Bayesian Learning, 7.5 hp

Home assignment - Part B

Thomas Mortensen, Julia Langenius, Alex Holmskär

2025-10-02

Table of contents

> **Note**
>
> This is the second part of the home assignment for the course. The first part had Problems 1-4, so this second part will continue with this numbering, and the first problem here will therefore be Problem 5.

> **Tip**
>
> It is **strongly advised** to study the notebook for the logistic regression applied to the Titanic data *before* embarking on this assignment. This notebook is here: html | quarto.

Problem 5 - Normal posterior approximation

The file `bugs.csv` contains a dataset with the number of bugs (`nBugs`) and some other explanatory variables for $n = 91$ releases of several software projects. We will use Poisson regression to the model the number of bugs as a function of the following *covariates/features*:

- *intercept* - this is columns of ones to model the intercept
- *nCommit* - the number of commits since the last release of the software
- *propC* - proportion of C/C++ code in the project
- *propJava* - proportion of Java code in the project

- *complexity* - a measure of code complexity that takes into account the frequency of if statements etc.

This code sets up the vector of response observations (`y`) and the 91×5 matrix of covariates `X`

```
library(latex2exp)
data = read.csv("https://github.com/mattiasvillani/BayesLearnCourse/raw/master/assignment/bug
                header = TRUE)
y = data$nBugs # response variable: the number of bugs, a vector with n = 91 observations
X = data[,-1]  # 91 x 5 matrix with covariates
X = as.matrix(X) # X was initially a data frame, but we want it to be matrix
head(X)
```

```
     intercept nCommits     propC  propJava complexity
[1,]         1        5 0.3191181 0.2307104  0.6050054
[2,]         1        4 0.4150649 0.3295860  0.7031914
[3,]         1       13 0.2532424 0.2821941  0.7550588
[4,]         1        1 0.4901135 0.2260926  0.2207819
[5,]         1       10 0.1888998 0.3671323  0.8765245
[6,]         1        7 0.3914173 0.3340347  0.8076927
```

We first consider the **Poisson regression model**

$$Y_i|x_i \overset{\text{ind}}{\sim} \text{Poisson}\Big(\lambda_i = \exp\left(x_i^\top \beta\right)\Big)$$

Note how each observation (software release) has its "own" λ_i parameter, which is modeled as the exponential of $x_i^\top \beta$. The exponential function is used to make sure that λ_i is always positive, as it has to be in the Poisson model.

We ignore here that some of the observations actually come from the same project at different releases, and assume that the response observations Y_i are *independent*, given the features.

The covariates/features for the ith observation $x_i = (x_{1,i}, \dots, x_{p,i})^\top$ is the ith row of the matrix `X` . For example, for the second observation we have `y[2] = 6`, so six bugs in the second release, and the covariate values for this second release are:

```
X[2,]
```

```
 intercept   nCommits      propC   propJava complexity
 1.0000000  4.0000000  0.4150649  0.3295860  0.7031914
```

That is, this release (observation) has 4 commits, approximately 41.4% C code, 32.9 % Java code and a Code Complexity of 0.7.

Problem 5a)

Compute a (multivariate) normal approximation of the joint posterior distribution for the vector of Poisson regression coefficients β. Use the prior $\beta \sim N(0, \tau^2 I_p)$ where I_p is the $p \times p$ identity matrix (obtained in R by `diag(5)` when $p = 5$). Set $\tau = 10$ to get a fairly non-informative prior. The library `mvtnorm` contains the multivariate normal density function `dmvnorm`.

```
library(mvtnorm)

logPostPoisReg = function(betaVec, y, X){
tau = 10
p = ncol(X)
logPrior = dmvnorm(betaVec, mean = rep(0,p), sigma = tau^2*diag(p), log = TRUE)
logLik = sum((X %*% betaVec * y) - exp(X %*% betaVec)) # proportional loglik, we ignore log(y

return(logLik + logPrior)
}

initVal <- as.vector(rep(0,ncol(X)));
OptimResults<-optim(initVal, logPostPoisReg, gr=NULL, y, X,
  method = c("BFGS"), control=list(fnscale=-1), hessian=TRUE) # optim minimizes by default, s
postMode = OptimResults$par
postCov = -solve(OptimResults$hessian) # inv(J) - Approx posterior covar matrix
postStd <- sqrt(diag(postCov))         # Approximate stdev
```

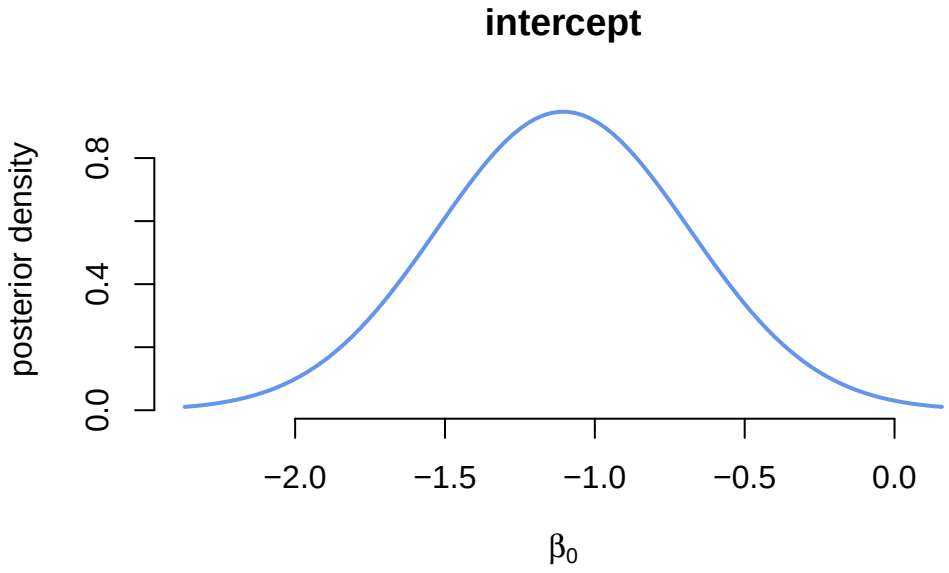
Problem 5b)

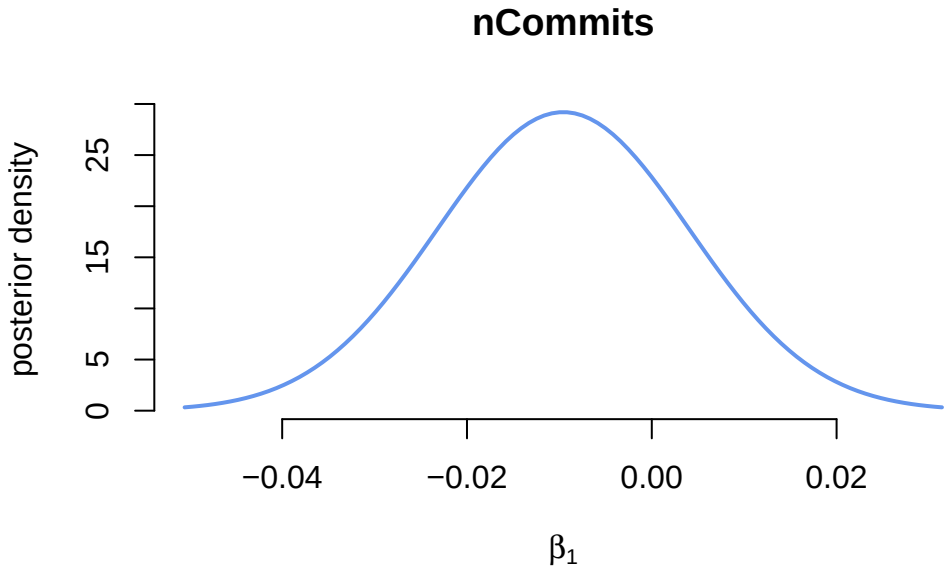
Plot the (approximate) marginal posterior distribution of each β_j for $j = 1, \ldots, 5$. Summarize each marginal posterior by a 95% interval (you are free to choose the type of interval). As a sort of Bayesian "significance" test, check if the value $\beta_j = 0$ is included in the interval for $j = 1, \ldots, 5$.

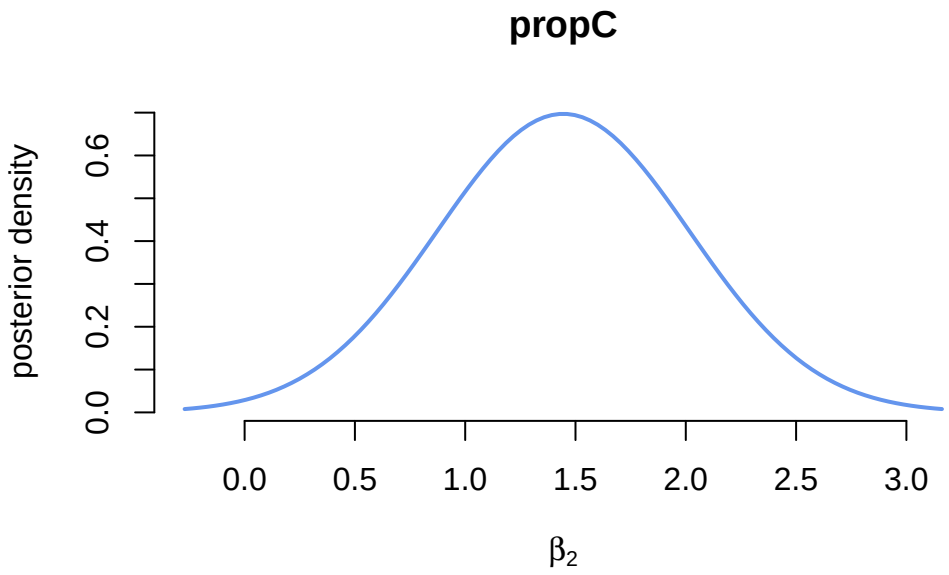
> **!** Important
>
> Note that this significance is not the usual frequentist interpretation. There are other Bayesian ways to assess "significance", for example by computing Bayes factors or using Bayesian variable selection. However, the simple version here based on credible intervals is often a more robust choice. In particular, the credible interval approach is less dependent on the prior for β, whereas Bayes factors are known to be very sensitive to the choice of prior and its prior hyperparameters.

