HW5

Jinhong Du - 12243476

May 15, 2020

Contents

Problem A: Gibbs Sampling

The Gibbs sampler in https://stephens999.github.io/fiveMinuteStats/gibbs_structure_simple.html assumes that $\Pr(z_i = k) = 0.5$ for each of the two groups/clusters. That is it sets the mixture component weights to 0.5. Your task here is to generalize this to estimate the mixture component weights from the data rather than fixing them to 0.5. Before attempting this you should read carefully through the related example in https://stephens999.github.io/fiveMinuteStats/gibbs2.html.

1. Write out the generalized model (likelihood and priors), introducing a parameter π to denote the mixture component weights $\Pr(z_i = k) = \pi_k$). (Note: in specifying a prior for π you will want to choose it so that you can exploit conjugacy in part 2 below.)

1. Model.

   Let $X = (X_1, \dots, X_n)^\top$ and $Z_i = (Z_1, \dots, Z_n)^\top$ where $X_i \in \{0,1\}^R$ and $Z_i \in \{0,1\}$. Let $P = (P_0, P_1)^\top$ and $P_k = (P_{k1}, \dots, P_{kR})^\top$, where P_{kj} denote the frequency of the "1" allele at locus j in group k ($j = 1, \dots, R$; $k = 0, 1$).

2. Likelihood.

   The likelihood is given by

$$p(X|Z,P) = \prod_{i=1}^n p(X_i|Z_i,P) = \prod_{i=1}^n \prod_{j=1}^R P_{Z_i,j}^{X_{ij}} (1 - P_{Z_i,j})^{1-X_{ij}} \mathbb{1}_{[0,1]}(P_{Z_i,j}).$$

3. Piror.

   We assume that P and Z are a prior independent, i.e., $p(P,Z) = p(P)p(Z)$. For the prior on P we will further assume that the allele frequencies in each group at each locus are independent and $P_{kj} \sim \text{Uniform}(0,1)$, so

$$p(P) = \prod_{k=0}^1 \prod_{j=1}^R p(P_{kj}).$$

   For Z we will assume that the origin of each individual is independent, with probabilities $(1-\pi, \pi)$ of arising from two groups, i.e. $\pi = \mathbb{P}(Z_i = 1)$. So

$$p(Z|\pi) = \prod_{i=1}^n p(Z_i|\pi) = \prod_{i=1}^n \pi^{Z_i}(1-\pi)^{1-Z_i}.$$

   We assume that $\pi \sim \text{Uniform}(0,1)$, i.e. $p(\pi) = \mathbb{1}_{[0,1]}(\pi)$.

2. Provide a mathematical derivation of the conditional distribution $(\pi|P, Z, X)$ under your model and prior in 1. (exploit conjugacy here!).

By Bayes Theorem the posterior distribution is given by

$$p(Z, P, \pi|X) \propto p(X|Z, P, \pi)p(Z, P, \pi) = p(X|Z, P)p(P)p(Z|\pi)p(\pi) = \prod_{i=1}^{n} p(X_i|Z_i, P)p(P)p(Z_i|\pi)p(\pi).$$

To sample from this distribution, we can use a Gibbs sampler for

1. sample from $P|X, Z, \pi$
2. sample from $Z|X, P, \pi$
3. sample from $\pi|X, Z, P$

Since

$$p(X, Z, P, \pi) = p(X|Z, P, \pi)p(Z, P, \pi) = \prod_{i=1}^{n} p(X_i|Z_i, P)p(P)p(Z_i|\pi)p(\pi),$$

the full conditional distributions are given by

$$\begin{aligned}
p(P|X, Z, \pi) &\propto p(X, Z, P, \pi) = \prod_{i=1}^{n} p(X_i|Z_i, P)p(P)p(Z_i|\pi)p(\pi) \\
&\propto \prod_{i=1}^{n}\prod_{j=1}^{R}\left[P_{Z_i,j}^{X_{ij}}(1 - P_{Z_i,j})^{1-X_{ij}}\left(\prod_{k=0}^{1} \mathbb{1}_{[0,1]}(P_{kj})\right)\right]\mathbb{1}_{[0,1]}(\pi) \\
p(Z|X, P, \pi) &\propto p(X, P, Z, \pi) = \prod_{i=1}^{n} p(X_i|Z_i, P)p(P)p(Z_i|\pi)p(\pi), \\
&\propto \prod_{i=1}^{n}\left[\pi^{Z_i}(1 - \pi)^{1-Z_i}\prod_{j=1}^{R}\left(P_{Z_i,j}^{X_{ij}}(1 - P_{Z_i,j})^{1-X_{ij}}\prod_{k=0}^{1}\mathbb{1}_{[0,1]}(P_{kj})\right)\right]\mathbb{1}_{[0,1]}(\pi) \\
p(\pi|X, Z, P) &\propto p(X, P, Z, \pi) = \prod_{i=1}^{n} p(X_i|Z_i, P)p(P)p(Z_i|\pi)p(\pi) \\
&\propto \prod_{i=1}^{n}\left(\pi^{Z_i}(1 - \pi)^{1-Z_i}\prod_{j=1}^{R}\prod_{k=0}^{1}\mathbb{1}_{[0,1]}(P_{kj})\right)\mathbb{1}_{[0,1]}(\pi)
\end{aligned}$$

