HW4

Yash Gangrade

April 8, 2018

Question 1

a) Running the code with just ising prior

Here, we just use the ising prior to get the images. We are also generating binary random images using the indigenous rand_generator function written. We are testing with different values of alpha, beta, and sigma. Different category of images are shown in the results. We are using 50 iterations to generate random binary images.

Through this experiment, we can observe the effects of alpha and beta over the image. We know that negative values of alpha favors more -1 (Black) pixels and similarly positive values of alpha favors more +1 (White) pixels. And, high beta favours bigger blobs of white or black pixels. This phenomenon can be observed in the test images of ques1 a.

b) Computing mean posterior images.

The clean images are shown in the results below. We used the following values.

Noisy Message -> alpha = 0, beta = 2.2, sigma^2 = 1.4, iterations = 100, burn-in = 20 Noisy Yinyang -> alpha = 0, beta = 5.5, sigma^2 = 2.5, iterations = 100, burn-in = 20

c) Final estimate of variance.

Here, we compute the variance estimate at each iteration. The final estimate that we get for both the images matches closely to the chosen variance values in b part. For noisy markov image, we get final estimate of variance as ~1.45 and for the noisy yinyang image, we get final estimate of variance as ~2.54.

CODE and RESULTS

```
source("Markov.r", echo = TRUE, keep.source = TRUE, max.deparse.length = 10000)
```

```
##
## > require(png)

## Loading required package: png

##
## > library(magick)

## Warning: package 'magick' was built under R version 3.4.4

## Linking to ImageMagick 6.9.9.14
## Enabled features: cairo, freetype, fftw, ghostscript, lcms, pango, rsvg, webp
## Disabled features: fontconfig, x11
```

```
##
## > read_image = function(file){
## +   x = readPNG(file, native = FALSE)
## +   x = x*20 - 10
## +
## +   t(x)[,nrow(x):1]
## + }
##
## > display_image = function(x, col=gray(seq(0,1,1/256)))
## + {
## +   w = dim(x)[1]
## +   h = dim(x)[2]
## +   par(mai = c(0,0,0,0))
## +   image(x, asp=h/w, col=col)
## + }
##
## > rand_generator = function()
## + {
## +   rand = runif(1)
## +   if(rand < 0.5)
## +     return (1)
## +   else
## +     return (-1)
## + }
##
## > checkerboard = function(row, col){
## +   check = matrix(0, row, col)
## +   for(i in 1:row){
## +     for(j in 1:col){
## +       if(check[i,j] == ((i+j) %% 2)){
## +         check[i,j] = 0
## +       }
## +       else{
## +         check[i,j] = 1
## +       }
## +     }
## +   }
## +   return(check)
## + }
##
## > # gen_ising_prior_term = function(img, alpha, beta){
## > #   num_col = dim(img)[2]
## > #   num_row = dim(img)[1]
## > #
## > #   #generating the new image, initializing it to 0 for all the values
## > #   img2 = matrix(0, num_row, num_col)
## > #
## > #   for(i in 1:num_row){
## > #     for(j in 1:num_col){
## > #       sum = 0
## > #       if(i != num_row){
## > #         sum = sum + img[i+1, j]
## > #       }
## > #       if(i != 1){
```

```
## > #         sum = sum + img[i-1, j]
## > #       }
## > #       if(j != num_col){
## > #         sum = sum + img[i, j+1]
## > #       }
## > #       if(j != 1){
## > #         sum = sum + img[i, j-1]
## > #       }
## > #       u_positive = alpha + beta*sum
## > #       u_negative = -alpha - beta*sum
## > #       probability_negative = exp(u_negative)
## > #       probability_positive = exp(u_positive)
## > #
## > #       final_probability = probability_positive / (probability_negative + probability_positive)
## > #       rand = runif(1)
## > #       if(final_probability < rand){
## > #         img2[i,j] = -1
## > #       }
## > #       else{
## > #         img2[i,j] = 1
## > #       }
## > #     }
## > #   }
## > #   return(img2);
## > # }
## >
## > gen_ising_prior_term = function(img, alpha, beta){
## +   num_col = dim(img)[2]
## +   num_row = dim(img)[1]
## +
## +   #generating the new image, initializing it to 0 for all the values
## +   img2 = matrix(0, num_row, num_col)
## +   checker = checkerboard(num_row, num_col)
## +   for(i in 1:num_row){
## +     for(j in 1:num_col){
## +       sum = 0
## +       if(i != num_row){
## +         sum = sum + img[i+1, j]
## +       }
## +       if(i != 1){
## +         sum = sum + img[i-1, j]
## +       }
## +       if(j != num_col){
## +         sum = sum + img[i, j+1]
## +       }
## +       if(j != 1){
## +         sum = sum + img[i, j-1]
## +       }
## +       u_positive = alpha + beta*sum
## +       u_negative = -alpha - beta*sum
## +       probability_negative = exp(u_negative)
## +       probability_positive = exp(u_positive)
## +
## +       final_probability = probability_positive / (probability_negative + probability_positive)
```

```
## +       rand = runif(1)
## +       if(final_probability < rand){
## +         img2[i,j] = -1
## +       }
## +       else{
## +         img2[i,j] = 1
## +       }
## +     }
## +   }
## +
## +
## +   # for(i in 1:num_row){
## +   #   for(j in 1:num_col){
## +   #     img2[i,j] = (img2[i,j] * checker[i,j]) + (img[i,j] * (1 - checker[i,j]))
## +   #   }
## +   # }
## +   return(img2);
## + }
##
## > gen_ising_prior_posterior_term = function(img, alpha, beta, sig){
## +   num_col = dim(img)[2]
## +   num_row = dim(img)[1]
## +
## +   #generating the new image, initializing it to 0 for all the values
## +   img2 = matrix(0, num_row, num_col)
## +
## +   for(i in 1:num_row){
## +     for(j in 1:num_col){
## +       sum = 0
## +       if(i != num_row){
## +         sum = sum + img[i+1, j]
## +       }
## +       if(i != 1){
## +         sum = sum + img[i-1, j]
## +       }
## +       if(j != num_col){
## +         sum = sum + img[i, j+1]
## +       }
## +       if(j != 1){
## +         sum = sum + img[i, j-1]
## +       }
## +       t1 = 1/(2*sig^2)
## +        num_positive = alpha + beta*sum - (t1*((1 - (1 - img[i,j])^2)))
## +        num_negative = -alpha - beta*sum - (t1*((-1 - (1 - img[i,j])^2)))
## +       #num_positive = alpha*img2[i,j] + beta*sum*img2[i,j] - (t1*((img2[i,j] - img[i,j])^2))
## +       #num_negative = -alpha*img2[i,j] - beta*sum*img2[i,j] - (t1*((img2[i,j] - img[i,j])^2))
## +
## +       probability_negative = exp(num_negative)
## +       probability_positive = exp(num_positive)
## +
## +       final_probability = probability_positive / (probability_negative + probability_positive);
## +       #print(final_probability)
## +       rand = runif(1);
## +
```

```
## +       if(final_probability > rand){
## +         img2[i,j] = 1
## +       } else{
## +         img2[i,j] = -1
## +       }
## +     }
## +   }
## +
## +   return(img2);
## + }
##
## > # gibbs_sampling_prior = function(img, alpha, beta, sig, iterations, burn){
## > #   num_col = dim(img)[2]
## > #   num_row = dim(img)[1]
## > #   num = 0
## > #   #generating the new image, initializing it to 0 for all the values
## > #   img2 = img
## > #   it = iterations + burn
## > #   for(i in 1:it){
## > #     img2 = gen_ising_prior_term(img2, alpha, beta)
## > #   }
## > #   return(img2)
## > # }
## >
## > gibbs_sampling_prior = function(img, alpha, beta, sig, iterations, burn){
## +   num_col = dim(img)[2]
## +   num_row = dim(img)[1]
## +   num = 0
## +   #generating the new image, initializing it to 0 for all the values
## +   img2 = img
## +   img_orig = img
## +   checker = checkerboard(num_row, num_col)
## +   it = iterations + burn
## +   for(i in 1:it){
## +     img2 = gen_ising_prior_term(img2, alpha, beta)
## +     img2 = img2*checker + (1 - checker)*img
## +     img = img2
## +     checker = 1 - checker
## +   }
## +   return(img2)
## + }
##
## > gibbs_sampling_prior_posterior = function(img, alpha, beta, sig, iterations, burn){
## +   num_col = dim(img)[2]
## +   num_row = dim(img)[1]
## +   num = 0
## +   #generating the new image, initializing it to 0 for all the values
## +   it = iterations + burn
## +   img2 = img
## +   for(i in 1:it){
## +     img3 = gen_ising_prior_posterior_term(img2, alpha, beta, sig)
## +     if(i > burn){
## +       num = num + 1
## +       img2 = (img2 * (num - 1) + img3)/num
```

```
## +     }
## +   }
## +   return(img2)
## + }
##
## > gen_estimated_variance = function(img, alpha, beta, sig, iterations, burn){
## +   num_col = dim(img)[2]
## +   num_row = dim(img)[1]
## +   num = 0
## +   sample_sig = matrix(0, iterations, 1)
## +   #img2 = matrix(0, num_row, num_col)
## +   img2 = img
## +   img_orig = img
## +   it = iterations + burn
## +   img3 = img
## +   for(i in 1:it){
## +     img3 = gen_ising_prior_posterior_term(img2, alpha, beta, sig)
## +     if(i > burn){
## +       num = num + 1
## +       img2 = (img2 * (num - 1) + img3)/num
## +       new_var = 0
## +       for(j in 1:num_row){
## +         for(k in 1:num_col){
## +           new_var = new_var + (img2[j,k] - img_orig[j,k])^2;
## +         }
## +       }
## +       sig = new_var/(num_col*num_row);
## +       #can be used for plotting
## +       sample_sig[num,] = sig
## +     }
## +   }
## +   cat("The sigma is ", sig, "\n")
## +   return(img2)
## + }
##
## > #
## > # get_estimated_variance_prior_posterior = function(img, alpha, beta, sig, iterations, burn){
## > #   num_col = dim(img)[2]
## > #   num_row = dim(img)[1]
## > #   num = 0
## > #   #generating the new image, initializing it to 0 for all the values
## > #   img2 = matrix(0, num_row, num_col)
## > #   img_orig = img
## > #   sample_sig = matrix(0, iterations, 1)
## > #   it = iterations + burn
## > #   img3 = img
## > #   for(i in 1:it){
## > #     img3 = gen_ising_prior_posterior_term(img, alpha, beta, sig)
## > #     if(i > burn){
## > #       num = num + 1
## > #       img2 = (img2 * (num - 1) + img3)/num
## > #       #display_image(img2)
## > #       new_var = 0
## > #         for(j in 1:num_row){
```

```
## > #           for(k in 1:num_col){
## > #             new_var = new_var + (img2[j,k] - img_orig[j,k])^2
## > #           }
## > #         }
## > #       #sample_sig[num,] = sig
## > #       sig = new_var/(num_row * num_col)
## > #     }
## > #     img = img3
## > #   }
## > #   #print(sum(sample_sig)/iterations)
## > #   cat("The sigma is ", sig)
## > #   return(img2)
## > # }
## >
## >
## >
## >
## >
## >
```

