Final Exam Submission

Spencer Hallyburton, Salvador Barragan

APMTH 207: Advanced Scientific Computing:

Stochastic Methods for Data Analysis, Inference and Optimization

Final ¶

Harvard University
Spring 2018
Instructors: Rahul Dave
Due Date: Saturday, May 12th, 2018 at 11:59am

Q1: GLMs with correlation

In Homework 10 you examined the effects of educational attainment and gender on the probability that an indivudal would earn more than $50K per year. However, even the models in the solutions (see Canvas) didn't achieve a perfect fit. Perhaps there was something missing?

The dataset: A Study of Census Data

In this problem, we are reprising our investigation into the data from the 1994 U.S. Census. The data has been processed so that only a subset of the features are present (for full dataset as well as the description see the UCI Machine Learning Repository).

We'll again want to aggregate the dataset into seven different categories. The categories we wish to consider are:

- 4 year college degree
- Some-college or two year academic college degree
- High school
- Professional, vocational school
- Masters
- Doctorate
- Some or no high school

Note that we had to combine some of the existing education categories in your dataframe. For each category, we kept track of a count of the number of males and females who make above (and resp. below) $50k

We provide the code below so that all students begin with the same data structure, names, and global parameters.

In [1]:
```
import pymc3 as pm
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from theano import tensor as tt
```

In [2]:
```
df = pd.read_csv('census_data.csv')
df.head()

cat_1 = ['11th', '7th-8th', '9th', '5th-6th', '10th', '1st-4th', 'Preschool', '12th']
cat_2 = ['HS-grad']
cat_3 = ['Some-college', 'Assoc-acdm']
cat_4 = ['Assoc-voc', 'Prof-school']
cat_5 = ['Bachelors']
cat_6 = ['Masters']
cat_7 = ['Doctorate']

labels = ["Some HS", "HS Grad", "Some College", "Prof School", "Bachelors", "Masters", "Doctor
ate"]

categories = [cat_1, cat_2, cat_3, cat_4, cat_5, cat_6, cat_7]
dataset = []
i = 1
for cat in categories:
    filtered = df[df['edu'].isin(cat)]
    dataset.append((i, 0, len(filtered[filtered['sex'] == 'Female']), len(filtered[(filtered[
'sex'] == 'Female') & (filtered['earning'] == '<=50K')]), len(filtered[(filtered['sex'] == 'Fe
male') & (filtered['earning'] == '>50K')])))
    dataset.append((i, 1, len(filtered[filtered['sex'] == 'Male']), len(filtered[(filtered['se
x'] == 'Male') & (filtered['earning'] == '<=50K')]), len(filtered[(filtered['sex'] == 'Male')
& (filtered['earning'] == '>50K')])))
    i += 1
dataset = np.array(dataset)
dataset = pd.DataFrame(dataset, columns=['edu', 'sex', 'total', '<=50', '>50'])
dataset['edu'] = dataset['edu']-1 #zero indexing
dataset
```

Out[2]:

	edu	sex	total	<=50	>50
0	0	0	1321	1298	23
1	0	1	2932	2711	221
2	1	0	3390	3164	226
3	1	1	7111	5662	1449
4	2	0	3227	2974	253
5	2	1	5131	3732	1399
6	3	0	592	483	109
7	3	1	1366	691	675
8	4	0	1619	1280	339
9	4	1	3736	1854	1882
10	5	0	536	357	179
11	5	1	1187	407	780
12	6	0	86	36	50
13	6	1	327	71	256

Part A: Model 1

In Chapter 13.2 McElreath iterates on his model for the UCB admissions data. (See the attached PDF of that section. We have included pdfs of excerpts from McElreath sections 10.1.3 and 13.2)

A1: Implement the upgraded model from section 13.2.1 on the income data. You may refer to Osvaldo Martin's pymc3 implementation of Statistical Rethinking if needed.

In [3]:
```
burnin=500
ssize = 10000
tune = 500
```

In [4]:
```
Nedu = len(categories)

with pm.Model() as edu_gender_13_2_1:
    # Set up the model
    a = pm.Normal('a',0, 10)
    bm = pm.Normal('bm',0, 1)
    sigma = pm.HalfCauchy('sigma', 2)
    alpha_ed = pm.Normal('alpha_ed', a, sigma, shape=Nedu)
    p = pm.math.invlogit(alpha_ed[dataset['edu']] + bm*dataset['sex'])
    income = pm.Binomial('income', p=p, n=dataset['total'], observed=dataset['>50'])
```

A2: You will notice here that the αs at different income levels are pooled

1. What criterion do you use to tell if full, partial, or no pooling is appropriate to the given data?

1.) It depends on the structure of the data as well as our underlying beliefs surrounding the data. If each "category" has a large collection of data and we believe the categories to be independent, then a no-pooling model may be appropriate (however, if this is not the case, we will definitely over-fit). If we believe that all our categories really come from one general place (aka that there is no variation between categories besides sampling variation), then we can pool all of our data together under a full-pooling model (however, if this is not the case, we will underfit the intricacies of our data). In most cases, the best approach is a partial-pooling model, where we expect variation between categories (and thus don't want complete pooling) yet some categories might have smaller samples (and therefore more variation) than others, thus leading us to want some pooling (vs. no pooling). Partial-pooling strikes this balance, as parameters for each category are taken to come from one distribution for the parameter.

2. Which type of pooling is implemented here. Why is it most appropriate to our income data?

2.) Partial pooling (we can see this as our intercepts - although different for each educational category - are coming from an assumed pooled, underlying distribution) - it's best here for two reasons. Firstly, because we believe there are differences in the earning potentials for the different sexes based on education level (therefore, don't want complete pooling). Secondly, because different categories have different sample sizes and therefore different levels of variation - pooling helps us take advantage of our large, overall data size, thus leading us to better models for all the categories.

A3: Sample from the model's posterior and comment on the quality of the samples obtained, examining at least two of the diagnostics discussed in class.

[Sampling this model took 2 minutes and 30 seconds per chain (~80 it/s) on a VM running on old hardware.]

In [5]:
```
# Sample the models
map1 = pm.find_MAP(model=edu_gender_13_2_1)

with edu_gender_13_2_1:
    stepper = pm.NUTS()
    traces_edu_13_2_1 = pm.sample(ssize, step=stepper, tune=tune, chains=3, start=map1)
```

```
logp = -71.454, ||grad|| = 0.40208: 100%|██████████| 48/48 [00:00<00:00, 276.86it/s]
Multiprocess sampling (3 chains in 2 jobs)
NUTS: [alpha_ed, sigma_log__, bm, a]
100%|██████████| 10500/10500 [00:33<00:00, 313.23it/s]
```

In [6]:
```
traces_edu_13_2_1
```

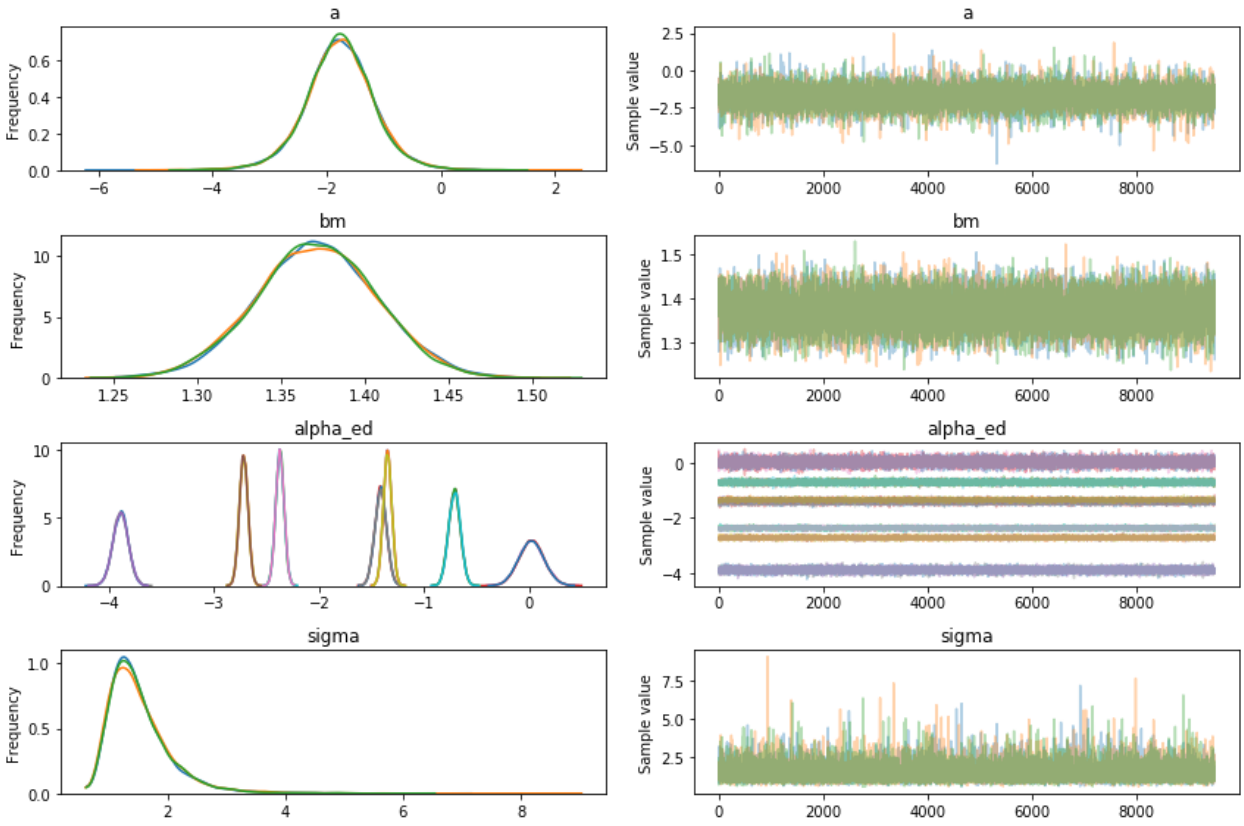
Out[6]:
```
<MultiTrace: 3 chains, 10000 iterations, 5 variables>
```

In [7]:
```
# Show summary plot
pm.summary(traces_edu_13_2_1).round(3)
```

Out[7]:

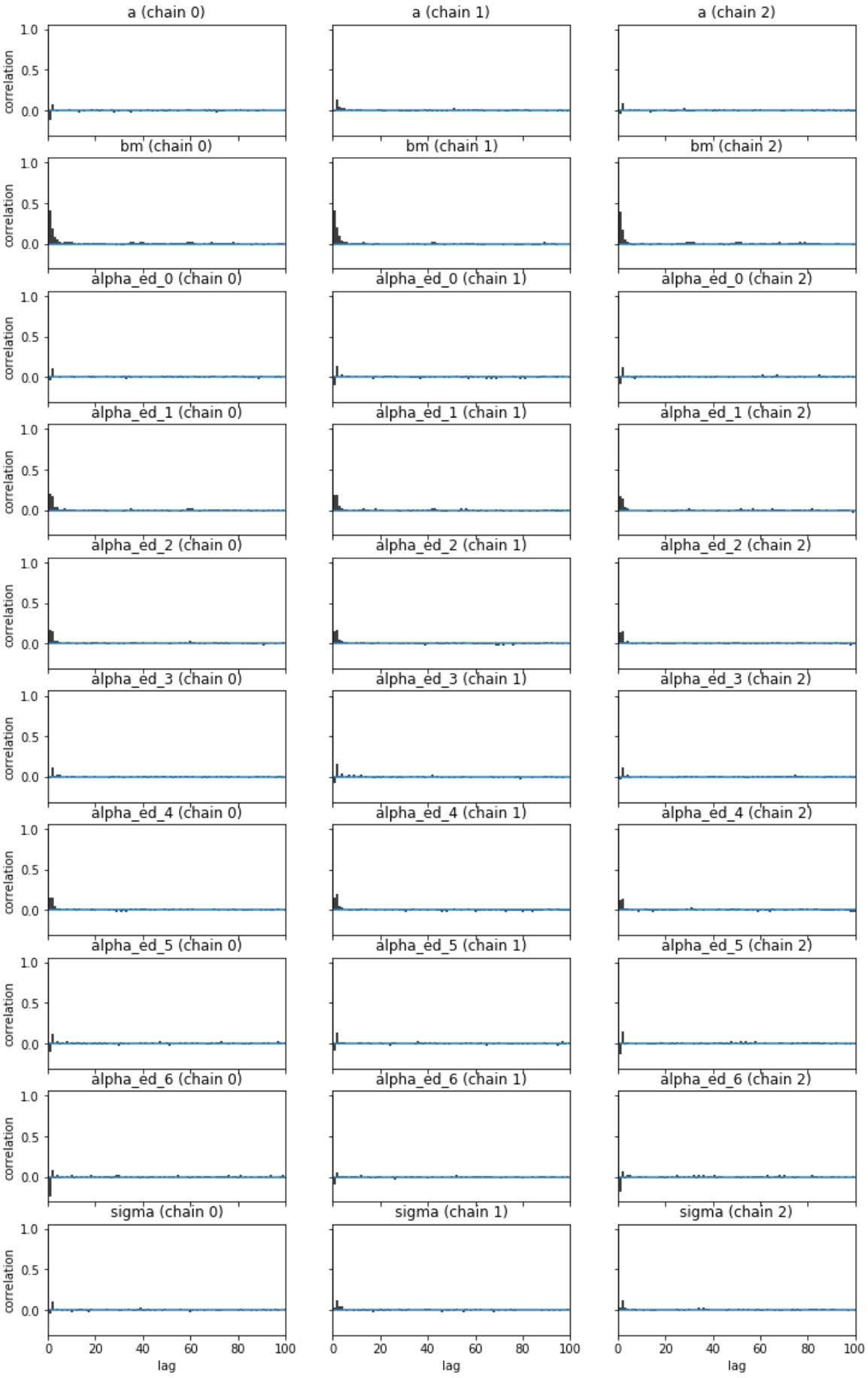
	mean	sd	mc_error	hpd_2.5	hpd_97.5	n_eff	Rhat
a	-1.771	0.616	0.003	-2.993	-0.532	23344.0	1.0
bm	1.371	0.036	0.000	1.301	1.442	12239.0	1.0
alpha_ed__0	-3.891	0.073	0.000	-4.033	-3.747	25097.0	1.0
alpha_ed__1	-2.719	0.041	0.000	-2.798	-2.637	15515.0	1.0
alpha_ed__2	-2.374	0.040	0.000	-2.455	-2.298	15891.0	1.0
alpha_ed__3	-1.415	0.056	0.000	-1.525	-1.308	21895.0	1.0
alpha_ed__4	-1.350	0.040	0.000	-1.431	-1.273	15991.0	1.0
alpha_ed__5	-0.712	0.056	0.000	-0.823	-0.604	26198.0	1.0
alpha_ed__6	0.020	0.119	0.001	-0.216	0.248	30000.0	1.0
sigma	1.539	0.530	0.004	0.784	2.573	21654.0	1.0

In [8]:
```
pm.traceplot(traces_edu_13_2_1[burnin:]);
```



Trace Comment: The above trace plots look pretty good. There are very limited number of regions where the sampler is sticky. The autocorrelation plots below show very quick correlation decreases.

In [9]:
```
pm.autocorrplot(traces_edu_13_2_1[burnin:]);
```



In [10]:
```
traces_edu_13_2_1.varnames
```

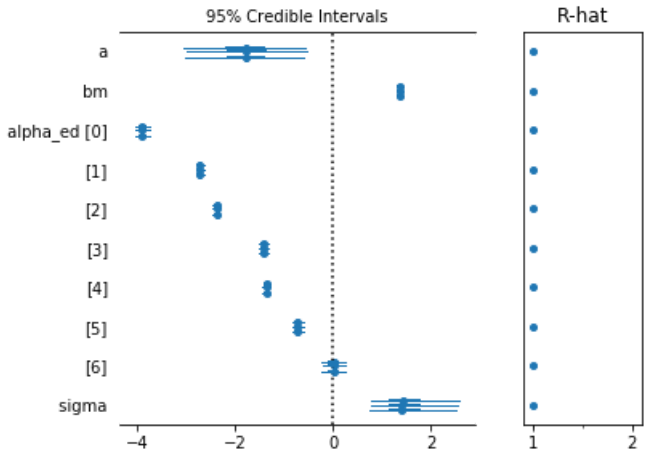
Out[10]:
```
['a', 'bm', 'sigma_log__', 'alpha_ed', 'sigma']
```

In [11]:
```
# 1. Gelman Rubin Statistic
pm.gelman_rubin(traces_edu_13_2_1)
```

Out[11]:
```
{'a': 1.000081697993225,
 'alpha_ed': array([ 0.99999305,  0.99995042,  0.99999093,  0.9999572 ,  0.99996376,
         0.99995284,  0.99996665]),
 'bm': 1.0000286496952051,
 'sigma': 1.0000680076533772}
```

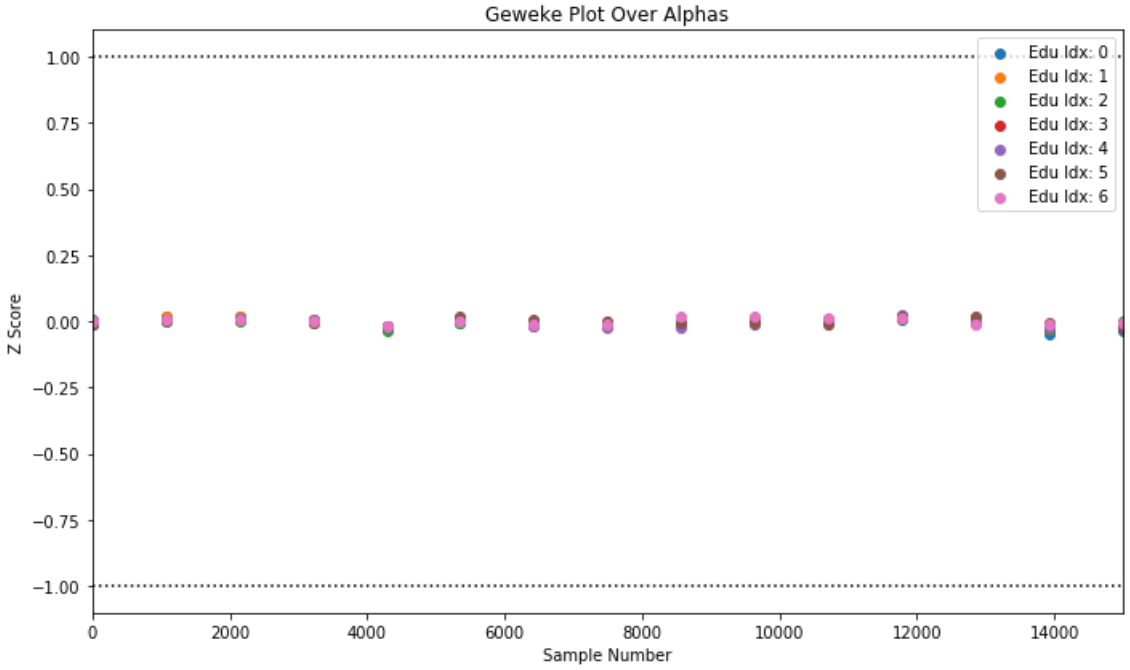
In [12]:
```
pm.forestplot(traces_edu_13_2_1)
```

Out[12]:
```
<matplotlib.gridspec.GridSpec at 0x1c165e7dd8>
```



The Gelman-Rubin statistic uses the fact of multiple chains to check if the sampling has converged. If there is convergence, we expect different chains to look very similar to one another, and the Gelman Rubin statistic checks for that. A value close to 1.0 indicates that sampling has converged and the chains look like each other. We see in the above results that all of the Gelman Rubin statistics are close to 1.0, indicating that our sampling did in fact converge. We can then use the Forest Plot function to get a sense of the behavior of each of the variables in our model. We see that hte Rhat values are very close to 1.0 and the 95% credible intervals are pretty tight along most of the parameters of the model.