So, when provided that the conditions are meaningful ($P_{kj}, \pi \in [0, 1]$), we have

$$\begin{aligned}
P_{kj}|X, Z_i, \pi &\sim \text{Beta}\left(1 + \sum_{i=1}^{n} X_{ij}\mathbb{1}_{\{Z_i=k\}}, 1 + \sum_{i=1}^{n}(1 - X_{ij})\mathbb{1}_{\{Z_i=k\}}\right) \\
\pi|X, Z, P &\sim \text{Beta}\left(1 + \sum_{i=1}^{n} Z_i, n + 1 - \sum_{i=1}^{n} Z_i\right)
\end{aligned}$$

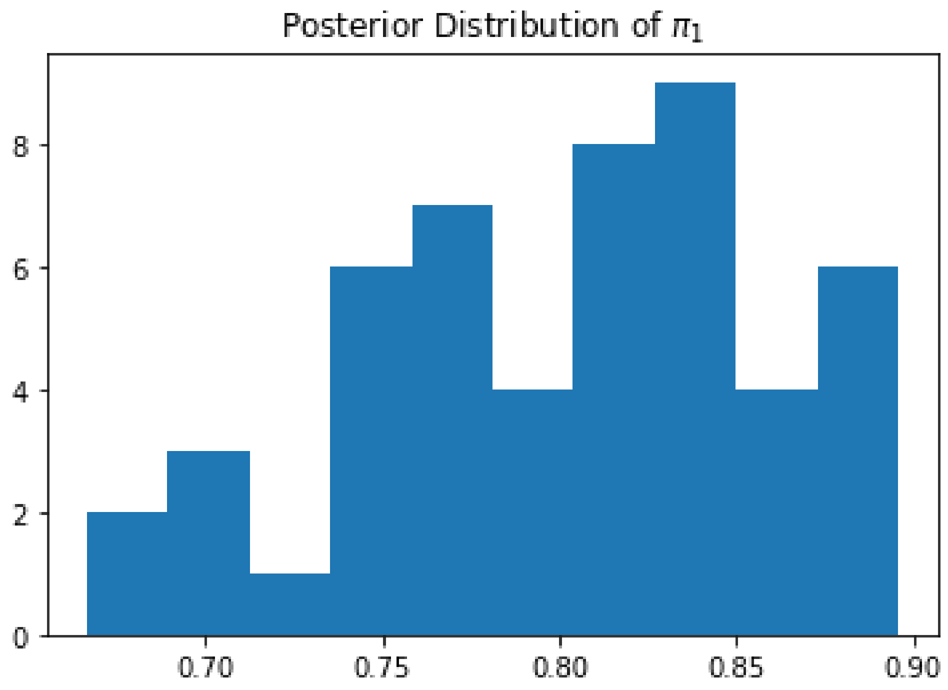
and 0 otherwise. As for $Z_i|X, P, \pi$, it does not have a simple distribution, but we can calculate the probability mass function since it is a discrete distribution.

3. Modify the code of the Gibbs sampler to update π as well as Z and P.

[1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

def generate_data(n,P,pi):
    '''
    Params:
        n  - number of observation
        P  - [2,R] array of frequencies
        pi - P(Z_i=1)
    Returns:
        Z  - [n, ] array of latent variables
        X  - [n,R] array of observations
    '''
    R = P.shape[1]
    Z = np.random.binomial(size=n, n=1, p=pi)
    X = np.random.binomial(size=(n,2,R), n=1, p=P)[np.arange(n), Z, :]
    return Z, X

def sample_P(X):
    '''
    Sample from P|X,Z,pi=P|X
    Params:
        X  - [n,R] array of observation
    Returns:
        P  - [2,R] array of frequencies
    '''
    R = X.shape[1]
    P = np.zeros((2,R))
    for k in range(2):
        P[k,:] = np.random.beta(size=R,
                                a=1+np.sum(X[Z==k,:], axis=0),
                                b=1+np.sum(1-X[Z==k,:], axis=0))
    return P