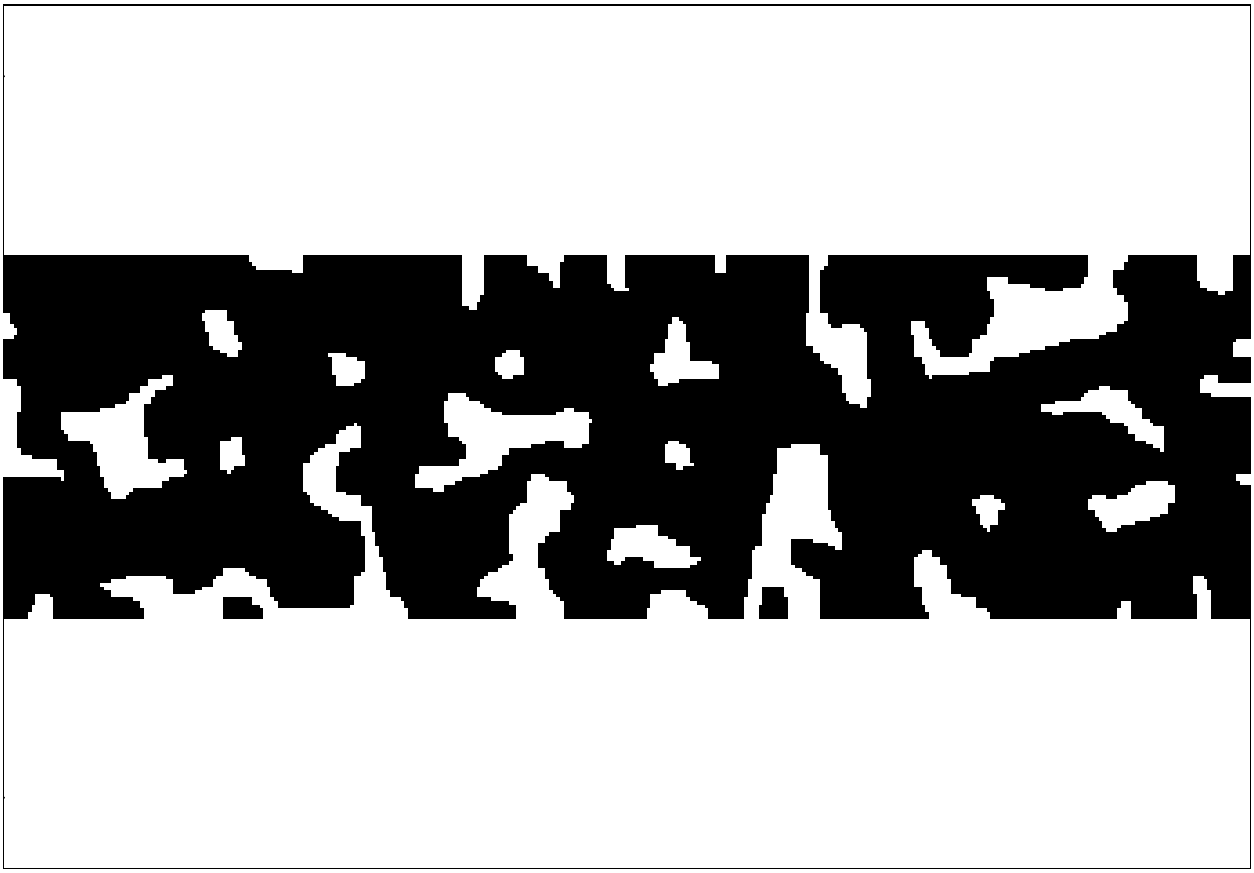
```
img1 = read_image("noisy-message.png")
img2 = read_image("noisy-yinyang.png")

# display_image(img1) display_image(img2)

num_col = dim(img1)[2]
num_row = dim(img1)[1]

binary_img = matrix(0, num_row, num_col)
binary_img = apply(binary_img, 1:2, function(x) rand_generator())

# checkerboard(num_row, num_col) display_image(binary_img)

## Question 1.1
cat("Trying to run for different alphas and betas \n")
```

```
## Trying to run for different alphas and betas
```

```
cat("Negative Alpha and Positive High Beta \n")
```

```
## Negative Alpha and Positive High Beta
```

```
resultant_img = gibbs_sampling_prior(binary_img, -0.1, 3, 1.5,
    50, 5)
display_image(resultant_img)
```



```
# #
cat("Negative Alpha and Low Beta \n")
```

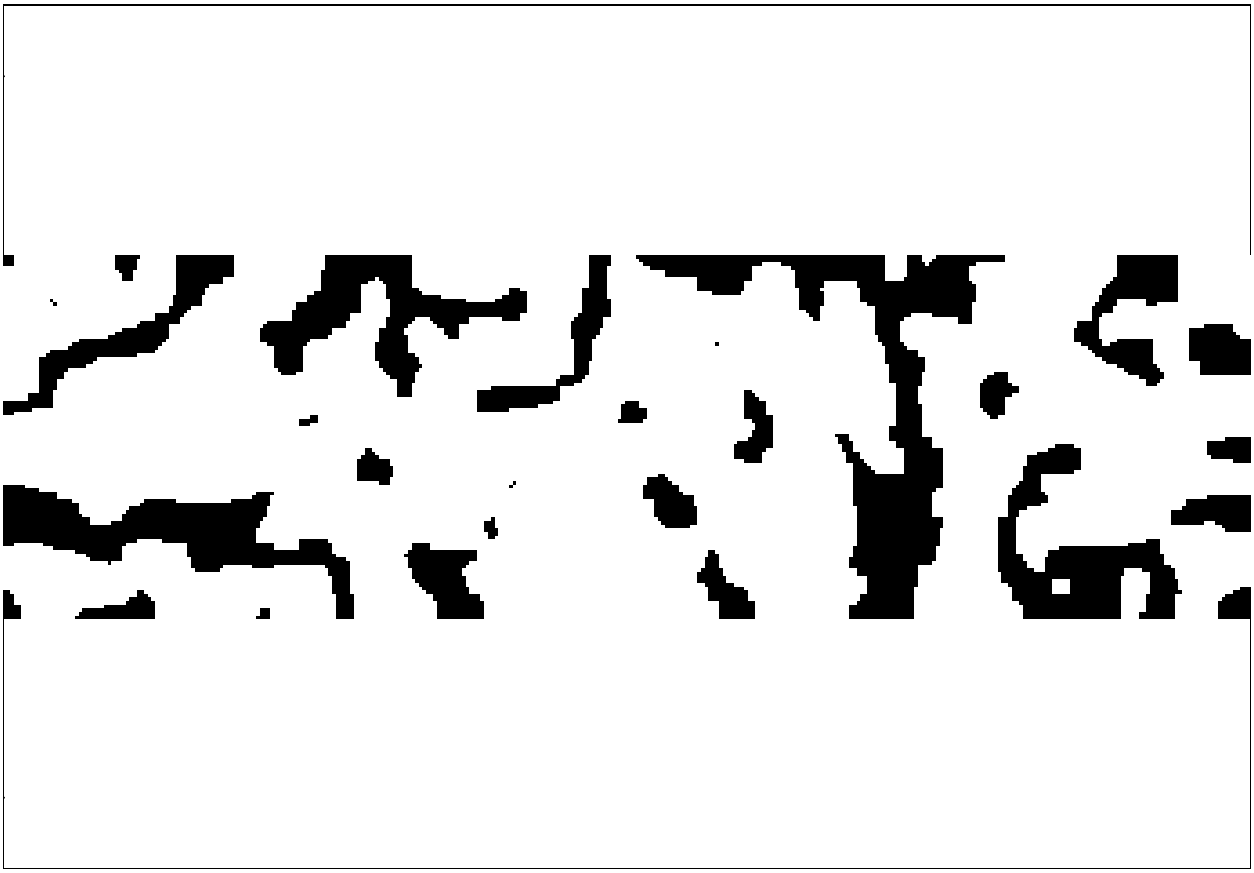
```
## Negative Alpha and Low Beta
```

```
resultant_img = gibbs_sampling_prior(binary_img, -0.1, 1, 1.5,
    50, 5)
display_image(resultant_img)
```

```
# #
cat("Positive Alpha and Positive High Beta \n")
```

```
## Positive Alpha and Positive High Beta
```

```
resultant_img = gibbs_sampling_prior(binary_img, 0.1, 3, 1.5,
    50, 5)
display_image(resultant_img)
```



```
# #
cat("Positive Alpha and Low Beta \n")
```

```
## Positive Alpha and Low Beta
```

```
resultant_img = gibbs_sampling_prior(binary_img, 0.1, 1, 1.5,
    50, 5)
display_image(resultant_img)
```

```
# ## Question 1.2
cat("Running for the Markov Noisy Image \n")
```

```
## Running for the Markov Noisy Image
```

```
#
cat("Here's the noisy image i.e. noisy-message.png \n")
```

```
## Here's the noisy image i.e. noisy-message.png
```

```
display_image(img1)
```


```
#
cat("We used alpha = 0, beta = 2.2, sigma = 1.4 with 100 iterations and 20 burn-in samples. The clean i
```

```
## We used alpha = 0, beta = 2.2, sigma = 1.4 with 100 iterations and 20 burn-in samples. The clean imag
```

```
resultant_img = gibbs_sampling_prior_posterior(img1, 0, 2.2,
    1.4, 100, 20)
display_image(resultant_img)
```