In [13]:
```
# 2. Geweke Statistic
plt.figure(figsize=(10,6))

z = pm.geweke(traces_edu_13_2_1['alpha_ed'], intervals=15)
for edu_idx in range(len(z)):
    plt.scatter(*z[edu_idx].T, label=r'Edu Idx: %d'%edu_idx)

plt.hlines([-1,1], 0, 15000, linestyles='dotted')
plt.xlim(0,15000)
plt.title('Geweke Plot Over Alphas')
plt.xlabel('Sample Number')
plt.ylabel('Z Score')
plt.legend()
plt.tight_layout()
plt.show()
```



The Geweke statistic compares the mean and variance of different segments in certain intervals of a single chain. You can find the equation online somewhere. If the z scores of different segments of the chain are similar, then we operate under the assumption that the sampler has converged. We see in the above figure that, over the number of samples, we do get a convergence of our chains.

In [14]:
```
# Let's also implement the old models and do a WAIC check
with pm.Model() as ps1:
    beta_m = pm.Normal("beta_m", 0, 10)
    alpha = pm.Normal('alpha', 0, 10)
    logitpi = alpha + (beta_m*dataset.sex)
    n_earn = pm.Binomial("n_earn", n=dataset.total, p=pm.math.invlogit(logitpi), observed=data
set['>50'])
```

In [15]:
```
with ps1:
    # obtain starting values via MAP
    start = pm.find_MAP(model=ps1)
    # instantiate sampler
    step = pm.NUTS()
    # draw posterior samples
    trace_1 = pm.sample(5000, step=step, start=start)
```

```
logp = -7,423.3, ||grad|| = 9,441.6: 100%|██████████| 11/11 [00:00<00:00, 835.61it/s]
Multiprocess sampling (2 chains in 2 jobs)
NUTS: [alpha, beta_m]
100%|██████████| 5500/5500 [00:11<00:00, 463.92it/s]
The acceptance probability does not match the target. It is 0.880953374945, but should be clo
se to 0.8. Try to increase the number of tuning steps.
The acceptance probability does not match the target. It is 0.889444950679, but should be clo
se to 0.8. Try to increase the number of tuning steps.
The number of effective samples is smaller than 25% for some parameters.
```

In [16]:
```
with pm.Model() as ps2:
    beta_m = pm.Normal("beta_m", 0, 10)
    alpha = pm.Normal('alpha', 0, 10, shape=7)
    logitpi = alpha[dataset['edu']] + (beta_m*dataset['sex'])
    n_earn = pm.Binomial("n_earn", n=dataset.total, p=pm.math.invlogit(logitpi), observed=data
set['>50'])
```

In [17]:
```
with ps2:
    # obtain starting values via MAP
    start = pm.find_MAP(model=ps2)
    # instantiate sampler
    step = pm.NUTS()
    # draw posterior samples
    trace_2 = pm.sample(5000, step=step, start=start)
```

```
logp = -79.535, ||grad|| = 0.10894: 100%|██████████| 26/26 [00:00<00:00, 1536.83it/s]
Multiprocess sampling (2 chains in 2 jobs)
NUTS: [alpha, beta_m]
100%|██████████| 5500/5500 [00:11<00:00, 487.77it/s]
```

We see that the newest model performs the best on the dataset as determined by the WAIC score.

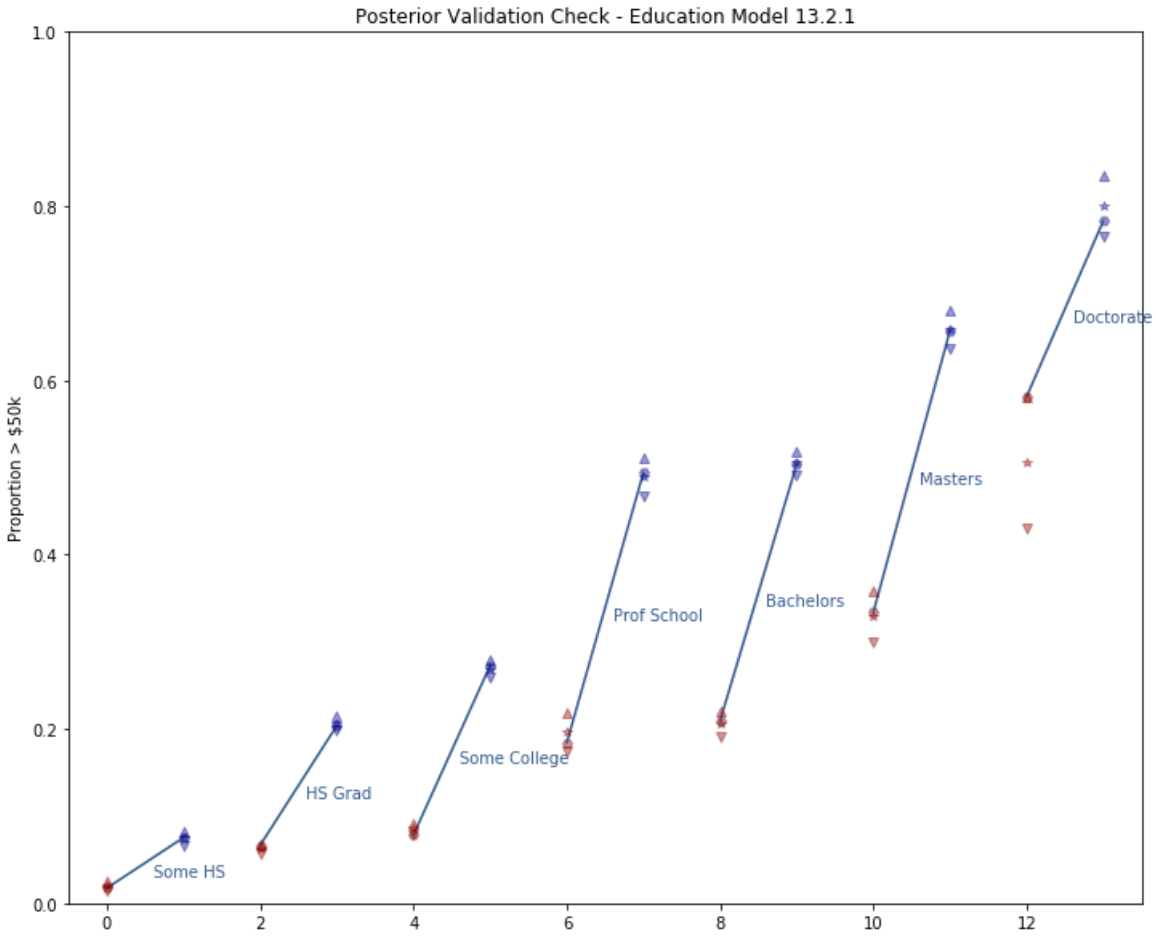
Now we will return back to the original code and do a posterior preductive sample on the original traces.

In [18]:
```
# Sample from posterior using sample ppc
with edu_gender_13_2_1:
    ppc_edu_13_2_1 = pm.sample_ppc(traces_edu_13_2_1)
```

```
100%|██████████| 10000/10000 [00:03<00:00, 2617.73it/s]
```

In [19]:
```
# CITATION: The following code was adapted from the solution set to HW 10 from this course.
# Posterior validation function
def plot_post_validation_check(ax, dataset, ppc_obs, labels, title=""):

    above_50k = dataset['>50']
    total = dataset['total']

    color_map = plt.cm.get_cmap("viridis")
    upper_quantile = np.percentile(ppc_obs, 89, axis=0)
    lower_quantile = np.percentile(ppc_obs, 11, axis=0)

    for i in range(0, 7):
        ax.plot([2*i], (above_50k/ total)[2*i], color= 'darkred', alpha=0.4, marker='H', lines
tyle='None')
        ax.plot([2*i+1], (above_50k/ total)[2*i+1], color= 'darkblue', alpha=0.4, marker='H',
linestyle='None')

        ax.plot([2*i, 2*i+1], (above_50k/ total)[2*i:2*i+2], color = color_map(2/7), linestyle
='-')

        ax.text(2*i+ 0.6, np.mean((above_50k/total)[2*i:2*i+2]), labels[i],  color = color_map
(2/7), ha='left', va='top')

        # plot ppc means
        ax.plot([2*i], (ppc_obs.mean(axis=0)/ total)[2*i], color= 'darkred', alpha=0.4, marker
='*', linestyle='None')
        ax.plot([2*i+1], (ppc_obs.mean(axis=0)/ total)[2*i+1], color= 'darkblue', alpha=0.4, m
arker='*', linestyle='None')


        # plot percentiles
        ax.plot([2*i], (upper_quantile/total)[2*i], color= 'darkred', alpha=0.4, marker='^', l
inestyle='None')
        ax.plot([2*i], (lower_quantile/total)[2*i], color= 'darkred', alpha=0.4, marker='v', l
inestyle='None')

        ax.plot([2*i+1], (upper_quantile/total)[2*i+1], color= 'darkblue',alpha=0.4, marker=
'^', linestyle='None')
        ax.plot([2*i+1], (lower_quantile/total)[2*i+1], color= 'darkblue',alpha=0.4, marker=
'v', linestyle='None')


    ax.set_xlim((-0.5, 13.5))
    ax.set_ylim((0, 1))
    ax.set_title("Posterior Validation Check - " + title)
    ax.set_ylabel("Proportion > \$50k")
```

In [20]:
```
fig, ax = plt.subplots(figsize=(12, 10))
plot_post_validation_check(ax, dataset, ppc_edu_13_2_1['income'], labels, "Education Model 13.2.1")
```



A4: Use the code below to plot the amount of shrinkage in our model. Think carefully about what quantity in the dataset and in the posterior you ought to compare. You should see minimal shrinkage in all categories except the last one.

In [21]:
```
def shrinkage_plot(observed_log_odds, posterior_means, posterior_sds):
    """
    observed_log_odds - an entry for each category giving the actual log-odds observed log(P(>50K)/(1-P(>50K)))
    posterior_means - an entry for each category giving the mean of the posterior distribution
    posterior_sds - an entry for each category giving the SD of the posterior distribution
    """
    plt.scatter(range(len(observed_log_odds)),observed_log_odds, label="Observed Log Odds", color='orange')
    plt.errorbar(range(len(posterior_means)), posterior_means, yerr = 2*posterior_sds, fmt='o', label="Posterior for Log Odds\n(mean and 2 SDs)",)
    plt.legend()
    plt.xlabel("Education Category")
```

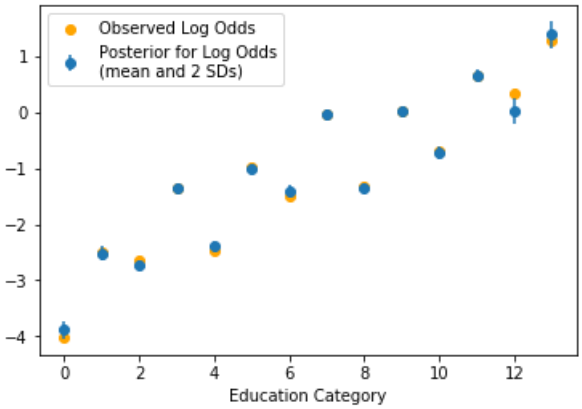
In [22]:
```
# Get the observed log odds:
obs_log_odds = np.log(dataset['>50'] / dataset['<=50']).values
```

In [23]:
```
obs_log_odds
```

Out[23]:
```
array([-4.03308568, -2.50691015, -2.63905733, -1.36290352, -2.46427364,
       -0.98118659, -1.48866877, -0.02342713, -1.32861525,  0.01498957,
       -0.69034998,  0.65048073,  0.32850407,  1.28249757])
```

In [24]:
```
# Get the posterior means
p_male = traces_edu_13_2_1['alpha_ed'] + traces_edu_13_2_1['bm'][:, np.newaxis]
p_female = traces_edu_13_2_1['alpha_ed']

# Calculate means and stds
p_male_mean = np.mean(p_male,axis=0)
p_male_sd = np.std(p_male,axis=0)
p_female_mean = np.mean(p_female,axis=0)
p_female_sd = np.std(p_female, axis=0)

# Restack them
p_mean = [val for pair in zip(p_female_mean, p_male_mean) for val in pair]
p_sd = [val for pair in zip(p_female_sd, p_male_sd) for val in pair]
```

In [25]:
```
shrinkage_plot(np.array(obs_log_odds), np.array(p_mean), np.array(p_sd))
```



A5: McElreath's new model showed substantial shrinkage within the categories. Why does our model show very minimal shrinkage?. For full credit, include Standard Error in your explanation.

In a typical hierarchical model with pooling, we expect to see some shrinkage in the predicted rates (or in this case, odds) between the model and the data that we are provided. This is a result of the pooling of certain variables in the modeling between different categories. In this case, we see that the total number of people in each category is large. Similarly to the some of the models we have looked at, this results in less shrinkage than one would otherwise expect. In the last case, we have smaller samples, and we do see the effect of shrinkage. In terms of the standard error, for a larger sample, you have a smaller standard error, meaning that your parameter estimates are more likely to reflect the data that you received.

Part B: Iterating further

After the pooling model above, McElreath builds a model where the alpha and beta in each unit might be correlated (perhaps the higher the admission rate, the less bias there is towards either gender).

B1: Implement the model in 13.2.2. Feel free to use code from Osvaldo Martin (https://github.com/aloctavodia/Statistical-Rethinking-with-Python-and-PyMC3/blob/master/Chp_13.ipynb) with attribution and understanding (it contains some sweet pymc3 technical wrangling).

The variable names in this model are unchanged and you may re-name them if you wish.

Note that this model builds the a 7 by 2 matrix with alpha values in the first column and beta values in the second. By assumption, the first column and the second column have correlation structure given by the LKJ prior, but there is no explicit correlation among the rows (aside from all the alphas coming from the same mean). In other words, the correlation matrix is 2x2 (not 14x14) and amongst the income-level-free parts of the intercept and slope.

In [26]:
```
# Code taken and adapted from Osvaldo Martin:
Nedu = len(categories)
edu_id = dataset['edu']
with pm.Model() as edu_gender_13_2_2:
    a = pm.Normal('a', 0, 10)
    b = pm.Normal('b', 0, 1)

    sd_dist = pm.HalfCauchy.dist(beta=2)
    packed_chol = pm.LKJCholeskyCov('chol_cov', eta=2, n=2, sd_dist=sd_dist)

    # compute the covariance matrix
    chol = pm.expand_packed_triangular(2, packed_chol, lower=True)
    cov = tt.dot(chol, chol.T)

    # Extract the standard deviations and rho
    sigma = pm.Deterministic('sigma', tt.sqrt(tt.diag(cov)))
    corr = tt.diag(sigma**-1).dot(cov.dot(tt.diag(sigma**-1)))
    r = pm.Deterministic('Rho', corr[np.triu_indices(2, k=1)])

    mu = pm.MvNormal('mu', mu=tt.stack([a, b]), chol=chol, shape=(Nedu, 2))

    alpha_ed = pm.Deterministic('alpha_ed', mu[:, 0])
    bm_ed = pm.Deterministic('bm_ed', mu[:, 1])

    p = pm.math.invlogit(mu[edu_id, 0] + mu[edu_id, 1] * dataset['sex'])
    income = pm.Binomial('income', p=p, n=dataset['total'], observed=dataset['>50'])
```

B2: Sample from the posterior of the model above *with a target acceptance rate of .9 or more*. (Sampling takes 10 minutes per chain on a VM on old hardware and 2 minutes 30 seconds on a recent-gen Mac). Analyze at least two diagnostics and comment on the quality of the samples obtained.