def sample_Z(X, P, pi):
    '''
    Sample from Z|X,P,pi
    Params:
        X  - [n,R] array of observation
        P  - [2,R] array of frequencies
        pi - P(Z_i=1)
    Returns:
        Z  - [n, ] array of latent variables
    '''
    # [n,2]
    p_X_ZP = np.exp(np.sum(
        np.tile(np.expand_dims(X, 1), (1,2,1)) * np.log(P) +
        np.tile(np.expand_dims(1-X, 1), (1,2,1)) * np.log(1-P),
        axis=-1)) * np.array([1-pi,pi])
    p_X_ZP = p_X_ZP/np.sum(p_X_ZP, axis=1, keepdims=True)
    Z = np.random.binomial(size=X.shape[0], n=1, p=p_X_ZP[:,1])
    return Z

def sample_pi(Z):
    '''
    Sample from pi|X,Z,P=pi|Z
    Params:
        Z  - [n, ] array of latent variables
    Returns:
        pi - P(Z_i=1)
    '''
    pi = np.random.beta(a=1+np.sum(Z), b=1+np.sum(1-Z))
    return pi

def gibbs(X, n_iter=100):
    n, R = X.shape

    res = {}
    res['Z'] = np.zeros((n_iter, n))
    res['P'] = np.zeros((n_iter, 2, R))
    res['pi'] = np.zeros(n_iter)

    Z = np.random.binomial(1, 0.5, size=n)
    pi = np.random.uniform()
    for i in range(n_iter):
        res['P'][i] = P = sample_P(X)
        res['Z'][i] = Z = sample_Z(X,P,pi)
        res['pi'][i] = pi = sample_pi(Z)
    return res
```

4. Illustrate your Gibbs sampler on simulated data where $\pi = (0.2, 0.8)$. Compare the posterior distribution you get from the sampled values of π with the true value of π.

[2]:
```
np.random.seed(1)
n = 50
R = 6
P = np.vstack((np.ones(R)*0.5, np.tile([0.001,0.009], 3)))
pi = 0.8

Z,X = generate_data(n,P,pi)
res = gibbs(X, n_iter=100)

