CS4070 - Multivariate Data Analysis - Part 2

Assignment 2

Konstantinos Krachtopoulos

Student Number: 5472539

Faculty of Electrical Engineering, Mathematics and Computer Science (EEMCS)

Technical University Delft

December 2021

1 Exercise 1

The log likelihood of the regression model, assuming that all y_i are independent is:

$$\begin{gathered} L(\theta; y) = \prod_{i=1}^{n} e^{-\mu_i} \mu_i^k/(k!) \Rightarrow \\ log(L(\theta; y)) = log(\prod_{i=1}^{n} e^{-\mu_i} \mu_i^k/(k!)) \Rightarrow \\ log(L(\theta; y)) = \sum_{i=1}^{n} (log(e^{-\mu_i}) + log(\mu_i^k) - log(k!)) \Rightarrow \\ log(L(\theta; y)) = \sum_{i=1}^{n} (-\mu_i + klog(\mu_i) - log(k!)) \Rightarrow \end{gathered} \tag{1}$$

Since $\mu_i = e^{x_i^T \theta}$,and $k = y_i$ the log likelihood becomes:

$$log(L(\theta; y)) = \sum_{i=1}^{n} (-e^{x_i^T \theta} + y_i x_i^T \theta - log(y_i!)) \tag{2}$$

2 Exercise 2

The log likelihood gradient is the first partial derivative of log likelihood with respect to θ:

$$\frac{\partial}{\partial \theta} log(L(\theta; y)) = \sum_{i=1}^{n} (-x_i^T e^{x_i^T \theta} + y_i x_i^T) \tag{3}$$

The Hessian of the log likelihood is the second partial derivative with respect to θ and θ^T:

$$\frac{\partial \partial}{\partial \theta \partial \theta^T} log(L(\theta; y)) = \sum_{i=1}^{n} (-x_i^T x_i e^{x_i^T \theta}) \tag{4}$$

3 Exercise 3

Based on the Newton Algorithm, can be updated iteratively using the following formula:

$$\theta^{j+1} = \theta^j - H_L(\theta^j)^{-1} \nabla L(\theta^j) \tag{5}$$

The Hessian Matrix and the Gradient of the Log Likelihood have already been calculated in Exercise 2. The initial θ will be sampled from a normal distribution $N \sim (0, \tilde{\sigma}^2 I_p)$, where $\tilde{\sigma} = 4$.

In order to compute the posterior distribution of θ, and because we know that its distribution is normal, we can use Laplace approximation. With Laplace Approximation, the posterior distribution is calculated as

$$f(\theta|x)_{\theta|x} \approx \Phi(\theta, \tilde{\Theta}, -H_L(\tilde{\Theta})) \tag{6}$$

where $\tilde{\Theta}$ is the posterior mode of θ, calculated with Newton Algorithm.

After implementing the above formulas in Python, the following values are derived for the mean vector and the covariance matrix:

$$m = \begin{bmatrix} 1.12777336 \\ 0.42858431 \\ 0.01512131 \\ -0.05416601 \end{bmatrix}, \tag{7}$$

$$Cov = \begin{bmatrix} 0.03140956 & -0.00820031 & 0.00116572 & -0.00139328 \\ -0.00820031 & 0.00305825 & -0.00031134 & 0.00066138 \\ 0.00116572 & -0.00031134 & 0.0148073 & -0.0014676 \\ -0.00139328 & 0.00066138 & -0.0014676 & 0.01173136 \end{bmatrix} \tag{8}$$