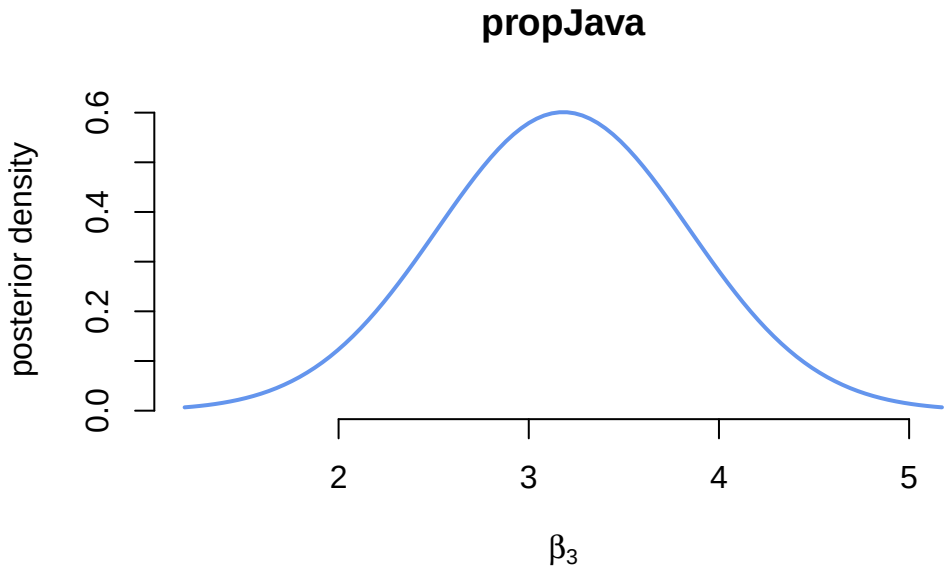
PropC and intercept just barely contain the value 0 so they can be "non-significant" but most likely not. The rest don't contain 0 except nCommits which is almost centered on 0 so our posterior tells us nCommits is most likely "non-significant" or has a very small coefficient.

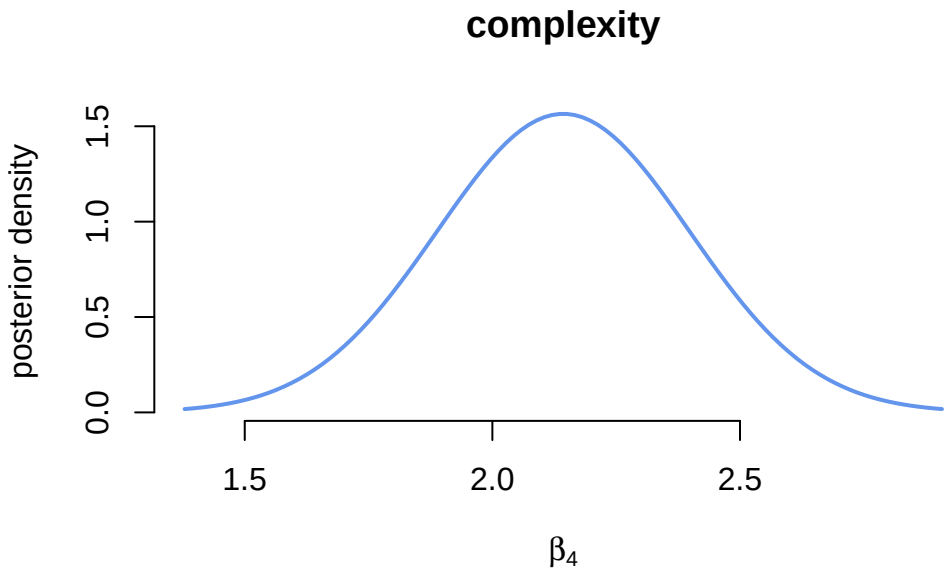
```
for (j in 1:5){
  gridVals = seq(postMode[j] - 3*postStd[j], postMode[j] + 3*postStd[j],
                 length = 100)
  plot(gridVals, dnorm(gridVals, mean = postMode[j], sd = postStd[j]),
    xlab = TeX(sprintf(r'($\beta_%d$)', j-1)), ylab= "posterior density",
    type ="l", bty = "n", lwd = 2, col = "cornflowerblue", main =(colnames(X)[j]))
}
```




nCommits
posterior density
0
5
15
25
−0.04
−0.02
0.00
0.02
β1


propC
posterior density
0.0
0.2
0.4
0.6
0.0
0.5
1.0
1.5
2.0
2.5
3.0
β2





Problem 6 - Posterior sampling with the Metropolis-Hastings algorithm

Problem 6a)

Write a function `RWMsampler` that implements the **Random Walk Metropolis algorithm** from Chapter 10 in the Bayesian Learning book. The `RWMsampler` function should have signature

```r
RWMsampler <- function(logPostFunc, initVal, nSim, nBurn, Sigma, c, ...){
  # Run the algorithm for nSim iterations
  # using the multivariate proposal N(theta_previous_draw, c*Sigma)
  # Return the posterior draws after discarding nBurn iterations as burn-in
  p = length(initVal)
  draws = matrix(NA, nrow = nSim + nBurn, ncol = p)
  proposal_theta = as.vector(rmvnorm(1,initVal, c*Sigma))
  accepted = logical(nSim + nBurn)
  for(i in 1:(nSim+nBurn)){
    old_theta = proposal_theta
    proposal_theta = as.vector(rmvnorm(1,proposal_theta, c*Sigma))
    acceptance_rate = min(1, exp(logPostFunc(proposal_theta, ...)-logPostFunc(old_theta, ...
    u = runif(1)
    if(u < acceptance_rate){
      draws[i,] = proposal_theta
      accepted[i] = TRUE
    } else{
      draws[i,] = proposal_theta
      proposal_theta = old_theta
      accepted[i] = FALSE
    }
  }
  drawsWithoutBurn = data.frame(draws[(nBurn+1):(nBurn + nSim),])
  accepted = accepted[(nBurn+1):(nBurn + nSim)]
  drawsWithoutBurn$accept = accepted
  return(drawsWithoutBurn)
}
```

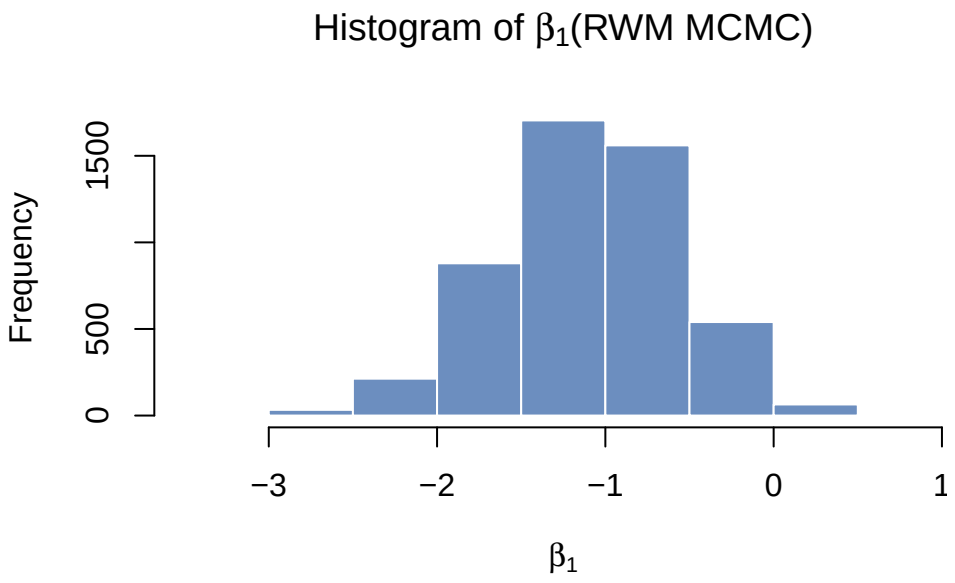
where `nSim` is the number of draws after the `nBurn` burn-in draws. `logPostFunc` is a function object that computes the log posterior in proportional form and the final triple dot argument … catches all additional arguments (data and the prior hyperparameters) needed to evaluate the `logPostFunc` function. See the document How to code up a general Metropolis sampler in R for details on all of this.