plt.hist(res['pi'][50:])
plt.title('Posterior Distribution of $\pi_1$')
plt.show()
```



[3]:
```
np.mean(res['pi'][50:])
```

[3]:
```
0.8001124686166173
```

We can see that the posterior mean of π is near to the true value 0.8.

Problem B: Fully Bayes Normal Means

In a previous homework you implemented Empirical Bayes (EB) shrinkage for the normal means problem with a normal prior. That is we have data $X = (X_1, \dots, X_n)$: $X_j|\theta_j, s_j \sim \mathcal{N}(\theta_j, s_j^2)$ and assume $\theta_j|\mu, \sigma \sim \mathcal{N}(\mu, \sigma^2)$ $j = 1, \dots, n$. The EB approach involved two steps:

1. Estimates (μ, σ) by maximizing the log-likelihood $l(\mu, \sigma) = \log p(X|\mu, \sigma)$.
2. Compute the posteriore distribution $p(\theta_j|\hat{\mu}, \hat{\sigma})$.

The EB approach can be criticized for ignoring uncertainty in the estimates of μ and σ. Here we will use MCMC to do a fully Bayesian analysis that takes account of this uncertainty.

To make this easier we will first re-parameterize to use $\eta = \log(\sigma)$, so η can take any value on the real line.

We will use a uniform prior on (μ, η), $p(\mu, \eta) \propto 1$ in the range $\mu = [-a, a]$ and $\eta \in [-b, b]$. You can use $a = 10^6$ and $b = 10$. (Because η is on the log scale, $b = 10$ covers a wide range of possible standard deviations). Thus the posterior distribution on (μ, η) is given by

$$p(\mu, \eta|X) \propto p(X|\mu, \eta)\mathbb{1}_{[-a,a]}(\mu)\mathbb{1}_{[-b,b]}(\eta),$$

where $\mathbb{1}$ denotes an indicator function.

1. Modify your log-likelihood computation code from your previous homework to compute the log-likelihood for (μ, η) given data X (and standard deviations s).

Since

$$\begin{aligned}
p(X_j|\mu, \sigma, s_j) &= \int_{\mathbb{R}} p(X_j, \theta_j|\mu, \sigma, s_j)\mathrm{d}\theta_j \\
&= \int_{\mathbb{R}} p(X_j|\theta_j, s_j)p(\theta_j|\mu, \sigma)\mathrm{d}\theta_j \\
&= \int_{\mathbb{R}} \frac{1}{2\pi s_j \sigma} e^{-\frac{1}{2s_j^2}(X_j-\theta_j)^2} e^{-\frac{1}{2\sigma^2}(\theta_j-\mu)^2}\mathrm{d}\theta_j \\
&= \frac{1}{2\pi s_j \sigma} e^{-\frac{1}{2s_j^2}X_j^2 - \frac{1}{2\sigma^2}\mu^2 + \frac{1}{2s_j^2\sigma^2(s_j^2+\sigma^2)}(X_j\sigma^2+\mu s_j^2)^2} \int_{\mathbb{R}} e^{-\frac{1}{2}\frac{s_j^2+\sigma^2}{s_j^2\sigma^2}\left(\theta_j - \frac{X_j\sigma^2+\mu+s_j^2}{s_j^2+\sigma^2}\right)^2} \mathrm{d}\theta_j \\
&= \frac{1}{\sqrt{2\pi(s_j^2+\sigma^2)}} e^{-\frac{1}{(s_j^2+\sigma^2)}(X_j-\mu)^2}
\end{aligned}$$

i.e., $X_j|\mu, \sigma, s_j \sim \mathcal{N}(\mu, s_j^2 + \sigma^2)$, the posterior distribution of (μ, η) is

$$p(\mu, \eta|X_j, s_j) \propto p(X_j|\mu, \eta, s_j)p(\mu, \eta) = p(X_j|\mu, \sigma, s_j)\mathbb{1}_{[-a,a]}(\mu)\mathbb{1}_{[-b,b]}(\eta).$$

So the log-likelihood of $(\mu, \sigma)|X_j, s_j$ is given by

$$\begin{aligned}
l(\mu, \sigma) &= \sum_{j=1}^n \log p(\mu, \eta|X_j, s_j) \\
&= \sum_{j=1}^n \left[-\frac{1}{2}\log(2\pi) - \frac{1}{2}\log(s_j^2+\sigma^2) - \frac{1}{2(s_j^2+\sigma^2)}(X_j-\mu)^2\right]\mathbb{1}_{[-a,a]}(\mu)\mathbb{1}_{[-b,b]}(\eta).
\end{aligned}$$

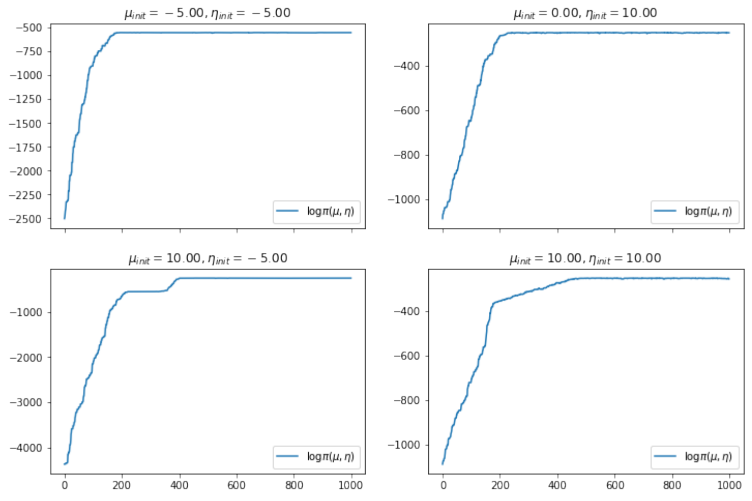
[4]:
```python
from scipy.stats import norm

