Bayesian Learning - Lab 3

Kevin Neville, Gustav Sternelöv

May, 3rd, 2016

Assignment 1 - Normal model, mixture of normal model with semi-conjugate prior.

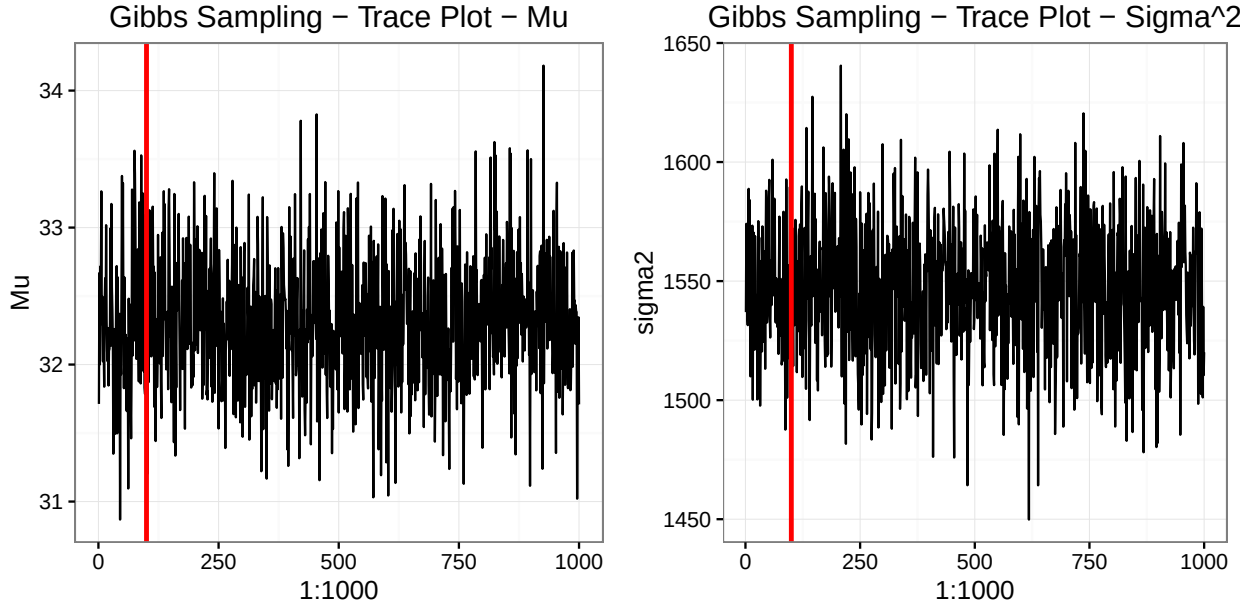
a) Normal model.

i)

The code used to implement the Gibbs sampler that simulates from the joint posterior can be seen in the appendix *R-code*.

ii)

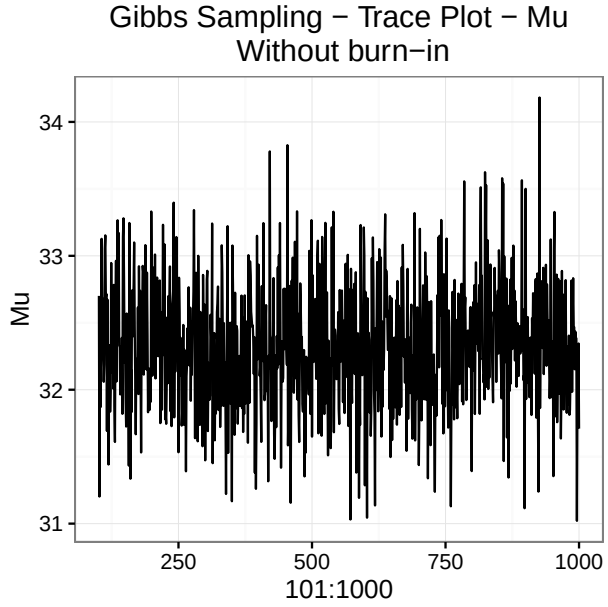
The Gibbs sampler from *i)* is tested and it is of interest to investigate the convergence of the chains and if the sampler is efficient. One way to check this is by looking at trace plots and auto-correlation plots.

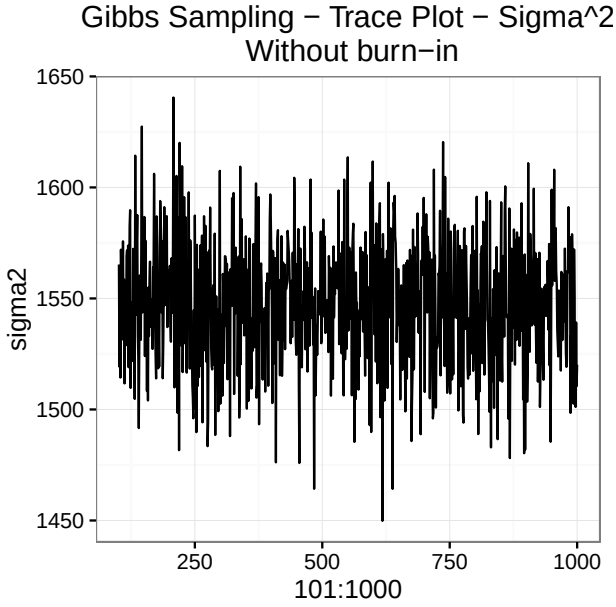


By the look of the plots above it seems like the chain has converged for both μ and σ^2. The chains converges quickly and if there is a burn-in period, it is thought to be short. Out of the 1000 iterations, perhaps 10 % of the iterations in both cases can be classified as belonging to the burn-in period. Even though it is hard to see a specific burn-in period it is reasonable to make the assumption that some proportion of the first iterations not have converged and should be discarded.

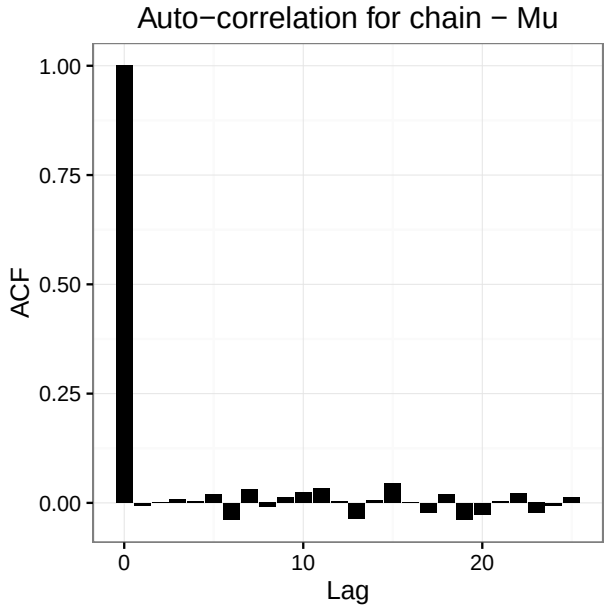
The chains are thought to have converged since they have settled rather well and appears to have a random pattern throughout the whole chain.

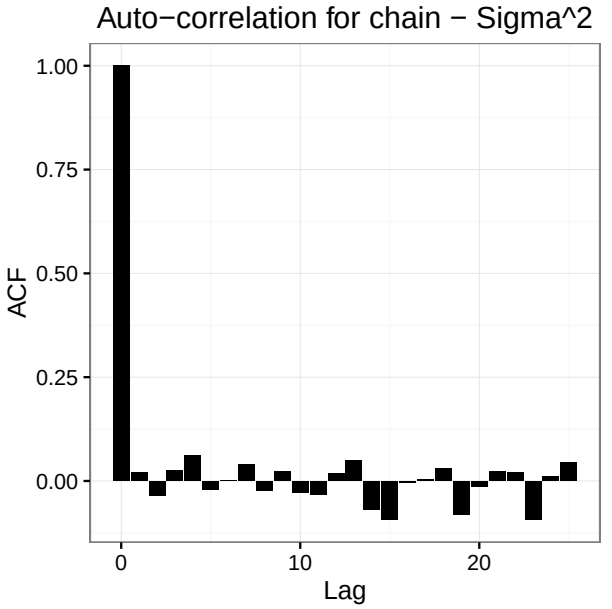
How the chains looks with the burn-in period discarded is shown by the trace plots below.





