PHP 2530: BAYESIAN STATISTICAL METHODS
HOMEWORK III PYTHON CODE APPENDIX

NICK LEWIS

Packages

```
#Problem 1 (BDA 3rd Ed. Exercise 5.3)

### PART A
school_df  = {
       'y': np.array([28,8,-3,7,-1,1,18,12]),
       'sd': np.array([15,10,16,11,9,11,10,18])
                   }
school_df = pd.DataFrame(school_df)
#Number of iterations
N = 1000
#range of tau as described on page 121
tau = np.linspace(start = 0.0001, stop = 40, num = N)
#best guess of mu
mu = np.linspace(start = -30, stop = 30, num = N)


#posterior density for (mu,tau)|y (bottom of page 116)
def school_post(data,sd,a,b):
    '''
    PARAMETERS:
    ----------
        data - estimated means
        sd - accompanying standard deviations
        a - gridspace for mu
        b - grid space for tau

    Returns:
    -------
    logpost: natural logarithm of the unnormalized posterior distribution
    '''

    #this is so we can adapt to grid sampling and single point values
    axes = 0 if type(b)==int else len(b.shape)-1

    loglik = -((data - a)**2 / (2*(b**2 + sd**2))) - 0.5*np.log(b**2 + sd**2)

    #prior distribution used for this problem, and log prior
    prior = 1
    logprior = np.log(prior)/len(data)

    logpost = np.sum(logprior + loglik, axis=axes)
    return logpost
```

```
#posterior distribution
school = school_post(data=school_df["y"].to_numpy(),
                     sd=school_df["sd"].to_numpy(),
                     a=mu[:,None],
                     b=tau[:,None,None])

#subtract to avoid overflow
school = np.exp( school - school.max())
#normalize the posterior
school = school / school.sum( )

#np.repeat repeats the vector; np.tile repeats the entries
mu_grid = np.repeat(mu,len(tau))
tau_grid = np.tile(tau,len(mu))

samples =  np.random.choice(school.size, size=N, p = school.ravel(order="F"))

#add some random jitter so the variables are continous random variables

#step sizes for our grids
d_mu = np.diff(mu)[0]/2
d_tau = np.diff(tau)[0]/2

mu_y = mu_grid[samples] -d_mu + d_mu*rand(N)
tau_y = tau_grid[samples] -d_tau + d_tau*rand(N)

#new updates mu and tau based on posterior draws
def theta_post(x,y,data,sd):
    '''
    PARAMETERS:
        x - posterior draws for mu
        y - posterior draws for tau
        data - estimated means
        sd - accompanying standard deviations

    Returns:
    -------
    Posterior draws theta_j | tau, mu, y for each school
    '''
    V = np.sqrt(1 / ( (1/y)**2 + (1/sd)**2 ) )
    theta_hat = ((x/ y**2) + (data/ sd**2))*(V**2)
    return norm.rvs(loc = theta_hat, scale = V)

#use updates to now make informed draws of theta_j
thetas = theta_post(x = mu_y[:,None],
                    y = tau_y[:,None],
                    data = school_df["y"].to_numpy(),
                    sd = school_df["sd"].to_numpy() )

#Letters for school labels
LETTERS = ["A","B","C","D","E","F","G","H"]
#smoother way to write our for loop for the best probability calculation
Best = [np.mean(thetas[:,i] == thetas.max(axis=1)) for i in range(school_df.shape[0])]
Best = pd.DataFrame(Best, index = LETTERS)
```

```python
Best.columns = ["Probability of Being Best School"]

#nicer way to write our probability matrix
Probs = np.array([np.mean(thetas[:,i] > thetas[:,j]) for
                 i in range(school_df.shape[0]) for j in range(school_df.shape[0])])

#turns array into matrix and turns it into dataframe
Probs = pd.DataFrame(Probs.reshape(school_df.shape[0],school_df.shape[0]),
                             index = LETTERS,
                             columns = LETTERS )

print(Best)
pd.set_option("display.max_columns", 8)
print(Probs)


#PART B
theta_inf = norm.rvs(loc = school_df['y'].to_numpy(),
                     scale = school_df['sd'].to_numpy(),
                     size = (N,school_df.shape[0]))

#nicer way to write our for loop and calculate probability of being best school
Best_inf = [np.mean(theta_inf[:,i] == theta_inf.max(axis=1))
            for i in range(school_df.shape[0])]

Best_inf = pd.DataFrame(Best_inf, index = LETTERS)
Best_inf.columns = ["Probability of Being Best School"]

#nicer way to write our probability matrix
Probs_inf = np.array([np.mean(theta_inf[:,i] > theta_inf[:,j]) for
                 i in range(school_df.shape[0]) for j in range(school_df.shape[0])])

#turns array into matrix and turns it into dataframe
Probs_inf = pd.DataFrame(Probs_inf.reshape(school_df.shape[0],school_df.shape[0]),
                             index = LETTERS,
                             columns = LETTERS )

print(Best_inf)
pd.set_option("display.max_columns", 8)
print(Probs_inf)

##### REPRODUCING CALCULATIONS IN SECTION 5.5 (FIGURES 5.5-5.7)

# Tau posterior Plot

plt.plot(tau,school.sum(axis=1),color="hotpink")
plt.title(r'marginal posterior density $p(\tau|y)$')
plt.xlabel(r'$\tau$')
plt.ylabel(r'$p(\tau | y)$')
plt.show()

# E(theta_j | tau,y) Plot

def expected_tau(x,data,sd):
    '''
    PARAMETERS:
    ----------
        x - grid for tau
        data - estimated means
        sd - accompanying standard deviations

    Returns:
    -------