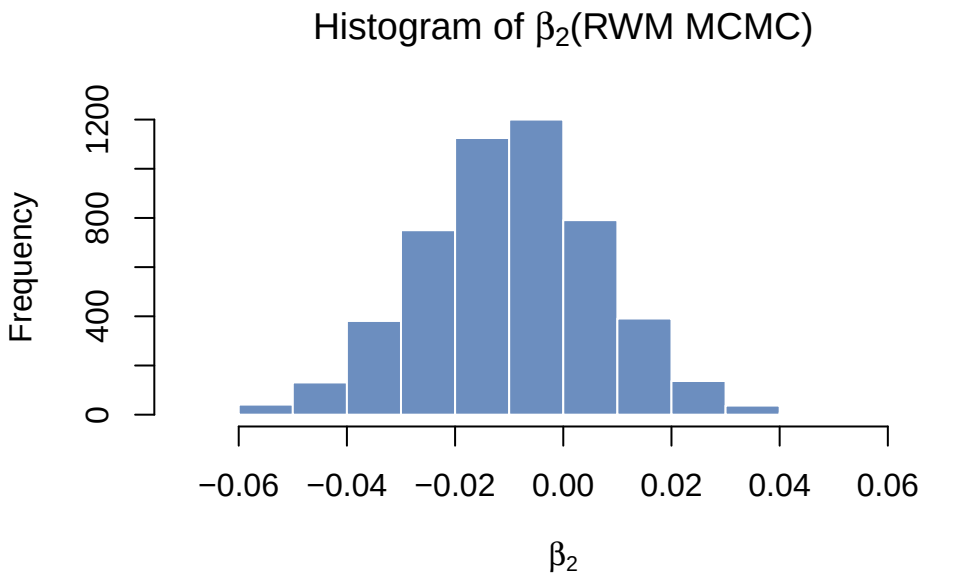
> **Caution**
>
> The `mvtnorm` package contains a random number generator `rmvnorm` for the multivariate normal distribution. The output of that function is an $n \times p$ matrix where each row is draw of the p-dimensional multivariate normal vector. So when used to generate a single draw $n = 1$, the output is $1 \times p$ matrix, **not** a vector (in R's sense). Use the `as.vector()` function to convert that $1 \times p$ matrix into a vector.

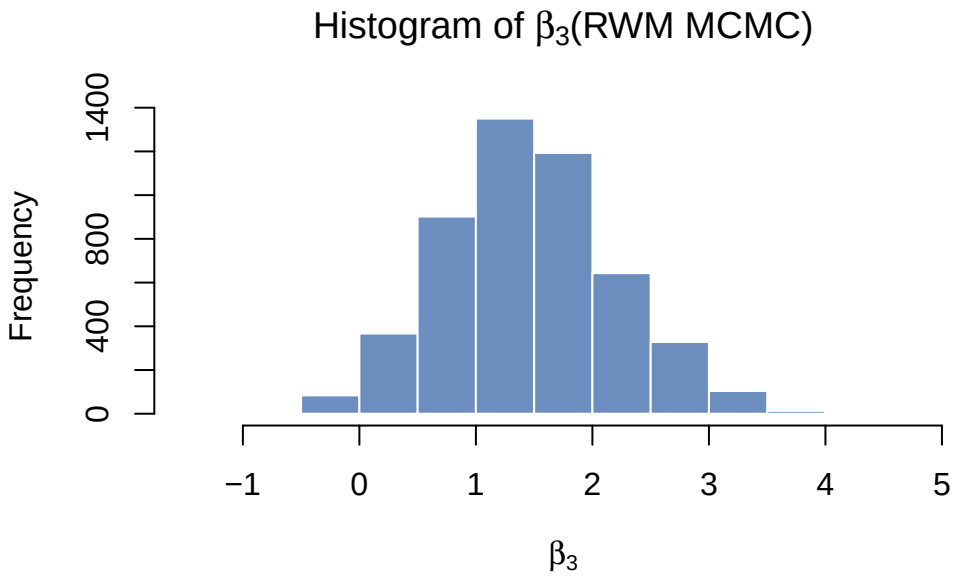
Problem 6b)

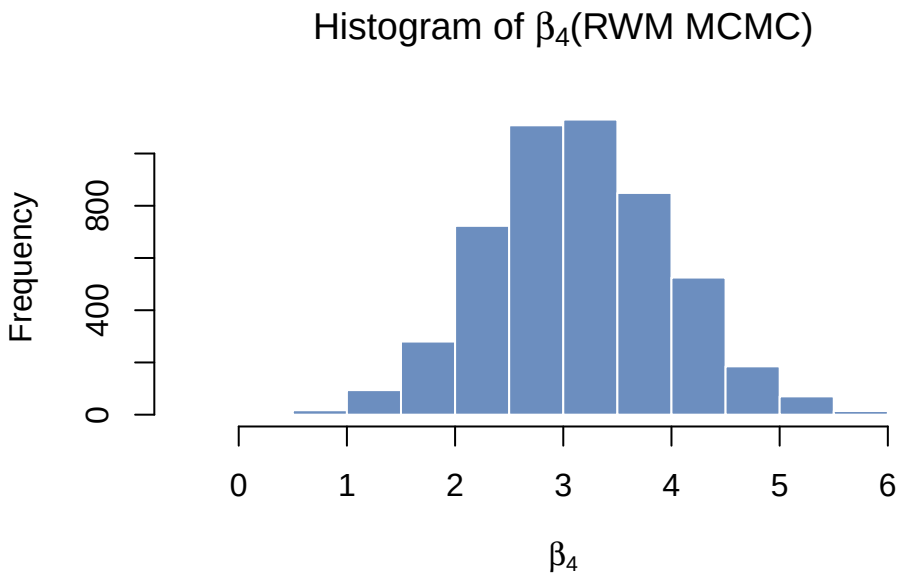
Use the RWM sampler from Problem 6a) to sample from the posterior of the Poisson regression model in Problem 5a). Set the proposal covariance matrix Σ equal to the posterior covariance matrix from the Normal approximation in Problem 5) and the RWM scaling constant to $c = 0.5$. Use `nSim = 5000` and `nBurn = 1000`. Use the zero vector as initial value for the algorithm. Plot histograms of the posterior draws to represent the marginal posterior densities.

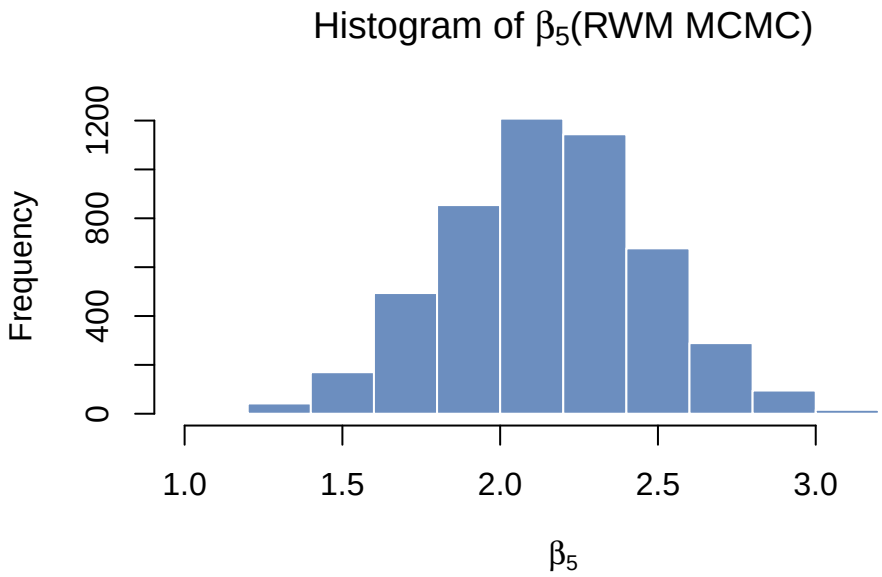
```
library(latex2exp)
PoisRWMDraws = RWMsampler(logPostPoisReg, initVal, nSim = 5000, nBurn = 1000, Sigma = postCov

for(i in 1:(ncol(PoisRWMDraws)-1)){
  hist(PoisRWMDraws[,i], main = TeX(paste0("Histogram of $\\beta_{", i, "}$(RWM MCMC)")), col
}
```




Histogram of β_2(RWM MCMC)
Frequency
0
400
800
1200
−0.06
−0.04
−0.02
0.00
0.02
0.04
0.06
β_2


Histogram of β_3(RWM MCMC)
Frequency
0
400
800
1400
−1
0
1
2
3
4
5
β_3





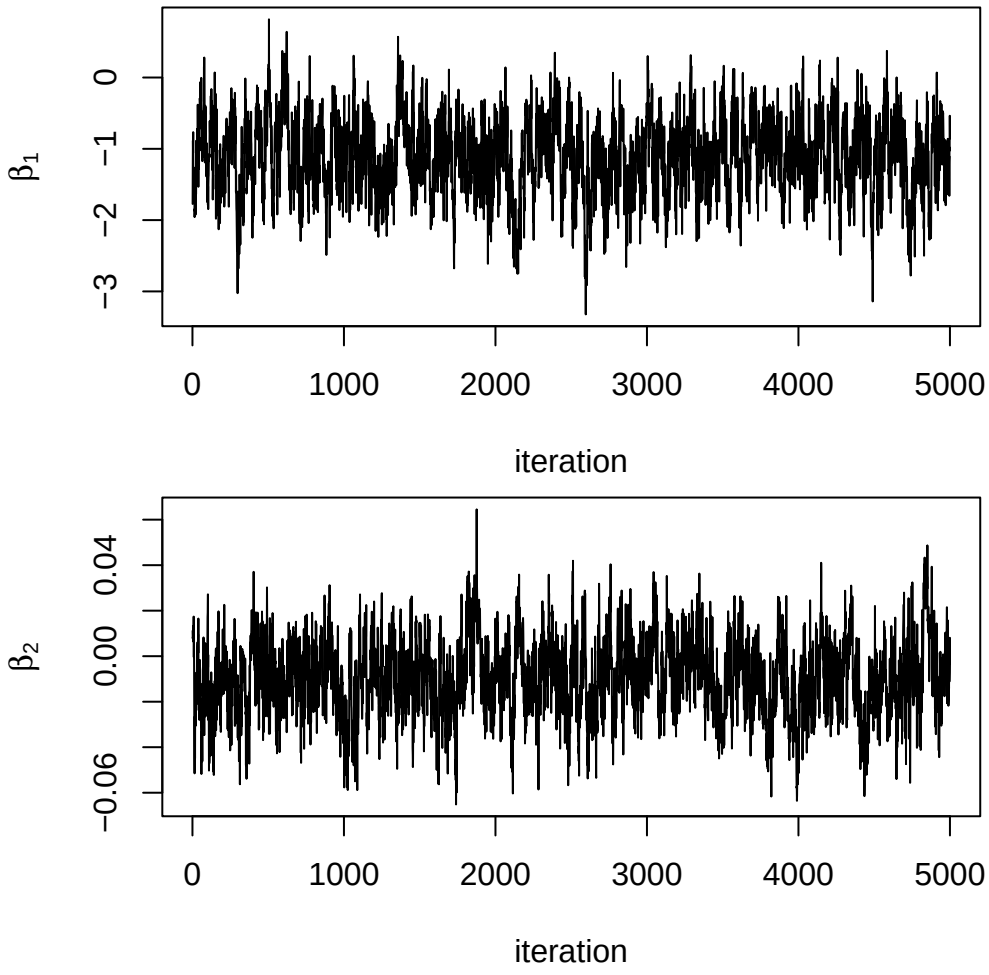
Problem 6c)