def log_likelihood(params, x, s, a=1e6, b=10):
    '''
    Evaulate log-likelihood l(mu,eta).
    Params:
        pars - params to be optimized
        x    - observation
        s    - posterior standard deviation of x
    Returns:
    '''
    mu, eta = params
    if np.abs(mu)>a or np.abs(eta)>b:
        l = - np.Inf
    else:
        l = np.sum(norm.logpdf(x, mu, np.sqrt(s**2+np.exp(eta)**2)))
    return l
```

2. Use this to implement a MH algorithm to sample from $\pi(\mu, \eta) = p(\mu, \eta|X)$. Note: In computing the MH acceptance probability you need to compute a ratio $\frac{L_1}{L_2}$. For numerical stability reasons you should always compute this ratio by $\exp(l_1 - l_2)$ where $l_i = \log(L_i)$ rather than directly computing L_1 and L_2 and then computing their ratio. (If both L_1 and L_2 are very small, they may be 0 to machine precision, which causes problems if you try to compute $\frac{L_1}{L_2}$ directly.)

[5]:
```python
def MH(x, s, n_iter, init_vals, proposal_sd):
    '''
    Params:
        x           - [n, ] observations
        s           - [n, ] standard deviation of x
        n_iter      - number of iterations
        init_vals   - [2, ] initial values of (mu, eta)
        proposal_sd - [2, ] standard deviation of the proposal distribution
    Returns:
        res         - [n_iter, 3] chain of sampled (mu,eta,l)
    '''
    res = np.zeros((n_iter, 3))
    current_vals = init_vals
    for i in range(n_iter):
        new_vals = np.random.normal(size=2, loc=current_vals, scale=proposal_sd)
        A = np.exp(log_likelihood(new_vals,x,s) -␣
 ↪log_likelihood(current_vals,x,s))
        res[i,:2] = new_vals if np.random.uniform()<A else current_vals
        res[i,2] = log_likelihood(res[i,:2],x,s)
        current_vals = res[i,:2]
    return res
```

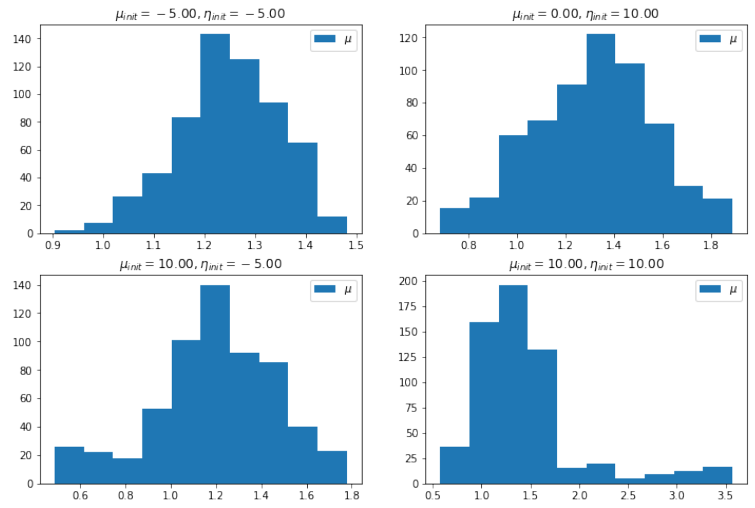
3. Apply your MH algorithm to simulated data where you know the answer. Run you MH algorithm multiple (at least 3) times from multiple different initializations. For each run plot how the value of $\log \pi(\mu_t, \eta_t)$ changes with iteration t. You should see that it starts from a low value (assuming you initialized to something that is not consistent with the data) and then gradually increases until it settles down to a "steady state" behavior. Use these plots to help decide how many iterations to run your algorithm to get reliable results (ie so results from different runs look similar) and how many iterations to discard as "burn-in". Compare your posterior distributions of μ and η with the true values you simulated (the distributions should cover the true values unless you did something wrong or are unlucky!)

[6]:
```python
np.random.seed(0)
mu = eta = 1.0
n = 100
n_iter = 1000
theta = np.random.normal(mu,np.exp(eta),n)
s = np.ones(n)
x = np.random.normal(theta,s,n)
init_vals_list = [[-5., -5.], [0.,10.], [10., -5.], [10., 10.]]
res_list = []
for i in range(4):
    res = MH(x, s, n_iter, init_vals_list[i], [0.1,0.1])
    res_list.append(res)
fig, axes = plt.subplots(2,2,sharex='all',figsize=(12,8))