In the comments above about the trace plots we mentioned that the values seem to be rather uncorrelated, that the correlation between one value and the next values seem to be low. This is investigated more closely by looking at the auto-correlation of the chains with the burn-in period removed.





As we thought, the generated values from the Gibbs sampler do not have a strong auto-correlation. The conlcusion from the visual examination of the Gibbs sampler then is that the chain both have converged and seem to be rather efficient.

b) Mixture normal model.

i)

Mattias's code is used for this assignment and added to the appendix at the end of the report. The values for the prior hyperparameters are presented below.

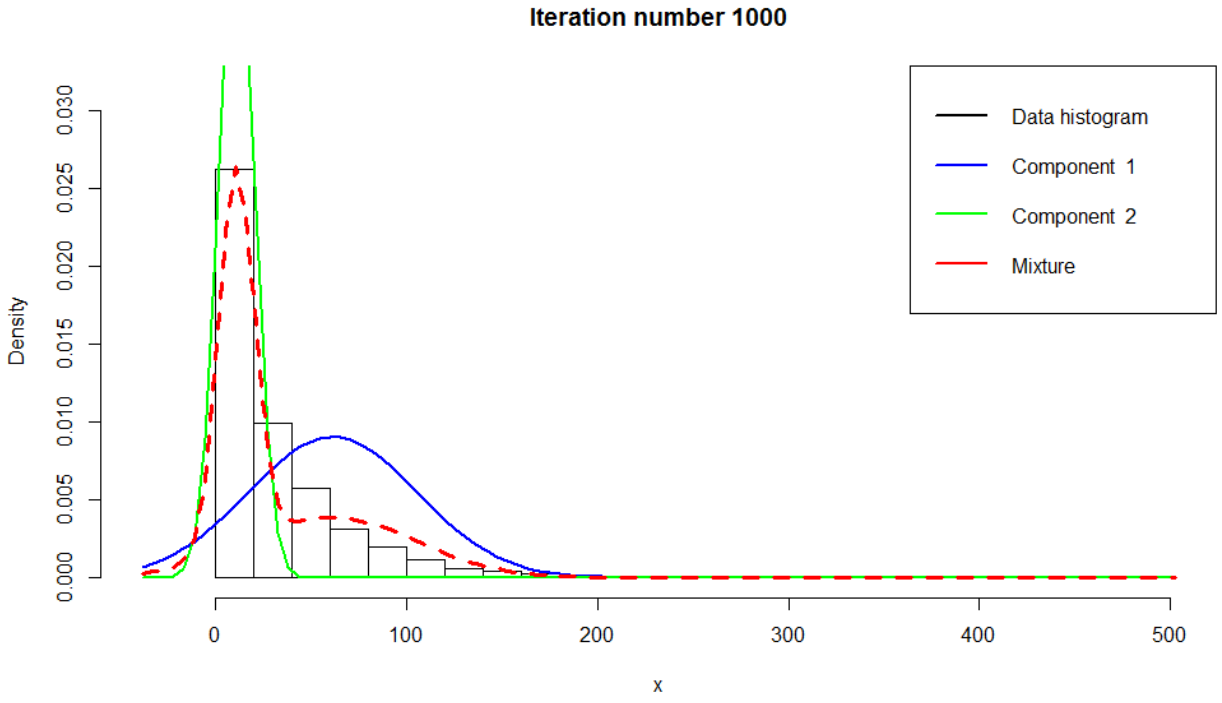
The priors for the two components, are set as following:

- α_0 = 10,10
- μ_0 = Mean of x, i.e 32.2681.
- τ_0^2 = 10,10
- σ_0^2 = Is set to the variance of x, i.e 1545.771.
- Degrees of freedom = 4

Some different priors were tested and the values presented above seemed to be the most suitable ones.

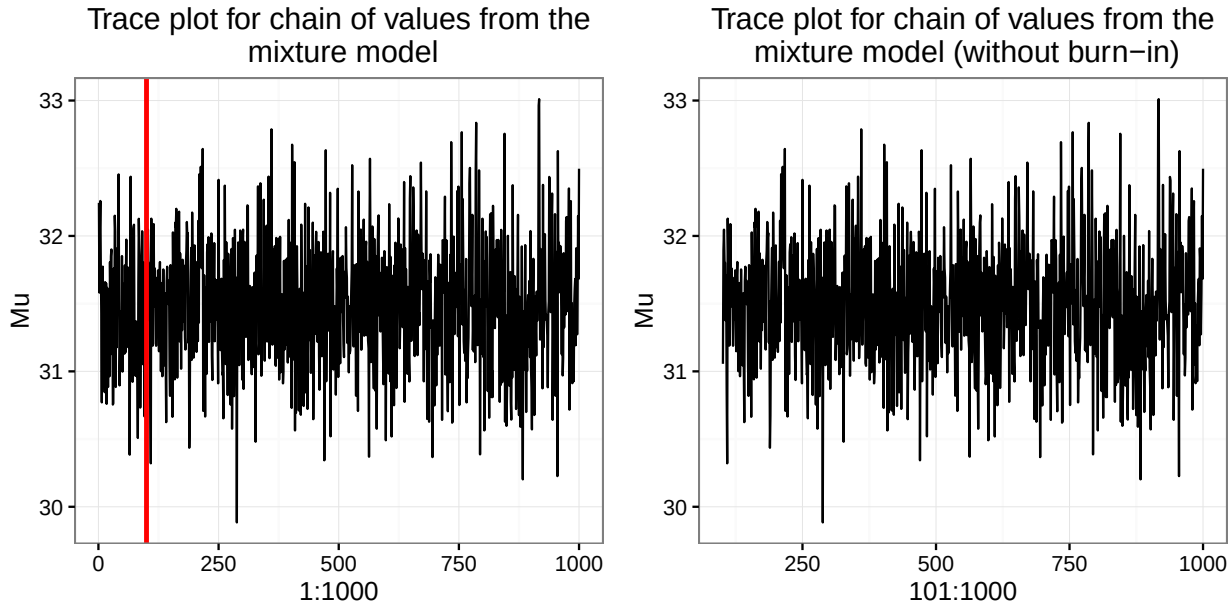
ii)

The resulting distribution given for both the normals and the mixture of the normals after 1000 iterations is shown by the plot below.



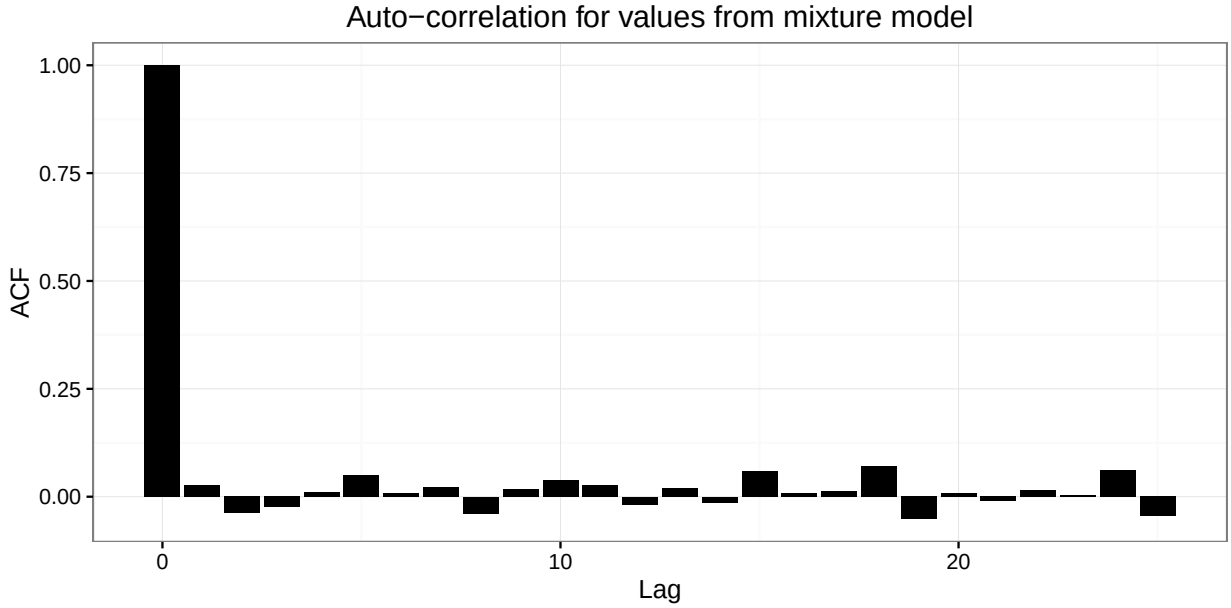
It can be seen the the mixture of the two normals quite well fits the observed data.