In [27]:
```
# Find map estimate
map2 = pm.find_MAP(model=edu_gender_13_2_2)
```

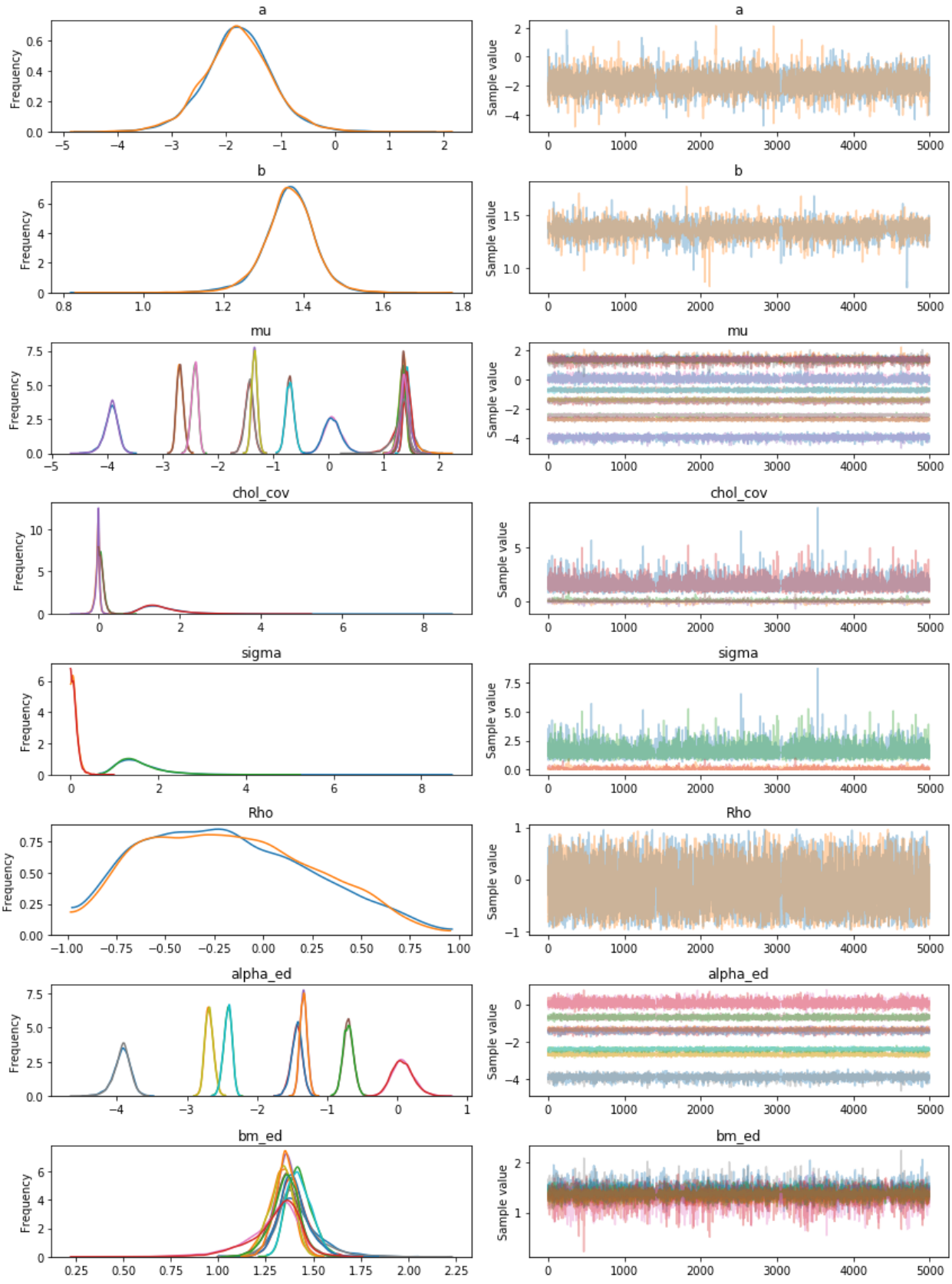
```
logp = -4.4541, ||grad|| = 11,861: 100%|██████████| 1801/1801 [00:03<00:00, 510.02it/s]
```

In [28]:
```
# Sample the models
with edu_gender_13_2_2:
   # stepper=pm.Metropolis()t
    #traces_13_2_2_preburn = pm.sample(10000, tune=200, step=stepper, start=map2, chains=2)
    traces_13_2_2 = pm.sample(5000, nuts_kwargs={'target_accept': 0.9}, tune=500, start=map2,
chains=2)
```

```
Auto-assigning NUTS sampler...
Initializing NUTS using jitter+adapt_diag...
Multiprocess sampling (2 chains in 2 jobs)
NUTS: [mu, chol_cov_cholesky_cov_packed__, b, a]
100%|██████████| 5500/5500 [05:50<00:00, 15.70it/s]
There were 22 divergences after tuning. Increase `target_accept` or reparameterize.
The acceptance probability does not match the target. It is 0.96128873938, but should be clos
e to 0.9. Try to increase the number of tuning steps.
There were 67 divergences after tuning. Increase `target_accept` or reparameterize.
The number of effective samples is smaller than 10% for some parameters.
```

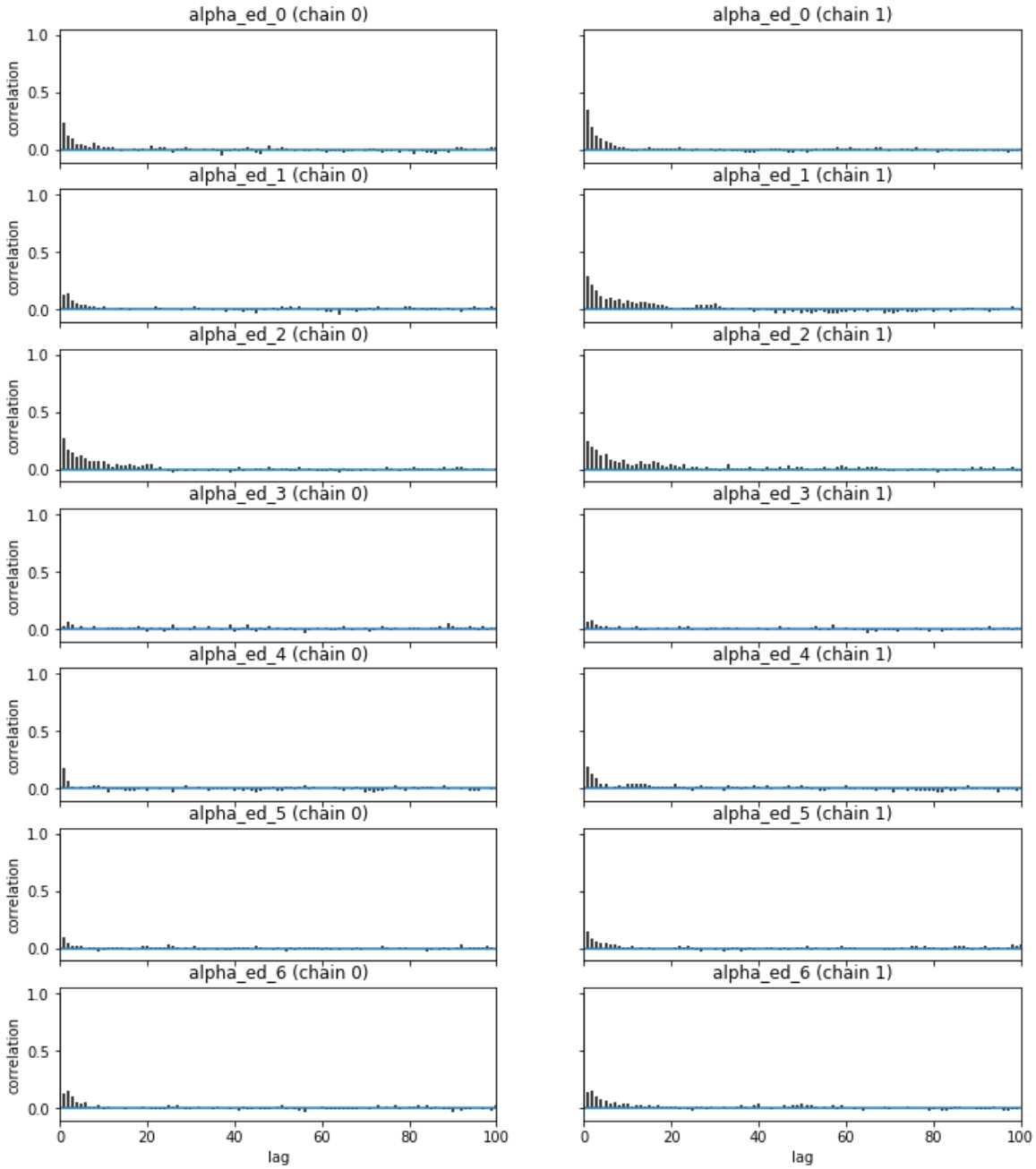
In [30]:
```
pm.traceplot(traces_13_2_2)
```

```
Out[30]: array([[<matplotlib.axes._subplots.AxesSubplot object at 0x1c1fe9cf98>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x1c200a1400>],
                [<matplotlib.axes._subplots.AxesSubplot object at 0x1c202eae10>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x1c202e4898>],
                [<matplotlib.axes._subplots.AxesSubplot object at 0x1c1e425518>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x1c20413198>],
                [<matplotlib.axes._subplots.AxesSubplot object at 0x1c1e424dd8>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x1c1d185be0>],
                [<matplotlib.axes._subplots.AxesSubplot object at 0x1c2027a0f0>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x1c1d19ec18>],
                [<matplotlib.axes._subplots.AxesSubplot object at 0x1c20339cc0>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x1c1df18978>],
                [<matplotlib.axes._subplots.AxesSubplot object at 0x1c1ee054e0>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x1c1ee75588>],
                [<matplotlib.axes._subplots.AxesSubplot object at 0x1c1e3c21d0>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x1c1ee55a20>]], dtype=object)
```


a
Frequency
Sample value
b
Frequency
Sample value
mu
Frequency
Sample value
chol_cov
Frequency
Sample value
sigma
Frequency
Sample value
Rho
Frequency
Sample value
alpha_ed
Frequency
Sample value
bm_ed
Frequency
Sample value

In [31]:
```
pm.autocorrplot(traces_13_2_2, varnames=['alpha_ed'])
```

Out[31]:
```
array([[<matplotlib.axes._subplots.AxesSubplot object at 0x1c206804e0>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x1c21ddd940>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x1c1f511518>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x1c2116eac8>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x1c1f39d860>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x1c1f3bf940>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x1c211bc518>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x1c211edda0>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x1c21225b38>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x1c22357748>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x1c2238ef28>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x1c223aaf28>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x1c22409128>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x1c22439e80>]], dtype=object)
```

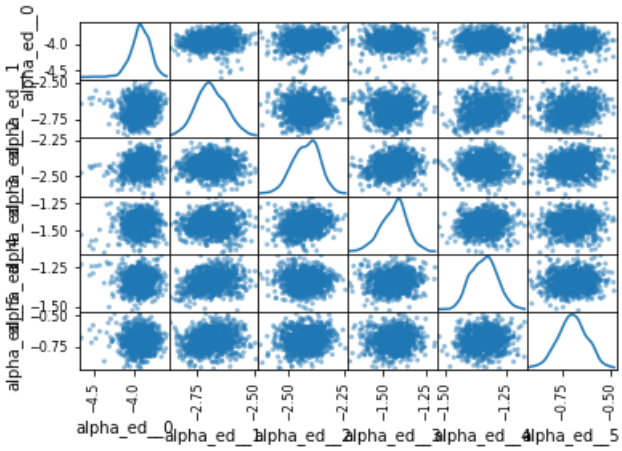


Below we will look at the parameter correlation plots of the alpha values and see that there are a lot of them!

In [32]:
```
df_trace=pm.trace_to_dataframe(traces_13_2_2, varnames=['alpha_ed'])
pd.plotting.scatter_matrix(df_trace.iloc[-1000:,0:6], diagonal='kde')
df_trace.iloc[burnin::,0:6].corr()
```

Out[32]:

	alpha_ed__0	alpha_ed__1	alpha_ed__2	alpha_ed__3	alpha_ed__4	alpha_ed__5
alpha_ed__0	1.000000	0.071102	0.238761	0.123161	0.036102	-0.010827
alpha_ed__1	0.071102	1.000000	-0.030842	0.034973	0.176217	0.135084
alpha_ed__2	0.238761	-0.030842	1.000000	0.188535	0.042014	0.062220
alpha_ed__3	0.123161	0.034973	0.188535	1.000000	0.092406	0.098340
alpha_ed__4	0.036102	0.176217	0.042014	0.092406	1.000000	0.148169
alpha_ed__5	-0.010827	0.135084	0.062220	0.098340	0.148169	1.000000



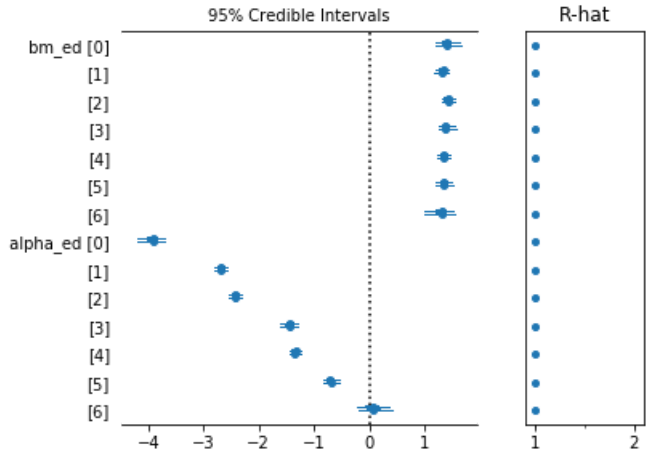
Perform diagnostic checks:

In [33]:
```
# 1. Gelman Rubin Statistic
pm.gelman_rubin(traces_13_2_2)
```

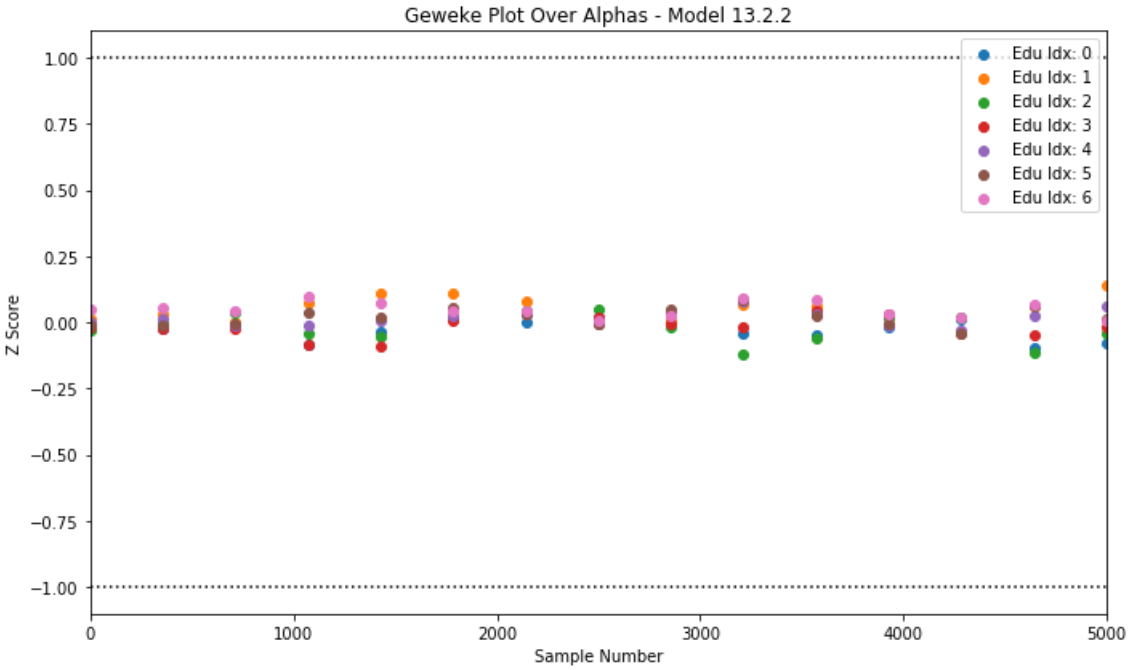
Out[33]:
```
{'Rho': array([ 1.00021545]),
 'a': 1.0000362772617883,
 'alpha_ed': array([ 0.99992316,  1.00025428,  0.99996157,  1.00002551,  0.99990476,
         1.00007319,  1.0012592 ]),
 'b': 1.0003742576488006,
 'bm_ed': array([ 0.99997415,  1.00061199,  0.99994155,  0.99995963,  1.00000217,
         1.00067966,  1.00122739]),
 'chol_cov': array([ 1.00058342,  1.00011026,  1.00114763]),
 'mu': array([[ 0.99992316,  0.99997415],
        [ 1.00025428,  1.00061199],
        [ 0.99996157,  0.99994155],
        [ 1.00002551,  0.99995963],
        [ 0.99990476,  1.00000217],
        [ 1.00007319,  1.00067966],
        [ 1.0012592 ,  1.00122739]]),
 'sigma': array([ 1.00058342,  1.00120922])}
```

In [34]:
```
pm.forestplot(traces_13_2_2, varnames=['bm_ed', 'alpha_ed'])
```

Out[34]: <matplotlib.gridspec.GridSpec at 0x1c20680b70>



In [35]:
```
# 2. Geweke Statistic
plt.figure(figsize=(10,6))

z = pm.geweke(traces_13_2_2['alpha_ed'], intervals=15)
for edu_idx in range(len(z)):
    plt.scatter(*z[edu_idx].T, label=r'Edu Idx: %d'%edu_idx)

plt.hlines([-1,1], 0, 5000, linestyles='dotted')
plt.xlim(0,5000)
plt.title('Geweke Plot Over Alphas - Model 13.2.2')
plt.xlabel('Sample Number')
plt.ylabel('Z Score')
plt.legend()
plt.tight_layout()
plt.show()
```



Comments on sample quality: We see that the sample quality is not too bad, but it could be better. The Rhat values are all close to 1.0 and the traceplots look fine, however, we see that the Geweke plot shows some signs of non-convergence in that the z values oscillate quite a bit. In addition, we see that there exist a large amount of divergences in our sampling, indicating that we can improve our model. We also see a small amount of autocorrelation in the plots above.