for i in range(2):
    for j in range(2):
        axes[i][j].plot(res_list[i*2+j][:,2], label='$\log\pi(\mu,\eta)$')
        axes[i][j].set_title('$\mu_{init}=%.2f,  \eta_{init}=%.2f$'%(
            init_vals_list[i*2+j][0], init_vals_list[i*2+j][1]))
        axes[i][j].legend()
plt.show()
```

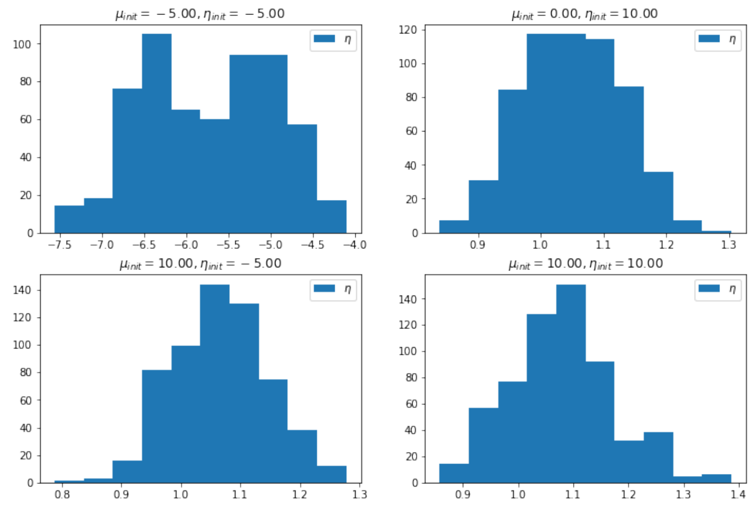


As we can see, the curves get stable after about 500 steps. So it is ok to run with 1000 steps and burn in the first 500 samples.

[7]:
```
fig, axes = plt.subplots(2,2,figsize=(12,8))
for i in range(2):
    for j in range(2):
        axes[i][j].hist(res_list[i*2+j][500:,0], label='$\mu$')
        axes[i][j].set_title('$\mu_{init}=%.2f,  \eta_{init}=%.2f$'%(
            init_vals_list[i*2+j][0], init_vals_list[i*2+j][1]))
        axes[i][j].legend()
plt.show()
```



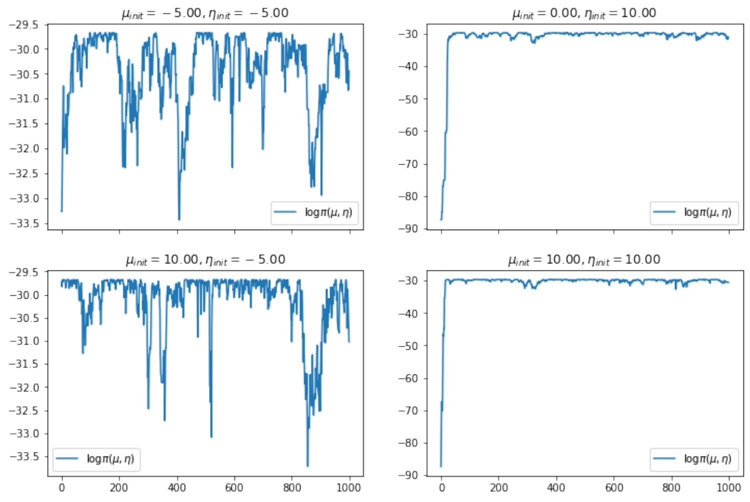
[8]:
```
fig, axes = plt.subplots(2,2,figsize=(12,8))
for i in range(2):
    for j in range(2):
        axes[i][j].hist(res_list[i*2+j][500:,1], label='$\eta$')
        axes[i][j].set_title('$\mu_{init}=%.2f,  \eta_{init}=%.2f$'%(
            init_vals_list[i*2+j][0], init_vals_list[i*2+j][1]))
        axes[i][j].legend()
plt.show()
```



We can see that for most cases, the posterior distribution covers the true values of $(\mu, \eta) = (1, 1)$, except when $(\mu_{init}, \eta_{init}) = (-5, -5)$ the estimated η is not near to the true value.

4. Repeat part 3 for the "8 schools data" here (omitting the comparisons with the true values, which of course you do not know here).

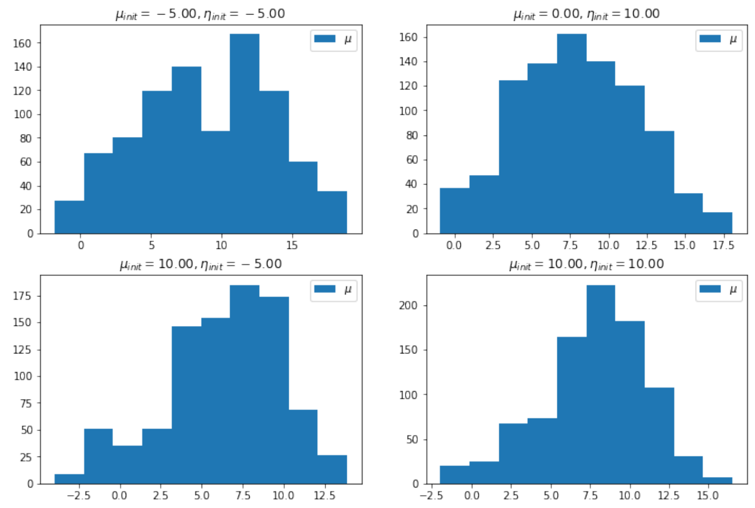
[9]:
```python
np.random.seed(0)
x = np.array([28,8,-3,7,-1,1,18,12])
s = np.array([15,10,16,11,9,11,10,18])
n_iter = 1000
init_vals_list = [[-5., -5.], [0.,10.], [10., -5.], [10., 10.]]
res_list = []
for i in range(4):
    res = MH(x, s, n_iter, init_vals_list[i], [1,1])
    res_list.append(res)
fig, axes = plt.subplots(2,2,sharex='all',figsize=(12,8))
for i in range(2):
    for j in range(2):
        axes[i][j].plot(res_list[i*2+j][:,2], label='$\log\pi(\mu,\eta)$')
        axes[i][j].set_title('$\mu_{init}=%.2f,  \eta_{init}=%.2f$'%(
            init_vals_list[i*2+j][0], init_vals_list[i*2+j][1]))
        axes[i][j].legend()
plt.show()
```



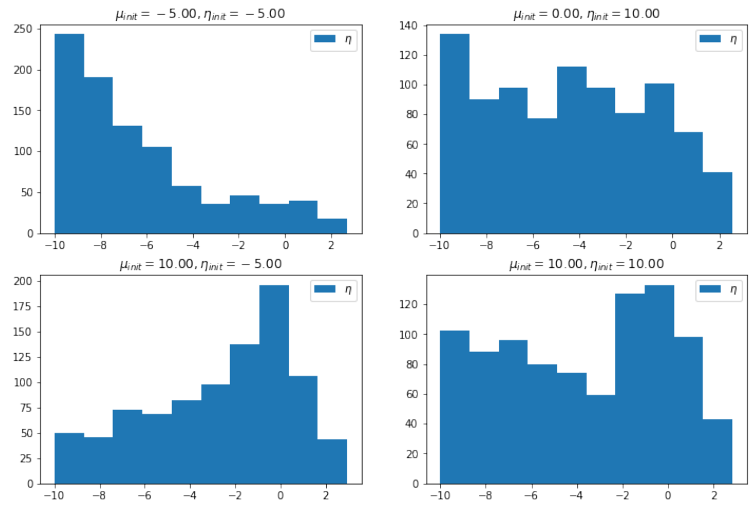
The curves on the left are actually near some values, i.e. they are actually stable already. So from plots on the right, we just need to burn in the first 100 samples.

As we can see from the below posterior distributions, the posterior mean of μ is near to 10 and the posterior mean of η is about -5.

[10]:
```
fig, axes = plt.subplots(2,2,figsize=(12,8))
for i in range(2):
    for j in range(2):
        axes[i][j].hist(res_list[i*2+j][100:,0], label='$\mu$')
        axes[i][j].set_title('$\mu_{init}=%.2f,  \eta_{init}=%.2f$'%(
            init_vals_list[i*2+j][0], init_vals_list[i*2+j][1]))
        axes[i][j].legend()
plt.show()
```



[11]:
```
fig, axes = plt.subplots(2,2,figsize=(12,8))
for i in range(2):
    for j in range(2):