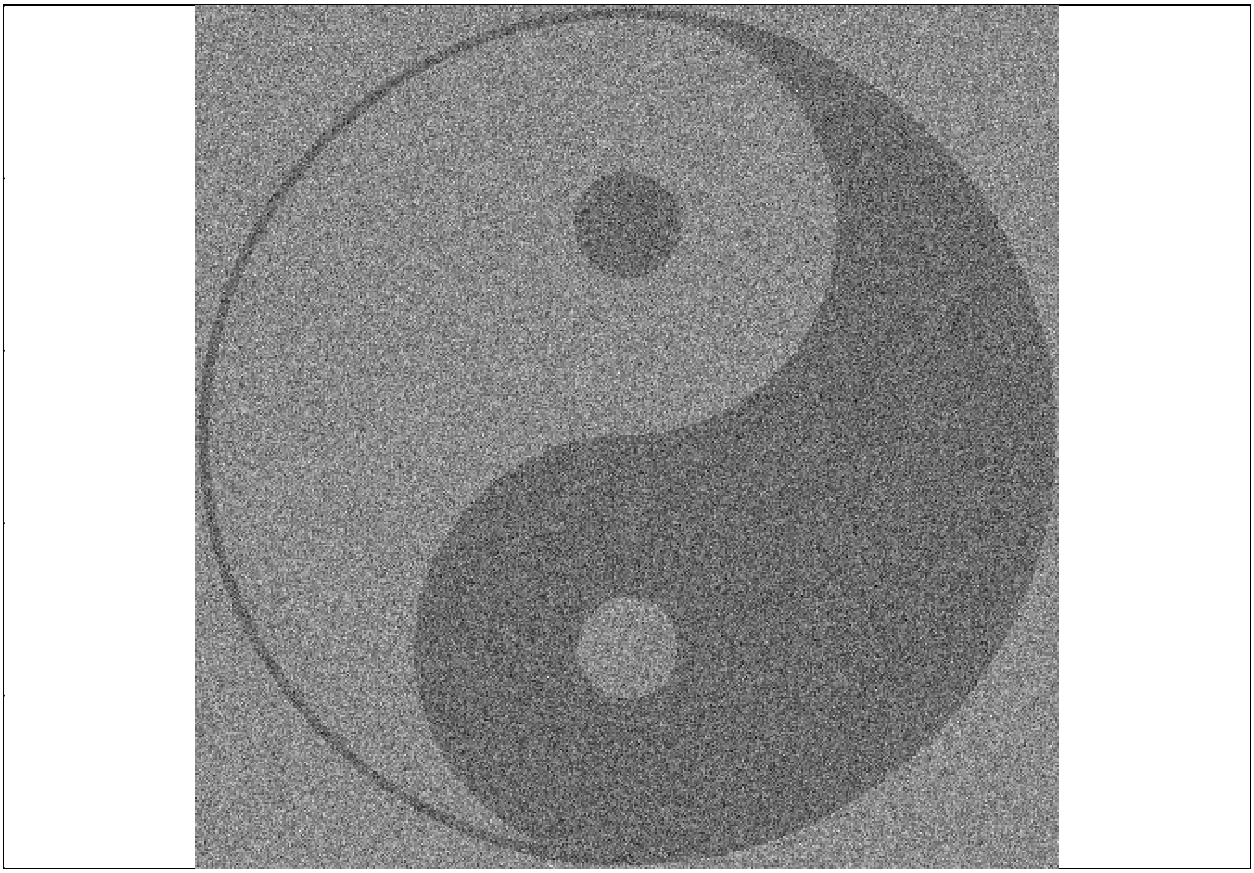

```
cat("Running for the Noisy Yinyang Image \n")
```

```
## Running for the Noisy Yinyang Image
```

```
#
cat("Here's the noisy image i.e. noisy-yinyang.png \n")
```

```
## Here's the noisy image i.e. noisy-yinyang.png
```

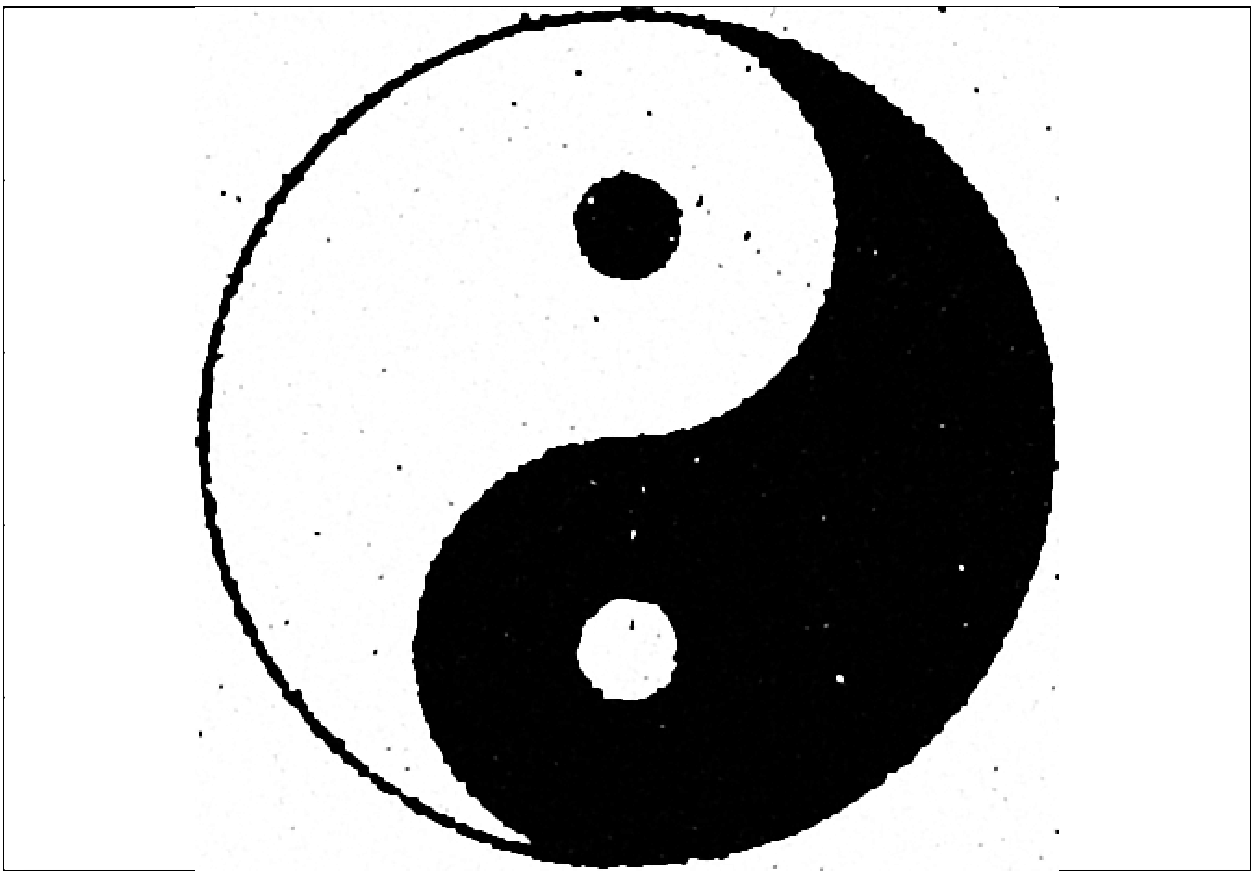
```
display_image(img2)
```



```
#
cat("We used alpha = 0, beta = 2.2, sigma = 1.4 with 100 iterations and 20 burn-in samples. The clean i
```

```
## We used alpha = 0, beta = 2.2, sigma = 1.4 with 100 iterations and 20 burn-in samples. The clean imag
```

```
resultant_img = gibbs_sampling_prior_posterior(img2, 0, 5.5,
    2.5, 100, 20)
display_image(resultant_img)
```



```
## Question 1.3

cat("Running for Noisy-Message MARKOV image. We get the cleaned image as well as the estimated sigma^2 f
```

```
## Running for Noisy-Message MARKOV image. We get the cleaned image as well as the estimated sigma^2 fo
```

```
resultant_img = gen_estimated_variance(img1, 0.2, 2, 1.5, 100,
    20)
```

```
## The sigma is  1.455798
```

```
display_image(resultant_img)
```

