AI701 HW2 Implementation Report

20204345 Sihwan Park

Code Implementation

Before begin, we introduce our algorithm. For more details on the algorithm, please refer the homework submission.

Algorithm 1 $P(\theta, Z|X)$ sampler using Gibbs + Metropolis-Hastings

---

Input: Dataset $X = \{x_1, \cdots, x_n\}$
  Initialize $\pi^{(1)}, \phi^{(1)} \sim p(\theta)$ and $Z^{(1)} \sim P(Z|\pi^{(1)})$
  for all $t = 1, \cdots$ **do**
    Set $Z^{(t+1)} = Z^{(t)}$
    for all $j = 1, \cdots, n$ **do**
      Sample $z_j^{(t+1)} \sim P\left(z_j | \left\{z_k^{(t+1)}\right\}_{k \neq j}\right) = P(z_j | Z^{(t+1)} \setminus \{z_j^{(t+1)}\}, X, \theta)$
    end for
    for all $j = 1, \cdots, K$ **do**
      $\alpha_j^{(t+1)}$ = number of i with $z_i^{(t+1)} = j$
    end for
    Sample $\pi^{(t+1)} \sim Dir(\mathbf{1} + \alpha^{(t+1)})$ where $\mathbf{1} = (1, 1, \cdots, 1) \in \mathbb{Z}^K$
    Propose $\phi' \sim q(\phi'|\phi^{(t)})$
    Compute $A(\phi'|\phi) = min\left\{1, \frac{P(X|Z,\phi')P(\phi')q(\phi|\phi')}{P(X|Z,\phi)P(\phi)q(\phi'|\phi)}\right\}$
    Sample $u \sim$ Unif [0,1]
    if $u \leq A(\phi'|\phi)$ **then**
      $\phi^{(t+1)} = \phi'$
    else
      $\phi^{(t+1)} = \phi^{(t)}$
    end if
  end for

Now, we will briefly explain our code implementation with code blocks. Before we start, please note that this code implementation is same as the algorithm. (This code is just a direct implementation of the algorithm).

```python
import numpy as np
from scipy.stats import multivariate_normal

import matplotlib
matplotlib.use('agg')
import matplotlib.pyplot as plt

# read data
x = []
with open('data.txt', 'r') as f:
    lines = f.readlines()
    for line in lines:
        x.append([float(val) for val in line.split()])
x = np.array(x)

# initialize
K = 3
alpha = [1 for _ in range(K)]
pi = np.random.dirichlet(alpha)

# sample z from categorical distribution conditioned on pi
z = np.random.choice([k+1 for k in range(K)], 1000, p = pi)

# sample mu, lambda, v from prior
mu = np.random.multivariate_normal([0,0], [[5., 0.],[0., 5.]], size = K)
lmda = np.random.lognormal(.1, .1, size = K)
v = np.random.multivariate_normal([0,0], [[.25, 0.],[0., .25]], size = K)
```

This code block is for reading data and initializing the parameters. In other words, initialize $\pi^{(1)}, \phi^{(1)} \sim p(\theta)$ and $Z^{(1)} \sim P(Z|\pi^{(1)})$.

```python
# probability density of lognormal distribution
def lognormal_pdf(x, mu, sigma):
    return (1/(x * sigma * np.sqrt(2*np.pi))) * np.exp(-1 * ((np.log(x)-mu)**2)/(2*(sigma**2)))

# calculate p(z_i | {z_k}_(k != i))
def p_z(i, pi, x, mu, lmda, v):
    K = len(pi)
    p = []

    for j in range(K):
        N = multivariate_normal(mean = mu[j], cov = lmda[j] * np.eye(2) + np.outer(v[j], v[j]))
        p.append(pi[j] * N.pdf(x[i]))

    # normalize
    sum = 0
    for i in range(len(p)):
        sum += p[i]

    return p/sum
```