        axes[i][j].hist(res_list[i*2+j][100:,1], label='$\eta$')
        axes[i][j].set_title('$\mu_{init}=%.2f,  \eta_{init}=%.2f$'%(
            init_vals_list[i*2+j][0], init_vals_list[i*2+j][1]))
        axes[i][j].legend()
plt.show()
```



5. Note that the posterior distribution on θ_j is given by:

$$p(\theta_j|X) = \int p(\theta_j|X,\mu,\eta)p(\mu,\eta|X),$$

which is the expectation of $p(\theta_j|X,\mu,\eta)$ over the posterior $p(\mu,\eta|X)$. Computing posterior distributions like this is sometimes referred to as "integrating out uncertainty in" (μ,η). (It is useful to compare this with the EB approach of just plugging in the maximum likelihood estimates and computing $p(\theta_j|X,\hat{\mu},\hat{\eta})$. Notice that the two will produce similar results if the posterior distribution $p(\mu,\eta|X)$ is very concentrated around the MLE.)

Given T samples $(\mu^{(1)},\eta^{(1)}),\ldots,(\mu^{(T)},\eta^{(T)})$ from the posterior distribution $p(\mu,\eta|X)$ you can approximate this expectation by

$$p(\theta_j|X) \approx \frac{1}{T}\sum_{t=1}^{T} p(\theta_j|X,\mu^{(t)},\eta^{(t)}).$$

So you can approximate the posterior mean by

$$\mathbb{E}(\theta_j|X) \approx \frac{1}{T}\sum_{t=1}^{T} \mathbb{E}(\theta_j|X,\mu^{(t)},\eta^{(t)}).$$

Using the same idea, given an expression to approximate the posterior second moment $\mathbb{E}(\theta_j^2|X)$, and so approximate the posterior variance (and hence posterior standard deviation).

Since

$$p(\theta_j|X) = \int p(\theta_j|X,\mu,\eta)p(\mu,\eta|X) \approx \frac{1}{T}\sum_{t=1}^{T} p(\theta_j|X,\mu^{(t)},\eta^{(t)}),$$

we have

$$\mathbb{E}(\theta_j^2|X) = \int \theta_j^2\cdot p(\theta_j|X)\mathrm{d}\theta_j \approx \frac{1}{T}\sum_{t=1}^{T}\int \theta_j^2\cdot p(\theta_j|X,\mu^{(t)},\eta^{(t)})\mathrm{d}\theta_j = \frac{1}{T}\sum_{t=1}^{T}\mathbb{E}(\theta_j^2|X,\mu^{(t)},\eta^{(t)})$$

and therefore

$$\mathrm{Var}(\theta_j|X) \approx \frac{1}{T}\sum_{t=1}^{T}\mathbb{E}(\theta_j^2|X,\mu^{(t)},\eta^{(t)}) - \left[\frac{1}{T}\sum_{t=1}^{T}\mathbb{E}(\theta_j|X,\mu^{(t)},\eta^{(t)})\right]^2.$$

From Homework 2 (b) we have $\theta_j|X_j,\mu,\sigma \sim \mathcal{N}\left(\frac{\sigma^2}{s_j^2+\sigma^2}X_j + \frac{s_j^2}{s_j^2+\sigma^2}\mu, \frac{s_j^2\sigma^2}{s_j^2+\sigma^2}\right)$.

6. Use the results from 4 and 5 to compute an approximate posterior mean and posterior standard deviation for θ_j for each school in the 8 schools data. Compare and contrast your results with the EB results and also the discussion in the initial blog-post https://statmodeling.stat.columbia.edu/2014/01/21/everything-need-know-bayesian-statistics-learned-eight-schools/.

[12]:
```
from scipy.optimize import minimize
def neg_log_likelihood(pars, x, s):
    '''
    Evaulate log-likelihood l(mu,sigma).
    Params:
        pars - params to be optimized
        x    - observation
        s    - posterior standard deviation of x
    Returns:
    '''
    mu = pars[0]
    sigma = np.exp(pars[1])
    L = np.sum(norm.logpdf(x, mu, np.sqrt(s**2+sigma**2)))
    return -L