B3: Propose at least two methods of improving the quality of the samples obtained and explain how/why each method would help. For **some extra credit** implement these.

Improvement 1: The first improvement would be to de-center the data such that it has zero mean. Doing so removes a lot of the correlation between the alpha and beta terms. We did not implement this method. Instead, we chose to implement method 2.

Improvement 2: The second improvement is to attempt to remove the divergences in the sampling by constructing a non-centered model. The model is created below

In [36]:
```
# NON-CENTERED MODEL:
# Code taken and adapted from Osvaldo Martin with adaptations for non-centering
Nedu = len(categories)
edu_id = dataset['edu']
with pm.Model() as edu_gender_13_2_2_NC:
    a = pm.Normal('a', 0, 10)
    b = pm.Normal('b', 0, 1)

    sd_dist = pm.HalfCauchy.dist(beta=2)
    packed_chol = pm.LKJCholeskyCov('chol_cov', eta=2, n=2, sd_dist=sd_dist)

    # compute the covariance matrix
    chol = pm.expand_packed_triangular(2, packed_chol, lower=True)
    cov = tt.dot(chol, chol.T)

    # Extract the standard deviations and rho
#     sigma = pm.Deterministic('sigma', tt.sqrt(tt.diag(cov)))
#     corr = tt.diag(sigma**-1).dot(cov.dot(tt.diag(sigma**-1)))
#     r = pm.Deterministic('Rho', corr[np.triu_indices(2, k=1)])

    mu_raw = pm.Normal('mu_raw', mu=0, sd=1, shape=(Nedu,2))
    mu = pm.Deterministic('mu', tt.dot(chol, mu_raw.T).T)

#     mu = pm.MvNormal('mu', mu=tt.stack([a, b]), chol=chol, shape=(Nedu, 2))

    alpha_ed = pm.Deterministic('alpha_ed', mu[:, 0])
    bm_ed = pm.Deterministic('bm_ed', mu[:, 1])

    p = pm.math.invlogit(mu[edu_id, 0] + mu[edu_id, 1] * dataset['sex'])
    income = pm.Binomial('income', p=p, n=dataset['total'], observed=dataset['>50'])
```

In [37]:
```
# Find map estimate
map3 = pm.find_MAP(model=edu_gender_13_2_2_NC)
```

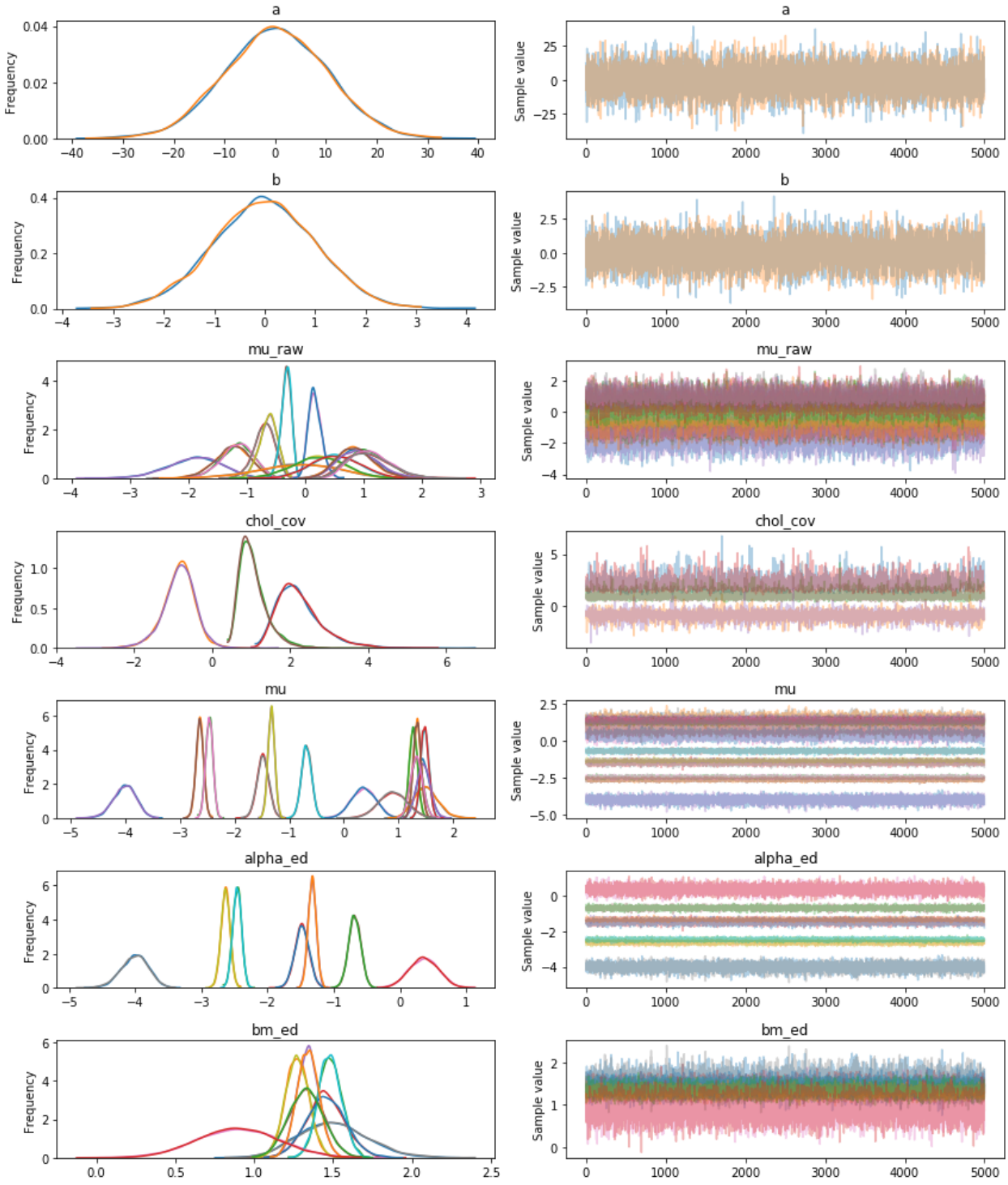
```
logp = -73.858, ||grad|| = 0.030473: 100%|██████████| 343/343 [00:00<00:00, 491.53it/s]
```

In [38]:
```
# Sample the models
with edu_gender_13_2_2_NC:
   # stepper=pm.Metropolis()t
    #traces_13_2_2_preburn = pm.sample(10000, tune=200, step=stepper, start=map2, chains=2)
    traces_13_2_2_NC = pm.sample(5000, nuts_kwargs={'target_accept': 0.9}, tune=500, start=map
3, chains=2)
```

```
Auto-assigning NUTS sampler...
Initializing NUTS using jitter+adapt_diag...
Multiprocess sampling (2 chains in 2 jobs)
NUTS: [mu_raw, chol_cov_cholesky_cov_packed__, b, a]
100%|██████████| 5500/5500 [11:04<00:00,  8.27it/s]
There were 1 divergences after tuning. Increase `target_accept` or reparameterize.
The number of effective samples is smaller than 25% for some parameters.
```

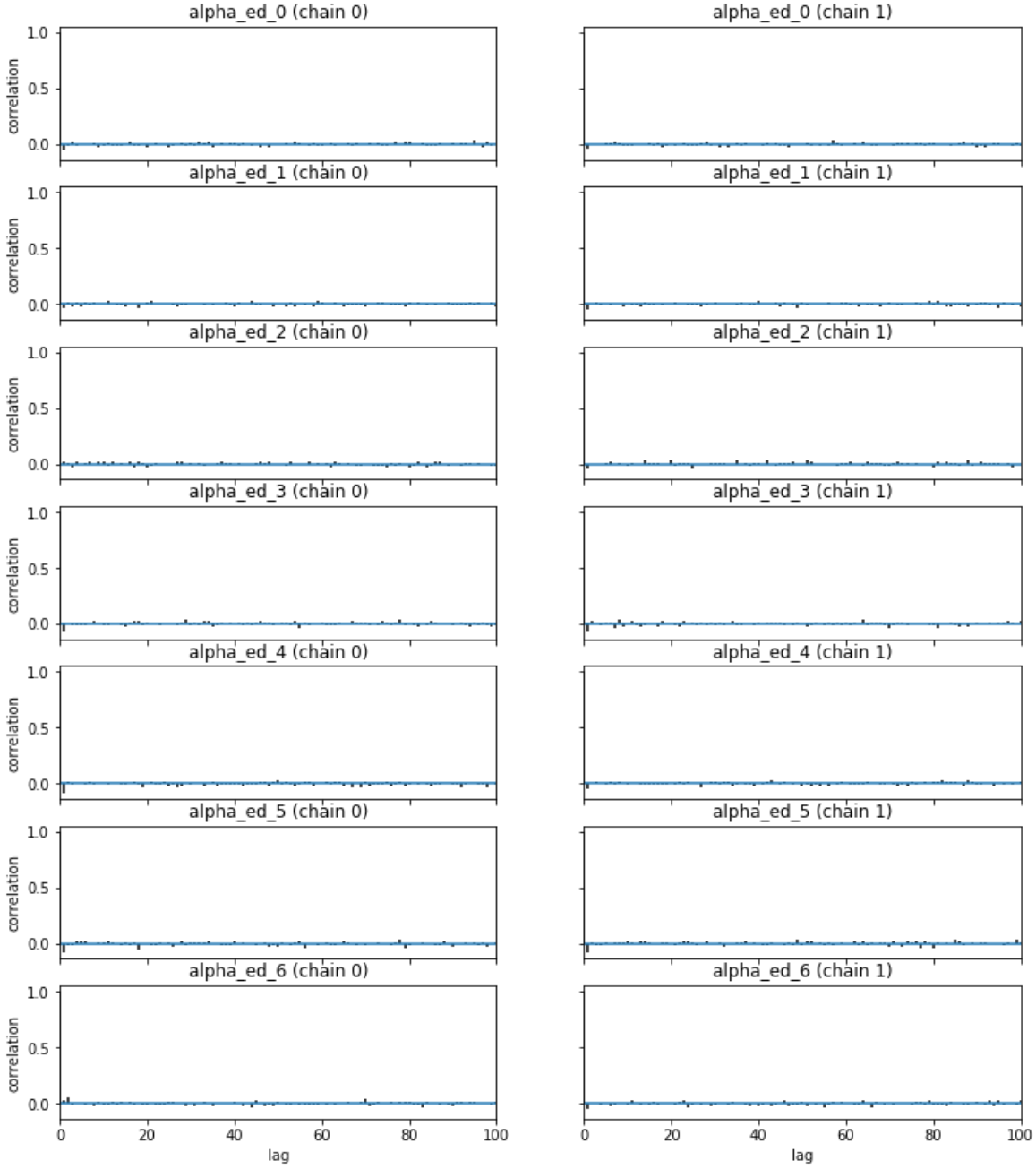
In [39]:
```
pm.traceplot(traces_13_2_2_NC)
```

Out[39]:
```
array([[<matplotlib.axes._subplots.AxesSubplot object at 0x1c22a63c88>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x1c22a7a7f0>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x1c24219438>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x1c24232048>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x1c242a7ba8>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x1c242d2c18>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x1c241ee860>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x1c244733c8>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x1c2449d390>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x1c244d0eb8>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x1c24508b00>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x1c245427b8>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x1c244edb70>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x1c245a7978>]], dtype=object)
```



In [40]:
```
pm.autocorrplot(traces_13_2_2_NC, varnames=['alpha_ed'])
```

Out[40]:
```
array([[<matplotlib.axes._subplots.AxesSubplot object at 0x1c22968e10>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x1c24abf470>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x1c2433fcf8>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x1c243aa780>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x1c24ae8080>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x1c24b120f0>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x1c24b41cf8>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x1c24b7d5c0>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x1c24bb83c8>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x1c24bd8e48>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x1c24c196d8>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x1c24c2fdd8>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x1c24c84898>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x1c24cc2550>]], dtype=object)
```

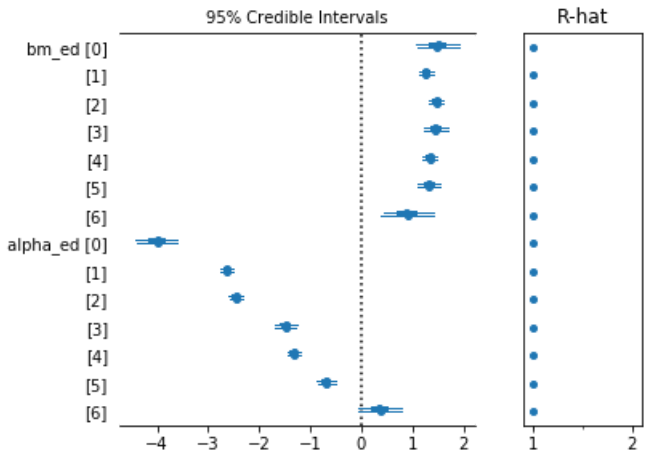


In [41]:
```
# 1. Gelman Rubin Statistic
pm.gelman_rubin(traces_13_2_2_NC)
```

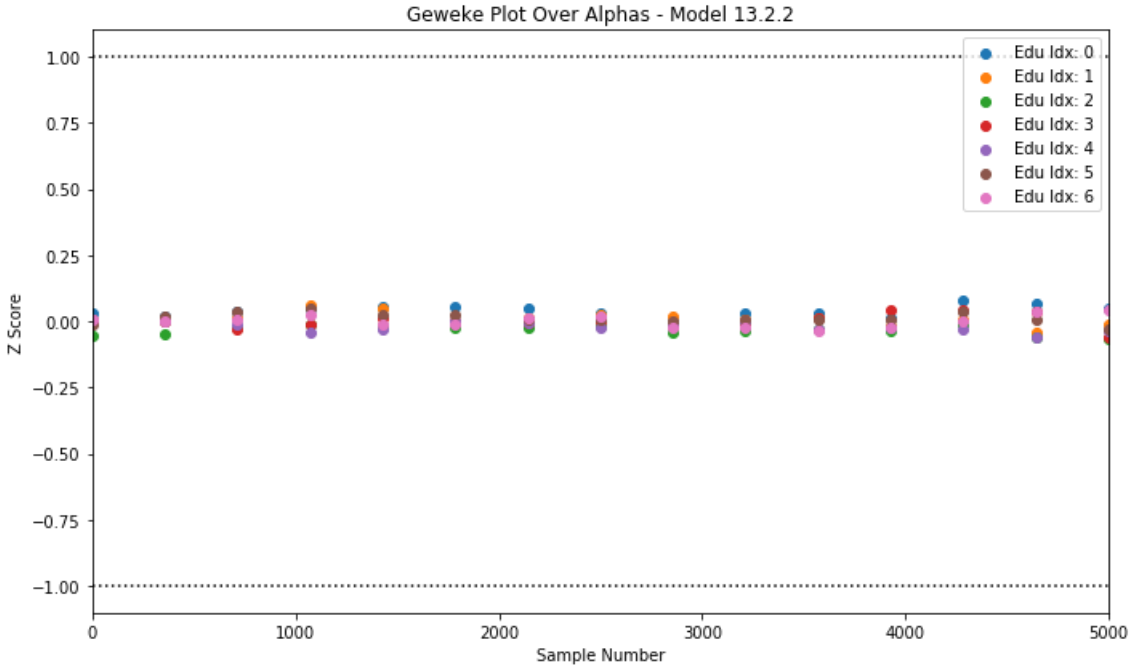
Out[41]:
```
{'a': 1.0000218130326497,
 'alpha_ed': array([ 1.00039548,  1.00007845,  1.00016427,  0.99994204,  0.99999485,
         1.00003384,  0.99992564]),
 'b': 1.0000225274708019,
 'bm_ed': array([ 1.00052708,  0.99995166,  1.00003586,  1.00000737,  1.00002602,
         0.99994808,  0.99997848]),
 'chol_cov': array([ 0.9999413 ,  0.99990218,  1.00136324]),
 'mu': array([[ 1.00039548,  1.00052708],
        [ 1.00007845,  0.99995166],
        [ 1.00016427,  1.00003586],
        [ 0.99994204,  1.00000737],
        [ 0.99999485,  1.00002602],
        [ 1.00003384,  0.99994808],
        [ 0.99992564,  0.99997848]]),
 'mu_raw': array([[ 0.99995637,  1.00002493],
        [ 1.00002346,  1.00010957],
        [ 1.0001002 ,  1.00034844],
        [ 0.99998523,  1.00127677],
        [ 1.00009835,  1.00108166],
        [ 0.99994005,  1.00152671],
        [ 0.99991656,  1.00057621]])}
```

In [42]:
```
pm.forestplot(traces_13_2_2_NC, varnames=['bm_ed', 'alpha_ed'])
```

Out[42]:
```
<matplotlib.gridspec.GridSpec at 0x1c24fd6e80>
```



In [43]:
```
# 2. Geweke Statistic
plt.figure(figsize=(10,6))

z = pm.geweke(traces_13_2_2_NC['alpha_ed'], intervals=15)
for edu_idx in range(len(z)):
    plt.scatter(*z[edu_idx].T, label=r'Edu Idx: %d'%edu_idx)

plt.hlines([-1,1], 0, 5000, linestyles='dotted')
plt.xlim(0,5000)
plt.title('Geweke Plot Over Alphas - Model 13.2.2')
plt.xlabel('Sample Number')
plt.ylabel('Z Score')
plt.legend()
plt.tight_layout()
plt.show()
```