4 Exercise 4

In order to implement Random Walk Metropolis-Hastings algorithm, we need to iteratively update θ with

$$\theta^o := \theta + \sigma_{proposal} N(0, I_p) \tag{9}$$

where the initial θ is sampled from the distribution $N(0, 4^2 I_p)$. The update takes place based on the Metropolis-Hastings acceptance probability:

$$\alpha(\theta, \theta^0) = min(1, \frac{\pi(\theta^0)}{\pi(\theta)} \frac{q(\theta^0, \theta)}{q(\theta, \theta^0)}) \tag{10}$$

where $q(.,.)$ is the transition probability between two states, and $\pi(\theta)$ is the posterior probability of θ. Because we consider proposals of θ coming from a normal distribution, the transition matrix is symmetric, i.e. $q(\theta, \theta^0) = q(\theta^0, \theta)$. Hence, the previous equation becomes:

$$\alpha(\theta, \theta^0) = min(1, \frac{\pi(\theta^0)}{\pi(\theta)}) \tag{11}$$

Moreover, in order to decide whether θ should be updated in the current iteration, we follow the relation:

$$\theta_{n+1} = \begin{Bmatrix} \theta^0, & with\, prob\, \alpha(\theta, \theta_0) \\ \theta, & with\, prob\, 1 - \alpha(\theta, \theta_0) \end{Bmatrix} \tag{12}$$

To implement the above equations, we calculate the log of the acceptance as $logA = log\pi(\theta^0) - log\pi(\theta)$, and then compare it with u, where u is a sampled from $Unif(0,1)$.

1. If $logA > logu$, the acceptance criterion isn't satisfied.

2. Else, the acceptance criterion isn't satisfied.

Finally, the parameter $\sigma_{proposal}$ needs tuning, in order to get an acceptance rate of 25-50%. After trying multiple values, the final value of $\sigma_{proposal}$ is set to be 0.08, resulting in an acceptance rate of 32.8%. The following graph shows the values of θ_2 versus θ_1, where the color indicates the iteration number. Green indicates the first iterations, while blue indicates the last iterations.

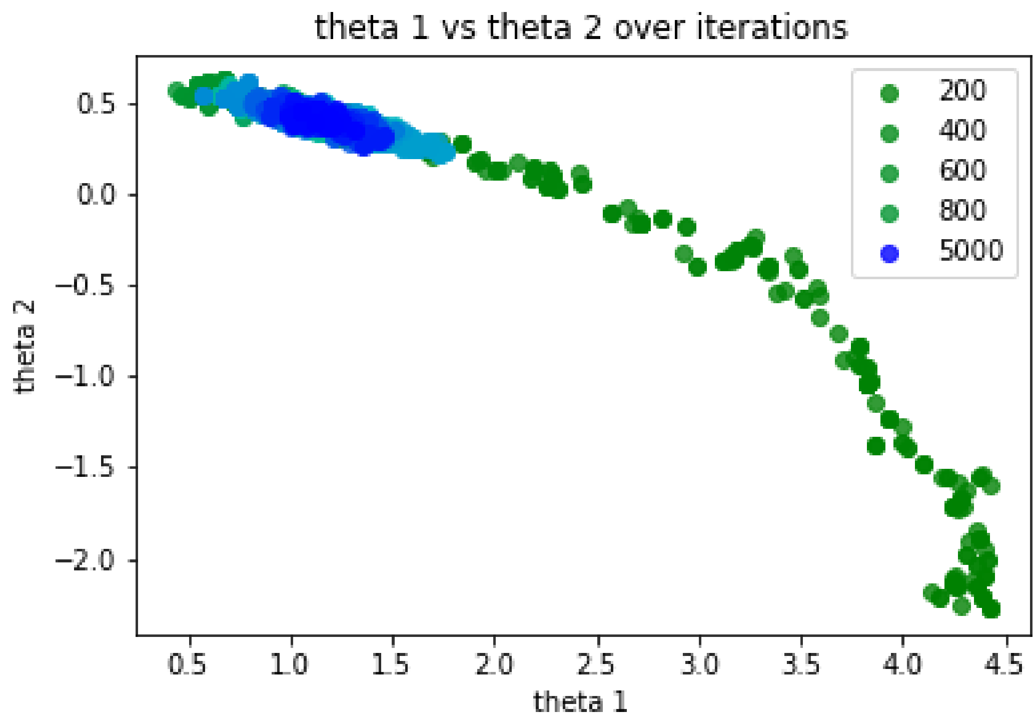

Figure 1: Random-walk iterations, θ_1, θ_2 comparison.