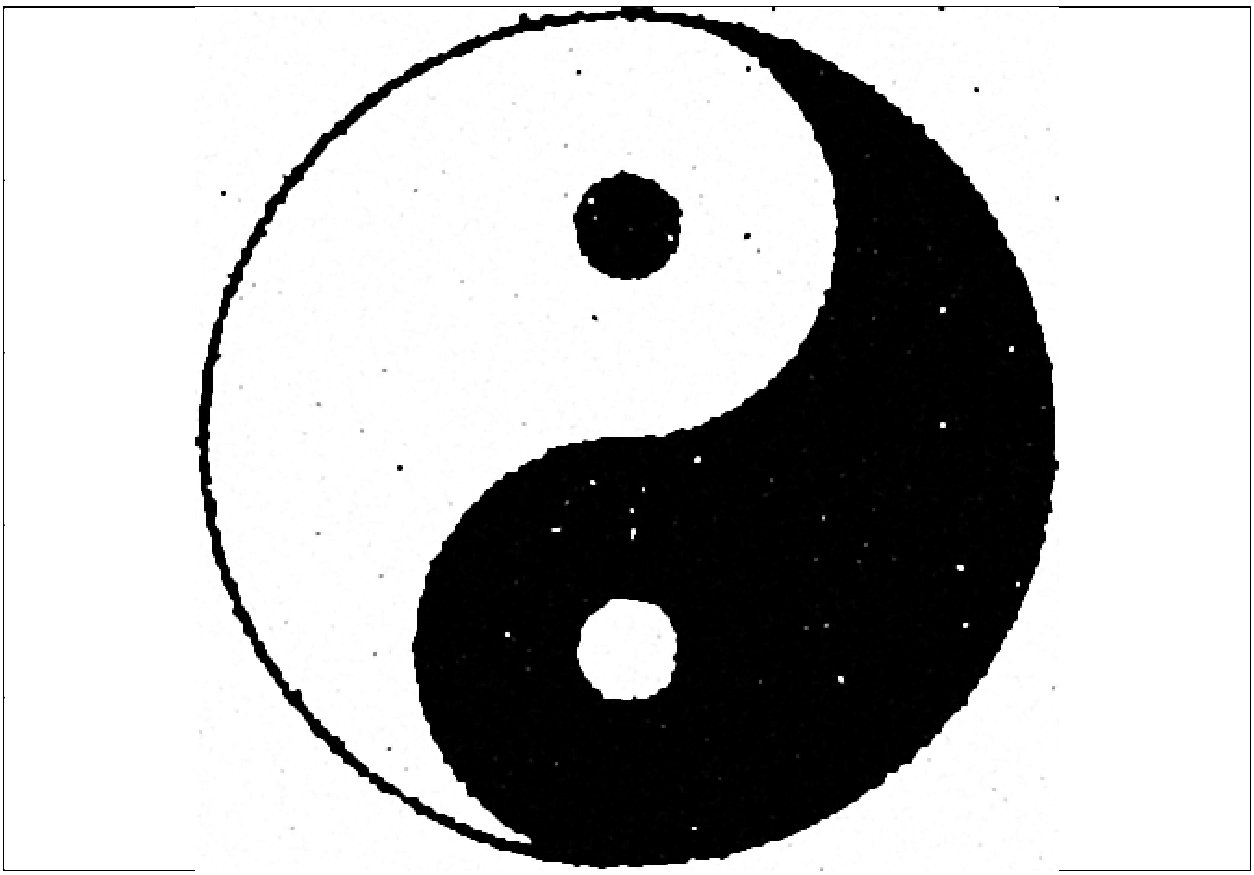

```
cat("Running for Noisy-yinyang image. We get the cleaned image as well as the estimated sigma^2 for this
```

```
## Running for Noisy-yinyang image. We get the cleaned image as well as the estimated sigma^2 for this.
```

```
resultant_img = gen_estimated_variance(img2, 0.2, 5.5, 2.5, 100,
    20)
```

```
## The sigma is  2.548276
```

```
display_image(resultant_img)
```



Question 2

(a) Write down the formula for the unnormalized posterior of $\beta|Y$

Solution:

We are given the data (X, Y), with $X \in R^d$ and $Y \in \{0, 1\}$. To train the classifier, we have the following model

$$Y \sim Ber\left(\frac{1}{1+e^{-X^T\beta}}\right)$$

$$X \sim N(0, \sigma^2 I)$$

Now, we need to write the formula of unnormalized posterior of $\beta|Y$ i.e.

$$p(\beta|y; x, \sigma) \propto \prod_{i=1}^{n} p(y_i|\beta; x_i)p(\beta; \sigma)$$

Now, to find this, we will first write the two terms in the product individually. So for $p(y_i|\beta; x_i)$ we can compute it using the properties of a bernoulli random variable. So we have,

$$p(y_i|\beta; x_i) = \left(\frac{1}{1+e^{-x_i^T\beta}}\right)^{y_i}\left(1-\frac{1}{1+e^{-x_i^T\beta}}\right)^{1-y_i}$$

Similarly, we have,

$$p(\beta, \sigma) \propto e^{\frac{-||\beta||^2}{2\sigma^2}}$$

Now, putting these two values together in the original unnormalized posterior formula, we have,

$$p(\beta|y; x, \sigma) \propto e^{\frac{-||\beta||^2}{2\sigma^2}} \cdot \left[\prod_{i=1}^{n} \left(\frac{1}{1+e^{-x_i^T\beta}}\right)^{y_i} \left(1 - \frac{1}{1+e^{-x_i^T\beta}}\right)^{1-y_i}\right]$$

(b) Show that this posterior is proportional to exp(-U(beta)).

Solution:

Here, we have to show that the posterior computed in the previous question is proportional to $exp(-U(\beta))$ where $U(\beta)$ is defined as:

$$U(\beta) = \sum_{i=1}^{n}(1-y_i)x_i^T\beta + \log(1+e^{-x_i^T\beta}) + \frac{1}{2\sigma^2}||\beta||^2$$

So, in order to prove this statement we will assume that $e^{-U(\beta)}$ is indeed equivalent to the posterior. We are gonna skip the proportionality constant in this case because it doesn't affect the proof. Then we will show that the $U(\beta)$ that we get now satisfies the aforementioned equation. Therefore, let us assume,

$$e^{-U(\beta)} = e^{\frac{-||\beta||^2}{2\sigma^2}} \cdot \left[\prod_{i=1}^{n} \left(\frac{1}{1+e^{-x_i^T\beta}}\right)^{y_i} \left(1 - \frac{1}{1+e^{-x_i^T\beta}}\right)^{1-y_i}\right]$$

Taking natural log of this equation

$$-U(\beta) = \frac{-||\beta||^2}{2\sigma^2} + \left[\sum_{i=1}^{n} y_i \log\left(\frac{1}{1+e^{-x_i^T\beta}}\right) + (1-y_i)\log\left(1 - \frac{1}{1+e^{-x_i^T\beta}}\right)\right]$$

$$-U(\beta) = \frac{-||\beta||^2}{2\sigma^2} + \left[\sum_{i=1}^{n} -y_i \log\left(1+e^{-x_i^T\beta}\right) + (1-y_i)\log\left(\frac{e^{-x_i^T\beta}}{1+e^{-x_i^T\beta}}\right)\right]$$

$$-U(\beta) = \frac{-||\beta||^2}{2\sigma^2} + \left[\sum_{i=1}^{n} -y_i \log\left(1+e^{-x_i^T\beta}\right) + (1-y_i)\left(\log\left(e^{-x_i^T\beta}\right) - \log\left(1+e^{-x_i^T\beta}\right)\right)\right]$$

$$U(\beta) = \frac{||\beta||^2}{2\sigma^2} + \left[\sum_{i=1}^{n} y_i \log\left(1+e^{-x_i^T\beta}\right) - (1-y_i)\log\left(e^{-x_i^T\beta}\right) + (1-y_i)\log\left(1+e^{-x_i^T\beta}\right)\right]$$

$$U(\beta) = \frac{||\beta||^2}{2\sigma^2} + \left[\sum_{i=1}^{n} \log\left(1+e^{-x_i^T\beta}\right) + (1-y_i)x_i^T\beta\right]$$

So we can see that it matches the original equation of $U(\beta)$, hence our assumption was right.

(c), (d), (e), and (f)

The code below contains the entire code for the parts aforementioned. I have added comments in between to demonstrate what a particular piece of code is doing. Please refer to the cat statements and comments in between. The results are summarized below before the code.

c)

The implementation of Hamiltonian Monte Carlo can be found in the code below. This code is derived from Radford-Neal Paper. Also, to ease up the process, we are labelling virginica as 0 and versicolor as 1.

d)

Firstly, the dataset is divided into training and testing datasets. Then, we are generating 11000 samples of beta which contains 1000 for burn-in and 10000 for the testing stage. We are also seeing that we are getting acceptance rate of ~98%. Now, for each sampled beta, we are computing a label for it. To compute a label, we are rounding up the probability to get the label. Following the aforementioned method, I got a classification accuracy of 97.5% for predicting the y labels.

e)

The trace plots and histogram for the 4 features and one constant features can be found at the end of code.

f)

For this part, we used sigma = 2, epsilon = 0.2, and leapfrog steps, L = 20. Here, we are doing the average classification and then computing the accuracy of classification. We got average classification as 97.5%.