The convergence and efficiency of the Gibbs sampler is analyzed by looking at a trace plot for the chain of μ values given by the mixture of the normals.



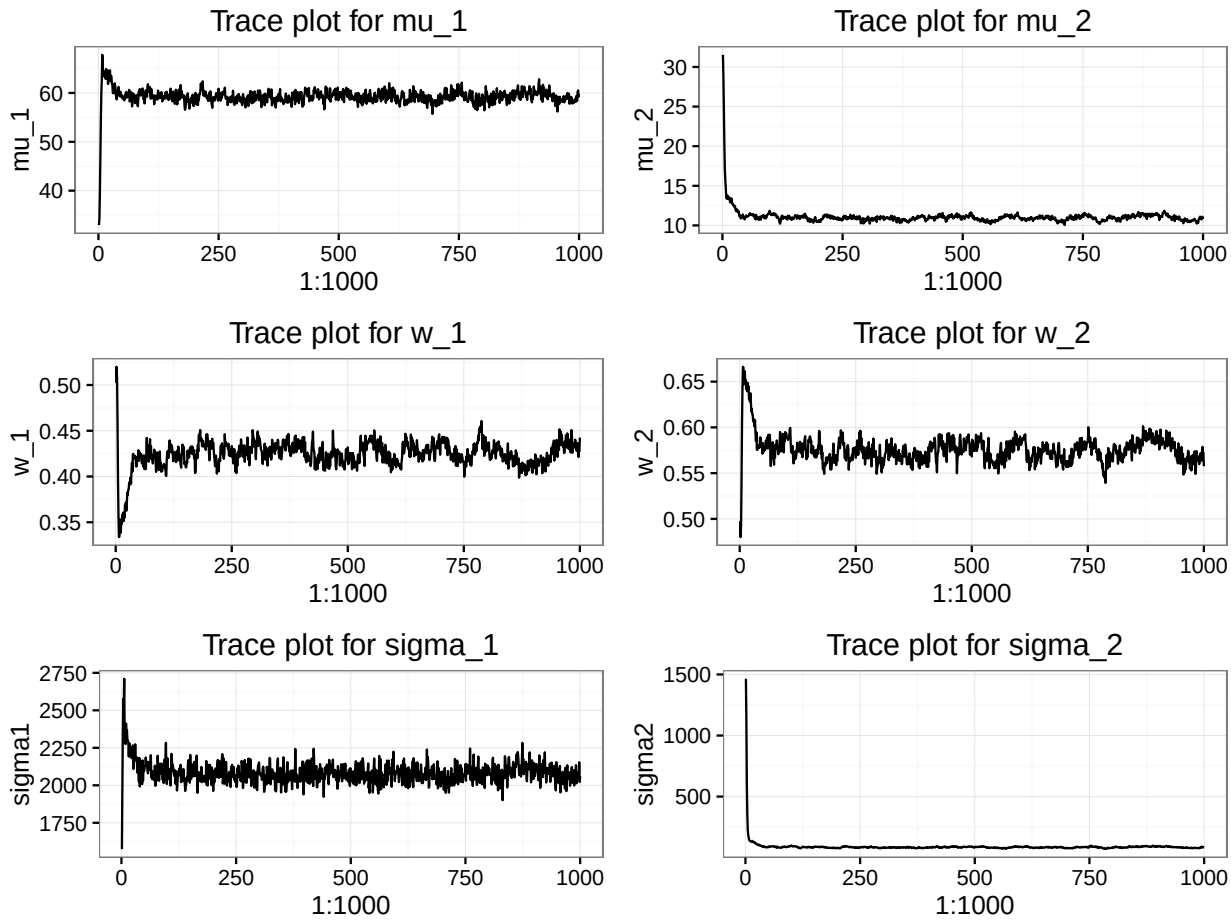
Again, it seems like the chain have converged rather rapidly. It settles quickly and the generated values does not get stuck nor follow any clear pattern. Looking at the burn-in period it is concluded that it probably is rather short. Hence, just the first 100 (10 %) observations are discarded because of the burn-in period.

The autocorrelation is also visualised in order to investigate the convergence and efficiency of the chain.



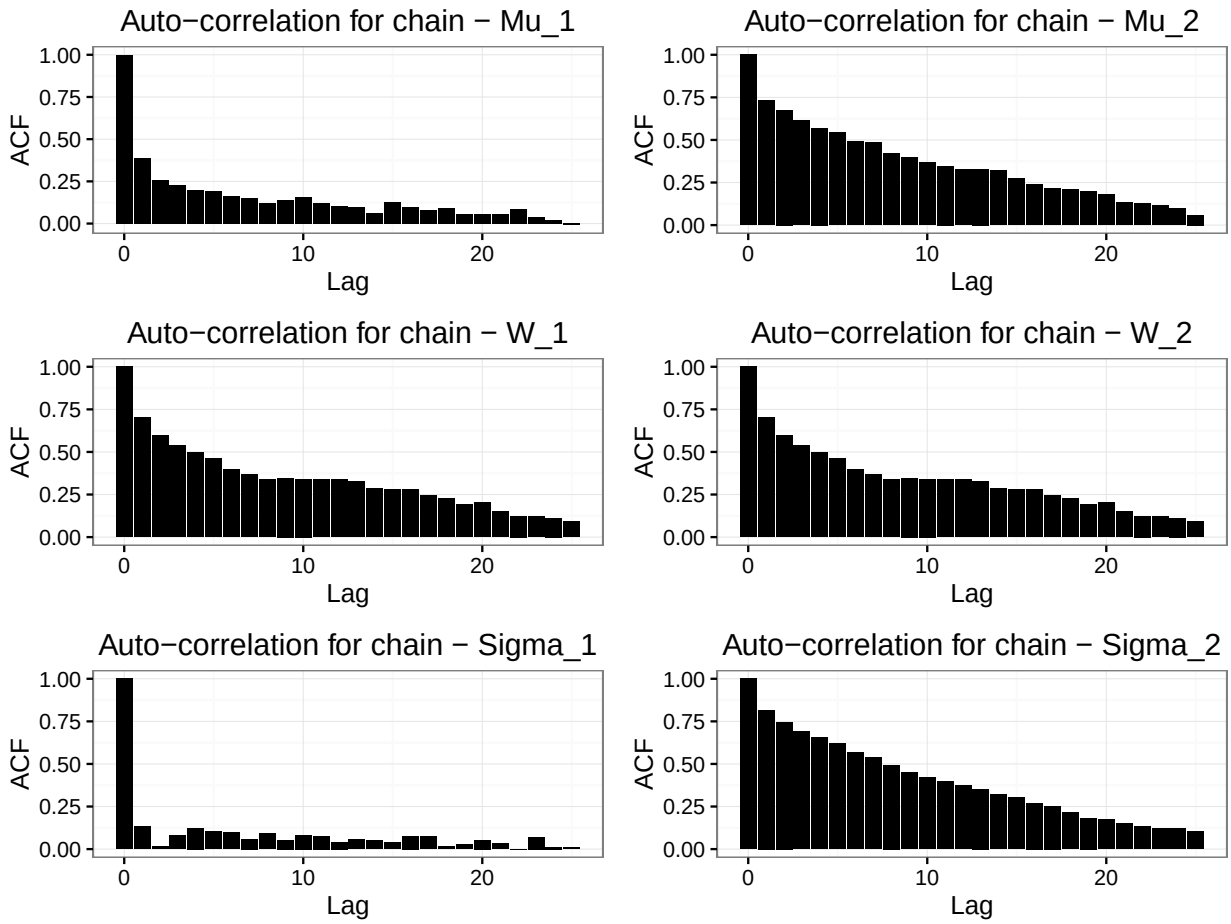
The correlation between values from iterations close to each other is low which speaks in favor of the efficiency of the sampler.