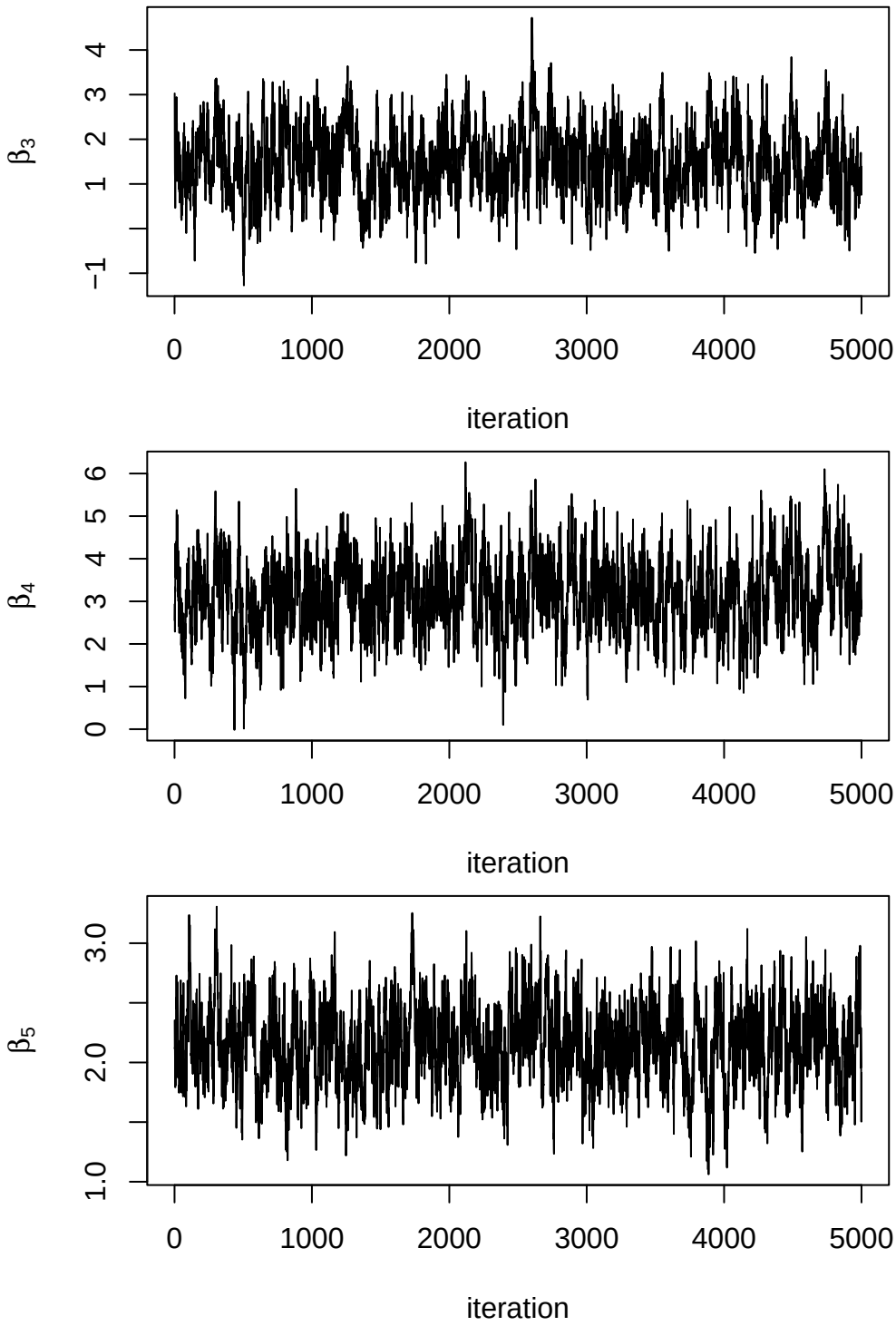
Check if the posterior samples from MCMC seems to have **converged** to the true posterior by:

1. **Plot the MCMC trajectories** (draws over the MCMC iterations) for all parameters.

They seem to walk around one mean so not random walk behavior, where it eratically goes off in one direction without returning regularly to the mean.

```r
for(i in 1:5){
  plot(PoisRWMDraws[,i], type = "l", xlab = "iteration", ylab = TeX(paste0("$\\beta_{",i,"}$
}
```

