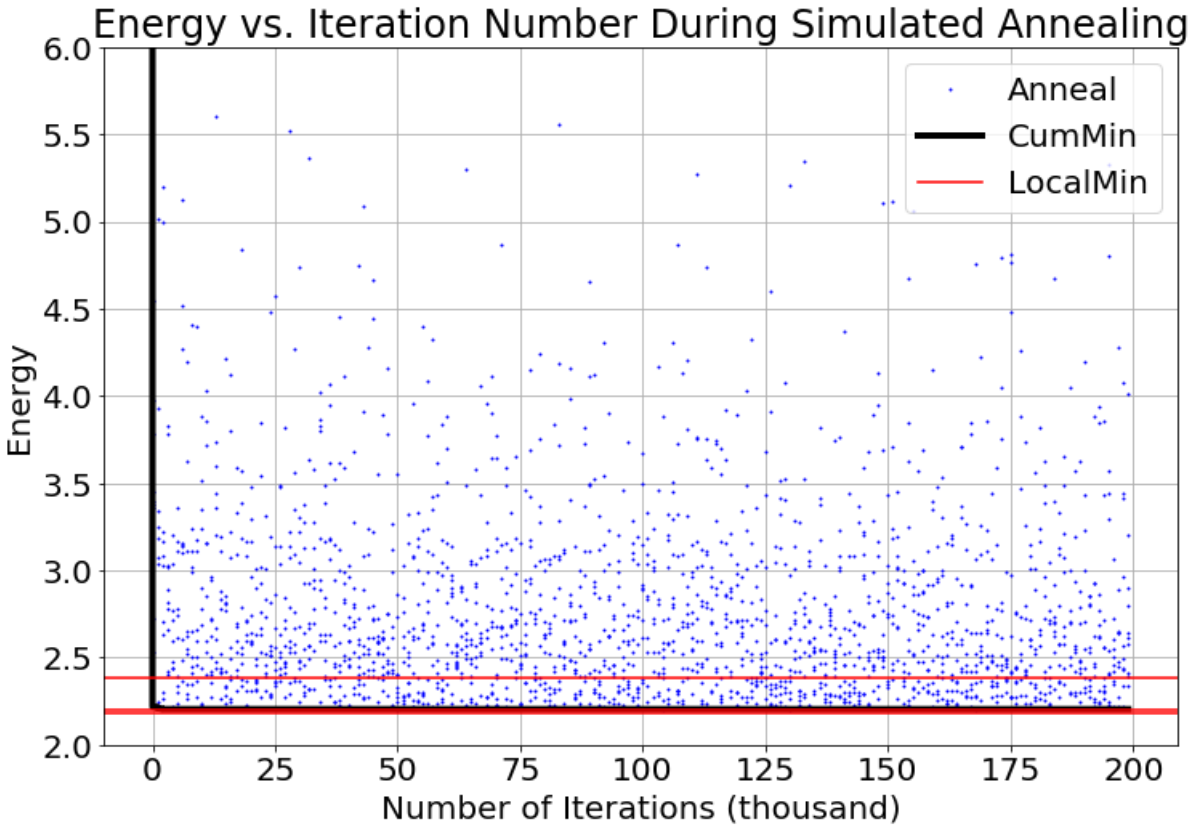
**Plot $E(x, y)$ vs iterations**

In [237]:
```
def anneal_plot(path, energy):
    """Plot the path and report results of simulated annealing process"""
    # Report the local minimum found
    X_min_anneal = path[-1]
    E_min_anneal = energy[-1]
    print(f'Simulated Annealing found local min at ({X_min_anneal[0]:0.4f}, {X_min_anneal[1]:0.4f}) '
          f'with energy {E_min_anneal:0.4f}')

    print(f'True local minimum is at ({mu_1[0]:0.4f}, {mu_1[1]:0.4f}) '
          f'with energy {E(mu_1):0.4f}')

    # Plot the path of simulated annealing
    fig = plot_path(path[::], x_grid, y_grid, p_grid, 'Simulated Annealing Path')
    ax = fig.axes[0]
    ax.plot(X_min_anneal[0], X_min_anneal[1], label='Anneal Min', marker='d', color='b', markersize=20)
    ax.plot(mus[:,0], mus[:,1], label='True Mins', linewidth=0, marker='+', color='k',
            markersize=20, markeredgewidth=5)
    ax.legend()
    return fig

# Plot the annealing path and report the results
fig = anneal_plot(anneal_path, anneal_energy)
```

```
Simulated Annealing found local min at (2.8063, -2.5010) with energy 2.4595
True local minimum is at (12.0000, 7.0000) with energy 2.1845
```



In [238]:
```
def plot_energy(anneal_energy):
    """Plot the energy during the annealing process"""
    # Cumulative minimum energy from annealing
    cum_min = np.minimum.accumulate(anneal_energy)
    plot_step: int = 100
    xx = np.arange(0,num_iters_anneal,plot_step) // 1000
    # Plot the energy
    fig, ax = plt.subplots(figsize=[12,8])
    ax.set_title('Energy vs. Iteration Number During Simulated Annealing')
    ax.set_xlabel('Number of Iterations (thousand)')
    ax.set_ylabel('Energy')
    ax.set_ylim(2,6)
    h1 = ax.plot(xx, anneal_energy[::plot_step], label='Anneal', color='b', linewidth=0, marker='.', marl
    h2 = ax.plot(xx, cum_min[::plot_step], label='CumMin', color='k', linewidth=4)
    h3 = ax.axhline(E(mu_1), label='LocalMin_1', color='r')
    ax.axhline(E(mu_2), label='LocalMin_2', color='r')
    ax.axhline(E(mu_3), label='LocalMin_3', color='r')
    ax.legend(handles=[h1[0], h2[0], h3], labels=['Anneal', 'CumMin', 'LocalMin'], loc='upper right')
    ax.grid()
    return fig

# Plot the annealing energy
fig = plot_energy(anneal_energy)
```




**Given how we constructed $p(x, y)$ it should be fairly straight-forward to observe the true optima by inspection. How does the optima found by SA compare to the true optima?**

This function has three local minima that are plotted above. They are at the centers of the three gaussians that are in the mixture. Of these, the point corresponding to $\mu_1$ at [12, 7] has the lowest energy, $E_1 = 2.18445$.

The path taken by the simulated annealing varies across trials due to randomness. With reasonable parameter settings as above it always appears to end up close to one of the local minima. It hit all three during just 5 trials, so they all seem to have a decent probability. The **cumulative minimum** found by this process is very close to the local minimum it's exploring. The last point visited is still not that close the local minimum. If we cooled the temperature further it could get closer.


**We have reached the finish line of Applied Math 207 2018!**

Dear teaching staff, it's been an arduous and enlightening journey for all of us. Thank you for all your hard work in grading these very long problem sets and best of luck in your future endeavors,
Sincerely,
Your friends on Team Braavos