So far so good, but we shall also look at the other parameters that have been sampled in the mixture model. The trace plots for μ_1, μ_2, W_1, W_2, σ_1 and σ_2 are presented below.



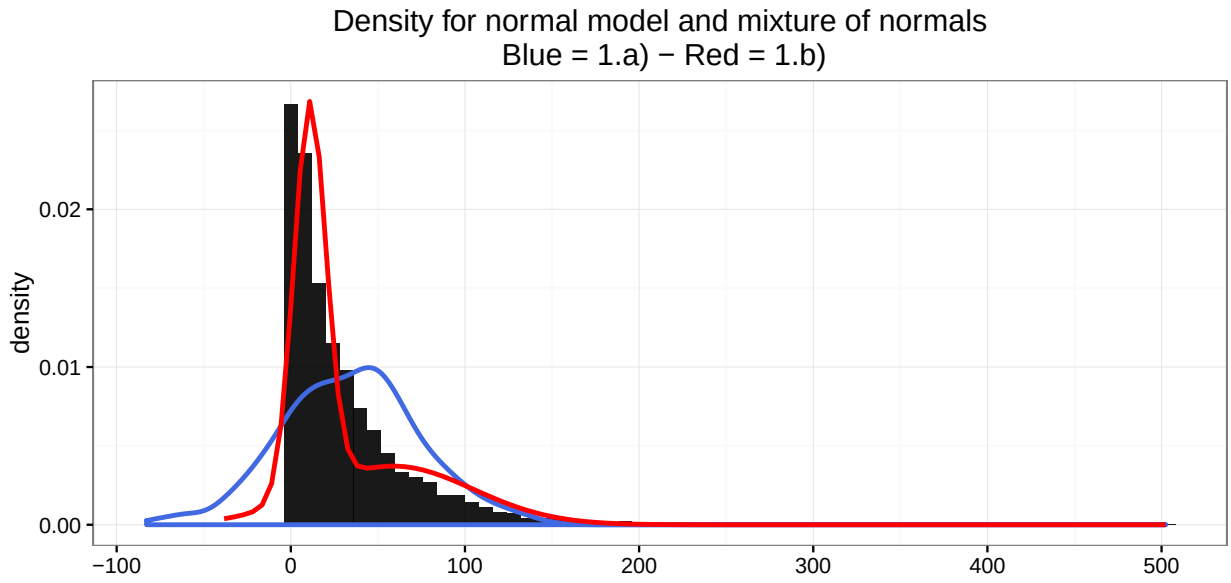
The trace plots appears to indicate that all the chains except the ones for *w_1* and *w_2* have converged.

We also investigate the convergence and efficiency of the parameters by the auto-correlation of the respective chains.



Compared to the sampler in *a)* it is evident that this sampler is less efficient. All parameters except sigma_1 have a high auto-correlation in their chains.

c) Graphical comparison.



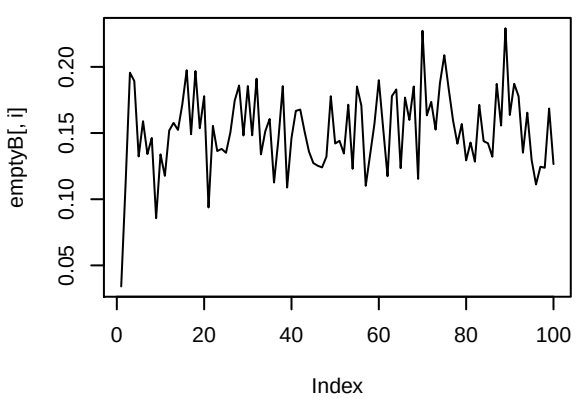
With no doubt, the mixture of normals model from *b)* is better in terms of fitting. The normal density is poorly fitted to the data, and even has a large probability mass for negative values. The mixture model also has some probability for negative values, but not as much as the normal model. The mixture model is also much better fitted to the observed values.

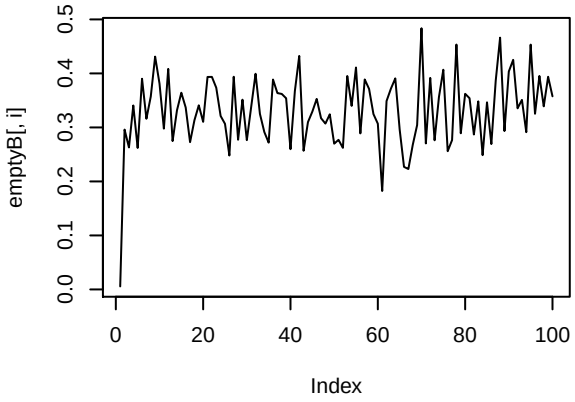
Assignment 2 - Binary regression models

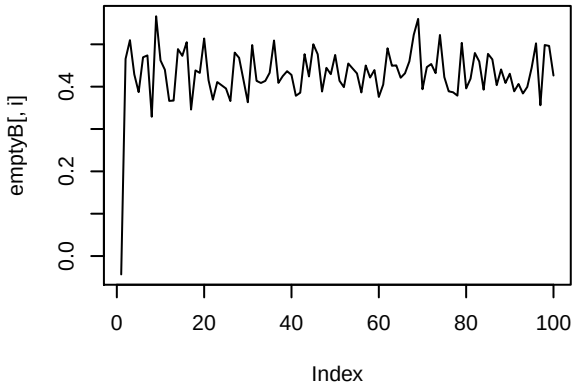
a-d)

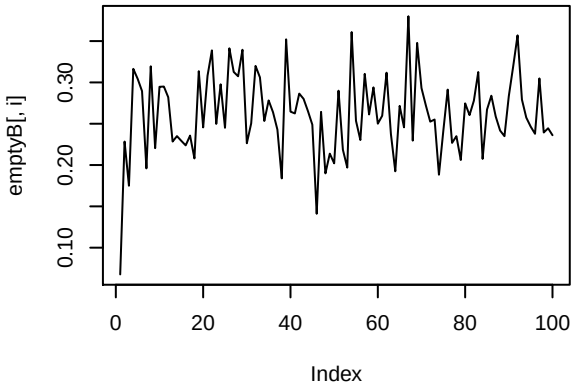
The code *OptimizeSpamR* is used to analyze the spam data set. A data augmentation Gibbs sampler is also implemented and the code for both methods can be seen in the appendix at the end of the report.

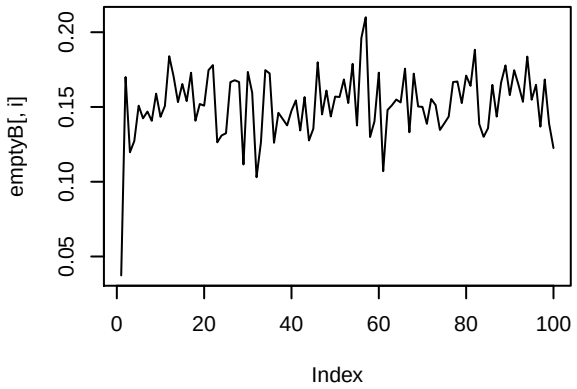
Only 100 draws are made with the Gibbs sampler since it take lots of time to run the sampler. The convergence for the β parameters is investigated with trace plots.