This code block is for helper functions. The ‘lognormal_pdf’ function calculates the probability density of the lognormal distribution. And the ‘p_z’ function calculates

$$P(z_i = j | Z \setminus \{z_i\}, X, \theta)$$

$$= \frac{\pi_j \mathcal{N}(x_i | \mu_j, \lambda_j I_d + v_j v_j^T)}{\sum_{k=1}^K \pi_k \mathcal{N}(x_i | \mu_k, \lambda_k I_d + v_k v_k^T)}$$ for all j = 1, ..., K

```python
# calculate the acceptance probability
def accept_prob(x, z, mu, lmda, v, mu_prime, lmda_prime, v_prime):
    # calculate p(X|Z,phi') / p(X|Z,phi)
    log_prob = 0.
    for i in range(len(x)):
        j = z[i] - 1
        N = multivariate_normal(mean = mu[j], cov = lmda[j] * np.eye(2) + np.outer(v[j], v[j]))
        N_prime = multivariate_normal(mean = mu_prime[j], cov = lmda_prime[j] * np.eye(2) + np.outer(v_prime[j], v_prime[j]))
        log_prob += np.log(N_prime.pdf(x[i])) - np.log(N.pdf(x[i])) # log((X|Z,phi') / p(X|Z,phi))

    # calculate p(phi), p(phi')
    N_mu = multivariate_normal(mean = [0,0], cov = [[5., 0.],[0., 5.]])
    N_v = multivariate_normal(mean = [0,0], cov = [[.25, 0.],[0., .25]])

    p_phi, p_phi_prime = 0., 0.
    for i in range(K):
        p_phi += (np.log(N_mu.pdf(mu[i])) + np.log(lognormal_pdf(lmda[i], .1, .1)) + np.log(N_v.pdf(v[i])))
        p_phi_prime += (np.log(N_mu.pdf(mu_prime[i])) + np.log(lognormal_pdf(lmda_prime[i], .1, .1))
                        + np.log(N_v.pdf(v_prime[i])))

    log_prob += p_phi_prime - p_phi

    # calculate q(phi'|phi), q(phi|phi')
    q_phi, q_phi_prime = 0., 0.
    for i in range(K):
        N_mu_prime = multivariate_normal(mean = mu[i], cov = sigma_q2 * np.eye(2))
        N_v_prime = multivariate_normal(mean = v[i], cov = sigma_q2 * np.eye(2))

        N_mu = multivariate_normal(mean = mu_prime[i], cov = sigma_q2 * np.eye(2))
        N_v = multivariate_normal(mean = v_prime[i], cov = sigma_q2 * np.eye(2))

        q_phi += (np.log(N_mu.pdf(mu[i])) + np.log(lognormal_pdf(lmda[i], lmda_prime[i], sigma_q2)) + np.log(N_v.pdf(v[i])))
        q_phi_prime += (np.log(N_mu_prime.pdf(mu_prime[i])) + np.log(lognormal_pdf(lmda_prime[i], lmda[i], sigma_q2))
                        + np.log(N_v_prime.pdf(v_prime[i])))

    log_prob += q_phi - q_phi_prime

    return min(np.exp(log_prob), 1.)
```

This code block is for calculating the acceptance probability

$$A(\phi'|\phi) = min\left\{1, \frac{P(X|Z,\phi')P(\phi')q(\phi|\phi')}{P(X|Z,\phi)P(\phi)q(\phi'|\phi)}\right\}$$

To prevent the likelihood overflow/underflow, use the log-exp trick. Calculate all the probabilities in log-scale, and exponent the log_prob at the end.

```python
def loglikelihood(x, z, pi, mu, lmda, v):
    log_prob = 0.

    N_mu = multivariate_normal(mean = [0,0], cov = [[5., 0.],[0., 5.]])
    N_v = multivariate_normal(mean = [0,0], cov = [[.25, 0.],[0., .25]])

    for i in range(K):
        log_prob += np.log(N_mu.pdf(mu[i])) + np.log(N_v.pdf(v[i])) + lognormal_pdf(lmda[i], .1, .1)

    for i in range(len(x)):
        j = z[i] - 1
        N = multivariate_normal(mean = mu[j], cov = lmda[j] * np.eye(2) + np.outer(v[j], v[j]))
        log_prob += np.log(pi[j] * N.pdf(x[i]))

    return log_prob
```