    E(theta_j | tau, y) for each school
    '''
    mu_hat = np.sum(data/(sd**2 + x**2)) / np.sum(1/(sd**2+x**2))
    V_tau = 1 / ( (1/x)**2 +  (1/sd)**2)
    return  ((mu_hat/ x**2) + (data/ sd**2))*(V_tau)

school_mean = expected_tau(x = tau[:,None],
                    data = school_df["y"].to_numpy(),
                    sd = school_df["sd"].to_numpy() )

plt.figure(figsize = (7,5))
[plt.plot(tau,school_mean[:,j]) for j in range(school_df.shape[0])]
plt.title(r'conditional posterior means of effects '
                  r'$\operatorname{E}(\theta_j|\tau,y)$')
plt.xlabel(r'$\tau$')
plt.ylabel(r'$\operatorname{E}(\theta_j | \tau, y)$')
plt.legend(['School '+ LETTERS[j] for j in range(school_df.shape[0])])
plt.show()


# sd(theta_j | tau,y) Plot

def sd_tau(x,data,sd):
    '''
    PARAMETERS:
    ----------
        x - grid for tau
        data - estimated means
        sd - accompanying standard deviations

    Returns:
    -------
    sd(theta_j | tau,y) for each school
    '''
    mu_hat = 1 / np.sum(1/(sd**2+x**2))
    V = 1 / ( (1/x)**2 +  (1/sd)**2)
    V_tau = (1/x)**2 / ( (1/x)**2 +  (1/sd)**2)
    return  np.sqrt(V + mu_hat*V_tau**2)

school_sd = sd_tau(x = tau[:,None],
                    data = school_df["y"].to_numpy(),
                    sd = school_df["sd"].to_numpy() )

#add little jitter so these are visible
school_sd[:,6] = school_sd[:,6] + 75*np.diff(school_sd[:,6])[0]
```

```
school_sd[:,5] = school_sd[:,5] + 75*np.diff(school_sd[:,5])[0]

plt.figure(figsize = (7,5))
[plt.plot(tau,school_sd[:,j]) for j in range(school_df.shape[0])]
plt.title(r'standard deviations of effects '
                  r'$\operatorname{sd}(\theta_j|\tau,y)$')
plt.xlabel(r'$\tau$')
plt.ylabel( r'$\operatorname{sd}(\theta_j | \tau, y)$')
plt.legend(['School '+ LETTERS[j] for j in range(school_df.shape[0])])
plt.show()
```

Problem 4

```
### PROBLEM 4 (BDA 3rd Ed. Exercise 5.14)

# number of vehicles on residential street with bike lane
#y-bikes for streets w/ bike lanes;v- vehicles for streets w/ bike lanes

res_df = {
       'bikes': np.array([16, 9, 10, 13,19, 20, 18, 17,35, 55]),
       'vehicles': np.array([58,90, 48, 57, 103, 57, 86,112, 273, 64])
                   }

res_df = pd.DataFrame(res_df)
res_df['total'] = res_df['bikes'] + res_df['vehicles']

#Grid for sampling
R = 1000
alpha = np.linspace(start = 0.0001, stop = 17, num = R)
betas = np.linspace(start = 0.0001, stop = 0.6, num = R)

#Posterior Function. It is of form seen in derivation
def vehicle_post(v,a,b):
    '''
    PARAMETERS:
    ----------
    v - data on number of vehicles on residential street
    a - alpha parameter
    b - beta parameter

    Returns:
    -------
    logpost : natural logarithm of unnormalized posterior density
    '''
    #prior distribution used for this problem
    prior = (a+b)**(-5/2)

    dummy = np.array(b) #dummy variable to test axis to take sum over
    axes = len(dummy.shape)-1 if len(dummy.shape) > 1 else 0

    # for brevity, split the likelihood into a numerator term and denominator
    loglik = gammaln(a+v) + a*np.log(b) - ( gammaln(a) + gammaln(v+1)
                                            + (v+a)*np.log(1+b) )
    logprior = np.log(prior)/len(v)
    logpost = np.sum(logprior + loglik, axis=axes)
    return logpost

posty = vehicle_post(v=res_df['total'].to_numpy(),
                     a = alpha[:,None],
                     b= betas[:,None,None])

#subtract to avoid overflow
posty = np.exp( posty - posty.max())
#normalize the posterior
posty = posty / posty.sum( )

#np.repeat repeats the vector; np.tile repeats the entries
alpha_grid = np.repeat(alpha,len(betas))
beta_grid = np.tile(betas,len(alpha))

#posterior draw indices
samples = np.random.choice(posty.size, size=R, p = posty.ravel(order="F"))

#step sizes for random jitter
d_alpha = np.diff(alpha)[0]/2
d_beta = np.diff(betas)[0]/2

#add some random jitter so the variables are continous random variables
alpha_post = alpha_grid[samples] - d_alpha + d_alpha*rand(R)
beta_post = beta_grid[samples] - d_beta + d_beta*rand(R)

#plots contours and simulated points
plt.figure(figsize = (10, 8))

lev = [ 0.001, 0.01,.025,0.05,0.25,0.50,0.75,0.90,0.95]
cont = np.quantile(np.linspace(posty.min(),posty.max(),10000),lev)
plt.contour(alpha, betas, posty, colors='red',levels=cont, zorder = 2)
plt.scatter(alpha_post, beta_post, zorder = 1)
plt.xlabel(r'$\alpha$')
plt.ylabel(r'$\beta$')
plt.ylim(0,.23)
plt.xlim(0,20)
plt.title(r'Posterior Distribution of p($\alpha$, $\beta$ | y)')
plt.show()

# Draw posterior values of the theta_j's

#this will hold ALL OF OUR samples
thetas = gamma.rvs(a = alpha_post[:,None] + res_df['total'].to_numpy(),
                   scale = 1/(beta_post[:,None] + 1),
                  size = (R,10))

#concatenate our hyperparameters and parameters together
param = np.hstack((alpha_post.reshape(R,1),beta_post.reshape(R,1),thetas))

param_names = np.concatenate([ ['\u03B1','\u03B2'],
                        ['\u03B8'+str(j) for j in range(1,res_df.shape[0]+1)]])
```

```
#The posterior distributions of each theta_j.