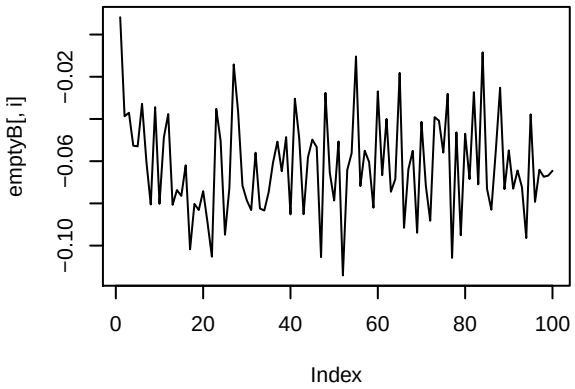

emptyB[, i]
0.05
0.10
0.15
0.20
0
20
40
60
80
100
Index

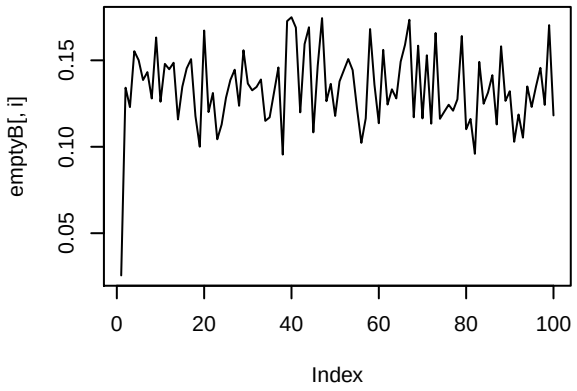

emptyB[, i]
0.0
0.1
0.2
0.3
0.4
0.5
0
20
40
60
80
100
Index

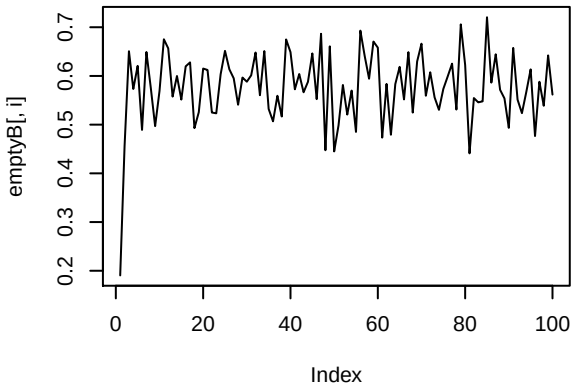

emptyB[, i]
0.0
0.2
0.4
0
20
40
60
80
100
Index

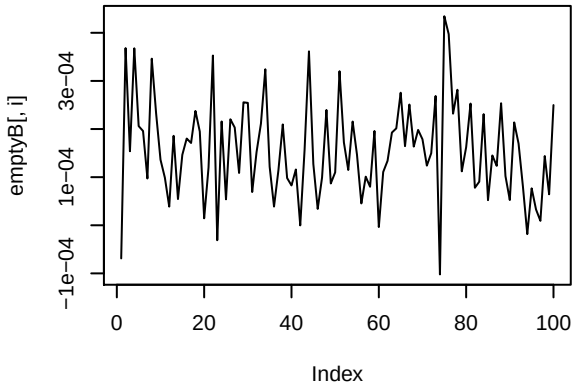

emptyB[, i]
0.10
0.20
0.30
0
20
40
60
80
100
Index

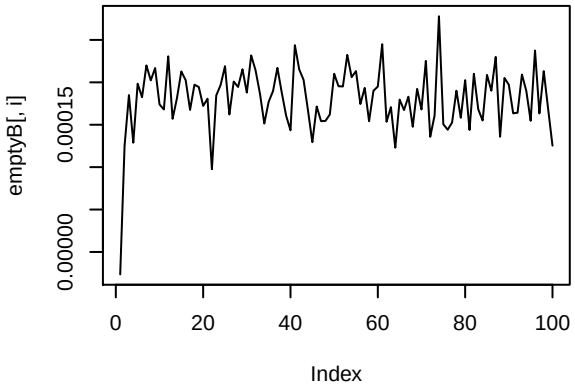

emptyB[, i]
0.05
0.10
0.15
0.20
0
20
40
60
80
100
Index

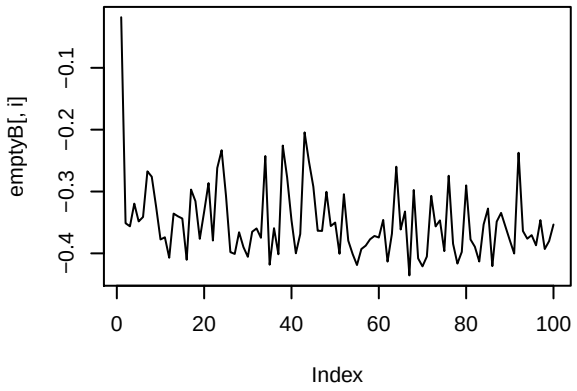

emptyB[, i]
−0.10
−0.06
−0.02
0
20
40
60
80
100
Index

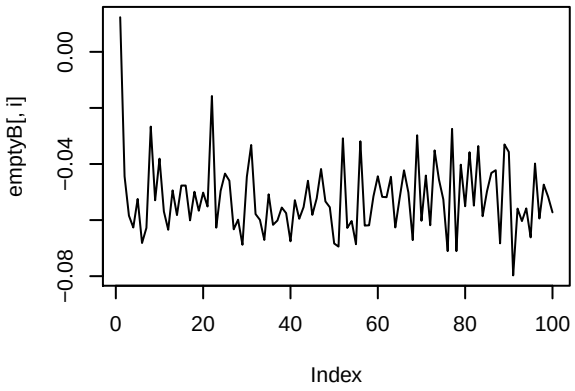

emptyB[, i]
0.05
0.10
0.15
0
20
40
60
80
100
Index

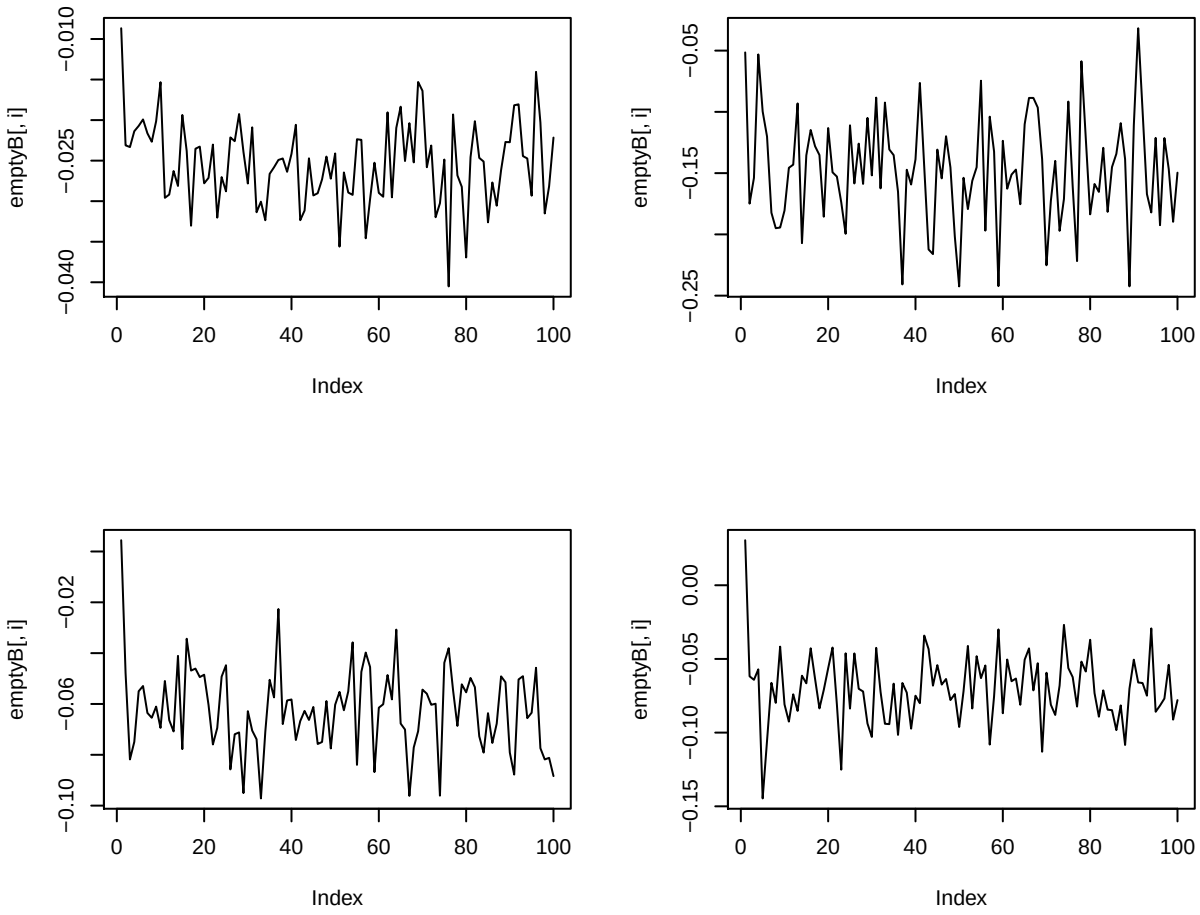

emptyB[, i]
0.2
0.3
0.4
0.5
0.6
0.7
0
20
40
60
80
100
Index


emptyB[, i]
−1e−04
1e−04
3e−04
0
20
40
60
80
100
Index


emptyB[, i]
0.00000
0.00015
0
20
40
60
80
100
Index


emptyB[, i]
−0.4
−0.3
−0.2
−0.1
0
20
40
60
80
100
Index


emptyB[, i]
−0.08
−0.04
0.00
0
20
40
60
80
100
Index



For all the trace plots it seems like the parameters have converged, even though the sample size is small. The effective sample size for the respective parameters is computed and it can be seen that the sampler is really effective.

```
##        X1        X2        X3        X4        X5        X6        X7
## 100.00000 100.00000 100.00000 100.00000 100.00000 100.00000 100.00000
```

```
##        X8        X9       X10       X11       X12       X13       X14
## 100.00000 100.00000 100.00000 100.00000 142.24277 100.00000 100.00000
##       X15       X16
##  73.42013 100.00000
```

The results for the respective method for both methods are compared against each other. First we look at the coefficients for the beta parameters and their standard deviation.

```