def ebnm_normal(x,s):
    '''
    Params:
        x        - array of size [n,], observation
        s        - array of size [n,], posterior standrad deviation of x
    Returns:
        mu       - MLE of mu
        sigma    - MLE of sigma
        E_theta - posterior mean of theta_j
        sd_theta- posterior sd of theta_j
    '''
    res = minimize(neg_log_likelihood, np.zeros(2),
                         args=(x, s),
                         method='L-BFGS-B')
    mu, sigma = res.x
    sigma = np.exp(sigma)
    E_theta = (sigma**2*x+s**2*mu)/(sigma**2+s**2)
    sd_theta = np.sqrt(sigma**2*s**2/(sigma**2+s**2))
    return mu, sigma, E_theta, sd_theta

mu_hat, sigma_hat, E_theta_hat, sd_theta_hat = ebnm_normal(x,s)
print(E_theta_hat, sd_theta_hat)
```

```
[7.68578326 7.6857597  7.68574757 7.68575732 7.68572986 7.68574392
 7.68578674 7.68576245] [0.01644182 0.0164418  0.01644182 0.01644181 0.0164418
0.01644181
 0.0164418  0.01644182]
```

```
[13]: mu_hat = np.tile(np.expand_dims(res_list[1][100:,0], -1), (1,len(s)))
      sigma_hat = np.tile(np.expand_dims(np.exp(res_list[1][100:,1]), -1), (1,len(s)))
      _s = np.tile(np.expand_dims(s, 0), (len(mu_hat),1))
      E_theta = np.mean((sigma_hat**2*x+_s**2*mu_hat)/(_s**2+sigma_hat**2), axis=0)
      E_theta2 = np.mean(_s**2*sigma_hat**2/(_s**2+sigma_hat**2) +
                         ((sigma_hat**2*x+_s**2*mu_hat)/(_s**2+sigma_hat**2))**2,␣
       ↪axis=0)
      sd_theta = np.sqrt(E_theta2 - E_theta**2)
      print(E_theta, sd_theta)
```

```
[8.18553118 8.02305702 7.93471882 8.00790362 7.85871153 7.92538941
 8.18161281 8.04096337] [4.15845636 4.06061427 4.15482736 4.07828056 4.1127635
4.10730682
 4.08970031 4.13740712]
```

As we can see, the approximate posterior means computed by the MH algorithm are near to and slightly larger than the posterior computed by the Empirical Bayes algorithm. While the estimated standard errors of the MH algorithm are much larger than those of the EM approach.