CODE

```
# source('HamiltonianMonteCarlo.r', echo = FALSE, keep.source
# = FALSE, max.deparse.length=10000)
require(animation)
```

```
## Loading required package: animation
```

```
## Warning: package 'animation' was built under R version 3.4.4
```

```
## Load and Store the data to use for the given question
dataset = iris
dataset = subset(dataset, Species != "setosa")

# if(dataset$Species == 'versicolor'){ dataset$Species = 1 }
# else{ dataset$Species = 0 }

# the upper solution was not working so found one alternative
# to that on StackOverflow Mapping the versicolor to 1 and
# virginica to 0
dataset$Species = ifelse(dataset$Species == "versicolor", 1,
    0)

## Debugging Statements print(dataset) dataset =
## as.matrix(dataset) print(dataset[,1])

# Seperate the dataset into features and labels
X = t(as.matrix(cbind(dataset[1:4], 1)))
Y = as.matrix(dataset[5])

# print(X[,1:30])
```

```
# print(t(rowMeans(X))) print(Y)

# Seperate the training and testing datasets
X_Train = matrix(0, 5, 60)
X_Test = matrix(0, 5, 40)
# print(dim(X_Train))

Y_Train = matrix(0, 60, 1)
Y_Test = matrix(0, 40, 1)

# For each species, first 30 rows will be training data and
# the rest 20 will be testing data
X_Train[, 1:30] = X[, 1:30]
X_Test[, 1:20] = X[, 31:50]
X_Train[, 31:60] = X[, 51:80]
X_Test[, 21:40] = X[, 81:100]

Y_Train[1:30, ] = Y[1:30, ]
Y_Test[1:20, ] = Y[31:50, ]
Y_Train[31:60, ] = Y[51:80, ]
Y_Test[21:40, ] = Y[81:100, ]

X_Train = X_Train - rowMeans(X_Train)
X_Test = X_Test - rowMeans(X_Test)
# print(X_Train) print(X_Test)

# initializing the beta, sigma and reject counter
beta_i = matrix(0, 5, 1)
sigma_i = 1
reject_counter = 0

#### Main methods start from here ####


# Method to calculate the norm of the beta vector
beta_norm = function(beta) {
    temp = as.vector(beta)
    temp = sqrt(sum(temp * temp))
    return(temp)
}

# The energy function
U = function(beta = beta_i, sigma = sigma_i, data = X_Train,
    labels = Y_Train) {
    temp_beta = beta_norm(beta)
    temp_mul = t(data) %*% beta
    temp_U = t(1 - labels) %*% temp_mul + sum(log(1 + exp(-temp_mul))) +
        (1/2 * sigma^2) * temp_beta^2
    return(temp_U)
}

# Gradient of the energy function
grad_U = function(beta = beta_i, sigma = sigma_i, data = X_Train,
```

```
    labels = Y_Train) {
    temp_mul = t(data) %*% beta
    temp_mat = 1 - (exp(-temp_mul)/(1 + exp(-temp_mul))) - labels
    temp_gradient = data %*% temp_mat + ((1/sigma^2) * beta)
    return(temp_gradient)
}

## Code for Q.2c i.e. implementation of Hamiltonian Monte
## Carlo routine for sampling from the posterior of Beta Code
## from Radford Neal's paper (slightly modified)

HMC = function(U, grad_U, epsilon, L, current_q, anim = FALSE) {
    q = current_q
    p = rnorm(length(q), 0, 1)  # independent standard normal variates
    current_p = p

    qList = q
    pList = p

    ## Make a half step for momentum at the beginning
    p = p - epsilon * grad_U(q)/2

    ## Alternate full steps for position and momentum
    for (i in 1:(L - 1)) {
        ## Make a full step for the position
        q = q + epsilon * p

        ## Make a full step for the momentum, except at end of
        ## trajectory
        p = p - epsilon * grad_U(q)

        if (anim) {
            H = function(x, y) {
                U(x) + 0.5 * y^2
            }

            t = seq(-2, 2, 0.01)
            Hvals = outer(t, t, H)
            contour(t, t, Hvals, levels = seq(0, 3, 0.1), asp = 1,
                drawlabel = FALSE, xlab = "q", ylab = "p")
            lines(qList, pList, lwd = 2, col = "red")
            ani.record()
            pList = c(pList, p)
            qList = c(qList, q)
        }
    }

    q = q + epsilon * p

    ## Make a half step for momentum at the end.
    p = p - epsilon * grad_U(q)/2

    ## Negate momentum at end of trajectory to make the proposal
```

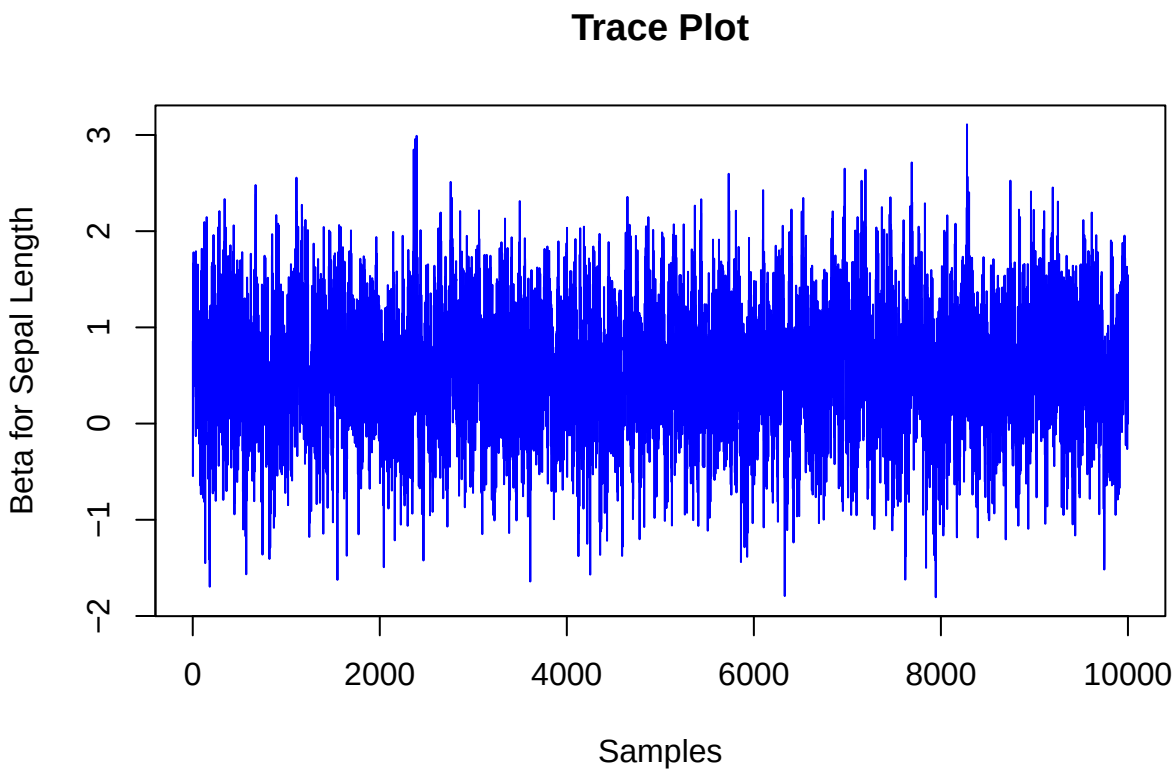
```
    ## symmetric
    p = -p

    ## Evaluate potential and kinetic energies at start and end of
    ## trajectory
    current_U = U(current_q)
    current_K = sum(current_p^2)/2
    proposed_U = U(q)
    proposed_K = sum(p^2)/2

    ## Accept or reject the state at end of trajectory, returning
    ## either the position at the end of the trajectory or the
    ## initial position
    if (current_U + current_K > proposed_U + proposed_K)
        return(q)  # reject
 else if (runif(1) < exp(current_U - proposed_U + current_K -
        proposed_K))
        return(q)  # accept
 else {
        reject_counter <<- reject_counter + 1
        return(current_q)  # reject
    }
}

# Driver function to run Hamiltonian Monte Carlo
HMC_util = function(epsilon, L, samples, burn) {
    reject = 0
    beta = beta_i

    # starting with the burn-in samples
    for (i in 1:burn) {
        beta = HMC(U, grad_U, epsilon, L, beta)
    }

    beta_l = matrix(0, samples, 5)
    beta_l[1, ] = beta

    # Start with the samples/iterations
    it = samples - 1
    beta_updated = beta

    for (i in 1:it) {
        beta_updated = HMC(U, grad_U, epsilon, L, beta_updated)
        temp = beta - beta_updated

        # if(beta_norm(temp) < 0.0001){ reject = reject + 1 }
        beta_l[i, ] = beta_updated
    }
    accept_percent = (1 - (reject_counter/samples)) * 100

    cat("Acceptance Rate is ", accept_percent, "\n")
    return(beta_l)
}
```

```
## Generating Plots - Trace Plots and Histogram

trace_beta = function(beta, ftr_num, feature) {
    row = dim(beta)[1]
    plot(1:row, beta[, ftr_num], lwd = 1, type = "l", col = "blue",
        main = "Trace Plot", xlab = "Samples", ylab = paste("Beta for ",
            feature, sep = ""))
}

hist_plot = function(beta, ftr_num, feature) {
    temp = beta[, ftr_num]
    hist(temp, freq = FALSE, breaks = 75, col = "blue", main = "Histogram",
        xlab = "Samples", ylab = paste("Beta for ", feature,
            sep = ""))
}

## Classification process

classifier = function(sample_beta, testing = X_Test) {

    data_points = dim(testing)[2]
    sample_count = dim(sample_beta)[1]
    updated_labels = matrix(0, data_points, 1)
    probability_matrix = matrix(0, data_points, 1)

    # loop through all the values and chose one label for each
    # beta by prediction
    for (i in 1:sample_count) {
        beta = sample_beta[i, ]
        temp_mat = t(testing) %*% beta
        temp_prob = 1 + exp(-temp_mat)
        probability_matrix = probability_matrix + (1/temp_prob)
    }

    probability_matrix = probability_matrix/sample_count
    updated_labels[probability_matrix > 0.5] = 1
    return(probability_matrix)
}

# To get the average prediction
average_classifier = function(sample_beta, testing = X_Test) {

    data_points = dim(testing)[2]
    sample_count = dim(sample_beta)[1]
    average_beta = colMeans(sample_beta)
    updated_labels = matrix(0, data_points, 1)
    probability_matrix = matrix(0, data_points, 1)
    temp_mat = t(testing) %*% average_beta
    temp_prob = 1 + exp(-temp_mat)
    probability_matrix = probability_matrix + (1/temp_prob)
    updated_labels[probability_matrix > 0.5] = 1
    return(updated_labels)
}
```