##           covs     OptimBeta     GibbsBeta      OptimStd      GibbsStd
## 1          our  0.2732466085  0.1265514755 0.03746767252 0.02246517374
## 2         over  0.6815867686  0.3576016888 0.09982156730 0.05492811498
## 3       remove  1.2486100444  0.4262770726 0.11709050618 0.03876139625
## 4     internet  0.4539457721  0.2361031012 0.07163539804 0.03708933048
## 5         free  0.6307512633  0.1225401494 0.05643740772 0.01823416387
## 6          hpl -0.7684908652 -0.0645575697 0.18306345858 0.01965785646
## 7           X.  0.1926618541  0.1181389788 0.02536962597 0.01856160710
## 8         X..1  3.1970606303  0.5616869282 0.24165952645 0.06270174572
## 9    CapRunMax  0.0052354006  0.0002498163 0.00066296373 0.00008693620
## 10 CapRunTotal  0.0004274572  0.0001253610 0.00005319823 0.00002818102
## 11       const -0.7398524569 -0.3533767510 0.04360858067 0.02338600660
## 12          hp -0.8940685712 -0.0572359251 0.11006094081 0.01042147004
## 13      george -4.0724676136 -0.0221541628 0.51974310474 0.00445690156
## 14       X1999 -0.3214764568 -0.1496230744 0.08927326680 0.03583939937
## 15          re -0.4075869443 -0.0883803885 0.06797134633 0.01479810880
## 16         edu -0.8881672861 -0.0779347075 0.11284434088 0.01651035636
```

For both the Optim and the Gibbs is the sign the same for all parameters. The beta parameters given by the Optim method has in general higher vales compared to those from the Gibbs sampler. The standard deviations are lower for the Gibbs sampler so they seem to be more precise, but perhaps this also is an effect of the Gibbs coefficients havning lower values.

We also compare the results by looking at confusion matrices for the Gibbs sampler and the optimizer.

```
##    y_fitGibbs
## y      0    1
##   0 2685  103
##   1  588 1225
```

```
##    y_fitOptim
## y      0    1
##   0 2656  132
##   1  286 1527
```

The misclassification rate for the Gibbs sampler is 0.1501847 and for the optimizer 0.0908498. For both methods is the misclassification rather low but of the two does the optimizer seem to perform slightly better than the Gibbs sampler.