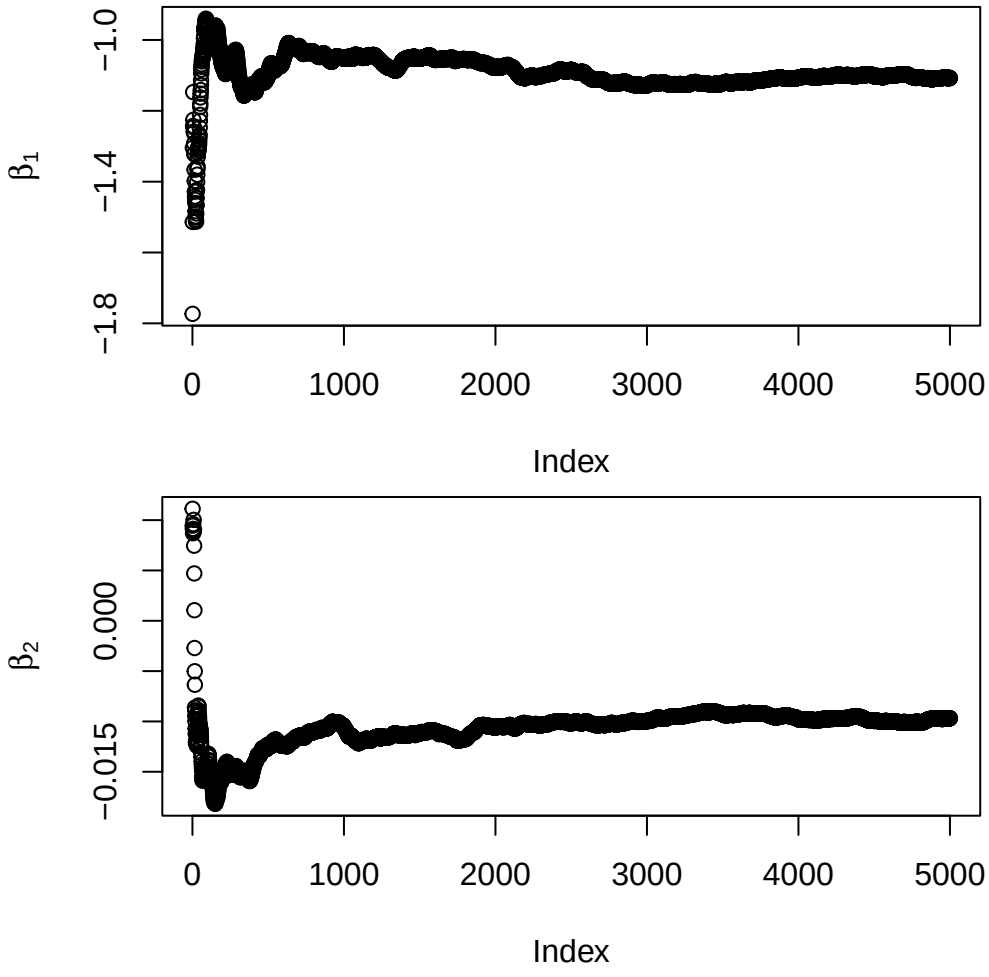


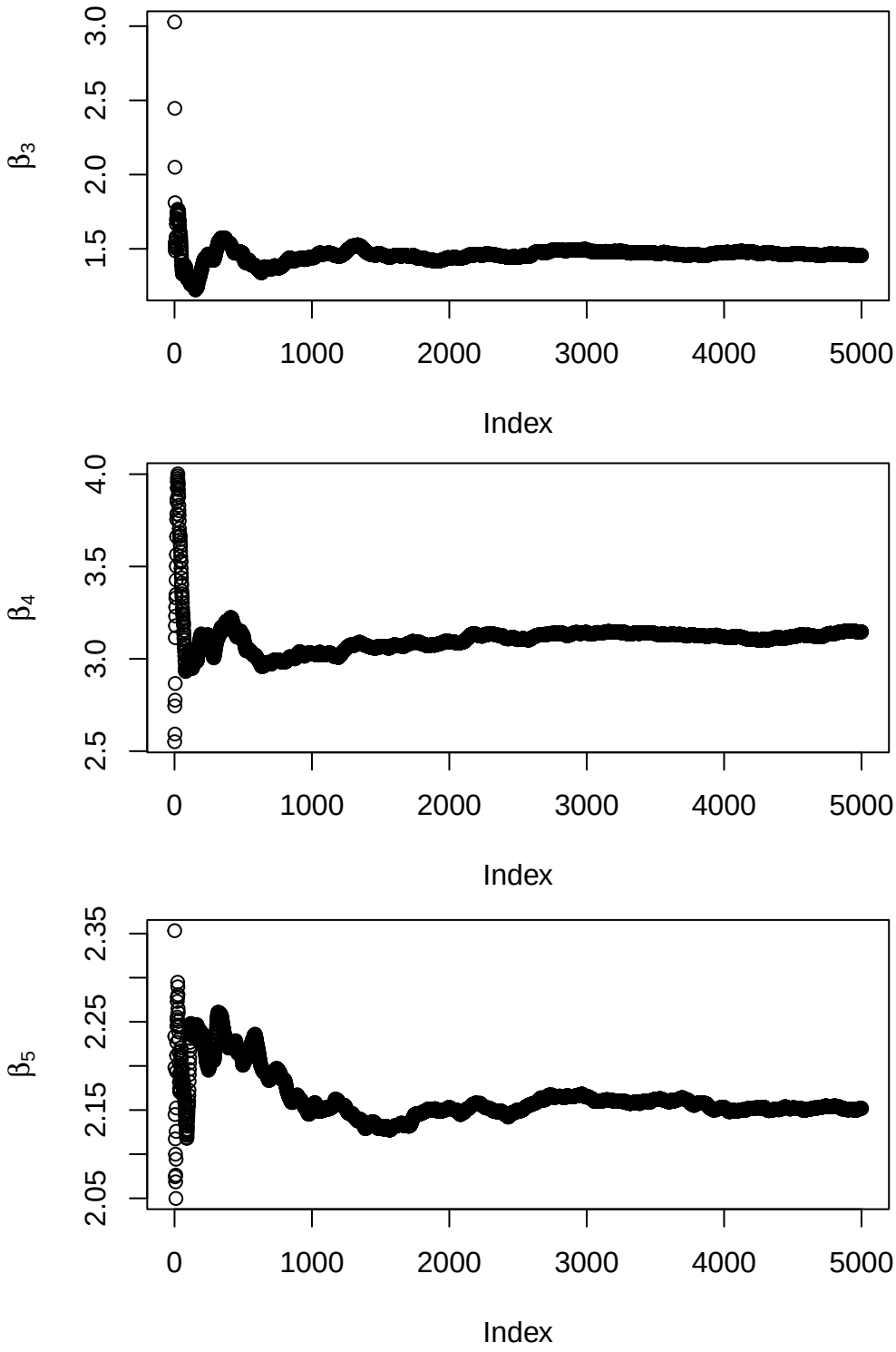


2. Plot the **cumulative estimates** of the posterior mean for the parameters based on increasing number of draws.

They seem to converge so we don't get insanely large values in either direction that would make our cumulative mean look wiggly/erratic. Instead we see that it regularly returns and varies around it, meaning the cumulative mean starts to converge instead of diverging.

```
cum_means = apply(PoisRWMDraws, 2, function(i) cumsum(i) / seq_along(i))
for(i in 1:5){
  plot(cum_means[,i], ylab = TeX(paste0("$\\beta_{",i,"}$")))
}
```





3. **Re-run the sampler** a second time, this time starting from the unit vector $(1, 1, 1, 1, 1)$,

and compare the posterior mean estimates from the two runs.

We seem to get about the same estimates for the posterior means of the marginals for X3 and X5, while X1, X2, X4 seem to differ within about 5% of eachother. So the starting point matters here, but not enough to completely twist our results.

```
PoisRWMDraws2 = RWMsampler(logPostPoisReg, initVal = rep(1,5), nSim = 5000, nBurn = 1000, Sig

SampledMeans = rbind(apply(PoisRWMDraws, 2,mean), apply(PoisRWMDraws2, 2,mean))

rownames(SampledMeans) = c("Initval c(0,0,0,0,0)","Initval c(1,1,1,1,1)")
SampledMeans
```

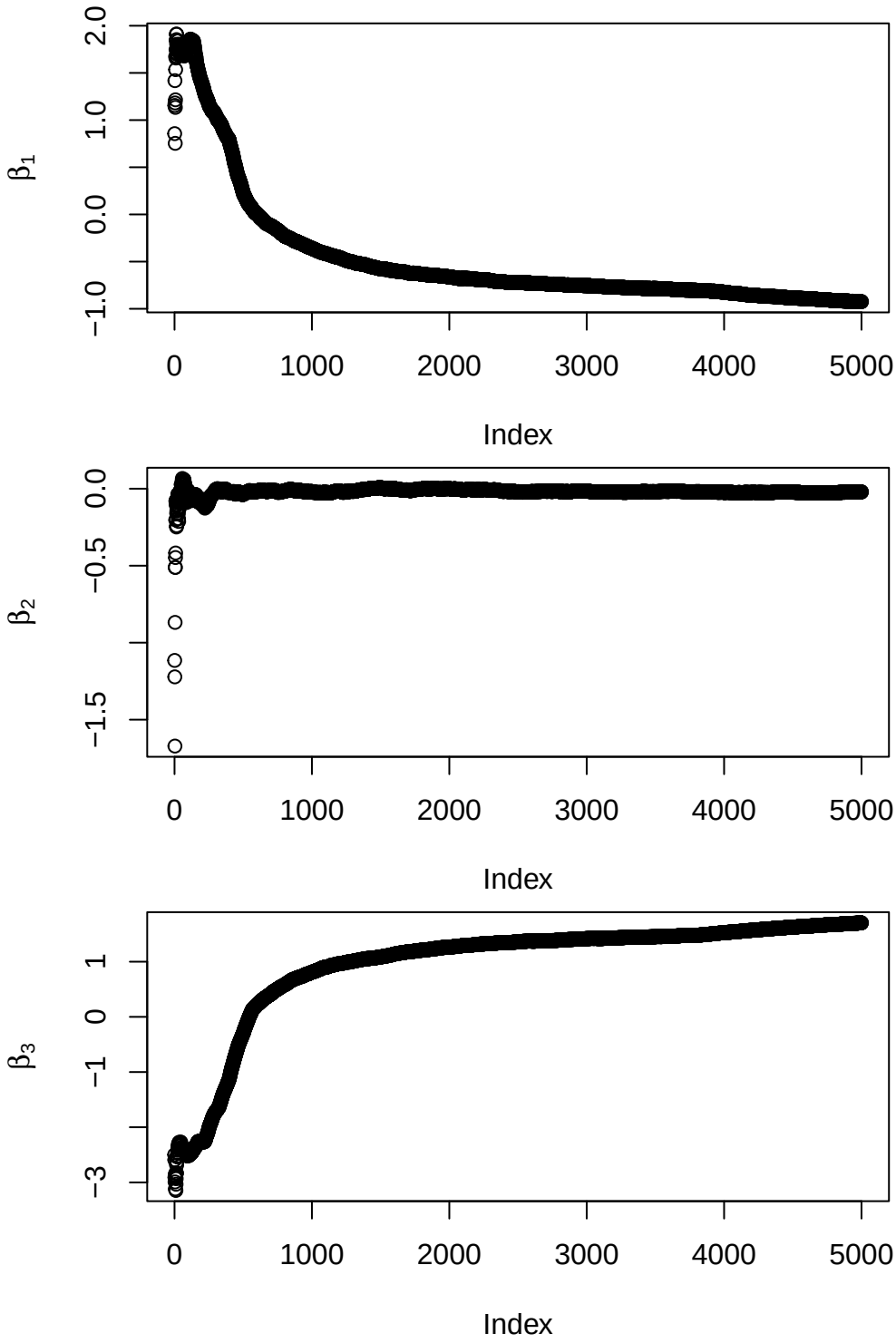
```
                            X1           X2       X3       X4       X5 accept
Initval c(0,0,0,0,0) -1.107331 -0.009683537 1.455263 3.145993 2.151792 0.4672
Initval c(1,1,1,1,1) -1.082358 -0.008820851 1.442373 3.105475 2.144302 0.4624
```