vehicle_stats  = {
        'Mean': param.mean(axis=0),
        'Standard Deviation': param.std(axis=0,ddof=1),
        '2.5%': np.percentile(param,2.5,axis=0),
        '25%': np.percentile(param,25.0,axis=0),
        '50%': np.percentile(param,50.0,axis=0),
        '75%': np.percentile(param,75.0,axis=0),
        '97.5%': np.percentile(param,97.5,axis=0)
                  }

vehicle_stats = np.round(pd.DataFrame(vehicle_stats,index=param_names),3)
pd.set_option("display.max_columns", 7)
print(vehicle_stats)
```

Problem 5

Python Code

```
#Problem 5 (BDA 3rd. Ed. Exercise 6.2)
#number of fatal accidents between 1976-1985

### STORES OUR INFORMATION

df  = {
       'Accidents':np.array([24, 25, 31, 31, 22, 21, 26, 20, 16, 22]),
       'Deaths': np.array([734, 516, 754, 877, 814, 362, 764, 809, 223, 1066]),
       'Year': np.arange(1,11),
       'Death Rate': np.array([0.19, 0.12, 0.15, 0.16, 0.14,
                               0.06, 0.13, 0.13, 0.03, 0.15])
                  }
df = pd.DataFrame(df)

#Adds new column of miles flown
df['Miles'] = df['Deaths']*1e8 / df['Death Rate']

#Model 1
#number of simulations for our posterior predictive check
N6 = 1000

#prior parameters
prior_shape = 0; prior_rate = 0

#size parameters for negative binomials

sizes =  np.array([df['Accidents'].sum(),df['Deaths'].sum()]) + prior_shape

#corresponding probability parameters
probs1 = np.repeat(df.shape[0]/(df.shape[0]+1+prior_rate),10)
probs2 = df['Miles'].sum()/(df['Miles'].sum()+np.asarray(df['Miles'])+prior_rate)

Model1_draws = nbinom.rvs(size=(N6,10),n = sizes[0], p= probs1)
Model2_draws = nbinom.rvs(size=(N6,10),n = sizes[0], p= probs2)
Model3_draws = nbinom.rvs(size=(N6,10),n = sizes[1], p= probs1)
```

```
Model4_draws = nbinom.rvs(size=(N6,10),n = sizes[1], p= probs2)

# (1) Independent Observations (Autocorrelations)

#Replications
Model1_Stat1 = np.array([sm.tsa.acf(Model1_draws[j,],fft=False)[1]
                    for j in range(N6)])
Model2_Stat1 = np.array([sm.tsa.acf(Model2_draws[j,],fft=False)[1]
                    for j in range(N6)])
Model3_Stat1 = np.array([sm.tsa.acf(Model3_draws[j,],fft=False)[1]
                    for j in range(N6)])
Model4_Stat1 = np.array([sm.tsa.acf(Model4_draws[j,],fft=False)[1]
                    for j in range(N6)])

#Observed Values
acc_obs_stat1 = sm.tsa.acf(df['Accidents'],fft=False)[1]
death_obs_stat1 = sm.tsa.acf(df['Deaths'],fft=False)[1]

fig, [[ax1, ax2],[ax3, ax4]]  = plt.subplots(2, 2,figsize=(10,5))

ax1.hist(x = Model1_Stat1,color='blue', alpha=0.7, rwidth=0.85)
ax1.set_xlabel('Test Statistic')
ax1.set_ylabel('Frequency')
ax1.axvline(acc_obs_stat1, color='black', linewidth=2)
ax1.text(-0.8, 150,'p = ' + str((Model1_Stat1 >= acc_obs_stat1).mean()), fontsize = 18)
ax1.set_title('Model 1')

ax2.hist(x = Model2_Stat1,color='red', alpha=0.7, rwidth=0.85)
ax2.set_xlabel('Test Statistic')
ax2.set_ylabel('Frequency')
ax2.axvline(acc_obs_stat1, color='black', linewidth=2)
ax2.text(-0.2, 150,'p = ' + str((Model2_Stat1 >= acc_obs_stat1).mean()), fontsize = 18)
ax2.set_title('Model 2')

ax3.hist(x = Model3_Stat1,color='yellow', alpha=0.7, rwidth=0.85)
ax3.set_xlabel('Test Statistic')
ax3.set_ylabel('Frequency')
ax3.axvline(death_obs_stat1, color='black', linewidth=2)
ax3.text(0.2, 150,'p = ' + str((Model3_Stat1 >= death_obs_stat1).mean()), fontsize = 18)
ax3.set_title('Model 3')

ax4.hist(x = Model4_Stat1,color='green', alpha=0.7, rwidth=0.85)
ax4.set_xlabel('Test Statistic')
ax4.set_ylabel('Frequency')
ax4.axvline(death_obs_stat1, color='black', linewidth=2)