By checking the values of θ over the 5000 iterations, we see that its values started converging after 280 iterations. Hence, "burning" these iterations, the Monte Carlo posterior mean is:

$$\theta_{mean} = \begin{bmatrix} 1.1471 \\ 0.4196 \\ 0.0197 \\ -0.0376 \end{bmatrix} \tag{13}$$

5 Exercise 5

We want to iteratively sample from the conditionals of θ and $\tilde{\sigma}^2$. The sampling for θ has already been implemented in the previous exercise. Moreover, since $\tilde{\sigma}^2$ follows an inverse Gamma distribution with, its density function is given by:

$$p(\tilde{\sigma}^2|\alpha, \beta) = \frac{\beta^\alpha}{\Gamma(\alpha)}(\tilde{\sigma}^2)^{-\alpha-1}exp(-\frac{\beta}{\tilde{\sigma}^2}) \tag{14}$$

For the conditional distribution $\tilde{\sigma}^2|\theta, y$:

$$\begin{gathered} p(\tilde{\sigma}^2|\theta, y) \propto p(y, \theta, \tilde{\sigma}^2) = \\ (\tilde{\sigma}^2)^{-p/2}exp(-\frac{1}{2\tilde{\sigma}^2}\left\|\theta\right\|^2)(\tilde{\sigma}^2)^{-A-1}\exp -\frac{\beta}{(\tilde{\sigma}^2)}\mathbf{1}_{(0,\infty)}(\tilde{\sigma}^2) \Rightarrow \\ \tilde{\sigma}^2|\theta, y \sim IG(\alpha + p/2, \beta + \left\|\theta\right\|^2/2) \end{gathered} \tag{15}$$

Hence, we follow the same approach as in Exercise 4. The only difference is that before sampling for θ, we sample for $\tilde{\sigma}^2$. The Traceplot below shows the values of $\tilde{\sigma}^2$ over the 5000 iterations.