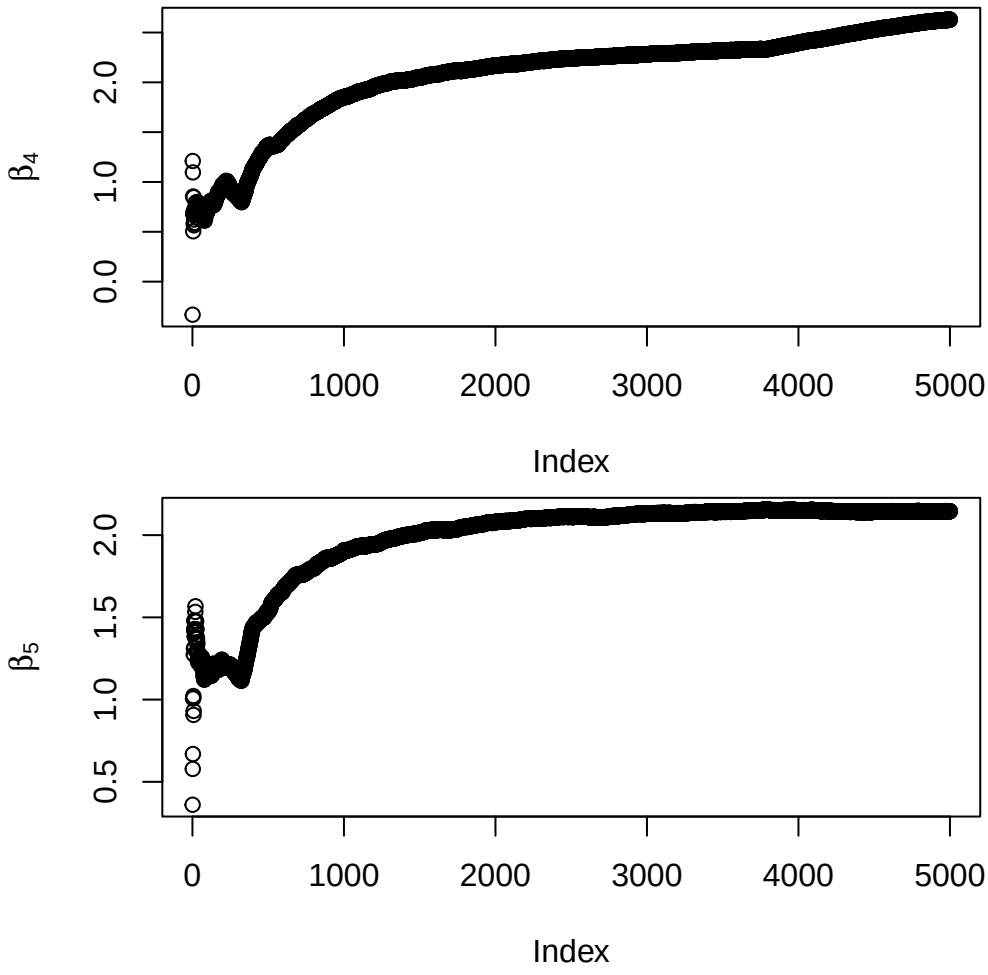
Problem 6d)

Re-run the RWM algorithm, this time using $\Sigma = I_p$ and with $c = 1$ (again using the zero vector as initial value). Is the mixing of the MCMC chain better or worse, compared to the samples obtained in Problem 6b)? Why?

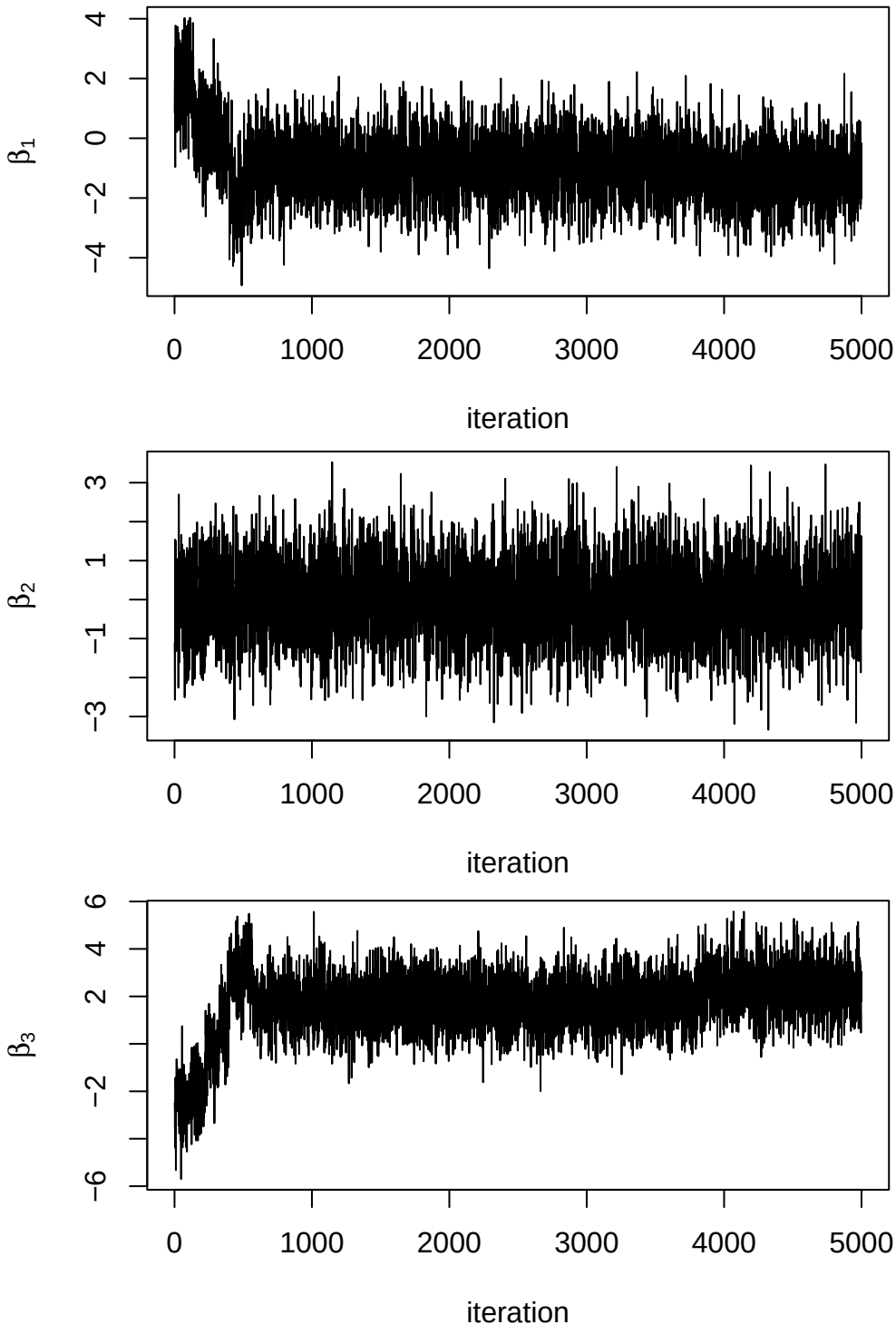
The mixing(How efficiently the chain explores our posterior) is worse as we converge slower and with no correlation structure assumed for our posterior when there is one, we reject a lot more, leading to us moving less and sampling more from the same old theta. This means the samples are highly correlated for many lags, while for good mixing that correlation tapers off.

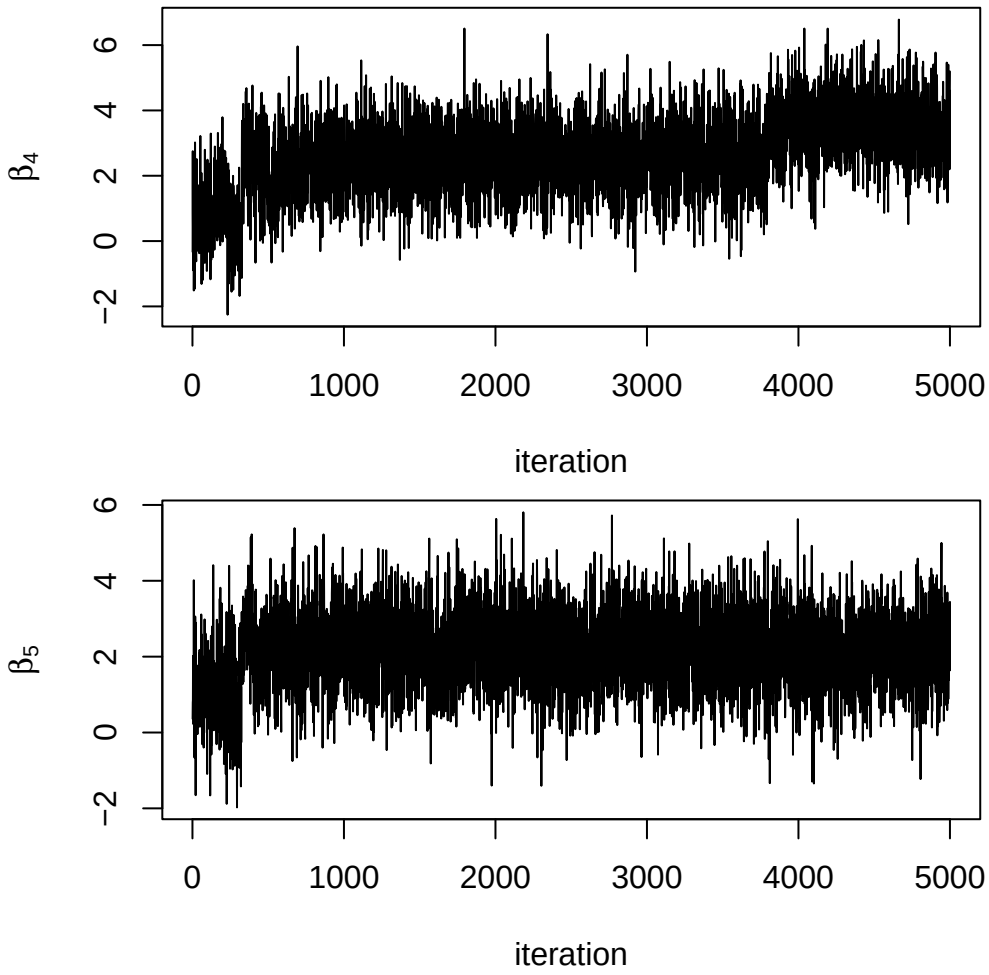
```
PoisRWMDraws3 = RWMsampler(logPostPoisReg, initVal = rep(0,5), nSim = 5000, nBurn = 1000, Sig

cum_means3 = apply(PoisRWMDraws3, 2, function(i) cumsum(i) / seq_along(i))
for(i in 1:5){
  plot(cum_means3[,i], ylab = TeX(paste0("$\\beta_{",i,"}$")))
}
```