ax4.text(0, 150,'p = ' + str((Model4_Stat1 >= death_obs_stat1).mean()), fontsize = 18)
ax4.set_title('Model 4')

fig.suptitle('Histogram of Lag-1 Autocorrelation', y = 1.05)
fig.tight_layout()

# (2) No Time Trend Over Time (Spearman Correlation)

Model1_Stat2 = np.array([spearmanr(Model1_draws[j,:],df['Year'])[0]
                    for j in range(N6)])
Model2_Stat2 = np.array([spearmanr(Model2_draws[j,:],df['Year'])[0]
                    for j in range(N6)])
Model3_Stat2 = np.array([spearmanr(Model3_draws[j,:],df['Year'])[0]
                    for j in range(N6)])
Model4_Stat2 = np.array([spearmanr(Model4_draws[j,:],df['Year'])[0]
                    for j in range(N6)])

#Observed Values
acc_obs_stat2 = spearmanr(df['Accidents'],df['Year'])[0]
death_obs_stat2 = spearmanr(df['Deaths'],df['Year'])[0]

fig, [[ax1, ax2],[ax3, ax4]]  = plt.subplots(2, 2,figsize=(10,5))

ax1.hist(x = Model1_Stat2,color='blue', alpha=0.7, rwidth=0.85)
ax1.set_xlabel('Test Statistic')
ax1.set_ylabel('Frequency')
ax1.axvline(acc_obs_stat2, color='black', linewidth=2)
ax1.text(-0.8, 150,'p = ' + str((Model1_Stat2 >= acc_obs_stat2).mean()), fontsize = 18)
ax1.set_title('Model 1')

ax2.hist(x = Model2_Stat2,color='red', alpha=0.7, rwidth=0.85)
ax2.set_xlabel('Test Statistic')
ax2.set_ylabel('Frequency')
ax2.axvline(acc_obs_stat2, color='black', linewidth=2)
ax2.text(-0.2, 150,'p = ' + str((Model2_Stat2 >= acc_obs_stat2).mean()), fontsize = 18)
ax2.set_title('Model 2')

ax3.hist(x = Model3_Stat2,color='yellow', alpha=0.7, rwidth=0.85)
ax3.set_xlabel('Test Statistic')
ax3.set_ylabel('Frequency')
ax3.axvline(death_obs_stat2, color='black', linewidth=2)
ax3.text(0.4, 150,'p = ' + str((Model3_Stat2 >= death_obs_stat2).mean()), fontsize = 18)
ax3.set_title('Model 3')

ax4.hist(x = Model4_Stat2,color='green', alpha=0.7, rwidth=0.85)
ax4.set_xlabel('Test Statistic')
ax4.set_ylabel('Frequency')
ax4.axvline(death_obs_stat2, color='black', linewidth=2)
ax4.text(0.5, 150,'p = ' + str((Model4_Stat2 >= death_obs_stat2).mean()), fontsize = 18)
ax4.set_title('Model 4')

fig.suptitle('Histogram of Spearman Correlations', y = 1.05)
fig.tight_layout()
```

Problem 6

Python Code

```
### PROBLEM 6 (BDA  3rd Ed., Exercise 6.7)

def post_pred(y,N):
  '''
      PARAMETERS:
          y - mean of the data
          N - number of samples in the data

      Returns:
      --------
      test statistic : the absolute value of the maximum from 100 samples
  '''
  p = norm.rvs(size=N,loc = y,scale=np.sqrt(1+1/N))
  return abs(max(p))


sample_max = np.array([post_pred(y=5.1,N=100) for j in range(1000)])

plt.hist(x = sample_max,color='gold', alpha=0.7, rwidth=0.85)
plt.xlabel('Test Statistic')
plt.ylabel('Frequency')
plt.axvline(8.1, color='black', linewidth=2)
plt.text(7, 150,'p = ' + str((sample_max >= 8.1).mean()), fontsize = 18)
plt.title(r'Posterior Predictive $T(y^{rep})$ = '
                  r'$\operatorname{max}_{j} |y_j|$')

def prior_pred(n,A):
  '''
  PARAMETERS:
    n - number of draws
    A - bounds of the uniform prior on theta (i.e. theta ~Unif(-A,A))

  Return:
  -------
   test statistic : the absolute value of the maximum from 100 samples
  '''
  theta = -A + A*rand(n)
  draws = norm.rvs(size = (n,100), loc = theta[:,None], scale = 1)
  reps = abs(draws.max(axis=1))
  return reps

prior_max = prior_pred(n=1000,A=1e5)

plt.hist(x = prior_max,color='gray', alpha=0.7, rwidth=0.85)
plt.xlabel('Test Statistic')
plt.ylabel('Frequency')
plt.axvline(8.1, color='black', linewidth=2)