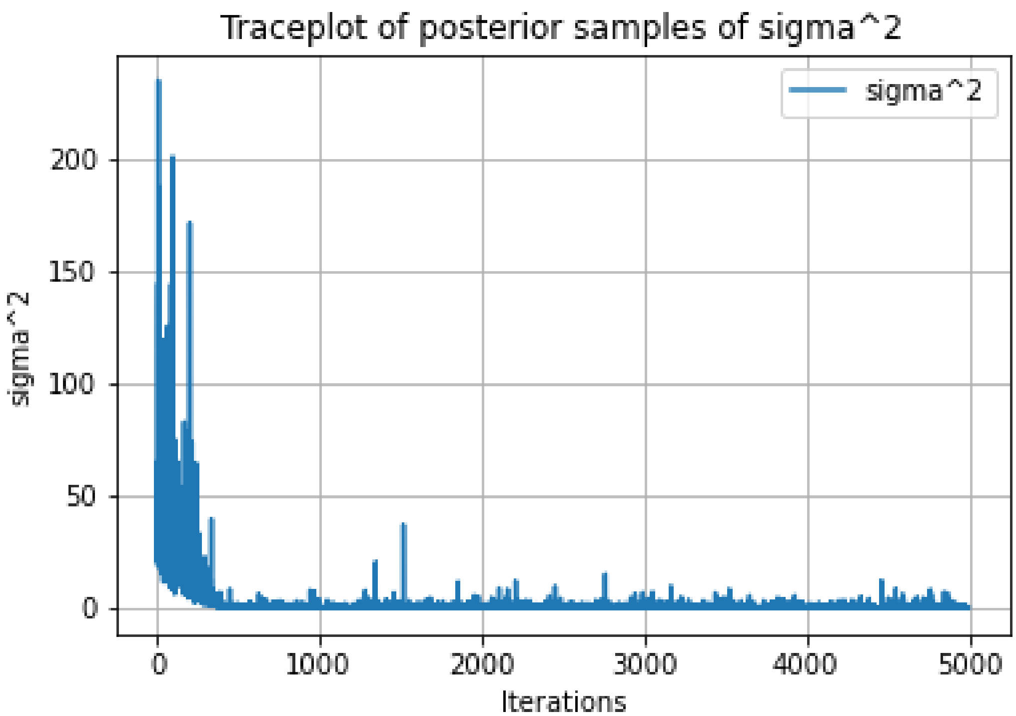

Figure 2: Traceplot of posterior samples of $\tilde{\sigma}^2$.

6 Appendix

The code used for computing the quantities of Exercises 3-4-5 is presented below, together with the appropriate comments:

6.1 Code - Exercise 3

```
# basic imports
import math

import numpy as np
import pandas as pd
from numpy.linalg import inv
import matplotlib.pyplot as plt

# Define functions to calculate foundamental values
def calc_gradient(x, y, thetas):
    # Calculate the gradient of the log likelihood
        return -x.T @ np.exp(x @ thetas.T) + x.T @ y

def calc_hessian_inverse(x, thetas):
    # Calculate the hessian matrix of the log likelihood
    hess = -x.T @ np.diag(np.exp(x @ thetas.T)) @ x
    return np.linalg.inv(hess)

# Load the data as a pandas dataframe
```

```python
df = pd.read_csv("dataexercise2.csv")

# Convert the dataframe to a numpy array (for easier manipulation)
arr = df.to_numpy()

# Split the dataset
# first 4 columns are features (x matrix)
# last column is label (y vector)
x = arr[:, :-1]
y = arr[:, -1]

# Sample thetas from prior distribution
featNum = x.shape[1]
std0 = 4
thetas = np.random.multivariate_normal(np.zeros(featNum), std0**2*np.eye(featNum))

# Newton algorithm implementation
iters = 1000
e = 1e-6

for i in range(iters):
    grad = calc_gradient(x, y, thetas)
    hess_inv = calc_hessian_inverse(x, thetas)
    update = grad @ hess_inv
    if np.any(np.abs(update)) > e:
        thetas = thetas - update
    else:
        print("Converged after {} iterations".format(i + 1))
        break

# Calculate the gaussian distribution of thetas with Laplace approximation
mean = thetas
cov = -calc_hessian_inverse(x, thetas)
print(mean)
print(cov)
```