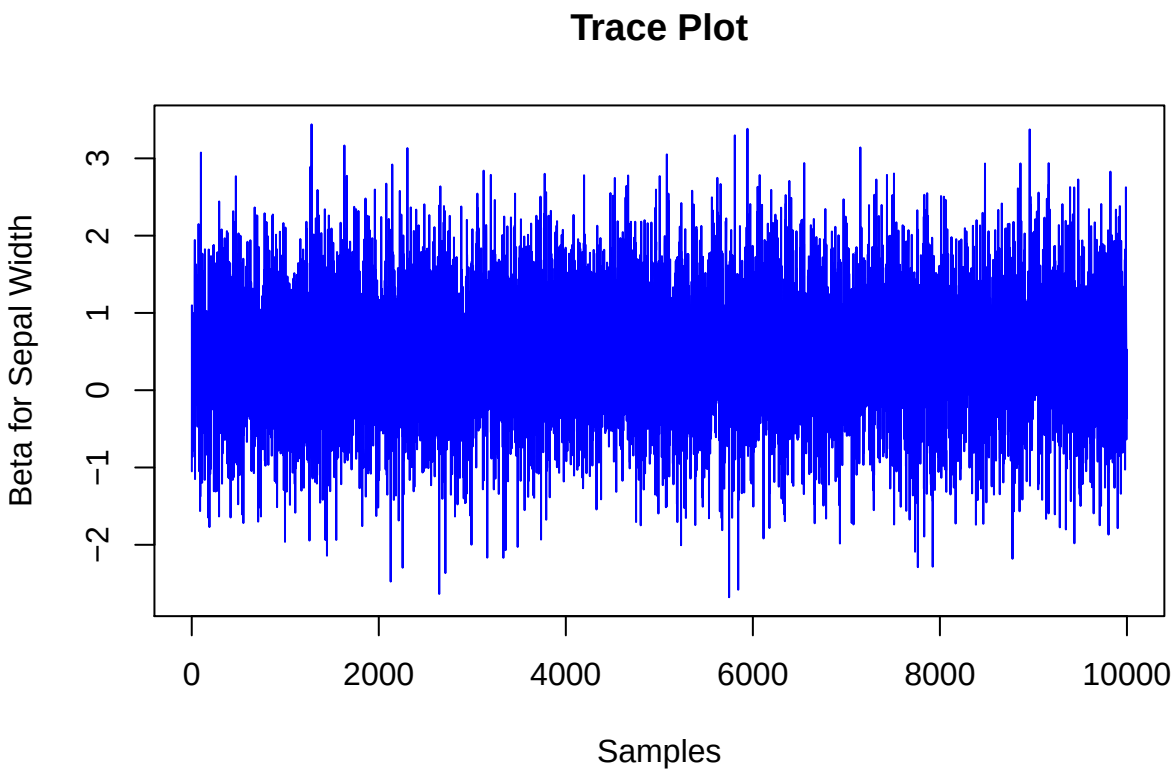
```r
# To get the Monte Carlo estimate of the Posterior Predictive
# probability
posterior_probability = function(sample_beta, testing = X_Test) {
    data_points = dim(testing)[2]
    sample_count = dim(sample_beta)[1]
    probability_matrix = matrix(0, data_points, 1)

    for (i in 1:sample_count) {
        beta = sample_beta[i, ]
        temp_mat = t(testing) %*% beta
        probability_matrix = probability_matrix + (1/(1 + exp(-temp_mat)))
    }
    probability_matrix = probability_matrix/sample_count
    return(probability_matrix)
}

# Code to get the accuracy of classification
testing_classification_accuracy = function(labels_predicted,
    labels_true = Y_Test) {
    data_count = dim(labels_predicted)[1]
    temp = matrix(0, data_count, 1)
    temp[labels_predicted == labels_true] = 1
    accuracy = (sum(temp)/data_count) * 100
    cat("The accuracy for classification is ", accuracy, "\n")
}

## Driver for Question 2

# Question 1c.

# Sample the 10000 samples of beta with epsilon = 0.2, L =
# 20, burn-in = 1000
sample_beta = HMC_util(0.2, 20, 10000, 1000)
```

```
## Acceptance Rate is  98.34
```

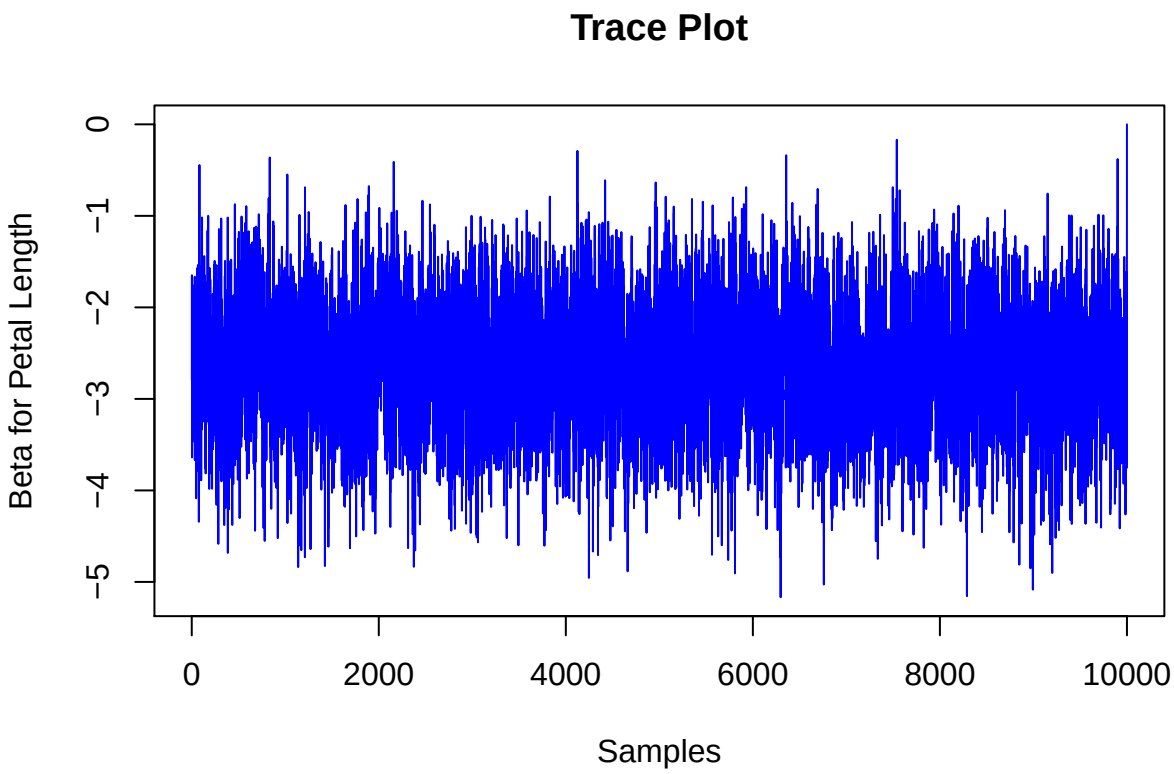
```r
# print(sample_beta)

## Question 1e

## Draw Trace Plots and Histograms of all the 5 variables
## (i.e. 1D constants)
trace_beta(sample_beta, 1, "Sepal Length")
```



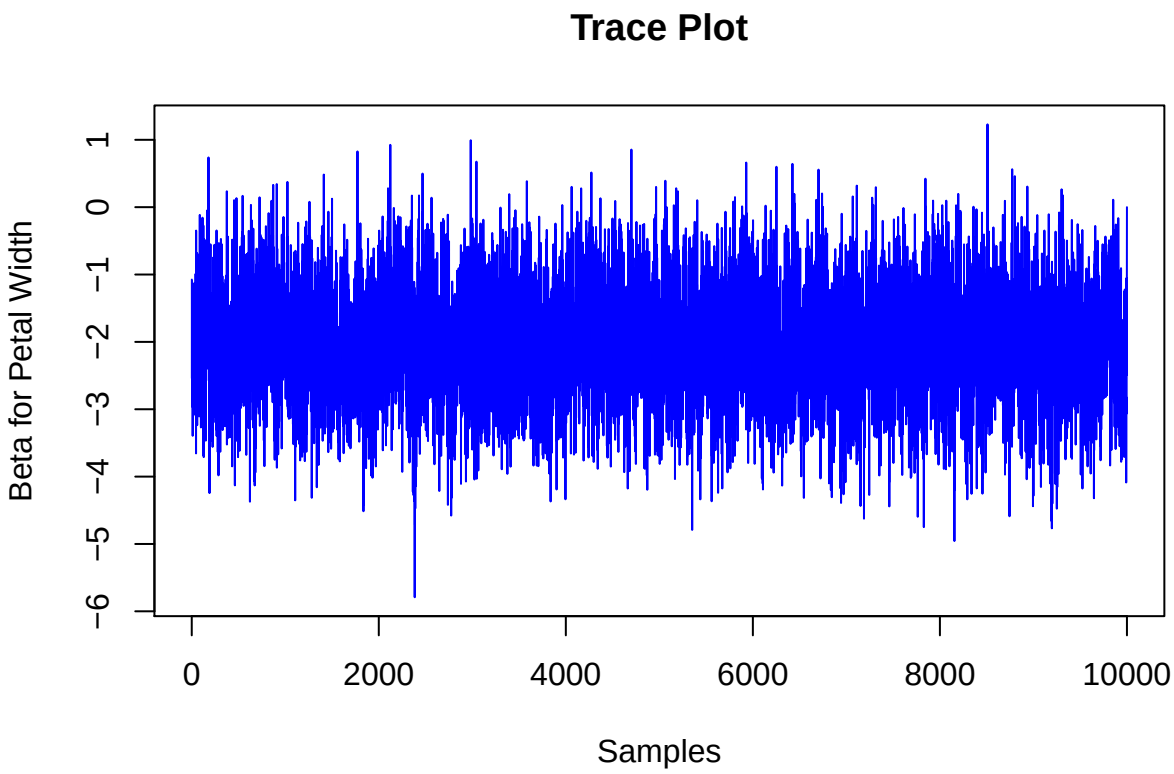

```
trace_beta(sample_beta, 2, "Sepal Width")
```