plt.text(7, 100,'p = ' + str((prior_max >= 8.1).mean()), fontsize = 18)
plt.title(r'Prior Predictive $T(y^{rep})$ = '
                  r'$\operatorname{max}_{j} |y_j|$')
```

Problem 7

Python Code

```
#Problem 7 (BDA 3rd Ed., Exercise 6.9)
#Rat Tumor Example

rat_df  = {
```

```python
        'rats':np.array([
            20, 20, 20, 20, 20, 20, 20, 19, 19, 19, 19, 18, 18, 17, 20, 20, 20,
            20, 19, 19, 18, 18, 25, 24, 23, 20, 20, 20, 20, 20, 20, 10, 49, 19,
            46, 27, 17, 49, 47, 20, 20, 13, 48, 50, 20, 20, 20, 20, 20, 20, 20,
            48, 19, 19, 19, 22, 46, 49, 20, 20, 23, 19, 22, 20, 20, 20, 52, 46,
            47, 24, 14
        ]),
        'tumors': np.array([
             0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  1,  1,  1,
             1,  1,  1,  1,  1,  2,  2,  2,  2,  2,  2,  2,  2,  2,  1,  5,  2,
             5,  3,  2,  7,  7,  3,  3,  2,  9, 10,  4,  4,  4,  4,  4,  4,  4,
            10,  4,  4,  4,  5, 11, 12,  5,  5,  6,  5,  6,  6,  6,  6, 16, 15,
            15,  9,  4
        ])
                    }
rat_df = pd.DataFrame(rat_df)


R2 = 1000

ralpha = np.linspace(start = 0.5, stop = 12, num = R2)
rbeta = np.linspace(start = 3, stop = 53, num = R2)

#of form seen in derivation
def bicycle_post(y,n,a,b):
    '''
    PARAMETERS:
    ----------
        y - data on bicyle proportions on residential street
        n - number of vehicles seen in total
        a - alpha parameter
        b - beta parameter
    Returns:
    -------
        logpost: natural logarithm of unnormalized posterior density
    '''

    #prior distribution used for this problem
    prior = (a+b)**(-5/2)

    dummy = np.array(b) #dummy variable to test axis to take sum over
    axes = len(dummy.shape)-1 if len(dummy.shape) > 1 else 0

    # for brevity, split the likelihood into a numerator term and denominator
    loglik = betaln(a+y,b+(n-y))-betaln(a,b)
    logprior =  np.log(prior)/len(y)
    logpost = np.sum(logprior+loglik,axis=axes)
    return logpost

rat_post = bicycle_post(y = np.asarray(rat_df['tumors']),
                        n = np.asarray(rat_df['rats']),
                        a = ralpha[:,None],
                        b = rbeta[:,None,None])
#subtract to avoid overflow
```

```
rat_post = np.exp( rat_post - rat_post.max())
#normalize the posterior
rat_post = rat_post / rat_post.sum( )


#np.repeat repeats the vector; np.tile repeats the entries
ralpha_grid = np.repeat(ralpha,len(rbeta))
rbeta_grid = np.tile(rbeta,len(ralpha))

samples = np.random.choice(rat_post.size, size=R2, p = rat_post.ravel(order="F"))

#add some random jitter so the variables are continous random variables

d_ralpha = np.diff(ralpha)[0]/2
d_rbeta = np.diff(rbeta)[0]/2

ralpha_post = ralpha_grid[samples] -d_ralpha + (d_ralpha)*rand(R2)
rbeta_post = rbeta_grid[samples] -d_rbeta + (d_rbeta)*rand(R2)

#plots contours and simulated points
lev = [0.001, 0.01,.025,0.05,0.25,0.50,0.75,0.90,0.95]
cont = np.quantile(np.linspace(rat_post.min(),rat_post.max(),100000),lev)
plt.contour(ralpha, rbeta, rat_post,levels=cont, colors='red', zorder = 2)
plt.scatter(ralpha_post, rbeta_post, zorder = 1)
plt.xlabel(r'$\alpha$')
plt.ylabel(r'$\beta$')
plt.title('Posterior Distribution of alpha, beta; Rat Tumors')
plt.show()

theta_r  = beta.rvs(a = ralpha_post[:,None] + (rat_df['tumors']).to_numpy(),
                    b = rbeta_post[:,None] + (rat_df['rats']- rat_df['tumors']).to_numpy(),
                    size = (R2,rat_df.shape[0]))

#The posterior distributions of each theta_j.