β1
2.0
1.0
0.0
−1.0
0
1000
2000
3000
4000
5000
Index
β2
0.0
−0.5
−1.5
0
1000
2000
3000
4000
5000
Index
β3
1
0
−1
−3
0
1000
2000
3000
4000
5000
Index



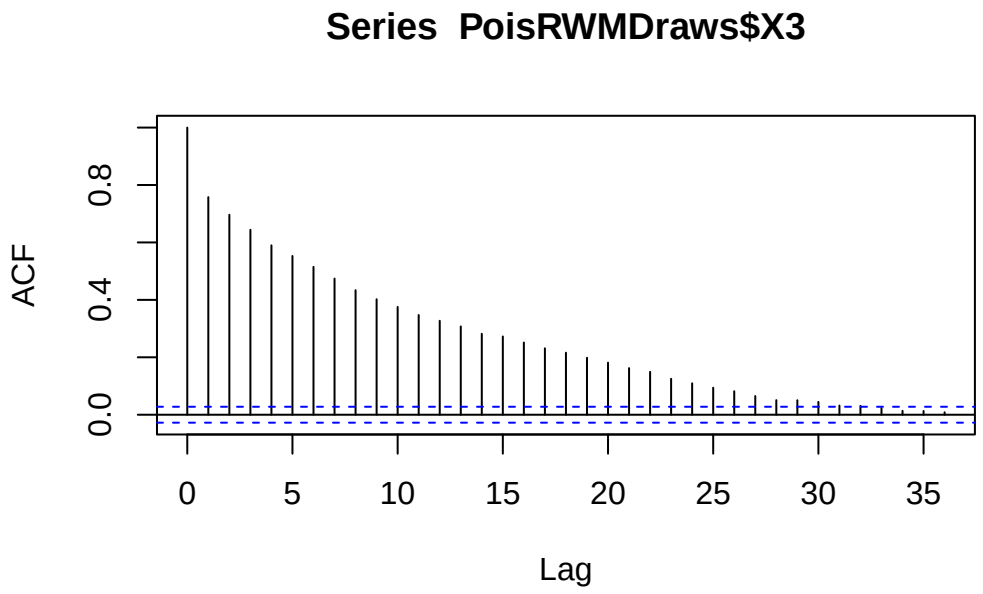

```
for(i in 1:5){
  plot(PoisRWMDraws3[,i], type = "l", xlab = "iteration", ylab = TeX(paste0("$\\beta_{",i,"}$
}
```


β1
4
2
0
−2
−4
0
1000
2000
3000
4000
5000
iteration
β2
3
1
−1
−3
0
1000
2000
3000
4000
5000
iteration
β3
6
4
2
−2
−6
0
1000
2000
3000
4000
5000
iteration

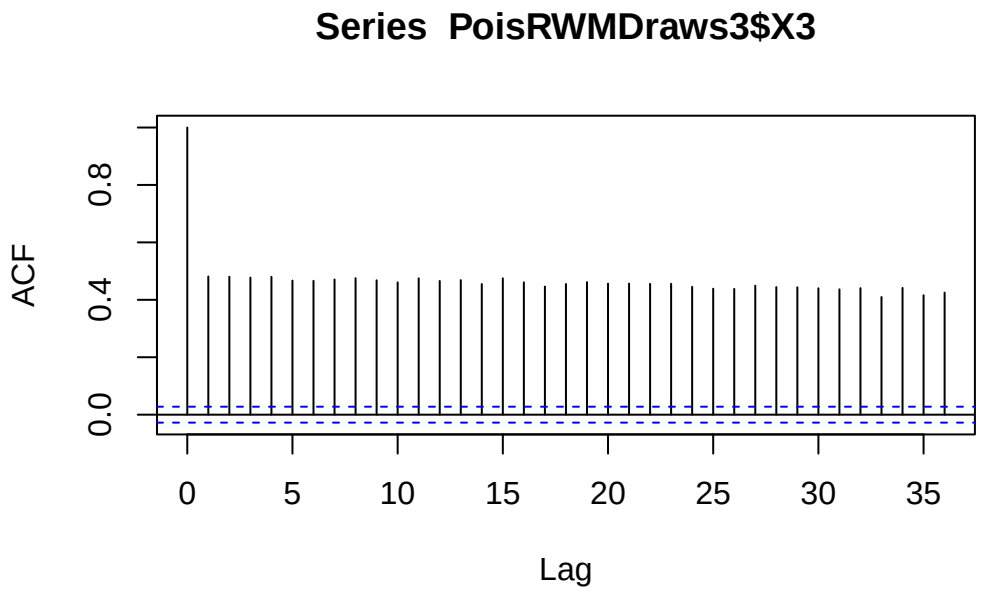


Here we can see the autocorrelation between samples(for those covariates/params where covariance structure matters) is very "sticky" for the samples using the identity covariance matrix.

```
acf(PoisRWMDraws$X3)
```




```
acf(PoisRWMDraws3$X3)
```



Problem 7 - Posterior sampling with the HMC algorithm in Stan

> **Note**
>
> This exercise uses Probabilistic programming in the Stan language. You need to use the `rstan` and `loo` packages. If you run into troubles installing `rstan` on your own computer, consult the Getting started with Rstan guide.

> **Important**
>
> `rstan` compiles your model the first time you define it. This takes some time. To avoid re-compilation every time you use the model, and to use all available cores on your computer, add the following settings in the beginning of your stan code:

```
library(rstan)
library(loo)
options(mc.cores = parallel::detectCores())
rstan_options(auto_write = TRUE)
```

Problem 7a)

Sample 1000 draws from the posterior distribution of β in the Poisson regression from Problem 5), but this time using the Hamiltonian Monte Carlo algorithm in the `rstan` package.

> **Tip**
>
> Stan includes a special function `poisson_log` that implements the Poisson distribution with a so called *log link*. This is the same as a Poisson distribution with $\lambda = \exp(\theta)$, so that the exponential function (which is the inverse function to the log function) is built-in from the start. Read this: StanUserGuide - Poisson regression.

```
# example(stan_model, package = "rstan", run.dontrun = TRUE)
Poisreg_dat = list(N = nrow(X), y = y, K = ncol(X), X = X, mu = rep(0,5), sigma = diag(100,5)

fit = stan(file = "Poisreg.stan", data = Poisreg_dat, iter = 1000)
```

```
fit
```

```
Inference for Stan model: anon_model.
4 chains, each with iter=1000; warmup=500; thin=1;
```

```
post-warmup draws per chain=500, total post-warmup draws=2000.

          mean se_mean   sd   2.5%    25%    50%    75%  97.5% n_eff Rhat
beta[1]  -1.10    0.02 0.43  -1.98  -1.38  -1.08  -0.81  -0.30   582    1
beta[2]  -0.01    0.00 0.01  -0.04  -0.02  -0.01   0.00   0.02  1117    1
beta[3]   1.43    0.02 0.57   0.35   1.06   1.41   1.78   2.57   816    1
beta[4]   3.16    0.03 0.67   1.90   2.72   3.14   3.59   4.48   664    1
beta[5]   2.16    0.01 0.26   1.67   1.98   2.16   2.34   2.67  1154    1
lp__    362.53    0.06 1.67 358.18 361.77 362.85 363.74 364.67   729    1

Samples were drawn using NUTS(diag_e) at Thu Oct  2 16:34:49 2025.
For each parameter, n_eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor on split chains (at
convergence, Rhat=1).
```

Problem 7b)

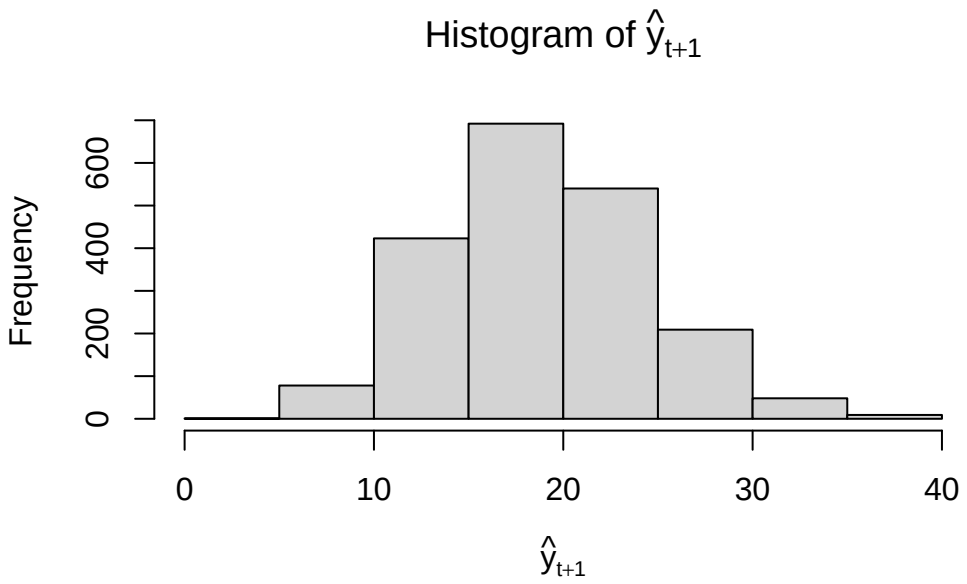
The upcoming release, has the following covariate vector:

```
xNew = c(1, 10, 0.45, 0.5, 0.89)

Poisreg_dat_new = list(N = nrow(X), y = y, K = ncol(X), X = X, mu = rep(0,5), sigma = diag(10

fit_new = stan(file = "PoisregPred.stan", data = Poisreg_dat_new, iter = 1000)
```

```
Running "C:/PROGRA~1/R/R-44~1.0/bin/x64/Rcmd.exe" SHLIB foo.c
using C compiler: 'gcc.exe (GCC) 13.3.0'
gcc  -I"C:/PROGRA~1/R/R-44~1.0/include" -DNDEBUG   -I"C:/Users/Thomas/AppData/Local/R/win-lib
cc1.exe: warning: command-line option '-std=c++14' is valid for C++/ObjC++ but not for C
In file included from C:/Users/Thomas/AppData/Local/R/win-library/4.4/RcppEigen/include/Eigen
                 from C:/Users/Thomas/AppData/Local/R/win-library/4.4/RcppEigen/include/Eigen
                 from C:/Users/Thomas/AppData/Local/R/win-library/4.4/StanHeaders/include/sta
                 from <command-line>:
C:/Users/Thomas/AppData/Local/R/win-library/4.4/RcppEigen/include/Eigen/src/Core/util/Macros
  679 | #include <cmath>
      |          ^~~~~~
compilation terminated.
make: *** [C:/PROGRA~1/R/R-44~1.0/etc/x64/Makeconf:289: foo.o] Error 1
```