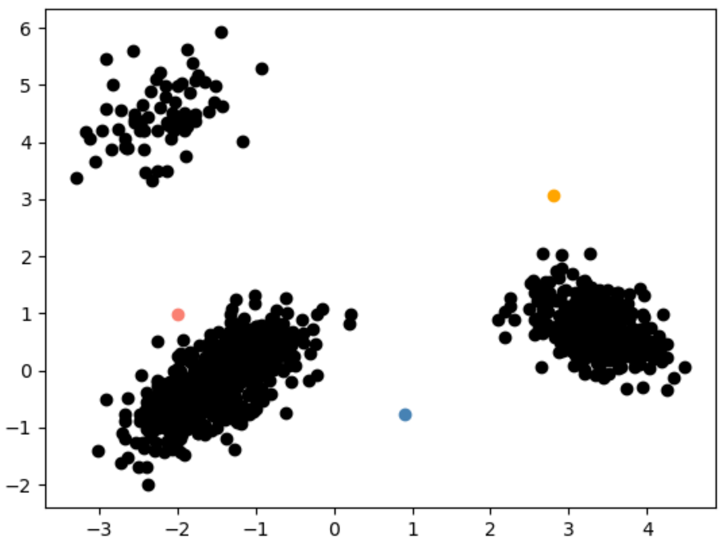
This code block is for calculating the marginal log-likelihood $\log p(X, Z, \theta)$ to trace the convergence.

```python
plt.scatter(x[:,0], x[:,1])
plt.savefig('data.png')

# main procedure
loglikelihoods = []
max_iter = 1000

for iter in range(max_iter):
    # sample Z using Gibbs sampling
    for j in range(len(x)):
        z[j] = np.random.choice([k+1 for k in range(K)], p = p_z(j, pi, x, mu, lmda , v))

    # sample pi from updated Dirichlet distribution
    new_alpha = []
    for j in range(1, K+1):
        new_alpha.append(list(z).count(j))

    alpha = [x+1 for x in new_alpha]
    pi = np.random.dirichlet(alpha)

    # sample mu, lambda, v using Metropolis-Hastings algorithm
    # propose mu', lmda', v' from proposal distribution
    mu_prime, lmda_prime, v_prime = [], [], []
    sigma_q2 = 1.
    for i in range(K):
        mu_prime.append(np.random.multivariate_normal(mu[i], sigma_q2 * np.eye(2)))
        lmda_prime.append(np.random.lognormal(np.log(lmda[i]), sigma_q2))
        v_prime.append(np.random.multivariate_normal(v[i], sigma_q2 * np.eye(2)))

    mu_prime = np.vstack(mu_prime)
    lmda_prime = np.hstack(lmda_prime)
    v_prime = np.vstack(v_prime)

    u = np.random.random()
    a = accept_prob(x, z, mu, lmda, v, mu_prime, lmda_prime, v_prime)
    print('Acceptance probability : %.4f'%(a))

    if u < a:
        print('Accepted')
        mu = mu_prime
        lmda = lmda_prime
        v = v_prime

        plt.scatter(x[:,0], x[:,1], color='black')
        plt.scatter(mu[0][0], mu[0][1], color='salmon')
        plt.scatter(mu[1][0], mu[1][1], color='orange')
        plt.scatter(mu[2][0], mu[2][1], color='steelblue')
        plt.savefig('results/estimation_iter%d.png'%(iter+1))
        plt.close()

    ll = loglikelihood(x, z, pi, mu, lmda, v)
    loglikelihoods.append(ll)

    plt.plot(range(1, iter+2), loglikelihoods)
    plt.xlabel('iteration')
    plt.ylabel('log-likelihood')
    plt.title('Trace of log-likelihood')
    plt.savefig('trace.png')
    plt.close()

    print('[%d/%d] Marginal LogLikelihood : %.4f'%(iter+1, max_iter, ll))
```