tumor_draws = binom.rvs(size=(R2,rat_df.shape[0]),
                        n = np.asarray(rat_df['rats'])[None,:], p = theta_r)

#Define the quantities of interest with the data

# (i) mean
Test1_rat = (tumor_draws / np.asarray(rat_df['rats'])[None,:]).mean(axis=1)
test1_obs_rat = (rat_df['tumors']/rat_df['rats']).mean()

# (ii) max
Test2_rat = (tumor_draws).max(axis=1)  # (ii) max
test2_obs_rat = (rat_df['tumors']).max()

# (iii) # of zeros
Test3_rat = (tumor_draws==0).sum(axis=1)
test3_obs_rat = (rat_df['tumors']==0).sum()


#posterior predictive draws
```

```python
gs = gridspec.GridSpec(10, 10)
gs.update(wspace=0.5)

fig = plt.subplots(figsize=(10,5))
#(i) Mean
ax1 = plt.subplot(gs[:4, :4])
ax1.hist(x = Test1_rat,color='blue',alpha=0.7, rwidth=0.85)
ax1.set_xlabel('Test Statistic')
ax1.set_ylabel('Frequency')
ax1.axvline(test1_obs_rat, color='black', linewidth=2)
ax1.text(0.10, 200, 'p = ' + str( (Test1_rat >= test1_obs_rat).mean() ), fontsize = 12)
ax1.set_title('Mean of Proportions')

#(ii) Max
ax2 = plt.subplot(gs[:4, 5:9])
ax2.hist(x = Test2_rat,color='red',alpha=0.7, rwidth=0.85)
ax2.set_xlabel('Test Statistic')
ax2.set_ylabel('Frequency')
ax2.axvline(test2_obs_rat, color='black', linewidth=2)
ax2.text(22, 150, 'p = ' + str( (Test2_rat >= test2_obs_rat).mean() ), fontsize = 12)
ax2.set_title('Maximum Number of Tumors')

#(iii) # of zeros
ax3 = plt.subplot(gs[6:, 2:6])
ax3.hist(x = Test3_rat,color='green',alpha=0.7, rwidth=0.85)
ax3.set_xlabel('Test Statistic')
ax3.set_ylabel('Frequency')
ax3.axvline(test3_obs_rat, color='black', linewidth=2)
ax3.text(2.5, 200, 'p = ' + str( (Test3_rat >= test3_obs_rat).mean() ), fontsize = 11)
ax3.set_title('Number of Tumor Free Rats')

plt.suptitle('Rat Model Test Statistics', y = 1.05,fontsize=20)
```

Problem 9

Python Code

```python
### Problem 9 (BDA 3rd Ed, Exercise 7.6)

'''
Convenient way to store our info. Can't make this into a dataframe since not
all of lengths are equal.
'''

#county radon measurements(excludes basement measurements)
radon  = {
       'Blue Earth':np.array([5,13,7.2,6.8,12.8,9.5,6,3.8,1.8,6.9,4.7,9.5]),
       'Clay':  np.array([12.9,2.6,26.6,1.5,13,8.8,19.5,9.0,13.1,3.6]),
       'Goodhue': np.array([14.3,7.6,2.6,43.5,4.9,3.5,4.8,5.6,3.5,3.9,6.7])
                  }

#PART A
N9 = 1000
phi = np.linspace(start = -5, stop = 5 , num = N9)
```

```python
#with our drawa od blue earth, we will update our values of phi
def phi_post(y,phi):
    '''
    PARAMETERS:
    ----------
        y-data
        phi - phi parameter
    Returns:
    -------
        logpost: natural logarithm of unnormalized posterior density
    '''
    n = len(y) #shorthand

    log_geo_mean = (phi-1)*(1-(1/n))*np.sum( np.log(y) ) #log geometric mean
    boxcox_y = boxcox(y,phi) #does boxcox transform
    log_boxcox_var = ((n-1)/2)*np.log( boxcox_y.var(ddof=1,axis=0) )
    logprior = np.log(1) + log_geo_mean
    logpost = logprior - log_boxcox_var #log posterior
    return logpost

#For pooled data

def pooled_phi_post(y,phi):
    '''
    PARAMETERS:
    ----------
        y-data (NOTE: data is in dictionary form)
        phi - phi parameter
    Returns:
    -------
        logpost: natural logarithm of unnormalized posterior density
    '''
    y_pooled = np.concatenate([y[x] for x in y.keys()],)
    N = len(y_pooled) #shorthand for pooled length
    n =  np.array([len(y[x]) for x in y.keys()]) #individual data lengths

    log_geo_mean = (phi-1)*(1-(1/N))*np.sum( np.log(y_pooled) ) #log geometric mean

    #does boxcox transform for each set, and takes the variance
    boxcox_y = np.array([boxcox(y[x][:,None],phi).var(ddof=1,axis=0)
                         for x in y.keys()] )

    #finishes calculation of the denominator seen in solutions
    log_boxcox_var = np.array([((n[x]-1)/2)*np.log( boxcox_y[x,:] )
                               for x in range(len(n)) ] )
    log_boxcox_var = log_boxcox_var.sum(axis=0)

    logprior = np.log(1) + log_geo_mean #log of the prior + log geometric mean