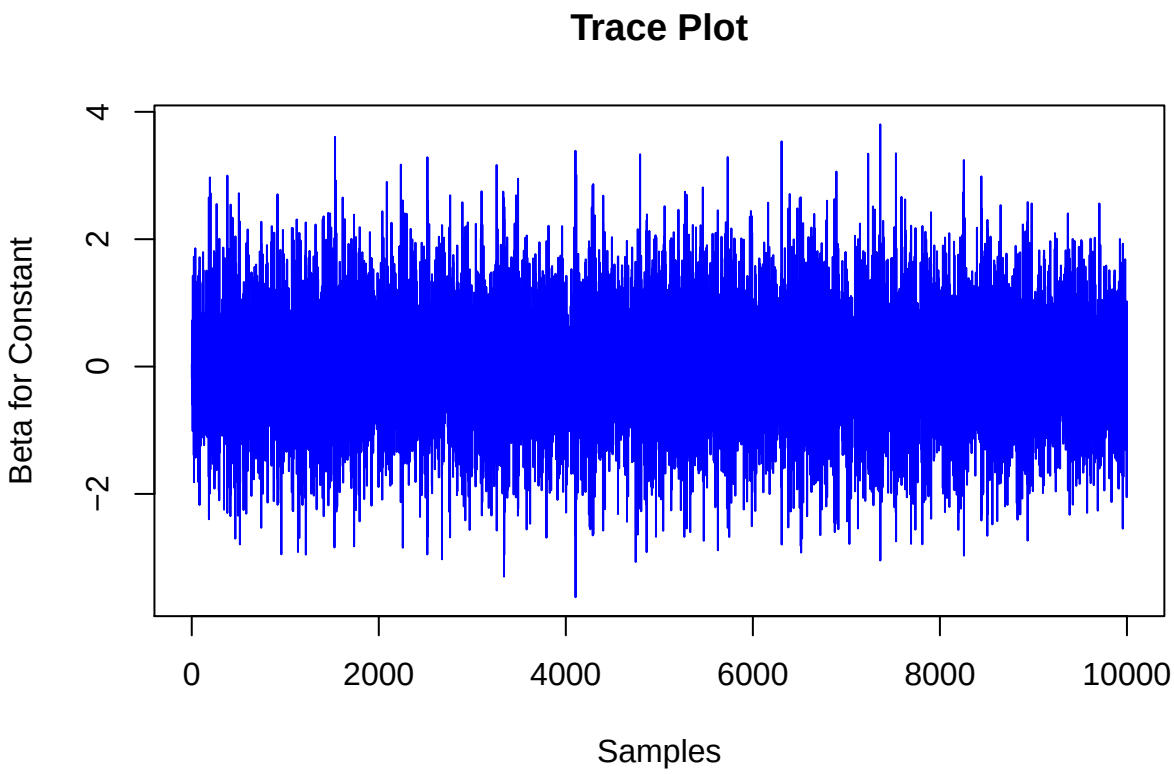

```
trace_beta(sample_beta, 3, "Petal Length")
```



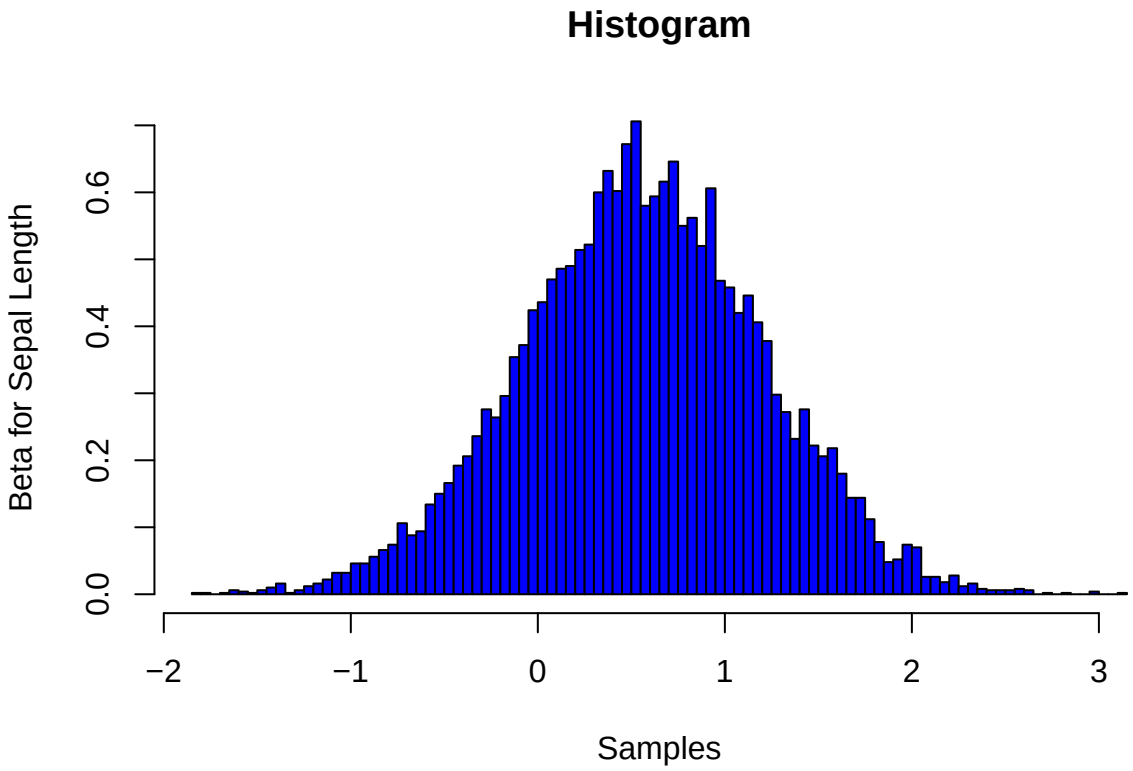

```
trace_beta(sample_beta, 4, "Petal Width")
```



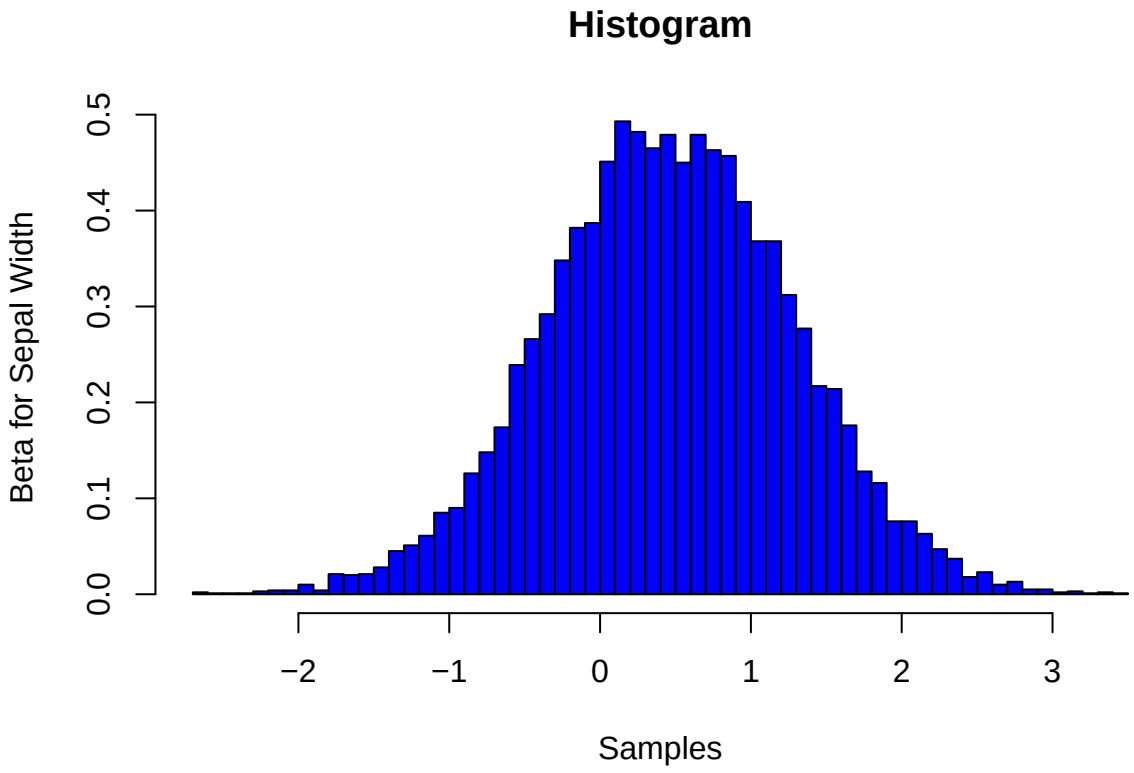

```
trace_beta(sample_beta, 5, "Constant")
```