Appendix

R-code

```
#### Assignment 1 ####
rainfall <- read.delim("C:/Users/Gustav/Documents/Bayesian-Learning/Lab3/rainfall.dat", sep="", header =
library(ggplot2)
library(gridExtra)
library(coda)
## a)
# priors and others
mu0 <- 1
kappa0 <- 1
v0 <- 1
sigma0 <- 1
n <- nrow(rainfall)
ybar <- colMeans(rainfall)
s2 <- var(rainfall[,1])

# Posteriors
muN <- (kappa0 / (kappa0 + n)) * mu0 + (n / (kappa0 + n)) * ybar
kappaN <- kappa0 + n
vN <- v0 +n
vNsigmaN <- v0*sigma0 + (n-1)*s2 + (kappa0*n / (kappa0 + n)) * (ybar - mu0)^2
sigmaN <- vNsigmaN / vN

# Simulations - Gibbs Sampling
sims <- data.frame(mu=0, sigma2=0)
muN <- (kappa0 / (kappa0 + n)) * mu0 + (n / (kappa0 + n)) * ybar
vNsigmaN <- v0*sigma0 + (n-1)*s2 + (kappa0*n / (kappa0 + n)) * (ybar - muN)^2
sigmaN <- vNsigmaN / vN
X <- rchisq(1, vN)
sigma2 <- (vN * sigmaN / X)
mu <- rnorm(1, muN, sqrt(sigma2/kappaN))
sims[1,1] <-  mu
sims[1,2] <-  sigma2
for (i in 2:1000){
  # Byter ut mu0 mot [i-1,1] i sims
  # Byter ut sigma0 mot [i-1,2] i sims
  muN <- (kappa0 / (kappa0 + n)) * sims[i-1,1] + (n / (kappa0 + n)) * ybar
  # Byter ut mu0 mot muN
  vNsigmaN <- v0*sims[i-1,2] + (n-1)*s2 + (kappa0*n / (kappa0 + n)) * (ybar - muN)^2
  sigmaN <- vNsigmaN / vN
  X <- rchisq(1, n-1)
  sigma2 <- (vN * sigmaN / X)
  mu <- rnorm(1, muN, sqrt(sigma2/kappaN))
  sims[i,1] <-  mu
  sims[i,2] <-  sigma2
}
#trace plots - with burn-in!
tr_w_burn <- ggplot(sims, aes(x=1:nrow(sims), y=mu)) + geom_line() + theme_bw() + xlab("1:1000") + ylab
tr_w_burn2 <- ggplot(sims, aes(x=1:nrow(sims), y=sigma2)) + geom_line() +
  theme_bw() +ggtitle("Gibbs Sampling - Trace Plot - Sigma^2")+ xlab("1:1000") + geom_vline(xintercept=
```

```r
grid.arrange(tr_w_burn, tr_w_burn2, ncol=2)
tr_wh_burn <- ggplot(sims[101:1000,], aes(x=101:nrow(sims), y=mu)) + geom_line() + theme_bw() + xlab("1
tr_wh_burn2 <- ggplot(sims[101:1000,], aes(x=101:nrow(sims), y=sigma2)) + geom_line() + theme_bw() + gg
grid.arrange(tr_wh_burn, tr_wh_burn2, ncol=2)
# Looks at efficieny in terms of auto-correlation
acf_m <- acf(sims[101:1000,1], lag.max = 25, type = c("correlation"), plot=FALSE)
acf_s <- acf(sims[101:1000,2], lag.max = 25, type = c("correlation"), plot=FALSE)
acf_m <- data.frame(ACF=as.numeric(acf_m$acf), Lag=0:25)
acf_s <- data.frame(ACF=as.numeric(acf_s$acf), Lag=0:25)
acf_mu <- ggplot(acf_m, aes(x=Lag, y=ACF))+geom_bar(stat="identity", fill="black")+theme_bw() + ggtitle
acf_sigma <- ggplot(acf_s, aes(x=Lag, y=ACF))+geom_bar(stat="identity", fill="black")+theme_bw() + ggti
grid.arrange(acf_mu, acf_sigma, ncol=2)
rainfall <- read.delim("C:/Users/Gustav/Documents/Bayesian-Learning/Lab3/rainfall.dat",
                       sep="", header = TRUE)
x <- as.matrix(rainfall['X136'])

# Model options
nComp <- 2    # Number of mixture components

# Prior options
alpha <- c(10, 10)
muPrior <- c(32.2681, 32.2681)
tau2Prior <- rep(10,nComp) # Prior std theta
sigma2_0 <- rep(var(x),nComp) # s20 (best guess of sigma2)
nu0 <- rep(4,nComp) # degrees of freedom for prior on sigma2


# MCMC options
nIter <- 1000 # Number of Gibbs sampling draws


# Plotting options
plotFit <- TRUE
lineColors <- c("blue", "green", "magenta", 'yellow')
################   END USER INPUT ###############

###### Defining a function that simulates from the
rScaledInvChi2 <- function(n, df, scale){
  return((df*scale)/rchisq(n,df=df))
}

####### Defining a function that simulates from a Dirichlet distribution
rDirichlet <- function(param){
  nCat <- length(param)
  thetaDraws <- matrix(NA,nCat,1)
  for (j in 1:nCat){
    thetaDraws[j] <- rgamma(1,param[j],1)
  }
  thetaDraws = thetaDraws/sum(thetaDraws) # Diving every column of ThetaDraws by the sum of the element
  return(thetaDraws)
}

# Simple function that converts between two different representations of the mixture allocation
```

```r
S2alloc <- function(S){
  n <- dim(S)[1]
  alloc <- rep(0,n)
  for (i in 1:n){
    alloc[i] <- which(S[i,] == 1)
  }
  return(alloc)
}

# Initial value for the MCMC
nObs <- length(x)
S <- t(rmultinom(nObs, size = 1 , prob = rep(1/nComp,nComp))) # nObs-by-nComp matrix with component all
theta <- quantile(x, probs = seq(0,1,length = nComp))
sigma2 <- rep(var(x),nComp)
probObsInComp <- rep(NA, nComp)

# Setting up the plot
xGrid <- seq(min(x)-1*apply(x,2,sd),max(x)+1*apply(x,2,sd),length = 100)
xGridMin <- min(xGrid)
xGridMax <- max(xGrid)
mixDensMean <- rep(0,length(xGrid))
effIterCount <- 0
ylim <- c(0,2*max(hist(x,plot = FALSE)$density))

simulations <- data.frame(w_1 = 0, w_2 = 0, mu_1 = 0, mu_2 = 0, sigma1=0, sigma2=0)

for (k in 1:nIter){
  alloc <- S2alloc(S) # Just a function that converts between different representations of the group al
  nAlloc <- colSums(S)
  # Update components probabilities
  w <- rDirichlet(alpha + nAlloc)
simulations[k,1] <- w[1]
simulations[k,2] <- w[2]

  # Update theta's
  for (j in 1:nComp){
    precPrior <- 1/tau2Prior[j]
    precData <- nAlloc[j]/sigma2[j]
    precPost <- precPrior + precData
    wPrior <- precPrior/precPost
    muPost <- wPrior*muPrior + (1-wPrior)*mean(x[alloc == j])
    tau2Post <- 1/precPost
    theta[j] <- rnorm(1, mean = muPost, sd = sqrt(tau2Post))
  }
simulations[k,3] <- theta[1]
simulations[k,4] <- theta[2]
simulations$Expected[k] <- sum(simulations$w_1[k] * simulations$mu_1[k] +
                               simulations$w_2[k] * simulations$mu_2[k])
  # Update sigma2's
  for (j in 1:nComp){
    sigma2[j] <- rScaledInvChi2(1, df = nu0[j] + nAlloc[j], scale = (nu0[j]*sigma2_0[j] + sum((x[alloc =
  }
simulations[k,5] <- sigma2[1]
```

```
simulations[k,6] <- sigma2[2]
  # Update allocation
  for (i in 1:nObs){
    for (j in 1:nComp){
      probObsInComp[j] <- w[j]*dnorm(x[i], mean = theta[j], sd = sqrt(sigma2[j]))
    }
    S[i,] <- t(rmultinom(1, size = 1 , prob = probObsInComp/sum(probObsInComp)))
  }

  # Printing the fitted density against data histogram
  if (plotFit && (k%%1 ==0)){
    effIterCount <- effIterCount + 1
    #hist(x, breaks = 20, freq = FALSE, xlim = c(xGridMin,xGridMax), main = paste("Iteration number",k)
    mixDens <- rep(0,length(xGrid))
    components <- c()
    for (j in 1:nComp){
      compDens <- dnorm(xGrid,theta[j],sd = sqrt(sigma2[j]))
      mixDens <- mixDens + w[j]*compDens
      #lines(xGrid, compDens, type = "l", lwd = 2, col = lineColors[j])
      components[j] <- paste("Component ",j)
    }
    mixDensMean <- ((effIterCount-1)*mixDensMean + mixDens)/effIterCount

    #lines(xGrid, mixDens, type = "l", lty = 2, lwd = 3, col = 'red')
    #legend("topleft", box.lty = 1, legend = c("Data histogram",components, 'Mixture'),
          # col = c("black",lineColors[1:nComp], 'red'), lwd = 2)
  }
}
Wh_b <- ggplot(simulations, aes(x=1:nrow(simulations), y=Expected)) + geom_line() + theme_bw() + geom_v
Wh_o_b <- ggplot(simulations[101:nrow(simulations),], aes(x=101:nrow(simulations), y=Expected)) + geom_
grid.arrange(Wh_b, Wh_o_b, ncol=2)
# Looks at efficieny in terms of auto-correlation
acf_mix <- acf(simulations[101:nrow(simulations),7], lag.max = 25, type = c("correlation"), plot=FALSE)
acf_mix <- data.frame(ACF=as.numeric(acf_mix$acf), Lag=0:25)
ggplot(acf_mix, aes(x=Lag, y=ACF))+geom_bar(stat="identity", fill="black")+theme_bw() + ggtitle("Auto-c
mu1_b <- ggplot(simulations, aes(x=1:1000, y=mu_1)) + geom_line() + theme_bw() + ggtitle("Trace plot fo