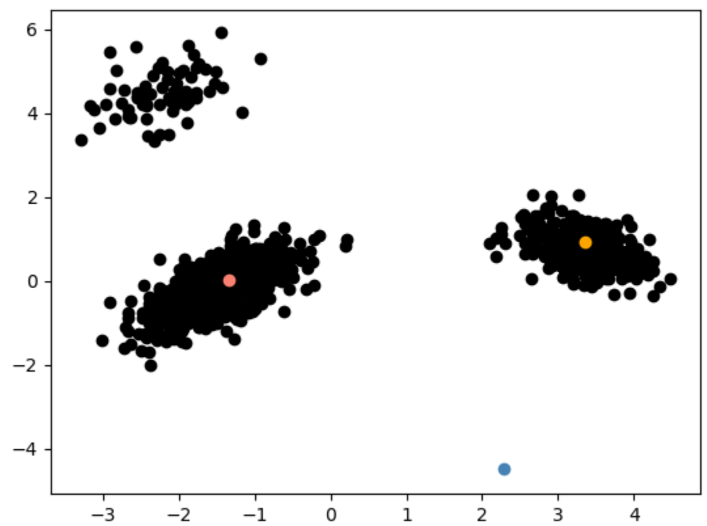
This code block is the implementation of our main algorithm. First, sample Z from the categorical distribution conditioned on pi using the Gibbs sampling. The, calculate the new alpha (will be the parameter of the Dirichlet distribution). And sample the pi from the resulting Dirichlet distribution. And then, sample mu, lambda, v using the Metropolis-Hastings algorithm. First propose mu', lambda', v' from the proposal distribution. Second, using 'accept_prob' function, calculate the acceptance probability. Next sample u from Uniform[0,1] and if u < acceptance probability, then accept the proposed parameters. Otherwise, do not update the parameters mu, lambda and v. When the new parameters are accepted, draw a picture to see the new parameter. At the end of each loop, calculate the marginal log-likelihood and print/plot it to trace the convergence.

Experimental Results

In the following figures, the black points are the data points and red, orange, blue points are the mean of each Gaussian component.



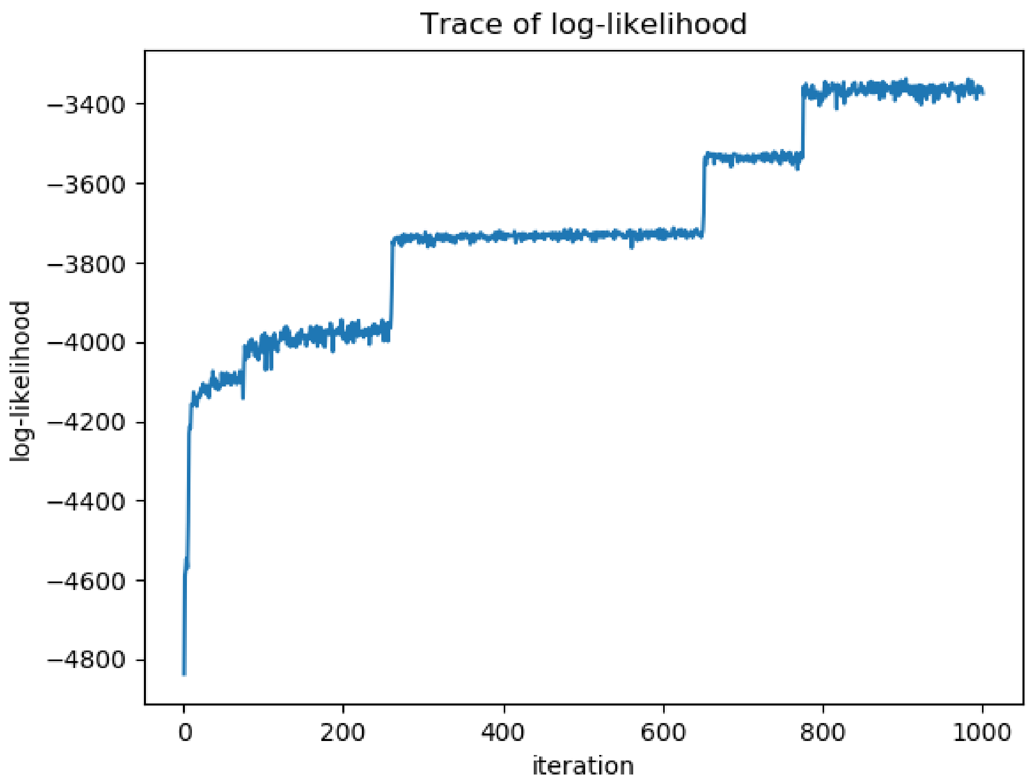
Initial parameters



After convergence

As we can see in the figure, means of two Gaussian components (red, orange points) are seems quite reasonable. However, the other mean (blue point) seems couldn't find the right position.

The following figure is the trace of the marginal log-likelihood.



Trace of the marginal log-likelihood

This figure shows that even if our sampler is a little slow, it's going in the right direction.