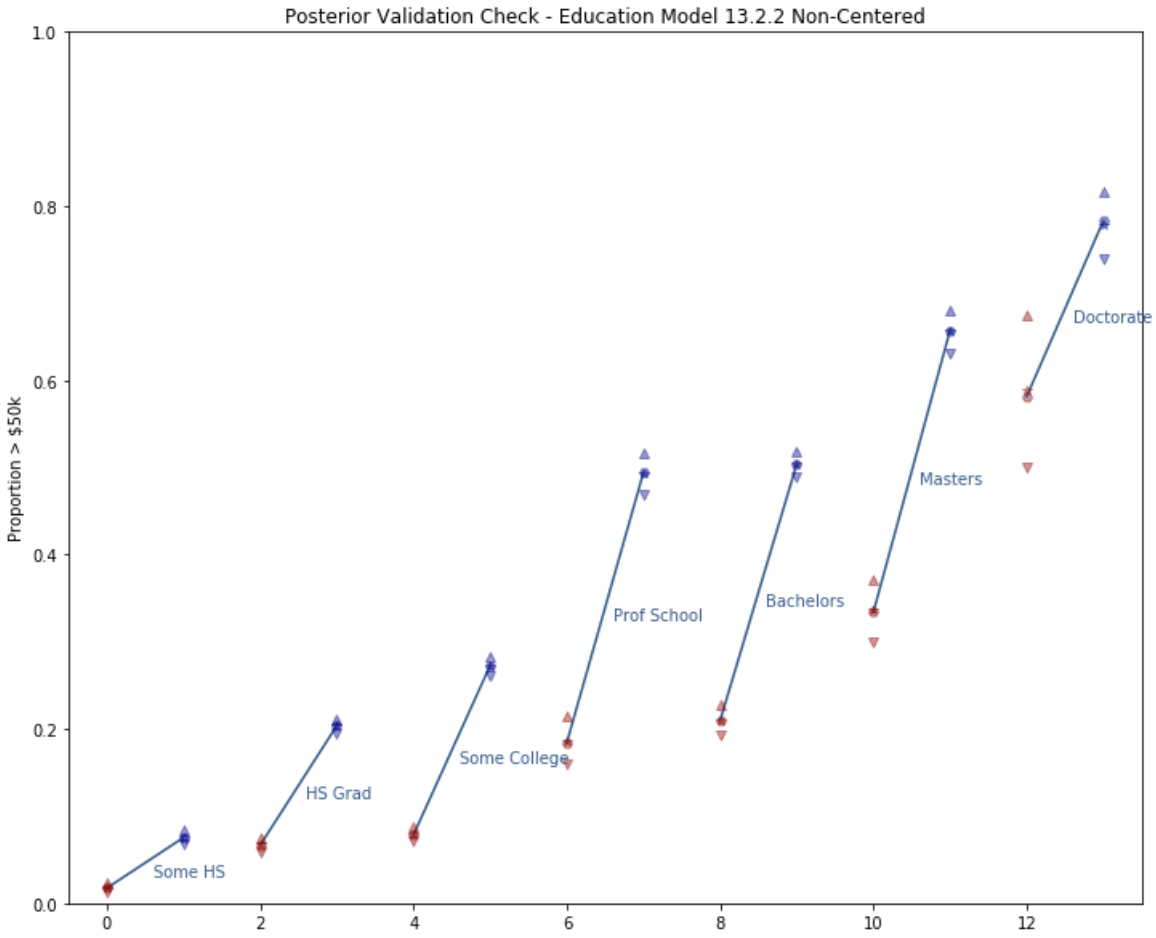


Now we see that the samples are much more converged. Our trace plots and autocorrelation plots indicate samples that are much better for the model! Next, we should plot a posterior predictive check to make sure that we are taking correct samples.

In [44]:
```
# Sample from posterior using sample ppc
with edu_gender_13_2_2_NC:
    ppc_edu_13_2_2_NC = pm.sample_ppc(traces_13_2_2_NC)
```

```
100%|██████████| 5000/5000 [00:03<00:00, 1492.61it/s]
```

In [45]:
```
fig, ax = plt.subplots(figsize=(12, 10))
plot_post_validation_check(ax, dataset, ppc_edu_13_2_2_NC['income'], labels, "Education Model
 13.2.2 Non-Centered")
```



Based on the above plot, it looks like sampling is going perfectly! Our improvement works!

B4: Regardless of your answer above, assume the samples are valid and give an analysis of whether the data and models support the hypothesis that males are more advantaged in lower income categories than they are in higher income categories. For full credit, include at least one formal model comparison or probability analysis.

In [46]:
```
# Compute WAIC scores:
def make_compare(names, traces, models, ic='WAIC'):
    compared=pm.compare(traces,models, method='pseudo-BMA')
    temp=compared.sort_index()
    temp['Model']=names
    compared = temp.sort_values(ic).set_index('Model')
    return compared
```

In [47]:
```
# Compare the previous two models using WAIC statistics:
names=['Edu. 1', 'Edu. 2', 'Edu 13.2.1', 'Edu 13.2.2', 'Edu 13.2.2 NC']
traces = [trace_1, trace_2, traces_edu_13_2_1, traces_13_2_2, traces_13_2_2_NC]
models = [ps1, ps2, edu_gender_13_2_1, edu_gender_13_2_2, edu_gender_13_2_2_NC]
dfc=make_compare(names, traces, models)
dfc
```

Out[47]:

	WAIC	pWAIC	dWAIC	weight	SE	dSE	warning
Model							
Edu 13.2.2	121.51	6.1	0	0.45	4.85	0	1
Edu 13.2.1	123.17	6.08	1.66	0.2	5.68	1.64	1
Edu 13.2.2 NC	123.24	7.14	1.73	0.19	4.58	1.87	1
Edu. 2	123.49	6.24	1.98	0.17	5.68	1.69	1
Edu. 1	4656.74	488.85	4535.23	0	1183.83	1182.64	1

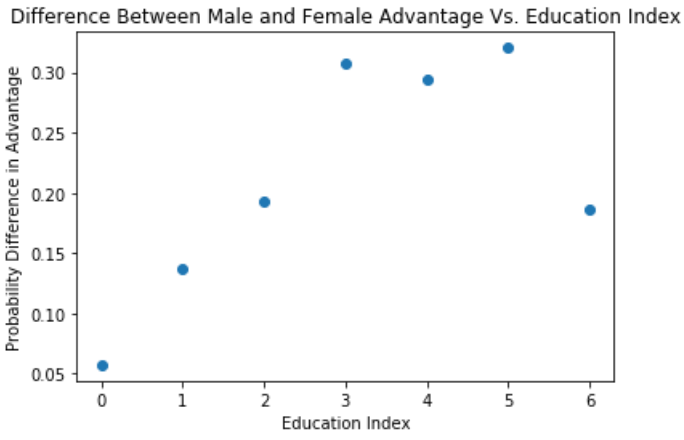
We see that the non-centered model contains the most weight in the WAIC model comparison tool. However, using the centered model and "assuming that our model is valid" does not provide us with the best overall model. Next is to test the hypothesis that males are more advantaged in lower income categories than they are in higher categories by doing a probability analysis.

In [48]:
```
ppc_edu_13_2_2_NC['income']
```

Out[48]:
```
array([[ 19, 242, 217, ..., 797,  46, 263],
       [ 37, 217, 238, ..., 832,  48, 248],
       [ 26, 203, 211, ..., 787,  53, 258],
       ...,
       [ 18, 207, 234, ..., 799,  49, 269],
       [ 20, 240, 214, ..., 802,  55, 244],
       [ 31, 187, 213, ..., 770,  54, 266]])
```

In [49]:
```
# Let's test two education categories using the posterior validation plot:
mid_quantile = np.percentile(ppc_edu_13_2_2_NC['income'], 50, axis=0)
total = dataset['total']
mid_pct = mid_quantile/total
mid_pct = mid_pct.values
```

In [50]:
```
# Now let's plot the mid quantile difference between male and female as a function of the educ
ation index:
diffs = [x - mid_pct[i-1] for i, x in enumerate(mid_pct)]
plt.plot(diffs[1::2], 'o')
plt.title('Difference Between Male and Female Advantage Vs. Education Index')
plt.xlabel('Education Index')
plt.ylabel('Probability Difference in Advantage')
plt.show()
```



The above figure supports the hypothesis that males are more advantaged at most of the higher education indices.

B5: Regardless of your answer above, assume the samples are valid and give an analysis of:

1. The posterior probability that males with a Bachelor's degree are more likely to have high incomes than males with a Professional School degree (education categories 3 and 4) [Specifically, how certain is it that one or the other class has a higher chance of having a high income?].
2. The posterior-predictive on the number of female PhDs earning more than 50K in similarly sized samples

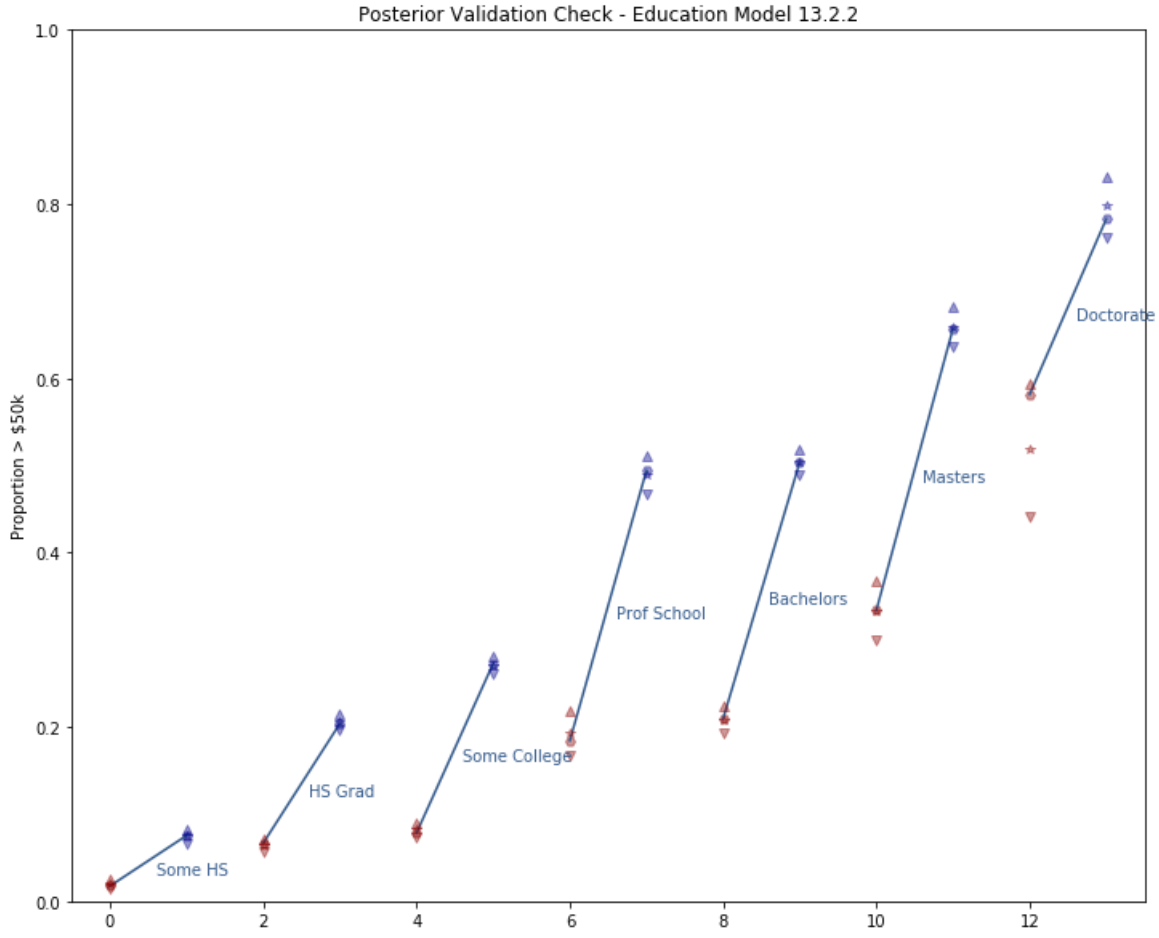
In [51]:
```
# 1. Certainty of one class having chance of higher income
# i) Get the probability values from the posterior

logp3_male = np.mean(traces_13_2_2['alpha_ed'][:,3] + traces_13_2_2['bm_ed'][:,3])
logp4_male = np.mean(traces_13_2_2['alpha_ed'][:,4] + traces_13_2_2['bm_ed'][:,4])

# ii) Get the difference in this log p
print('Posterior Probability of Bachelors>Professional: %.3f%%' % (np.exp(logp4_male - logp3_m
ale)*100))
```

```
Posterior Probability of Bachelors>Professional: 105.714%
```

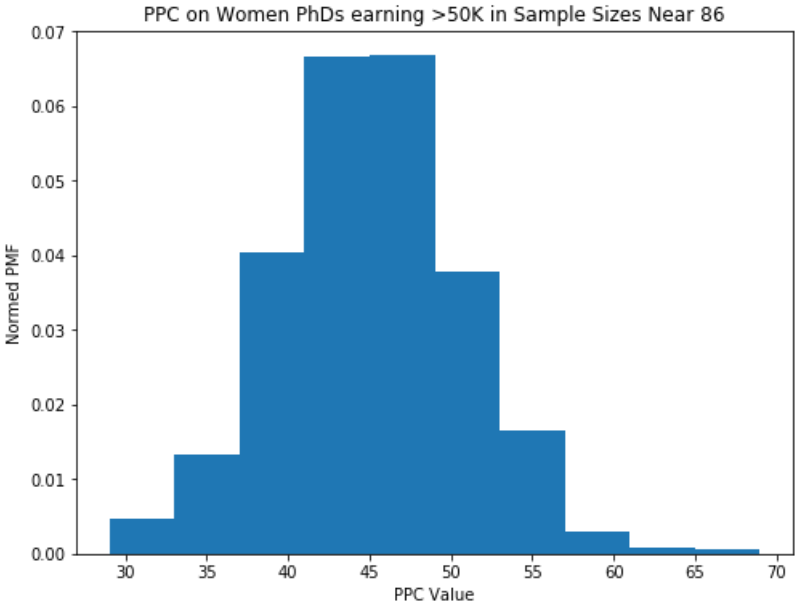
In [52]:
```
# Sample the posterior predictive on the full set of data
with edu_gender_13_2_2:
    ppc_edu_13_2_2 = pm.sample_ppc(traces_13_2_2, samples=1000)
```

```
100%|██████████| 1000/1000 [00:00<00:00, 1349.47it/s]
```

In [53]:
```
fig, ax = plt.subplots(figsize=(12, 10))
plot_post_validation_check(ax, dataset, ppc_edu_13_2_2['income'], labels, "Education Model 13.
2.2")
```



In [54]:
```
# 2. Posterior predictive on number of female PhDs earning more than 50K
cat = 12

# Plot a histogram of the ppc of the women PhDs earning more than 50K
fig = plt.figure(figsize=(8, 6))
plt.hist(ppc_edu_13_2_2['income'][:,cat], normed=True)
plt.title('PPC on Women PhDs earning >50K in Sample Sizes Near 86')
plt.xlabel('PPC Value')
plt.ylabel('Normed PMF')
plt.show()
```



Q2: Using 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 read this data in...