6.2 Code - Exercise 4

```python
# basic imports
import numpy as np
import pandas as pd
import math
import matplotlib.pyplot as plt
from scipy.stats import multivariate_normal
from colour import Color

# Define functions to modularize the script
def calcPriorTheta(thetas, std0=4, featNum=4):
    # Calculate the prior of thetas, based on normal distribution
    # thetas: 4x1 array of parameters
    # std0: standard deviation set by the problem statement
    # featNum: Number of features loaded
    dist = multivariate_normal(mean=np.zeros(featNum), cov=std0**2*np.eye(featNum))
    return dist.pdf(thetas.T)
```

```python
def calcLogPostTheta(thetas, x, y):
    # Calculate the posterior of theta given data, without the
    # regularization parameter
    # thetas: 4x1 array of parameters
    # x: 2d given dataset
    # y: 1d labels of given dataset
    log_likelihood = 0
    for i in range(y.shape[0]):
        log_likelihood += -np.exp(x[i].T @ thetas)\
                          + y[i]*x[i].T @ thetas \
                          - np.log(math.factorial(y[i]))
    log_prior = calcPriorTheta(thetas)
    return (log_likelihood + log_prior)[0]

def acceptUpdate(logPostThetasCur, logPostThetasNew):
    # Calculate whether currents thetas will be updated with the new thetas
    # based on the acceptance criterion.
    # logPostThetasCur
    # logPostThetasNew
    logAcceptance = logPostThetasNew - logPostThetasCur
    if logAcceptance > 0: # then a = 1, sure update
        return True
    else:
        # Sample from uniform distribution
        logU = np.log(np.random.uniform(0, 1))
        return logAcceptance > logU

def execMetropolisHastings(thetasInit, iters, x, y):
    # logPosterior: the log posterior probability of thetas
    # thetasInit: initialization of theta parameters
    # iters: number of iterations
    # x: 2d given dataset
    # y: 1d labels of given dataset

    # Create lists to store the accepted, rejected and current thetas
    acc_thetas = []
    rej_thetas = []
    cur_thetas = np.zeros((iters, featNum))

    # Initialize theta parameters
    thetasCur = np.reshape(thetasInit, (-1, 1))

    # Initialized proposed sigma
    sigma_prop = 0.08

    # Begin the loop
    for iter in range(iters):
        # Compute the new theta parameters with Random Walk
        thetasNew = thetasCur + np.random.normal(0, sigma_prop, (featNum, 1))
        thetasCurPost = calcLogPostTheta(thetasCur, x, y)
        thetasNewPost = calcLogPostTheta(thetasNew, x, y)
        if acceptUpdate(thetasCurPost, thetasNewPost):
            # Update thetas
```

```
            thetasCur = thetasNew
            acc_thetas.append(thetasNew)
        else:
            rej_thetas.append(thetasNew)
        # Collect current theta parameters
        cur_thetas[iter] = thetasCur[:, 0]

    return cur_thetas, acc_thetas, rej_thetas

# Main code execution
# Load the data as a pandas dataframe
df = pd.read_csv("dataexercise2.csv")

# Convert the dataframe to a numpy array (for easier manipulation)
arr = df.to_numpy()

# Split the dataset
# first 4 columns are features (x matrix)
# last column is label (y vector)
x = arr[:, :-1]
y = arr[:, -1]

# Set initial parameters
featNum = x.shape[1]
std0 = 4
thetasInit = np.random.multivariate_normal(np.zeros(featNum), std0**2*np.eye(featNum))
iters = 5000

cur_thetas, acc_thetas, rej_thetas = \
    execMetropolisHastings(thetasInit, iters, x, y)

acceptance_rate = len(acc_thetas)/iters
print("Acceptance rate is {}".format(acceptance_rate))

# plot theta1 vs theta2
initCol = Color("green")
finCol = Color("blue")
colorSpectrum = list(initCol.range_to(finCol, len(cur_thetas)))
fig, ax = plt.subplots()
for i in range(len(colorSpectrum)):
    if (((i+1) % 200 == 0) and (i+1 < 1000)) or (i+1 == iters):
            ax.scatter(cur_thetas[i, 0], cur_thetas[i, 1], alpha=0.8, c=str(colorSpectrum[i]),
                   label=i+1)
    else:
        ax.scatter(cur_thetas[i, 0], cur_thetas[i, 1], alpha=0.8, c=str(colorSpectrum[i]))

plt.xlabel("theta 1")
plt.ylabel("theta 2")
plt.title("theta 1 vs theta 2 over iterations")
plt.legend()
plt.show()
fig.savefig('plot4.png')

# Compute the mean posterior
```

```
lim = 280
meanPost = np.mean(cur_thetas[280:, :], axis=0)
print("Mean posterior theta is: {}".format(meanPost))
```