mu2_b <- ggplot(simulations, aes(x=1:1000, y=mu_2)) + geom_line() + theme_bw()+ ggtitle("Trace plot for
w1_b <- ggplot(simulations, aes(x=1:1000, y=w_1)) + geom_line() + theme_bw()+ ggtitle("Trace plot for w
w2_b <- ggplot(simulations, aes(x=1:1000, y=w_2)) + geom_line() + theme_bw()+ ggtitle("Trace plot for w
sigma1_b <- ggplot(simulations, aes(x=1:1000, y=sigma1)) + geom_line() + theme_bw()+ ggtitle("Trace plo
sigma2_b <- ggplot(simulations, aes(x=1:1000, y=sigma2)) + geom_line() + theme_bw()+ ggtitle("Trace plo
grid.arrange(mu1_b, mu2_b,w1_b,w2_b,sigma1_b,sigma2_b, ncol=2)
# Autocorrelation plots for the other parameters
acf_mu1 <- acf(simulations[101:1000,3], lag.max = 25, type = c("correlation"), plot=FALSE)
acf_mu2 <- acf(simulations[101:1000,4], lag.max = 25, type = c("correlation"), plot=FALSE)
acf_w1 <- acf(simulations[101:1000,1], lag.max = 25, type = c("correlation"), plot=FALSE)
acf_w2 <- acf(simulations[101:1000,2], lag.max = 25, type = c("correlation"), plot=FALSE)
acf_s1 <- acf(simulations[101:1000,5], lag.max = 25, type = c("correlation"), plot=FALSE)
acf_s2 <- acf(simulations[101:1000,6], lag.max = 25, type = c("correlation"), plot=FALSE)
acf_mu1 <- data.frame(ACF=as.numeric(acf_mu1$acf), Lag=0:25)
acf_mu2 <- data.frame(ACF=as.numeric(acf_mu2$acf), Lag=0:25)
acf_w1 <- data.frame(ACF=as.numeric(acf_w1$acf), Lag=0:25)
acf_w2 <- data.frame(ACF=as.numeric(acf_w2$acf), Lag=0:25)
```

```r
acf_s1 <- data.frame(ACF=as.numeric(acf_s1$acf), Lag=0:25)
acf_s2 <- data.frame(ACF=as.numeric(acf_s2$acf), Lag=0:25)

acf_mu1 <- ggplot(acf_mu1, aes(x=Lag, y=ACF))+geom_bar(stat="identity", fill="black")+theme_bw() + ggti
acf_mu2 <- ggplot(acf_mu2, aes(x=Lag, y=ACF))+geom_bar(stat="identity", fill="black")+theme_bw()+ ggtitl
acf_w1 <- ggplot(acf_w1, aes(x=Lag, y=ACF))+geom_bar(stat="identity", fill="black")+theme_bw()+ ggtitle
acf_w2 <- ggplot(acf_w2, aes(x=Lag, y=ACF))+geom_bar(stat="identity", fill="black")+theme_bw()+ ggtitle
acf_s1 <- ggplot(acf_s1, aes(x=Lag, y=ACF))+geom_bar(stat="identity", fill="black")+theme_bw()+ ggtitle
acf_s2 <- ggplot(acf_s2, aes(x=Lag, y=ACF))+geom_bar(stat="identity", fill="black")+theme_bw() + ggtitl
grid.arrange(acf_mu1, acf_mu2,acf_w1,acf_w2,acf_s1,acf_s2,ncol=2)

## c)
a1 <- data.frame(x=rnorm(1000, 32.27564, sqrt(1546.53868)))
b1 <- data.frame(y=mixDensMean, x=xGrid)
ggplot(rainfall, aes(X136)) + geom_histogram(aes(y = ..density..),alpha=0.9,
  fill="black", binwidth=8) + theme_bw() +
  geom_density(data=a1, aes(x),col="royalblue", size=1.05) +
  geom_line(data=b1, aes(x=x, y=y), col="red", size=1.05) +
  ggtitle("Density for normal model and mixture of normals\n Blue = 1.a) - Red = 1.b)") + xlab("")
############   BEGIN USER INPUTS   #################
Probit <- 1 # If Probit <-0, then logistic model is used.
chooseCov <- c(1:16) # Here we choose which covariates to include in the model
tau <- 10; # Prior scaling factor such that Prior Covariance = (tau^2)*I
############     END USER INPUT    #################

# install.packages("mvtnorm") # Loading a package that contains the multivariate normal pdf
library("mvtnorm") # This command reads the mvtnorm package into R's memory. NOW we can use dmvnorm fun

# Loading data from file
Data<-read.table("C:/Users/Gustav/Documents/Bayesian-Learning/Lab3/SpamReduced.dat",header=TRUE)  # Spa
y <- as.vector(Data[,1]); # Data from the read.table function is a data frame. Let's convert y and X to
X <- as.matrix(Data[,2:17]);
covNames <- names(Data)[2:length(names(Data))];
X <- X[,chooseCov]; # Here we pick out the chosen covariates.
covNames <- covNames[chooseCov];
nPara <- dim(X)[2];

# Setting up the prior
mu <- as.vector(rep(0,nPara)) # Prior mean vector
Sigma <- tau^2*diag(nPara);

# Defining the functions that returns the log posterior (Logistic and Probit models). Note that the fir

# this function must be the one that we optimize on, i.e. the regression coefficients.

LogPostProbit <- function(betaVect,y,X,mu,Sigma){
  nPara <- length(betaVect);
  linPred <- X%*%betaVect;
  # MQ change:
  # instead of logLik <- sum(y*log(pnorm(linPred)) + (1-y)*log(1-pnorm(linPred)) ) type in the equivale
  # much more numerically stable;
  logLik <- sum(y*pnorm(linPred, log.p = TRUE) + (1-y)*pnorm(linPred, log.p = TRUE, lower.tail = FALSE))
  # The old expression: logLik2 <- sum(y*log(pnorm(linPred)) + (1-y)*log(1-pnorm(linPred)) )
```

```
  abs(logLik) == Inf
  #print('-----------------')
  #print(logLik)
  #print(logLik2)
  #if (abs(logLik) == Inf) logLik = -20000; # Likelihood is not finite, stear the optimizer away from h
  logPrior <- dmvnorm(betaVect, matrix(0,nPara,1), Sigma, log=TRUE);
  return(logLik + logPrior)
}

# Calling the optimization routine Optim. Note the auxilliary arguments that are passed to the function
# Note how I pass all other arguments of the function logPost (i.e. all arguments except betaVect which
# The argument control is a list of options to the optimizer. Here I am telling the optimizer to multip
# Optim finds a minimum, and I want to find a maximum. By reversing the sign of logPost I can use Optim

# Different starting values. Ideally, any random starting value gives you the same optimum (i.e. optimu
initVal <- as.vector(rep(0,dim(X)[2]));
# Or a random starting vector: as.vector(rnorm(dim(X)[2]))
# Set as OLS estimate: as.vector(solve(crossprod(X,X))%*%t(X)%*%y); # Initial values by OLS

if (Probit==1){
  logPost = LogPostProbit;
} else{
  logPost = LogPostLogistic;
}

OptimResults<-optim(initVal,logPost,gr=NULL,y,X,mu,Sigma,method=c("BFGS"),control=list(fnscale=-1),hessi

library(msm)
library(mvtnorm)

tau <- 10
mu <- as.vector(rep(0,nPara)) # Prior mean vector
Sigma <- tau^2*diag(nPara)
nPara <- dim(X)[2]

mean_p <- t(as.matrix(as.vector(rep(0,dim(X)[2])), ncol=1))
sigma_p <- diag(x=tau^2, 16, 16)
sigma_p2 <- ( as.matrix(diag(sigma_p)))

emptyB <- data.frame(matrix(vector(), 100, 16))
u <- data.frame(matrix(vector(), 4601, 100))
set.seed(311015)
u[,1] <- rtnorm(4601, X%*%t(mean_p), sd = rep(1, 16))

for (i in 1:100){
  B_n <- solve(t(X)%*%X + sigma_p2%*%mean_p) %*% t(X)%*%u[,i]
  mean_p <- t(as.matrix(B_n))
  sigma_p <- solve(t(X)%*%X + sigma_p)
  sigma_p2 <- ( as.matrix(diag(sigma_p))) #same as sigma_p, just modified format

  emptyB[i,] <-  rmvnorm(1, mean_p, sigma_p)
  newB <- t(matrix(as.numeric(emptyB[i,])))
```

```
  for(j in 1:4601){
    if(Data$spam[j] == 1){
      u[j,i+1] <- rtnorm(1, t(X[i,] %*% newB),sd = rep(1, 16), lower=0, upper=Inf)
    }else{
      u[j,i+1] <- rtnorm(1, t(X[i,] %*% newB),sd = rep(1, 16), lower=-Inf, upper=0)
    }
  }
}
par(mfrow=c(3,2))
for(i in 1:16){
  plot(emptyB[,i], type="l")
}
effectiveSize(emptyB)
Betas <- matrix(as.numeric(emptyB[100,]))
options(scipen = 999)
Res <- data.frame(covs=covNames,OptimBeta = as.numeric(OptimResults$par), GibbsBeta=as.numeric(emptyB[1
Res
Betas <- matrix(as.numeric(emptyB[100,]))

y_fitGibbs <- (X) %*% Betas
for(i in 1:4601){
  if(y_fitGibbs[i] > 0){
    y_fitGibbs[i] = 1
  }else{
    y_fitGibbs[i] = 0
  }
}
table(y,y_fitGibbs)

betasOptim <- matrix(as.numeric(OptimResults$par))
y_fitOptim <- (X) %*% betasOptim
for(i in 1:4601){
  if(y_fitOptim[i] > 0){
    y_fitOptim[i] = 1
  }else{
    y_fitOptim[i] = 0
  }
}
table(y,y_fitOptim)
## NA
```