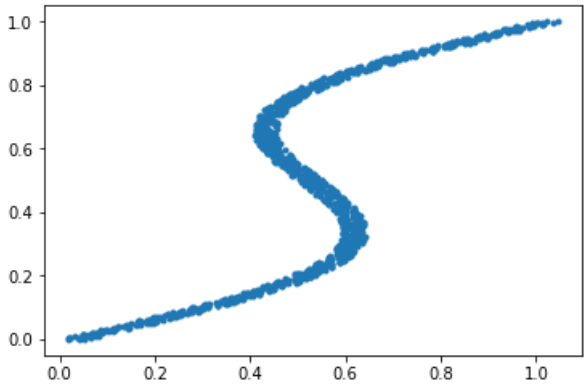
In [55]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
df = pd.read_csv("one-to-many.csv")
df.head()
```

Out[55]:

	target	x
0	0.000000	0.018727
1	0.001001	0.050424
2	0.002002	0.042375
3	0.003003	0.038596
4	0.004004	0.019352

...and then plot it. Notice both the uneven sampling (more towards the center), and the "more than one y" for a given x.

In [56]:
```
plt.plot(df.x, df.target, '.');
```



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.

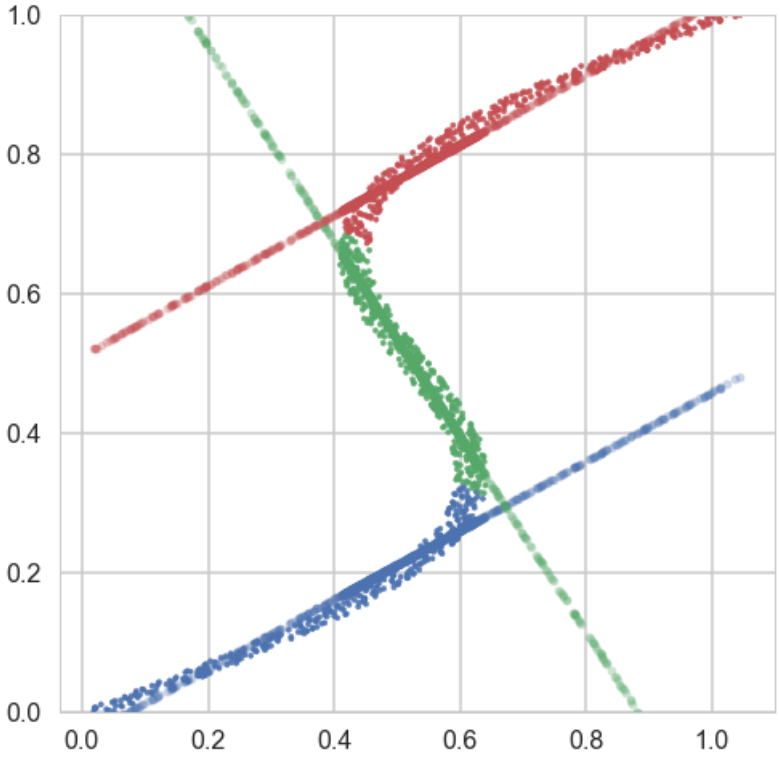
See https://www.cs.toronto.edu/~hinton/absps/hme.pdf (https://www.cs.toronto.edu/~hinton/absps/hme.pdf) and http://www.ee.hacettepe.edu.tr/~eyuksel/Publications/2012_TwentyYearsofMixtureofExperts.pdf (http://www.ee.hacettepe.edu.tr/~eyuksel/Publications/2012_TwentyYearsofMixtureofExperts.pdf) for more 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: Iterative Linear Regression

The basic idea, illustrated below, is to fit piecewise linear regressions iteratively in a EM style algorithm.



The algorithm looks like this: initialize the 1000 points in this dataset as randomly belonging to one of 3 "classes" or "clusters" or "experts". This takes a x-data problem and augments it with a fake z data point that tells us whether the point belongs in cluster 0, 1, or 2. (It helps convergence to say that points in the leftmost 0.2 belong to cluster 0 and righmost 0.2 to cluster 2).

Once we have the fake z data we can separate out the 1000 points into 3 sets of points and fit 3 linear regression models.

We can make predictions for all 1000 points on the data for each of the 3 regression lines. We now assign a data point to the cluster by calculating the squared distances between its actual y value and the 3 cluster predictions and choosing the smallest one.

We then rinse and repeat.

A NOTE FOR THIS PROBLEM:

Occasionally I screw up the coloring for the labels, but a simple switching around of the colors will do the trick and fix the problem! Do to the stochasticity of the pymc3 algorithms, we will occasionally get different assignments of the different regions to different cluster indices, but this does not change the overall results of the problem.

A1 Implement this algorithm along with a convergence criterion of your choice

In [57]:
```
from sklearn import linear_model
from sklearn.metrics import mean_squared_error
import time
```

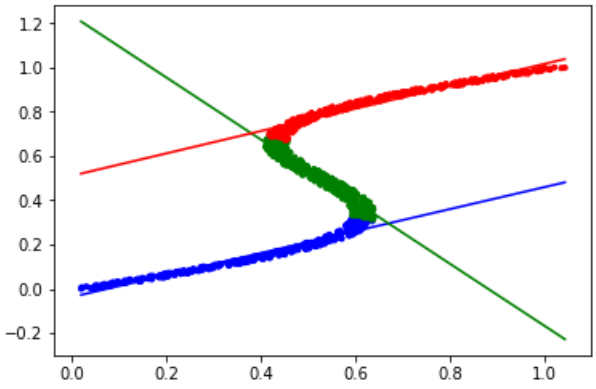
In [58]:
```
# Step 1: Initialize the points randomly with a z value
df['z'] = np.random.randint(0,3,size=(len(df)))

# Force initialization of the leftmost .2 and rightmost .2
df.loc[df.x < 0.3, 'z'] = 0
df.loc[df.x > 0.7, 'z'] = 2

# Step 2: Separate 1000 points into 3 sets of points and fit 3 regression models
cats = [0,1,2]

mse = np.ones((len(df), len(cats)))

mse_new = np.ones((len(df), 1))
mse_old = np.zeros((len(df), 1))

# Loop until convergence:
k=0
while np.abs((np.mean(mse_new) - np.mean(mse_old))/np.mean(mse_old)) > 1e-4:
    mse_old = mse_new
    regr = []

    for i, cat in enumerate(cats):
        # Create linear regression object
        regr.append(linear_model.LinearRegression())

        # Train the model on this category
        regr[i].fit(df.loc[df.z==cat, 'x'].values.reshape(-1, 1), df.loc[df.z==cat,'target'].v
alues.reshape(-1, 1))

        # Step 3: Make predictions on all points for this line
        preds = regr[i].predict(df['x'].values.reshape(-1, 1))
        mse[:,i] = (1/2 * (df['target'].values - preds.reshape(1, -1))**2)

    # Step 3: Get the minimum value of the MSE and reassign the labels
    zvals = np.argmin(mse, axis=1)

    # Get the proper mse for comparison
    mse_new = mse[:, zvals]

    # Reassign dataframe
    df['z'] = zvals

    # Count iterations
    k+=1
```

A2 Produce a diagram like the one above which illustrates the straight line fits as well as the cluster belonging.

In [59]:
```
print('Number of Iterations:', k)
```

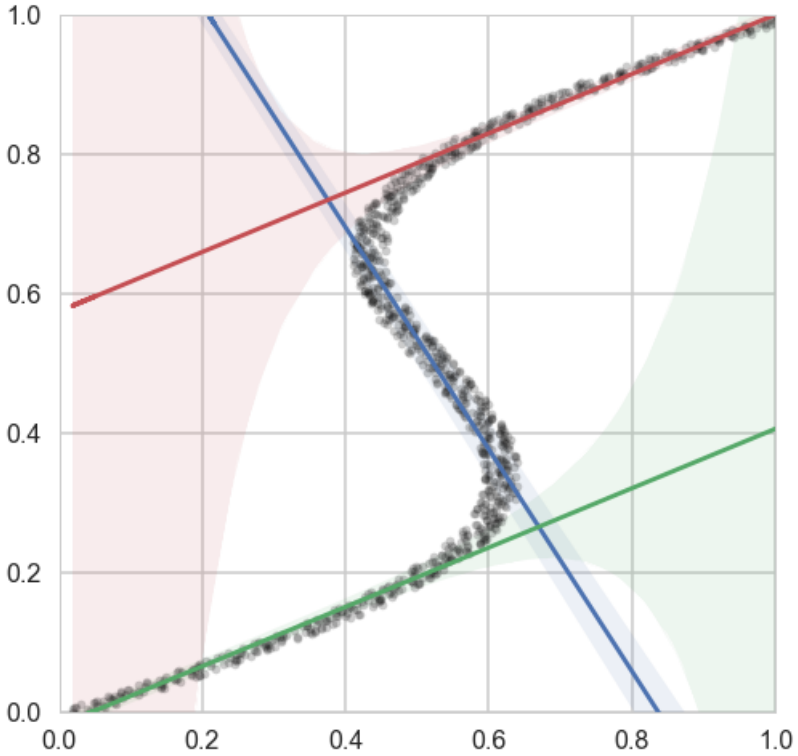
```
Number of Iterations: 8
```

In [60]:
```
# Plot the output classes
colors = ['b','g','r']

cs = [colors[i] for i in df.z.values]

plt.scatter(df.x, df.target, marker='.', c=cs);
# Add regression lines
for cat in cats:
    plt.plot([df.x.min(), df.x.max()], [regr[cat].predict(df.x.min())[0][0], regr[cat].predict
(df.x.max())[0][0]],
             c=colors[cat])
plt.plot()
plt.show()
```



Part B: Variational Mixture of Experts

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



Here again the three regression lines work in different regions of f. We want a principled way to sample from this model and to be able to produce posterior and posterior-predictive regions.

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 (σ needs to be positive).

We now need to model the mixture probabilities, i.e., the probabilities required to choose one or the other gaussian. We'll model these probabilities i.e. the λs using softmax regression (ie do a linear regression and softmax it to get 3 probabilities).

B1 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 σs to make sure you dont have a collapsed 0 width gaussian, ie we want some data in every cluster. (Thus to get the final σ, you will exponentiate your regression for $log(\sigma)$ and add 0.01.)

In [61]:
```
import theano.tensor as T
```

In [62]:
```
# Initial values
Nmix = 3
Npts = len(df)
data = df.x.values.reshape(-1,1)
targ = df.target.values.reshape(-1,1)

# The model
with pm.Model() as mix_exp_1:
    # Means
    w1 = pm.Normal('w1',0,5, shape=(1,Nmix))
    b1 = pm.Normal('b1',0,5, shape=(1,Nmix))
    means = pm.Deterministic('means', pm.math.dot(data,w1) + b1)

    # Log sigmas
    w2 = pm.Normal('w2',0,5, shape=(1,Nmix))
    b2 = pm.Normal('b2',0,5, shape=(1,Nmix))
    logsigs = pm.math.dot(data,w2) + b2
    sigmas = pm.Deterministic('sigmas',np.exp(logsigs) + 0.01)

    # Mixture probabilities
    w3 = pm.Normal('w3',0,5, shape=(1,Nmix))
    b3 = pm.Normal('b3',0,5, shape=(1,Nmix))
    mixprob = pm.Deterministic('mixprob', T.nnet.softmax(pm.math.dot(data,w3) + b3))

    # Likelihood model
    y_obs = pm.NormalMixture('y_obs', w=mixprob, mu=means, sd=sigmas, observed=targ)
```

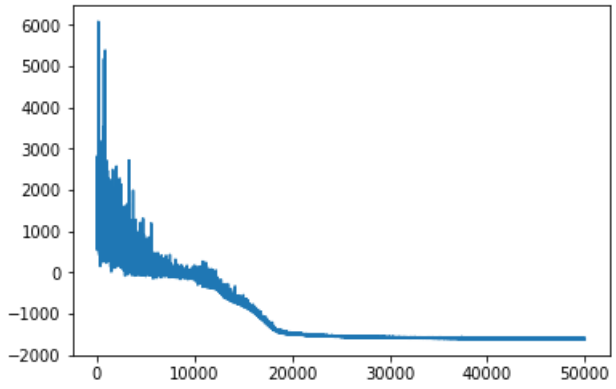
B2 Fit this model variationally for about 50,000 iterations using the adam optimizer. (`obj_optimizer=pm.adam()`) Plot the ELBO to make sure you have converged. Print summaries and traceplots for the means, σs and probabilities.

In [63]:
```
# Sample from this model variationally:
advifit = pm.ADVI( model=mix_exp_1)
advifit.fit(n=50000, obj_optimizer=pm.adam())
```

```
Average Loss = -1,602.2: 100%|██████████| 50000/50000 [01:15<00:00, 661.23it/s]
Finished [100%]: Average Loss = -1,602.2
```

Out[63]:
```
<pymc3.variational.approximations.MeanField at 0x1c20c00080>
```

In [64]:
```
# Plot the ELBO
plt.plot(advifit.hist);
plt.show()
```



In [65]:
```
advifit.approx.shared_params, type(advifit.approx.shared_params)
```

Out[65]:
```
({'mu': mu, 'rho': rho}, dict)
```

In [66]:
```
advifit.approx.mean.eval(), advifit.approx.std.eval()
```

Out[66]:
```
(array([  0.57413719,  -0.01985809,   1.33142905,   1.80657288,
        -10.1136197 ,  -4.33282952,  -4.98127101,   4.40097302,
          0.43948129,   8.42966149,  -9.65745385,  -0.18719354,
          0.4251922 ,   0.42521375,  -1.59251939, -12.14805071,
         10.03671333,   1.47963325]),
 array([ 0.00074848,  0.00074735,  0.00200256,  0.10851681,  0.14583494,
         0.06111863,  0.08722488,  0.088171  ,  0.0697588 ,  0.14664913,
         0.18176122,  0.13172588,  0.00097798,  0.00241191,  0.00388593,
         0.20356999,  0.26899589,  0.10854015]))
```

In [67]:
```
trace = advifit.approx.sample(10000)
```

In [68]:
```
trace.varnames
```

Out[68]:
```
['w1', 'b1', 'w2', 'b2', 'w3', 'b3', 'means', 'sigmas', 'mixprob']
```

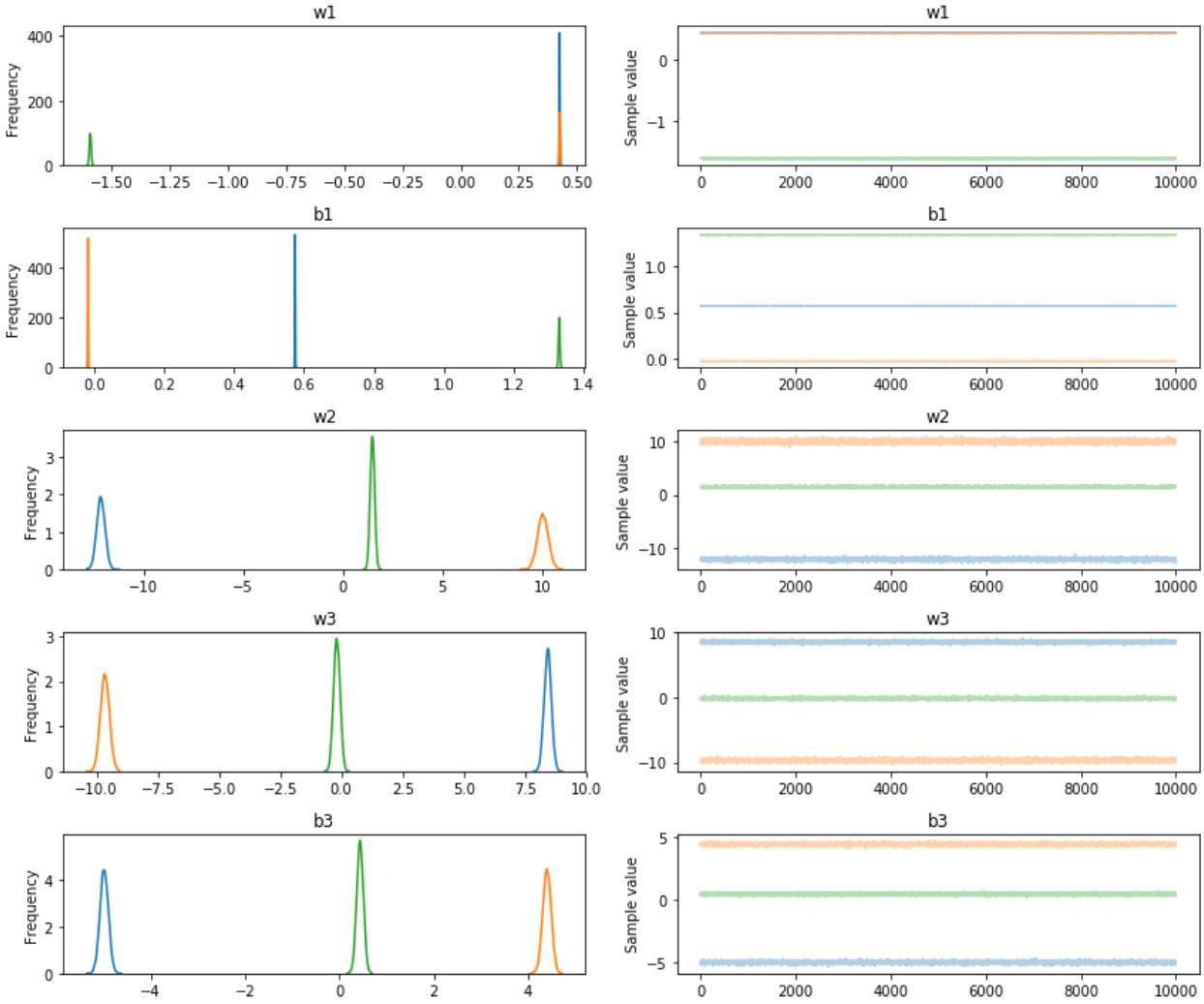
In [69]:
```
# Summaries and traceplots for a few
pm.summary(trace, varnames=['w1','b1','w2','b2','w3','b3'])
```

Out[69]:

	mean	sd	mc_error	hpd_2.5	hpd_97.5
w1__0_0	0.425195	0.000980	0.000010	0.423168	0.427014
w1__0_1	0.425248	0.002376	0.000023	0.420449	0.429840
w1__0_2	-1.592514	0.003876	0.000035	-1.600269	-1.585204
b1__0_0	0.574136	0.000744	0.000007	0.572669	0.575577
b1__0_1	-0.019868	0.000741	0.000007	-0.021308	-0.018415
b1__0_2	1.331435	0.001997	0.000019	1.327607	1.335422
w2__0_0	-12.151831	0.205528	0.002028	-12.556547	-11.752397
w2__0_1	10.038399	0.267112	0.002551	9.480963	10.535999
w2__0_2	1.482613	0.109049	0.001083	1.269466	1.694790
b2__0_0	1.805494	0.109013	0.001115	1.593770	2.025180
b2__0_1	-10.114884	0.145073	0.001484	-10.405707	-9.841834
b2__0_2	-4.333247	0.061415	0.000606	-4.453423	-4.212533
w3__0_0	8.427485	0.146882	0.001449	8.149000	8.725220
w3__0_1	-9.659581	0.181110	0.001640	-10.019787	-9.309528
w3__0_2	-0.187904	0.131151	0.001417	-0.442491	0.062670
b3__0_0	-4.981268	0.087982	0.000894	-5.149354	-4.804861
b3__0_1	4.401042	0.088233	0.000977	4.235116	4.577251
b3__0_2	0.439583	0.069820	0.000760	0.307621	0.577646

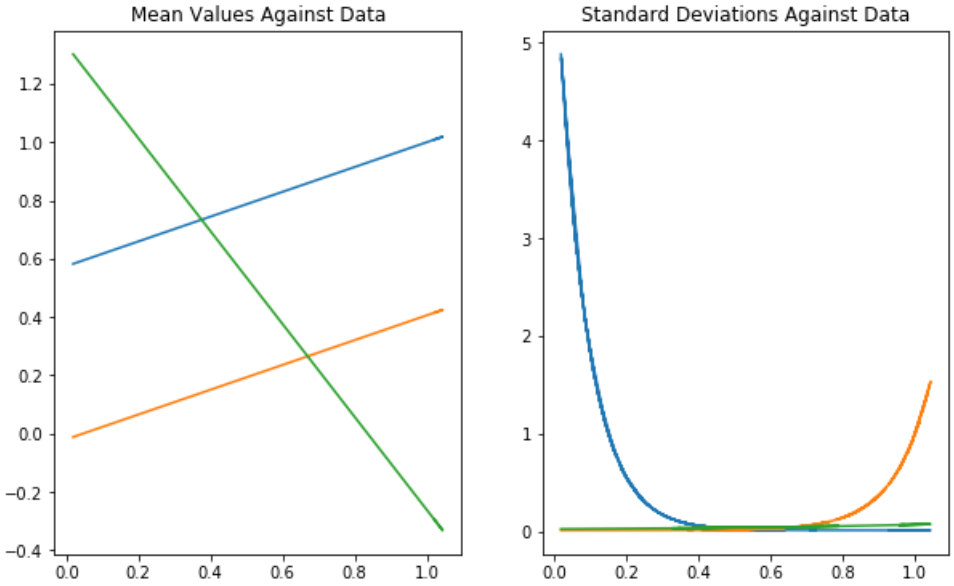
In [70]:
```
pm.traceplot(trace, varnames=['w1','b1','w2','w3','b3'])
```

Out[70]:
```
array([[<matplotlib.axes._subplots.AxesSubplot object at 0x1c2312b518>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x1a0b08b0f0>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x1a0b0bc2b0>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x1a0b0ebdd8>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x1a0b125e48>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x1a0b162a58>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x1a0b19f550>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x1c231665f8>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x1c231a40f0>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x1c231ced68>]], dtype=object)
```

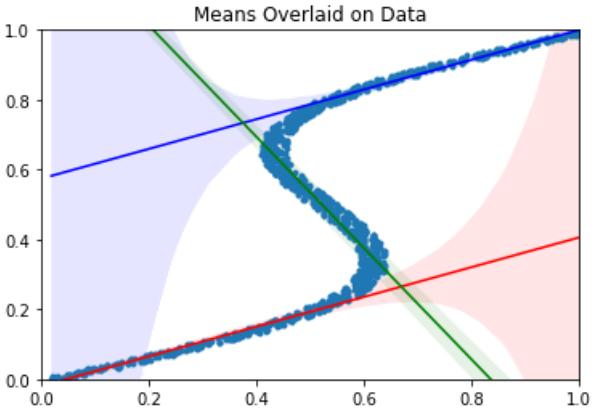


B3 Plot the mean posteriors with standard deviations against x. Also produce a diagram like the one above to show the means with standard deviations showing their uncertainty overlaid on the data.

In [71]:
```
# Plotting mean posteriors and standard deviations against x
means_out = np.mean(trace['means'], axis=0).T
sigmas_out = np.mean(trace['sigmas'], axis=0).T