6.3 Code - Exercise 5

```
# basic imports
import numpy as np
import pandas as pd
import math
import matplotlib.pyplot as plt
from scipy.stats import multivariate_normal
from scipy.stats import invgamma

# Define functions to modularize the script
def calcPriorTheta(thetas, std0=4, featNum=4):
        # Calculate the prior of thetas, based on normal distribution
        # thetas: 4x1 array of parameters
        # std0: standard deviation set by the problem statement
        # featNum: Number of features loaded
        dist = multivariate_normal(mean=np.zeros(featNum), cov=std0**2*np.eye(featNum))
        return dist.pdf(thetas.T)

def calcLogPostTheta(thetas, x, y, sigma=4):
        # Calculate the posterior of theta given data, without the
        # regularization parameter
        # thetas: 4x1 array of parameters
        # x: 2d given dataset
        # y: 1d labels of given dataset
        log_likelihood = 0
        for i in range(y.shape[0]):
                log_likelihood += -np.exp(x[i].T @ thetas)\
                          + y[i]*x[i].T @ thetas\
                          - np.log(math.factorial(y[i]))
        log_prior = calcPriorTheta(thetas, std0=sigma)
        return (log_likelihood + log_prior)[0]

def acceptUpdate(logPostThetasCur, logPostThetasNew):
        # Calculate whether currents thetas will be updated with the new thetas
        # based on the acceptance criterion.
        # logPostThetasCur
        # logPostThetasNew
        logAcceptance = logPostThetasNew - logPostThetasCur
        if logAcceptance > 0: # then a = 1, sure update
                return True
        else:
                # Sample from uniform distribution
                logU = np.log(np.random.uniform(0, 1))
                return logAcceptance > logU

def execGibbsSamples(thetasInit, iters, x, y):
        # logPosterior: the log posterior probability of thetas
        # thetasInit: initialization of theta parameters
        # iters: number of iterations
```

```
        # x: 2d given dataset
        # y: 1d labels of given dataset

        # Create lists to store the accepted, rejected and current thetas
    acc_thetas = []
    rej_thetas = []
    cur_thetas = np.zeros((iters, featNum))
    sigmas = []

        # Initialize theta parameters
    thetasCur = np.reshape(thetasInit, (-1, 1))

        # Initialized proposed sigma
    sigma_prop = 0.08
    alpha = 0.2
    beta = 0.2

        # Begin the loop
    for iter in range(iters):
        # First update the sigma, based on the inversed gamma distribution
        sigma = invgamma.rvs(a=alpha + featNum/2, scale=beta + 0.5*np.linalg.norm(thetasCur)**2)
        sigmas.append(sigma)
        # Compute the new theta parameters with Random Walk
        thetasNew = thetasCur + np.random.normal(0, sigma_prop, (featNum, 1))
        thetasCurPost = calcLogPostTheta(thetasCur, x, y, sigma=sigma)
        thetasNewPost = calcLogPostTheta(thetasNew, x, y, sigma=sigma)
        if acceptUpdate(thetasCurPost, thetasNewPost):
                        # Update thetas
            thetasCur = thetasNew
            acc_thetas.append(thetasNew)
        else:
            rej_thetas.append(thetasNew)
                # Collect current theta parameters
        cur_thetas[iter] = thetasCur[:, 0]

    return cur_thetas, acc_thetas, rej_thetas, sigmas

# Main code execution
# Load the data as a pandas dataframe
df = pd.read_csv("dataexercise2.csv")

# Convert the dataframe to a numpy array (for easier manipulation)
arr = df.to_numpy()

# Split the dataset
# first 4 columns are features (x matrix)
# last column is label (y vector)
x = arr[:, :-1]
y = arr[:, -1]

# Set initial parameters
featNum = x.shape[1]
std0 = 4
thetasInit = np.random.multivariate_normal(np.zeros(featNum), std0**2*np.eye(featNum))
```

```
iters = 5000

cur_thetas, acc_thetas, rej_thetas, sigmas =\
    execGibbsSamples(thetasInit, iters, x, y)

acceptance_rate = len(acc_thetas)/iters
print("Acceptance rate is {}".format(acceptance_rate))

# plot traceplot
plt.plot(np.arange(len(sigmas)), sigmas, label="sigma^2")
plt.xlabel("Iterations")
plt.ylabel("sigma^2")
plt.title("Traceplot of posterior samples of sigma^2")
plt.legend()
plt.grid()
plt.savefig('plot5_1.png')
plt.show()
```