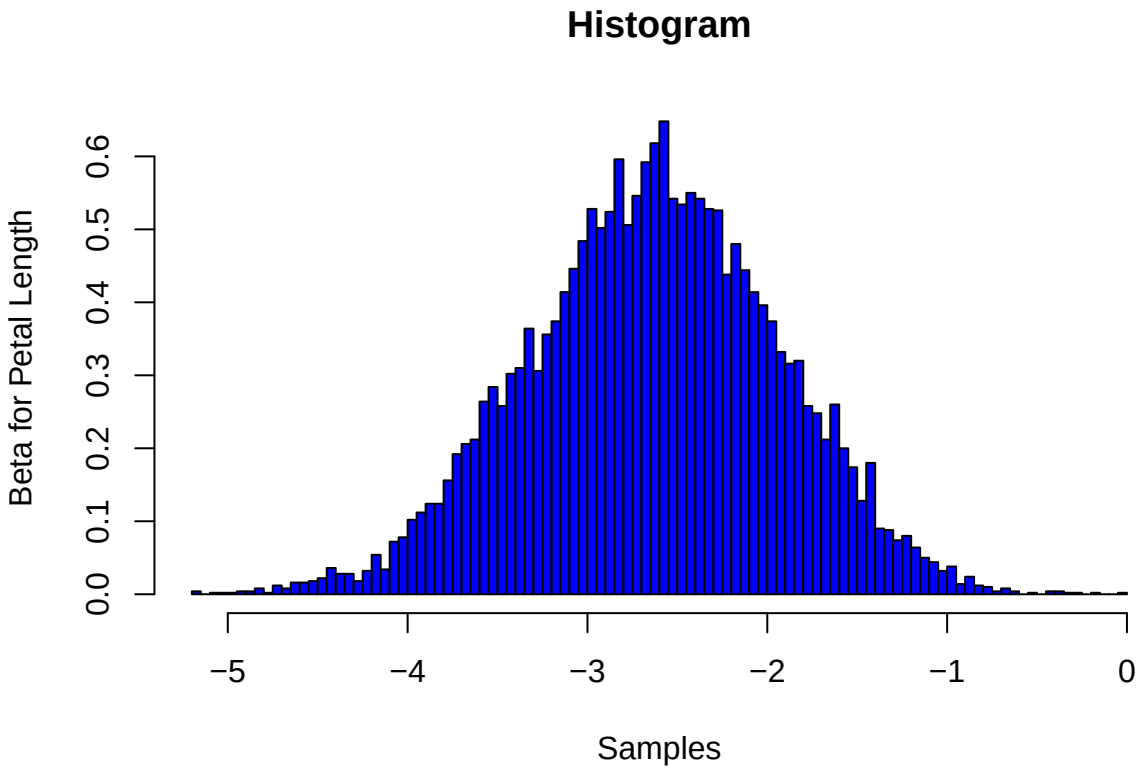

```
hist_plot(sample_beta, 1, "Sepal Length")
```



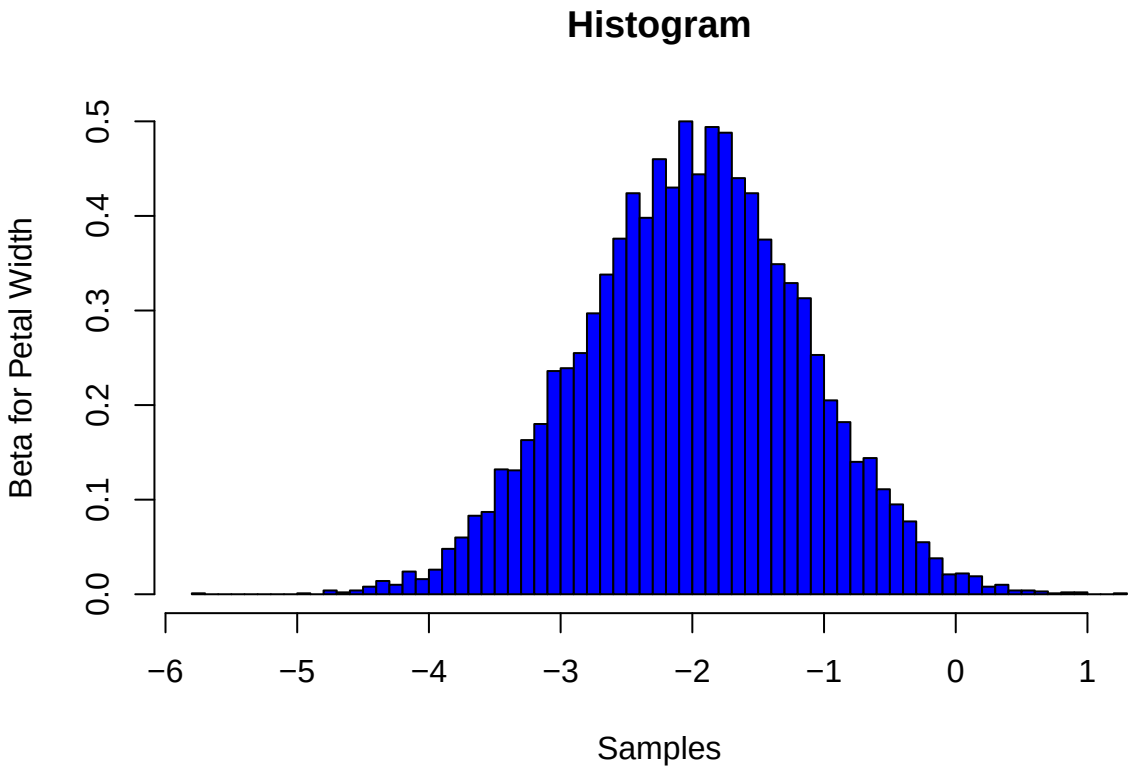

```
hist_plot(sample_beta, 2, "Sepal Width")
```



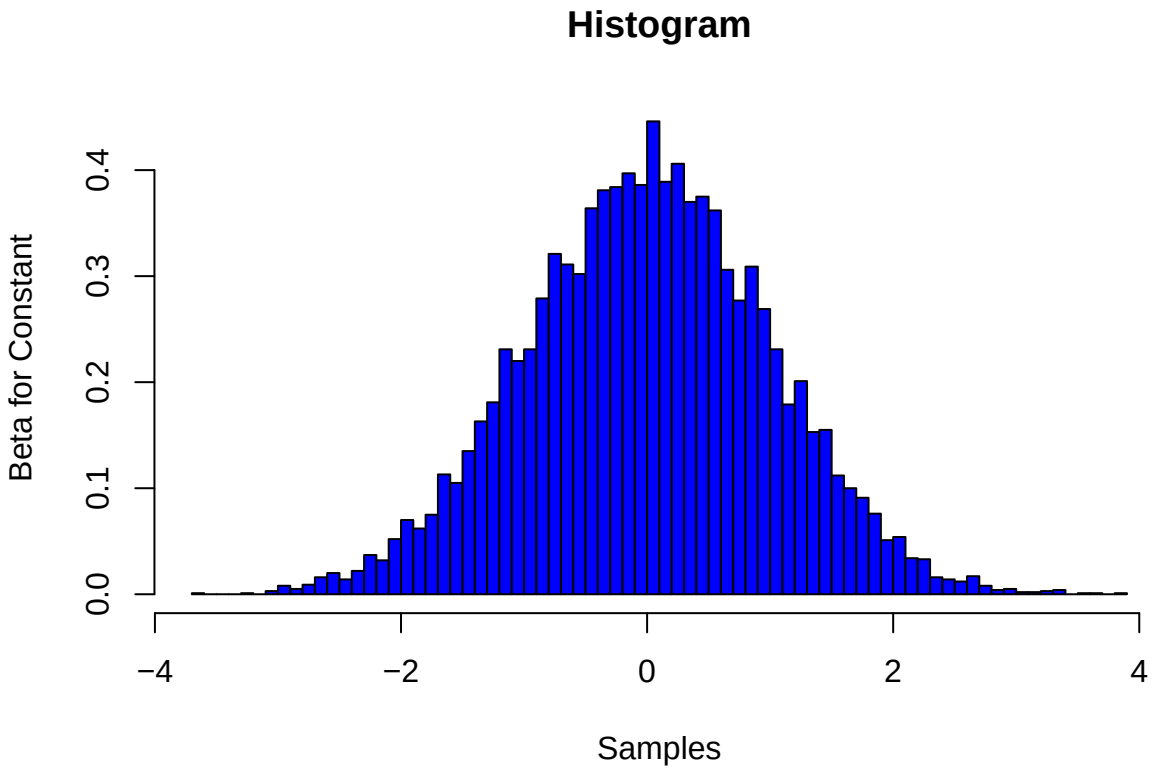

```
hist_plot(sample_beta, 3, "Petal Length")
```




```
hist_plot(sample_beta, 4, "Petal Width")
```




```
hist_plot(sample_beta, 5, "Constant")
```




```
# print(sample_beta) print(sample_beta[,1])

## Question 1d.

## Get the prediction for y of the class labels for the
## testing data
prediction = classifier(sample_beta)
prediction = round(prediction)
# print(Y_Test)

cat("For the predicted labels, the accuracy of classification is calculated here\n")
```

```
## For the predicted labels, the accuracy of classification is calculated here
```

```
testing_classification_accuracy(prediction)
```

```
## The accuracy for classification is  97.5
```

```
cat("The posterior predictive probability for each testing data point \n")
```

```
## The posterior predictive probability for each testing data point
```

```
posterior = posterior_probability(sample_beta)
print(posterior)
```

```
##              [,1]
##  [1,] 0.95185875
##  [2,] 0.96759841
##  [3,] 0.94821368
```

```
##  [4,] 0.31435518
##  [5,] 0.67808550
##  [6,] 0.74185977
##  [7,] 0.73083507
##  [8,] 0.82067522
##  [9,] 0.90427908
## [10,] 0.90022871
## [11,] 0.80813273
## [12,] 0.75035486
## [13,] 0.93193507
## [14,] 0.98080501
## [15,] 0.86706045
## [16,] 0.90409649
## [17,] 0.88350529
## [18,] 0.88669944
## [19,] 0.98995972
## [20,] 0.90299057
## [21,] 0.05437595
## [22,] 0.05547215
## [23,] 0.05187342
## [24,] 0.40710662
## [25,] 0.17465048
## [26,] 0.03635869
## [27,] 0.04784326
## [28,] 0.14466315
## [29,] 0.43415536
## [30,] 0.14200251
## [31,] 0.04997256
## [32,] 0.20010320
## [33,] 0.18791055
## [34,] 0.03269844
## [35,] 0.03744700
## [36,] 0.13984200
## [37,] 0.26455079
## [38,] 0.19634230
## [39,] 0.08450093
## [40,] 0.25181948
```

```
## Question 1f.

## Do for the average classification
cat("For the average classification, the accuracy of classification is \n")
```

```
## For the average classification, the accuracy of classification is
```

```
prediction_avg = average_classifier(sample_beta)
prediction_avg = round(prediction_avg)
testing_classification_accuracy(prediction_avg)
```

```
## The accuracy for classification is  97.5
```