fig,ax = plt.subplots(1,2, figsize=(10,6))
ax[0].plot(df.x, means_out.T)
ax[0].set_title('Mean Values Against Data')
ax[1].plot(df.x, sigmas_out.T)
ax[1].set_title('Standard Deviations Against Data')
plt.show()
```



In [72]:
```
# Means with standard deviations showing their uncertainty overlaid on the data.
plt.plot(df.x, df.target, '.');
new_colors = ['b','r','g']
for cat in cats:
    plt.plot(df.x, means_out[cat,:], c=new_colors[cat])
    plt.fill_between(df.x, means_out[cat,:] + sigmas_out[cat,:], means_out[cat,:] - sigmas_out
[cat,:],
                     facecolor=new_colors[cat], alpha=0.1)
plt.ylim([0,1.0])
plt.xlim([0,1.0])
plt.title('Means Overlaid on Data')
plt.show()
```

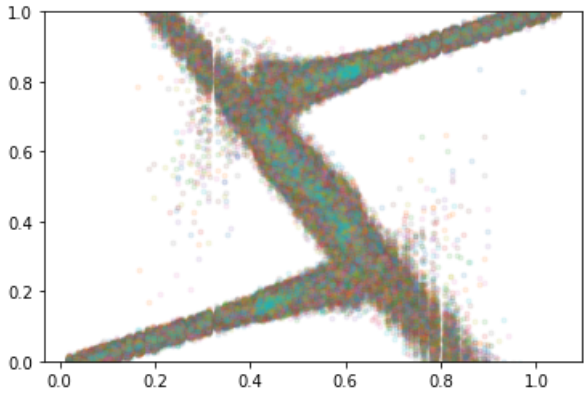


B4 Plot the posterior predictive (mean and variance) as a function of x for this model (using `sample_ppc` for example). Why does the posterior predictive look nothing like the data?

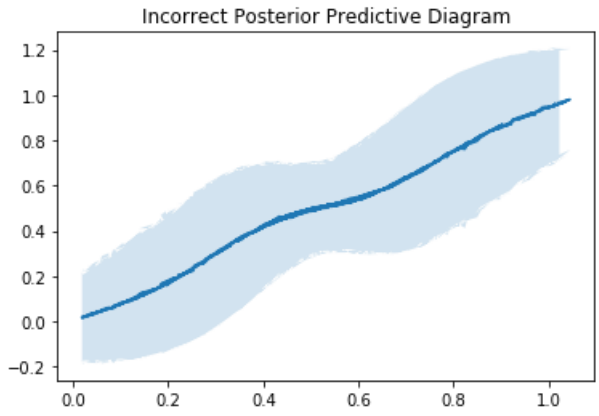
In [73]:
```
# Get ppc samples
with mix_exp_1:
    traces_ppc = pm.sample_ppc(trace, samples=5000)
```

```
100%|██████████| 5000/5000 [03:05<00:00, 26.92it/s]
```

In [74]:
```
plt.plot(df.x, traces_ppc['y_obs'][::10,:].T, '.', alpha=.1)
plt.ylim([0,1.0])
plt.show()
```

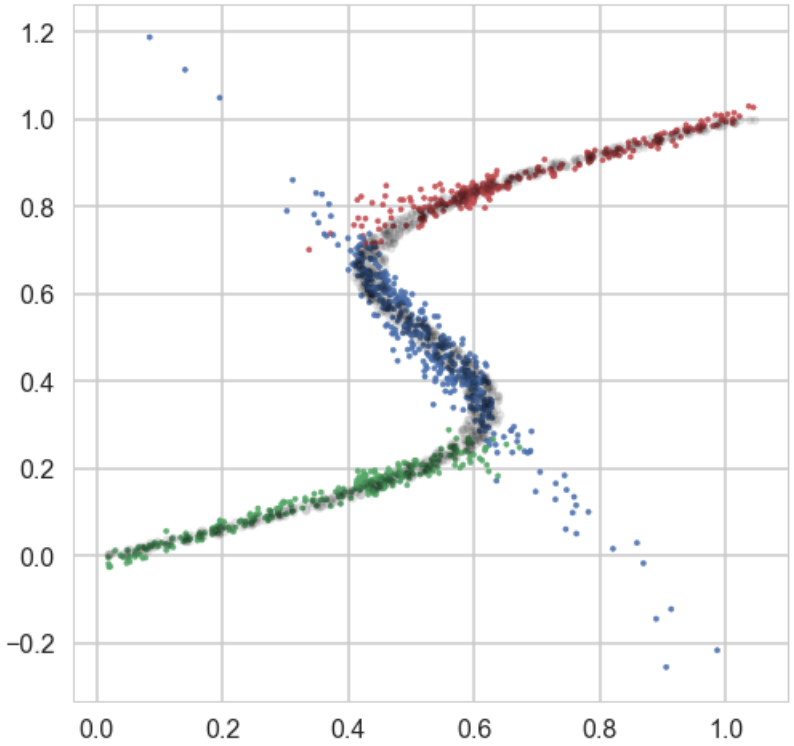


In [75]:
```
# Plotting the posterior predictive
plt.plot(df.x, traces_ppc['y_obs'].mean(axis=0))
plt.fill_between(df.x, traces_ppc['y_obs'].mean(axis=0) + traces_ppc['y_obs'].std(axis=0),
                       traces_ppc['y_obs'].mean(axis=0) - traces_ppc['y_obs'].std(axis=0),
                 alpha=0.2)
plt.title('Incorrect Posterior Predictive Diagram')
plt.show()
```



The reason that this posterior predictive is incorrect is because, predictions are not able to provide a one-to-many output for the ppc samples. It only draws one value at a time for each value of x. As a result, the mean value of the regions where the function behaves as a one-to-many will be the mean of the "many" and will exist somewhere in the middle. As a result, there is no way to get the one-to-many behavior.

B5 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. This diagram may look something like this:

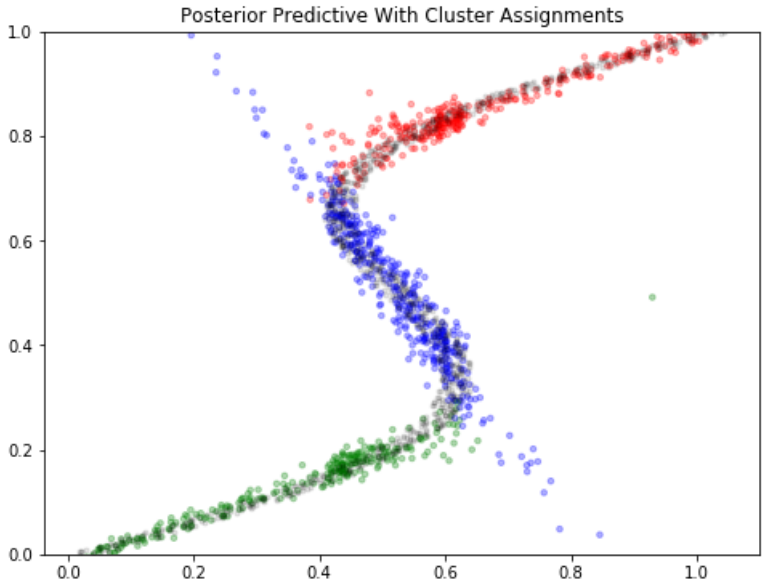


In [76]:
```
# Perform Gumbel softmax sampling.
# Algorithm extracted from: http://timvieira.github.io/blog/post/2014/07/31/gumbel-max-trick/
def gumbel_max_sample(x):
    '''x here is the pre-softmax weighting'''
    z = np.random.gumbel(loc=0, scale=1, size=x.shape)
    return (x + z).argmax(axis=1)
```

In [77]:
```
# Sample pre-softmax weighting using Gumbel Max
idx = np.random.randint(len(trace))
pin = np.dot(data.reshape(-1,1), trace[idx]['w3']) + trace[idx]['b3']
cat_samples = gumbel_max_sample(pin)
```

In [78]:
```
# Based on the category, pick the associated mean and sigma then sample from normal
means_samp = trace[idx]['means'][np.arange(len(cat_samples)),cat_samples]
sigmas_samp = trace[idx]['sigmas'][np.arange(len(cat_samples)),cat_samples]
data = np.random.normal(loc=means_samp, scale=sigmas_samp)
```

In [79]:
```
colors = ['r','g','b']
col = [colors[i] for i in cat_samples]

# Make the plot
fig = plt.figure(figsize=(8,6))
plt.plot(df.x, df.target, 'k.', alpha=.1);
plt.scatter(df.x, data, s=50, marker='.', color=col, alpha=.3 )
plt.ylim([0,1.0])
plt.title('Posterior Predictive With Cluster Assignments')
plt.show()
```



Part C. 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 B4 and B5 above). Thus we'll use the negative loss likelihood of the gaussian mixture model explicitly.

C1: 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]:
```
import torch
import torch.nn as nn
from torch.autograd import Variable
```

In [81]:
```
# CITATION: Code taken from the link provided above:
# https://github.com/hardmaru/pytorch_notebooks/blob/master/mixture_density_networks.ipynb
class MixtureDensityNetwork(nn.Module):
    def __init__(self, n_hidden, n_gaussians):
        super(MixtureDensityNetwork, self).__init__()
        self.z_h = nn.Sequential(
            nn.Linear(1, n_hidden),
            nn.Tanh()
        )
        self.z_pi = nn.Linear(n_hidden, n_gaussians)
        self.z_sigma = nn.Linear(n_hidden, n_gaussians)
        self.z_mu = nn.Linear(n_hidden, n_gaussians)

    def forward(self, x):
        z_h = self.z_h(x)
        pi = nn.functional.softmax(self.z_pi(z_h), -1)
        sigma = torch.exp(self.z_sigma(z_h))
        mu = self.z_mu(z_h)
        return pi, sigma, mu
```

In [82]:
```
# Implement cost functions from:
# https://github.com/hardmaru/pytorch_notebooks/blob/master/mixture_density_networks.ipynb
oneDivSqrtTwoPI  = 1.0 / np.sqrt(2.0*np.pi) # normalization factor for Gaussians
def gaussian_distribution(y, mu, sigma):
    # make |mu|=K copies of y, subtract mu, divide by sigma
    result = (y.expand_as(mu) - mu) * torch.reciprocal(sigma)
    result = -0.5 * (result * result)
    return (torch.exp(result) * torch.reciprocal(sigma)) * oneDivSqrtTwoPI

def mdn_loss_fn(pi, sigma, mu, y):
    result = gaussian_distribution(y, mu, sigma) * pi
    result = torch.sum(result, dim=1)
    result = -torch.log(result)
    return torch.mean(result)
```

In [83]:
```
network = MixtureDensityNetwork(n_hidden=20, n_gaussians=3)
```

In [84]:
```
xdata = np.float32(df.x.values.reshape(-1,1))
ydata = np.float32(targ)

xtensor = torch.from_numpy(xdata)
ytensor = torch.from_numpy(ydata)

xvariable = Variable(xtensor)
yvariable = Variable(ytensor, requires_grad=False)
```

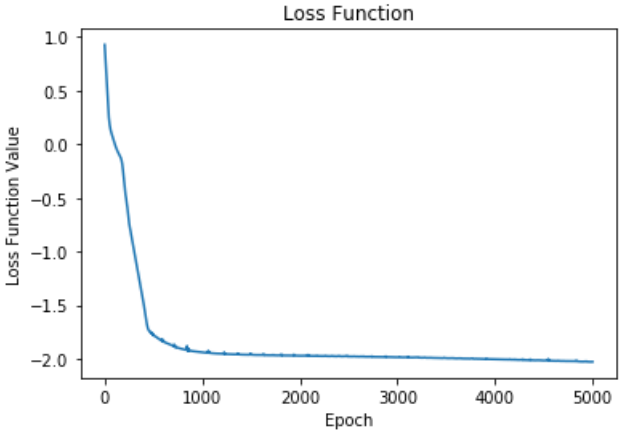
C2: Train the network using the Adam or similiar optimizer. Make sure your loss converges and plot this convergence.

In [85]:
```
optimizer = torch.optim.Adam(network.parameters())
```

In [86]:
```
# Training functions
def train_MDN(epochs):
    loss_accum = []
    for epoch in range(epochs):
        pi_variable, sigma_variable, mu_variable = network(xvariable)
        loss = mdn_loss_fn(pi_variable, sigma_variable, mu_variable, yvariable)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        loss_accum.append(loss.data[0])
        if epoch % 500 == 0:
            print(epoch, loss.data[0])
    return loss_accum