    logpost = logprior - log_boxcox_var #log posterior
    return logpost

#this becomes a dataframe that contains all of the posteriors for phi
radon_post = {
        'Blue Earth': phi_post(y = radon["Blue Earth"][:,None], phi = phi),
        'Clay':phi_post(y = radon["Clay"][:,None], phi = phi),
        'Goodhue': phi_post(y = radon["Goodhue"][:,None], phi = phi),
        'Pooled': pooled_phi_post(y = radon, phi = phi)
        }

radon_post = pd.DataFrame(radon_post)

#normalize the columns
radon_post = radon_post.apply(lambda x: np.exp(x)/np.sum(np.exp(x)))

#This is what our plot of p(phi|y) should look like
[plt.plot(phi,radon_post[x]) for x in radon_post.columns]
plt.title("Marginal Posterior of "+r'p($\phi$ |y)'+", Counties")
plt.xlabel(r'$\phi$')
plt.ylabel(r'p($\phi$ |y)')
plt.legend(["Blue Earth County","Clay County","Goodhue County","Pooled"])
plt.show()

#now based on the vector, we draw most likely values of phi
BE_samples = np.random.choice(phi, size=len(phi), replace=True,
                              p = radon_post['Blue Earth'])

#add jitter
d_phi = np.diff(phi)[0]/2

#Posterior Draws
blue_earth_samples += -d_phi + d_phi*rand(len(phi))

### PART B (+ statistics from A)

#now based on the vector, we draw most likely values of phi
pooled_phi = np.random.choice(phi, size=len(phi),
                                      replace=True, p = radon_post['Pooled'])
pooled_phi += -d_phi + d_phi*rand(len(phi))

#Posterior draws for mu, sigma
def norm_draws(y,phi):
    '''
    PARAMETERS:
    ----------
        y - data
        phi - posterior draws from phi | y
    Returns:
    -------
        tuple of (mu,sigma) draws the same length of phi
    '''
    n =  len(y) #individual data lengths
    boxcox_samples = boxcox(y[:,None],phi)

    sample_mean = boxcox_samples.mean(axis=0)
    sample_var = boxcox_samples.var(ddof=1,axis=0)
    sigma = np.sqrt((n-1)*sample_var / chi2.rvs(df = n - 1, size = len(phi)) )
    mu = norm.rvs(loc = sample_mean,scale = sigma/np.sqrt(n),size = len(phi))
    return mu, sigma
```

```python
#statistics for part a model
BE_data = {
        'Phi': BE_samples,
        'Blue Earth mu': norm_draws(radon["Blue Earth"], phi = BE_samples)[0],
        'Blue Earth sigma': norm_draws(radon["Blue Earth"], phi = BE_samples)[1],
        }

BE_data = pd.DataFrame(BE_data)

BE_stats = {
        'Mean': BE_data.mean(axis=0),
        'Standard Deviation': BE_data.std(axis=0,ddof=1),
        '2.5%': BE_data.quantile(0.025,axis=0),
        '25%': BE_data.quantile(0.25,axis=0),
        '50%': BE_data.quantile(0.5,axis=0),
        '75%': BE_data.quantile(0.75,axis=0),
        '97.5%': BE_data.quantile(0.975,axis=0)
        }

BE_stats = pd.DataFrame(BE_stats)
#so we can see all the data
pd.set_option("display.max_columns", 8)
print(BE_stats)

#statistics for part b model

county_stats = {
        'Phi': pooled_phi,
        'Blue Earth mu': norm_draws(radon["Blue Earth"], phi = pooled_phi)[0],
        'Blue Earth sigma': norm_draws(radon["Blue Earth"], phi = pooled_phi)[1],
        'Clay mu': norm_draws(radon["Clay"], phi = pooled_phi)[0],
        'Clay sigma': norm_draws(radon["Clay"], phi = pooled_phi)[1],
        'Goodhue mu': norm_draws(radon["Goodhue"], phi = pooled_phi)[0],
        'Goodue sigma': norm_draws(radon["Goodhue"], phi = pooled_phi)[1]
        }

county_stats = pd.DataFrame(county_stats)

county_stats = {
        'Mean': county_stats.mean(axis=0),
        'Standard Deviation':county_stats.std(axis=0,ddof=1),
        '2.5%': county_stats.quantile(0.025,axis=0),
        '25%':county_stats.quantile(0.25,axis=0),
        '50%': county_stats.quantile(0.5,axis=0),
        '75%':county_stats.quantile(0.75,axis=0),
        '97.5%':county_stats.quantile(0.975,axis=0)
        }

county_stats = pd.DataFrame(county_stats)
#so we can see all the data
pd.set_option("display.max_columns", 8)
print(county_stats)
```

```python
#PART C: Posterior Predictive Simulations

def boxcox_samples(n,y,phi):
    '''