So, the release is based on 10 commits, good proportions of C and Java code and a high code complexity of 0.89. Use Stan to simulate from the predictive distribution of the number of bugs in this release.

```r
y_new_pred = extract(fit_new, pars = c("y_new[1]"))

hist(y_new_pred[[1]], main = TeX(paste("Histogram of", "$\\hat{y}_{t+1}$")), xlab = TeX("$\\
```



> **Tip**
>
> Add a `generated quantities` section to your Stan model to produce the predictive distribution, see Stan User Guide - Prediction.

Problem 7c)

Consider the negative binomial regression for the same bugs data:

$$Y_i|x_i \overset{\text{ind}}{\sim} \text{NegBin}\Big(r, \mu_i = \exp\left(x_i^\top \beta\right)\Big)$$

> **Note**
>
> We are here using the parameterization of the negative binomial distribution where the **second parameter is the mean**; this corresponds to using the `mean` argument in `dnbinom` function in plain R. In Stan, this distribution (again, with the log link built-in) is called `neg_binomial_2_log`, see Stan User Guide - negative binomial regression.

The parameters in this model are therefore the vector of negative binomial regression coefficients β **and** the scalar parameter $r > 0$. Use HMC in Stan to sample 1000 draws from joint

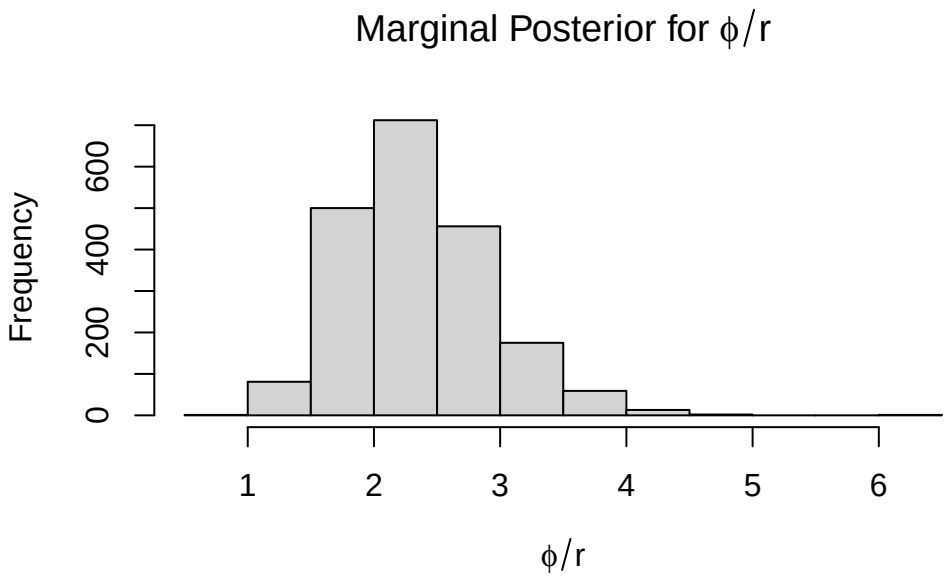
posterior $p(\beta, r|y, X)$. Plot the marginal posterior for r. What does this posterior tell you about the suitability of the Poisson regression model in 7a)?

```
NegbinData = list(N = nrow(X), y = y, K = ncol(X), X = X, mu = rep(0,5), sigma = diag(100,5))

fitNegBin = stan(file = "NegbinReg.stan", data = NegbinData, iter = 1000) # Should we use a p
```

```
fitNegBin
```

```
Inference for Stan model: anon_model.
4 chains, each with iter=1000; warmup=500; thin=1;
post-warmup draws per chain=500, total post-warmup draws=2000.

          mean se_mean   sd    2.5%     25%     50%     75%   97.5% n_eff Rhat
beta[1]   -1.14    0.03 0.71   -2.57   -1.62   -1.14   -0.68    0.30   803 1.00
beta[2]   -0.01    0.00 0.03   -0.07   -0.03   -0.01    0.01    0.04  1487 1.00
beta[3]    1.29    0.03 1.02   -0.70    0.63    1.27    1.95    3.30  1173 1.00
beta[4]    3.07    0.04 1.17    0.74    2.30    3.05    3.85    5.38   984 1.00
beta[5]    2.39    0.01 0.45    1.50    2.10    2.39    2.68    3.30  1480 1.00
phi        2.35    0.01 0.57    1.42    1.94    2.28    2.70    3.62  1663 1.00
lp__    -235.56    0.06 1.68 -239.69 -236.48 -235.29 -234.28 -233.22   916 1.01

Samples were drawn using NUTS(diag_e) at Thu Oct  2 16:35:44 2025.
For each parameter, n_eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor on split chains (at
convergence, Rhat=1).
```

```
hist(extract(fitNegBin, par = "phi")[[1]], breaks = 15, main = TeX(paste("Marginal Posterior
```



For poisson the variance is equal to the mean and this is true for the negative binomial too in this parametrization if ϕ/r goes to infinity(seen in the expression below) so for poisson to be a good model, we would expect this property of variance being equal to the mean of the distribution. Here marginal posterior shows ϕ/r is not very large at all, quite small in fact with a posterior mode of about 2. This means we have evidence that poisson is not very suitable as one of its inherent properties required is shown to not be true for this data.

Problem 7d)

Compare the Poisson regression and Negative binomial regression models using Bayesian **leave-one-out (LOO) cross-validation**. Use the posterior samples from `rstan` and the `loo` package to compute the expected log predictive density from leave-one-out cross-validation (**ELPD-LOO**) for each model. Which model is preferred?

> **Note**
>
> Computing the ELPD-LOO from the Stan output requires that we tell Stan to store the log-likelihood values for each HMC draw. Similar to prediction, we achieve this by adding the log-likelihood computation to the `generated quantities` section. See the Writing Stan programs for use with the loo package.

```
fitPoisLoo = stan(file = "PoisRegLoo.stan", data = Poisreg_dat, iter = 1000)
```

```
loglikPois = extract_log_lik(fitPoisLoo, merge_chains = FALSE)
reffPois = relative_eff(loglikPois)
```

```
looPois = loo(loglikPois, r_eff = reffPois, cores = 2)
looPois
```

```
Computed from 2000 by 91 log-likelihood matrix.

         Estimate   SE
elpd_loo   -301.1 35.4
p_loo        18.6  4.9
looic       602.2 70.8
------
MCSE of elpd_loo is 0.2.
MCSE and ESS estimates assume MCMC draws (r_eff in [0.3, 1.1]).

All Pareto k estimates are good (k < 0.7).
See help('pareto-k-diagnostic') for details.
```

```
fitNegBinLoo = stan(file = "NegbinRegLoo.stan", data = NegbinData,iter = 1000)
#loglikNegBin = extract_log_lik(fitNegBinLoo)
```

```
loglikNegBin = extract_log_lik(fitNegBinLoo, merge_chains = FALSE)
reffNegBin = relative_eff(loglikNegBin)
looNegBin = loo(loglikNegBin, r_eff = reffNegBin, cores = 2)
looNegBin
```

```
Computed from 2000 by 91 log-likelihood matrix.

         Estimate   SE
elpd_loo   -239.8 10.4
p_loo         6.5  1.6
looic       479.7 20.8
------
MCSE of elpd_loo is 0.1.
MCSE and ESS estimates assume MCMC draws (r_eff in [0.4, 1.0]).

All Pareto k estimates are good (k < 0.7).
See help('pareto-k-diagnostic') for details.
```

The table below from the leave one out package, shows the difference in expected log predictive densities between the negative binomial versus the poisson models. We can see clearly that the negative binomial has a higher ELPD which makes sense as the poisson property of variance in

the data being equal to the mean which we showed evidence against earlier in section Problem 7c.

```
compareLoo = loo_compare(looPois, looNegBin)
print(compareLoo, simplify = FALSE)
```

```
       elpd_diff se_diff elpd_loo se_elpd_loo p_loo  se_p_loo looic  se_looic
model2    0.0       0.0  -239.8     10.4         6.5    1.6    479.7   20.8
model1  -61.2      26.6  -301.1     35.4        18.6    4.9    602.2   70.8
```