loss_accum = train_MDN(epochs=5000)
```

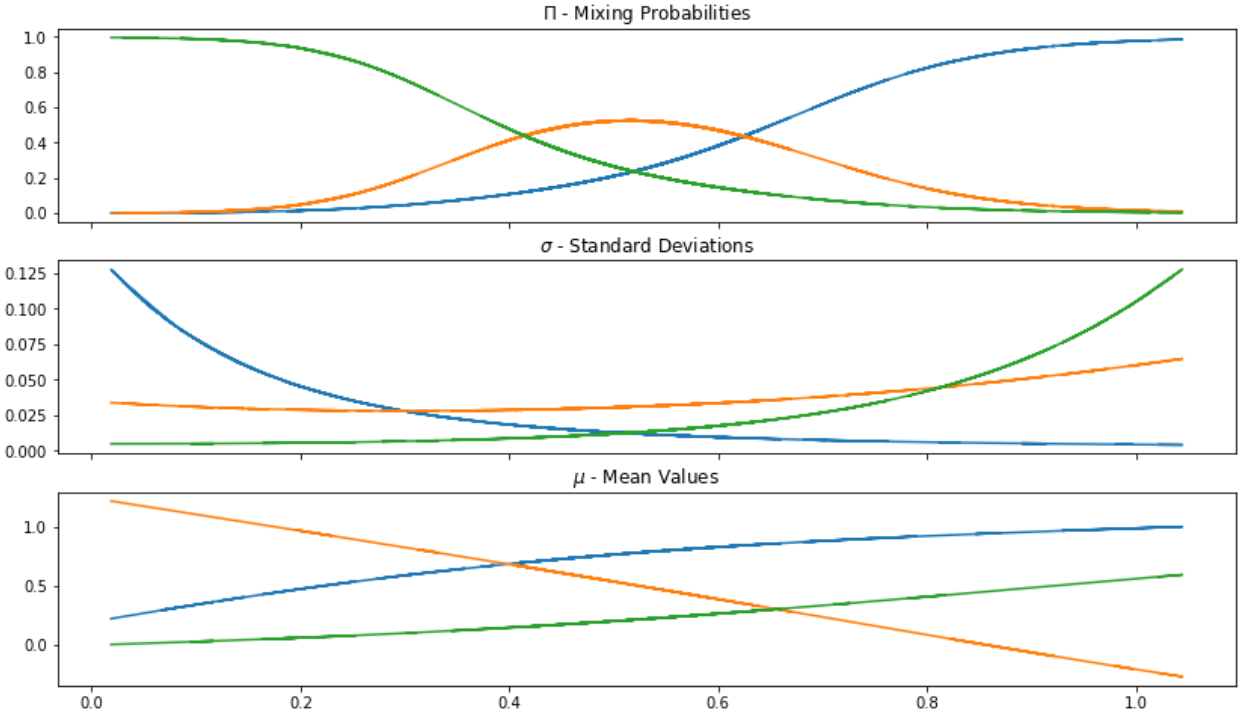
```
0 0.9280447363853455
500 -1.7795270681381226
1000 -1.9409044981002808
1500 -1.9603720903396606
2000 -1.9722895622253418
2500 -1.9782878160476685
3000 -1.9866528511047363
3500 -1.9938396215438843
4000 -2.00559663772583
4500 -2.017199754714966
```

In [87]:
```
# Plot loss function:
plt.plot(loss_accum)
plt.title('Loss Function')
plt.xlabel('Epoch')
plt.ylabel('Loss Function Value')
plt.show()
```

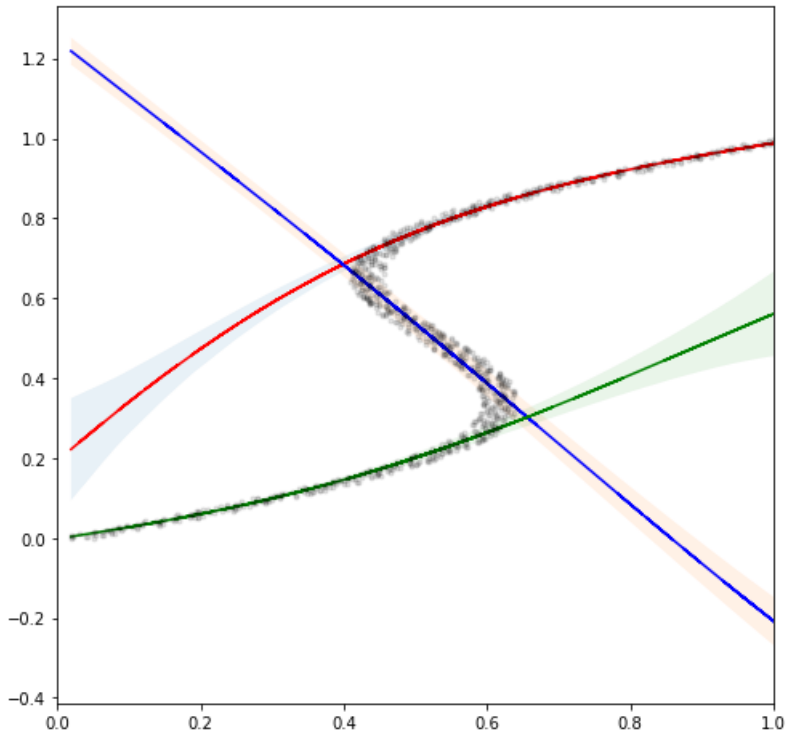


C3: Plot the MLE parameters against x. Make a plot similar to B3 above where you overlay the "means" of the gaussians against the data.

In [88]:
```
pi_variable, sigma_variable, mu_variable = network(xvariable)

pi_data = pi_variable.data.numpy()
sigma_data = sigma_variable.data.numpy()
mu_data = mu_variable.data.numpy()

fig, (ax1, ax2, ax3) = plt.subplots(3, 1, sharex=True, figsize=(14,8))
ax1.plot(xdata, pi_data)
ax1.set_title('$\Pi$ - Mixing Probabilities')
ax2.plot(xdata, sigma_data)
ax2.set_title('$\sigma$ - Standard Deviations')
ax3.plot(xdata, mu_data)
ax3.set_title('$\mu$ - Mean Values')
plt.show()
```



In [89]:
```
# Make plot like B3
colors = ['r','b','g']
plt.figure(figsize=(8, 8), facecolor='white')
for i, (mu_k, sigma_k) in enumerate(zip(mu_data.T, sigma_data.T)):
    plt.plot(xdata.reshape(-1), mu_k, c=colors[i])
    plt.fill_between(xdata.reshape(-1), mu_k-sigma_k, mu_k+sigma_k, alpha=0.1)
plt.plot(df.x, df.target, 'k.', alpha=0.1)
plt.xlim([0.0,1.0])
plt.show()
```

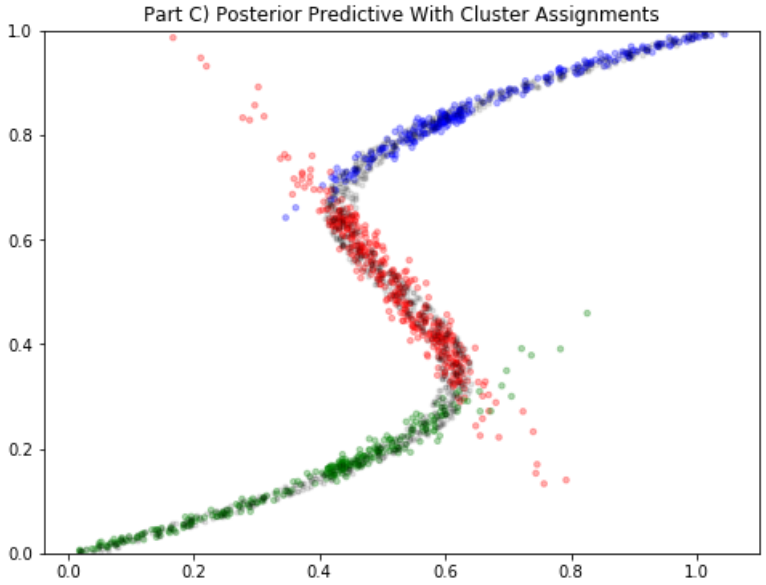


C4: Sample from the sampling distributions to make a plot similar to B5 above

In [90]:
```
# Define another gumbel sample by taking the log of the input weight probability
def gumbel_sample(x, axis=1):
    z = np.random.gumbel(loc=0, scale=1, size=x.shape)
    return (np.log(x) + z).argmax(axis=axis)
```

In [91]:
```
# Get samples
n_samples = len(df.x)
k = gumbel_sample(pi_data)
indices = (np.arange(n_samples), k)
rn = np.random.randn(n_samples)
sampled = rn * sigma_data[indices] + mu_data[indices]
```

In [92]:
```
col = [new_colors[i] for i in k]

# Make the plot
fig = plt.figure(figsize=(8,6))
plt.plot(xdata, ydata, 'k.', alpha=.1);
plt.scatter(xdata, sampled, s=50, marker='.', color=col, alpha=.3 )
plt.ylim([0,1.0])
plt.title('Part C) Posterior Predictive With Cluster Assignments')
plt.show()
```



To think about 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?

Part D Variational Mixture Density Network

We want to implement the Mixture Density Metwork model that we constructed in Part C in 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.

D1: Write out the equivalent pymc3 version of the MDN and generate posterior samples with ADVI.

In [93]:
```
# Pytorch network model
print(network)
```

```
MixtureDensityNetwork(
  (z_h): Sequential(
    (0): Linear(in_features=1, out_features=20)
    (1): Tanh()
  )
  (z_pi): Linear(in_features=20, out_features=3)
  (z_sigma): Linear(in_features=20, out_features=3)
  (z_mu): Linear(in_features=20, out_features=3)
)
```

Reminder that

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
```

In [94]:
```
n_hidden=20 # Number of hidden neurons
n_gaussians=3
with pm.Model() as nn_mixture:
    # Weights from input to hidden layer
    w1 = pm.Normal('w1', 0, 5, shape=(xdata.shape[1],n_hidden) )
    b1 = pm.Normal('b1', 0, 5, shape=(1,n_hidden))
    layer_1 = pm.math.tanh( pm.math.dot(xdata, w1) + b1 )

    # Weights from hidden to output for mean
    w_m = pm.Normal('w_m', 0, 5, shape=(n_hidden,n_gaussians) )
    b_m = pm.Normal('b_m', 0, 5, shape=(1,n_gaussians))

    # Weights for log sigma
    w_s = pm.Normal('w_s', 0, 5, shape=(n_hidden,n_gaussians) )
    b_s = pm.Normal('b_s', 0, 5, shape=(1,n_gaussians))

    # Weights for mixing probabilities
    w_p = pm.Normal('w_p', 0, 5, shape=(n_hidden,n_gaussians) )
    b_p = pm.Normal('b_p', 0, 5, shape=(1,n_gaussians))

    # Getting mu
    mu = pm.Deterministic('mu', pm.math.dot( layer_1, w_m ) + b_m)

    # Getting sigma matrix
    sig = pm.Deterministic('sig', pm.math.exp( pm.math.dot(layer_1,w_s)+b_s)+.01)

    # Getting mixing probabilities
    pmix = pm.Deterministic('pmix', T.nnet.softmax(pm.math.dot(layer_1,w_p)+b_p))

    # Y observed likelihood
    y_obs = pm.NormalMixture('y_obs', w=pmix, mu=mu, sd=sig, observed=ydata)
```

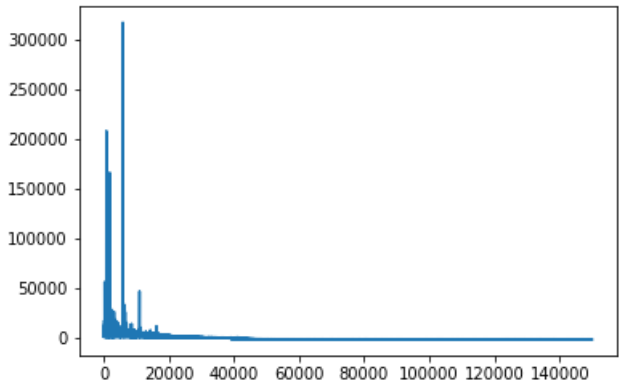
D2: Sample from the posterior predictive as you did in B4 and produce a diagram like C4 and B5 for this model.

In [95]:
```
# Sample from this model variationally:
advifit_2 = pm.ADVI( model=nn_mixture)
advifit_2.fit(n=15000, obj_optimizer=pm.adam())
```

```
Average Loss = -1,289.5: 100%|██████████| 150000/150000 [06:19<00:00, 395.26it/s]
Finished [100%]: Average Loss = -1,289.4
```

Out[95]:
```
<pymc3.variational.approximations.MeanField at 0x1c5bf58fd0>
```

In [96]:
```
# Show ELBO plot for convergence
plt.plot(advifit_2.hist);
plt.show()
```



In [97]:
```
trace_2 = advifit_2.approx.sample(10000)
```

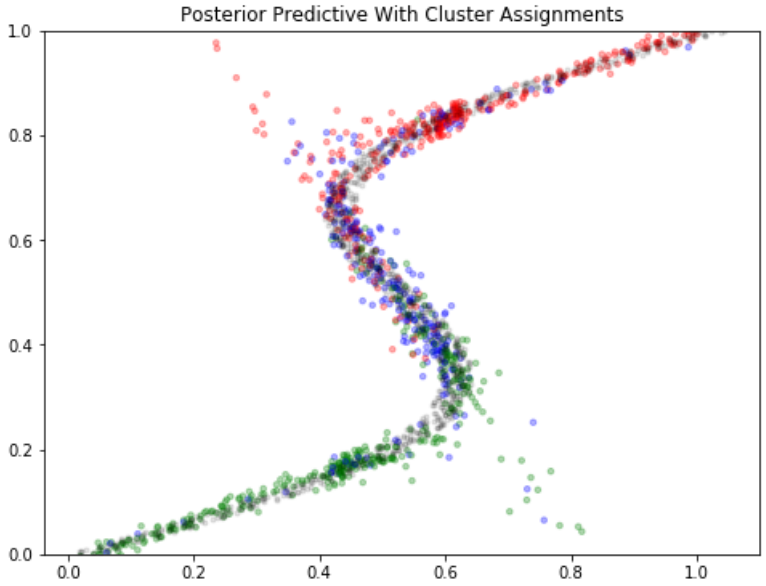
In [98]:
```
# Get ppc samples
with nn_mixture:
    traces_ppc_2 = pm.sample_ppc(trace_2, samples=5000)
```

```
100%|██████████| 5000/5000 [03:01<00:00, 27.50it/s]
```

In [99]:
```
# Sample pre-softmax weighting using Gumbel Max
pin = np.exp(trace_2['pmix'])
cat_samples = gumbel_max_sample(pin)
```

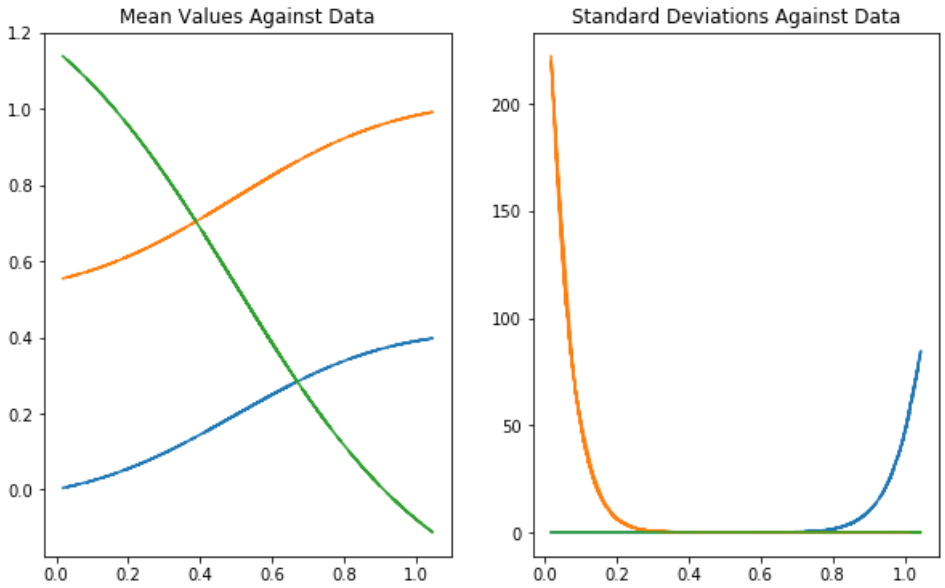
In [100]:
```
# Sample pre-softmax weighting using Gumbel Max
idx = np.random.randint(len(trace))
pin_2 = np.dot(data.reshape(-1,1), trace[idx]['w3']) + trace[idx]['b3']
cat_samples = gumbel_max_sample(pin_2)

# Based on the category, pick the associated mean and sigma then sample from normal
means_samp_2 = trace[idx]['means'][np.arange(len(cat_samples)),cat_samples]
sigmas_samp_2 = trace[idx]['sigmas'][np.arange(len(cat_samples)),cat_samples]
ydata_2 = np.random.normal(loc=means_samp, scale=sigmas_samp)
```

In [101]:
```
colors = ['r','g','b']
col = [colors[i] for i in cat_samples]

# Make the plot
fig = plt.figure(figsize=(8,6))
plt.plot(xdata, ydata, 'k.', alpha=.1);
plt.scatter(xdata, ydata_2, s=50, marker='.', color=col, alpha=.3 )
plt.ylim([0,1.0])
plt.title('Posterior Predictive With Cluster Assignments')
plt.show()
```



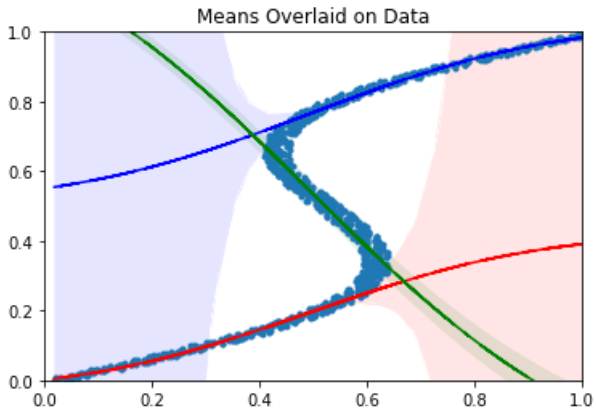
D3: Plot the "mean" regression curves (similar to C3 and B3). Do the "mean" regression curves in this model look the same from those in Part C? If they differ why so?

In [102]:
```
# Plotting mean posteriors and standard deviations against x
means_out_2 = np.mean(trace_2['mu'], axis=0).T
sigmas_out_2 = np.mean(trace_2['sig'], axis=0).T

fig,ax = plt.subplots(1,2, figsize=(10,6))
ax[0].plot(xdata, means_out_2.T)
ax[0].set_title('Mean Values Against Data')
ax[1].plot(xdata, sigmas_out_2.T)
ax[1].set_title('Standard Deviations Against Data')
plt.show()
```



The mean regression curves look slightly different than they did in part c and that is simply a function of the differences in the model specification. In general, they look mostly similar.

In [103]:
```
# Means with standard deviations showing their uncertainty overlaid on the data.
plt.plot(xdata, ydata, '.');
new_colors = ['r','b','g']
for cat in cats:
    plt.plot(xdata, means_out_2[cat,:], c=new_colors[cat])
    plt.fill_between(df.x, means_out_2[cat,:] + sigmas_out_2[cat,:], means_out_2[cat,:] - sigm
as_out_2[cat,:],
                     facecolor=new_colors[cat], alpha=0.1)
plt.ylim([0,1.0])
plt.xlim([0,1.0])
plt.title('Means Overlaid on Data')
plt.show()
```



Part E: Approximate EM for model in part B (EXTRA CREDIT)

Part B can be implemented using Expectation Maximization using iteratively-reweighted least squares. Our method in A was merely an approximation to this process. You can find update equations here (https://people.eecs.berkeley.edu/~jordan/papers/jordan-xu.ps)