    PARAMETERS:
    ----------
      n - number of samples
      y - data
      phi - posterior draw of phi
    Returns:
    -------
     matrix of replications of original data ( n by length of data)
    '''

    #draw samples
    #First, create your boxcox data
    boxcox_data =  boxcox(y,phi)

    #Second,now get sufficient statistics for the data
    y_mean, y_var = boxcox_data.mean(), boxcox_data.var(ddof=1)
    N = len(y)

    #get our samples of mu, sigma
    sigma = np.sqrt((N-1)*y_var / chi2.rvs(df = N - 1, size = n) )
    mu = norm.rvs(loc = y_mean,scale = sigma/np.sqrt(N),size = n)

    #use samples of mu, sigma to get boxcox draws
    data = norm.rvs(loc = mu[:,None],scale = sigma[:,None],size = (n,N))
    #reverse samples
    invdata = inv_boxcox(data,phi)
    return invdata

#mode of p(phi|y)
pooled_mode = phi[radon_post['Pooled'].argmax()]

#Replications
BlueEarth_reps = boxcox_samples(n=1000,y=radon["Blue Earth"],phi=pooled_mode)
np.sum( np.isnan(BlueEarth_reps) )

Clay_reps = boxcox_samples(n=1000,y=radon["Clay"],phi=pooled_mode)
np.sum(np.isnan(Clay_reps) )

Goodhue_reps = boxcox_samples(n=1000,y=radon["Goodhue"],phi=pooled_mode)
np.sum(np.isnan(Goodhue_reps) )


### TEST STATISTICS

#(i) Maximum
BlueEarth_max = BlueEarth_reps.max(axis=1)
Clay_max = Clay_reps.max(axis=1)
Goodhue_max = Goodhue_reps.max(axis=1)
```

```python
BE_max_obs = radon['Blue Earth'].max()
Clay_max_obs = radon['Clay'].max()
Goodhue_max_obs = radon['Goodhue'].max()

gs = gridspec.GridSpec(10, 10)
gs.update(wspace=0.5)
fig = plt.subplots(figsize=(10,5))

ax1 = plt.subplot(gs[:4, :4])
ax1.hist(x = BlueEarth_max,color='blue',alpha=0.7, rwidth=0.85)
ax1.set_xlabel('Test Statistic')
ax1.set_ylabel('Frequency')
ax1.axvline(BE_max_obs, color='black', linewidth=2)
ax1.text(0.10, 200, 'p = ' + str( (BlueEarth_max >= BE_max_obs).mean() ))
ax1.set_title('Blue Earth County')

ax2 = plt.subplot(gs[:4, 5:9])
ax2.hist(x = Clay_max,color='red',alpha=0.7, rwidth=0.85)
ax2.set_xlabel('Test Statistic')
ax2.set_ylabel('Frequency')
ax2.axvline(Clay_max_obs, color='black', linewidth=2)
ax2.text(22, 150, 'p = ' + str( (Clay_max >= Clay_max_obs).mean() ))
ax2.set_title('Clay County')

ax3 = plt.subplot(gs[6:, 2:6])
ax3.hist(x = Goodhue_max,color='green',alpha=0.7, rwidth=0.85)
ax3.set_xlabel('Test Statistic')
ax3.set_ylabel('Frequency')
ax3.axvline(Goodhue_max_obs, color='black', linewidth=2)
ax3.text(2.5, 200, 'p = ' + str( (Goodhue_max >= Goodhue_max_obs).mean() ))
ax3.set_title('Goodhue County')

plt.suptitle('Radon Boxcox Model Maximum Test Statistic', y = 1.05,fontsize=20)


#(ii) Standard Deviation
BlueEarth_sd = BlueEarth_reps.std(ddof=1,axis=1)
Clay_sd = Clay_reps.std(ddof=1,axis=1)
Goodhue_sd = Goodhue_reps.std(ddof=1,axis=1)

BE_sd_obs = radon['Blue Earth'].std(ddof=1)
Clay_sd_obs = radon['Clay'].std(ddof=1)
Goodhue_sd_obs = radon['Goodhue'].std(ddof=1)

gs = gridspec.GridSpec(10, 10)
gs.update(wspace=0.5)
fig = plt.subplots(figsize=(10,5))

ax1 = plt.subplot(gs[:4, :4])
ax1.hist(x = BlueEarth_sd,color='blue',alpha=0.7, rwidth=0.85)
ax1.set_xlabel('Test Statistic')
ax1.set_ylabel('Frequency')
ax1.axvline(BE_sd_obs, color='black', linewidth=2)
ax1.text(0.10, 200, 'p = ' + str( (BlueEarth_sd >= BE_sd_obs).mean() ))
```

```
ax1.set_title('Blue Earth County')

ax2 = plt.subplot(gs[:4, 5:9])
ax2.hist(x = Clay_sd,color='red',alpha=0.7, rwidth=0.85)
ax2.set_xlabel('Test Statistic')
ax2.set_ylabel('Frequency')
ax2.axvline(Clay_sd_obs, color='black', linewidth=2)
ax2.text(22, 150, 'p = ' + str( (Clay_sd >= Clay_sd_obs).mean() ))
ax2.set_title('Clay County')

ax3 = plt.subplot(gs[6:, 2:6])
ax3.hist(x = Goodhue_sd,color='green',alpha=0.7, rwidth=0.85)
ax3.set_xlabel('Test Statistic')
ax3.set_ylabel('Frequency')
ax3.axvline(Goodhue_sd_obs, color='black', linewidth=2)
ax3.text(2.5, 200, 'p = ' + str( (Goodhue_sd >= Goodhue_sd_obs).mean() ))
ax3.set_title('Goodhue County')

plt.suptitle('Radon Boxcox Model Std. Error Test Statistic', y = 1.05,fontsize=20)
```