Homework 4

Amy Kuang, Shravan Shenoy - PSTAT 115, Spring 2021

Due on June 12, 2021 at 11:59 pm

```
library(knitr)
knitr::opts_chunk$set(echo=TRUE,
                      cache=FALSE,
                      fig.width=5,
                      fig.height=5,
                      fig.align='center')

r = function(x, digits=2){ round(x, digits=digits) }
indent1 = '    '
indent2 = paste(rep(indent1, 2), collapse='')

options(tinytex.verbose = TRUE)
options(buildtools.check = function(action) TRUE )
knitr::opts_chunk$set(echo = TRUE, eval=TRUE)
suppressPackageStartupMessages(library(tidyverse))
suppressPackageStartupMessages(library(rstan))
suppressPackageStartupMessages(library(coda))
suppressPackageStartupMessages(library(testthat))
```

Problem 1. Logistic regression for toxicity data (part 1)

A environmental agency is testing the effects of a pesticide that can cause acute poisoning in bees, the world's most important pollinator of food crops. The environmental agency collects data on exposure to different levels of the pestidicide in parts per million (ppm). The agency also identifies collapsed beehives, which they expect could be due to acute pesticide poisoning. In the data they collect, each observation is pair (x_i, y_i), where x_i represents the dosage of the pollutant and y_i represents whether or not the hive survived. Take $y_i = 1$ means that the beehive has collapsed from poisoning and $y_i = 0$ means the beehive survived. The agency collects data at several different sites, each of which was exposed to a different dosages. The resulting data can be seen below:

```
x <- c(1.06, 1.41, 1.85, 1.5, 0.46, 1.21, 1.25, 1.09,
       1.76, 1.75, 1.47, 1.03, 1.1, 1.41, 1.83, 1.17,
       1.5, 1.64, 1.34, 1.31)

y <- c(0, 1, 1, 1, 0, 1, 1, 1, 1, 1,
       1, 0, 0, 1, 1, 0, 0, 1, 1, 0)
```

Assume that beehiv collapse, y_i, given pollutant exposure level x_i, is $Y_i \sim \text{Bernoulli}(\theta(x_i))$, where $\theta(x_i)$ is the probability of death given dosage x_i. We will assume that $\text{logit}(\theta_i(x_i)) = \alpha + \beta x_i$ where $\text{logit}(\theta)$ is defined as $\log(\theta/(1-\theta))$. This model is known as *logistic regression* and is one of the most common methods for modeling probabilities of binary events.

1a. Solve for $\theta_i(x_i)$ as a function of α and β by inverting the logit function. If you haven't seen logistic

regression before (it is covered in more detail in PSTAT 127 and PSTAT131), it is essentially a generalization of linear regression for binary outcomes. The inverse-logit function maps the linear part, $\alpha + \beta x_i$, which can be any real-valued number into the interval [0, 1] (since we are modeling probabilities of binary outcome, we need the mean outcome to be confined to this range).

Solving for $\theta_i(x_i)$ as follows:

$$\text{logit}(\frac{(\theta_i(x_i))}{1-\theta_i(x_i)}) = \alpha + \beta x_i$$

$$e^{\text{logit}(\frac{(\theta_i(x_i))}{1-\theta_i(x_i)})} = e^{\alpha+\beta x_i}$$

$$\frac{(\theta_i(x_i))}{1-\theta_i(x_i)} = e^{\alpha+\beta x_i}$$

$$\theta_i(x_i) = (1-\theta_i(x_i))(e^{\alpha+\beta x_i})$$

$$\theta_i(x_i) = e^{\alpha+\beta x_i} - \theta_i(x_i) \cdot e^{\alpha+\beta x_i}$$

$$\theta_i(x_i) + \theta_i(x_i) \cdot e^{\alpha+\beta x_i} = e^{\alpha+\beta x_i}$$

$$\theta_i(x_i)(1 + e^{\alpha+\beta x_i}) = e^{\alpha+\beta x_i}$$

$$\therefore \theta_i(x_i) = \frac{e^{\alpha+\beta x_i}}{1 + e^{\alpha+\beta x_i}}$$

1b The dose at which there is a 50% chance of beehvive collapse, $\theta(x_i) = 0.5$, is known as LD50 ("letha dose 50%"), and is often of interest in toxicology studies. Solve for LD50 as a function of α and β.

Solve for LD50 as a function of α and β as follows:

Plug in $\theta(x_i) = 0.5$ into 1a and solve:

$$0.5 = \frac{e^{\alpha+\beta x_i}}{1 + e^{\alpha+\beta x_i}}$$

$$0.5(1 + e^{\alpha+\beta x_i}) = e^{\alpha+\beta x_i}$$

$$0.5 + 0.5e^{\alpha+\beta x_i} = e^{\alpha+\beta x_i}$$

$$0.5 = 0.5e^{\alpha+\beta x_i}$$

$$1 = e^{\alpha+\beta x_i}$$

$$log(1) = \alpha + \beta x_i$$

$$0 = \alpha + \beta x_i$$

$$-\alpha = \beta x_i$$

$$\therefore -\frac{\alpha}{\beta} = x_i$$

1c Implement the logistic regression model in stan by reproducing the stan model described here: https://mc-stan.org/docs/2_18/stan-users-guide/logistic-probit-regression-section.html. Run the stan model on the beehive data to get Monte Carlo samples. Compute Monte Carlo samples of the LD50 by applying the function derived in the previous part to your α and β samples. Report and estimate of the posterior mean of the LD50 by computing the sample average of all Monte Carlo samples of LD50.

```
# YOUR CODE HERE
input_dat = list(N=20, y=y, x=x )
stan_fit = stan(file ="beehive.stan",data=input_dat)
```

```
##
## SAMPLING FOR MODEL 'beehive' NOW (CHAIN 1).
## Chain 1:
## Chain 1: Gradient evaluation took 1.5e-05 seconds
## Chain 1: 1000 transitions using 10 leapfrog steps per transition would take 0.15 seconds.
## Chain 1: Adjust your expectations accordingly!
## Chain 1:
## Chain 1:
## Chain 1: Iteration:    1 / 2000 [  0%]  (Warmup)
## Chain 1: Iteration:  200 / 2000 [ 10%]  (Warmup)
## Chain 1: Iteration:  400 / 2000 [ 20%]  (Warmup)
## Chain 1: Iteration:  600 / 2000 [ 30%]  (Warmup)
## Chain 1: Iteration:  800 / 2000 [ 40%]  (Warmup)
## Chain 1: Iteration: 1000 / 2000 [ 50%]  (Warmup)
## Chain 1: Iteration: 1001 / 2000 [ 50%]  (Sampling)
## Chain 1: Iteration: 1200 / 2000 [ 60%]  (Sampling)
## Chain 1: Iteration: 1400 / 2000 [ 70%]  (Sampling)
## Chain 1: Iteration: 1600 / 2000 [ 80%]  (Sampling)
## Chain 1: Iteration: 1800 / 2000 [ 90%]  (Sampling)
## Chain 1: Iteration: 2000 / 2000 [100%]  (Sampling)
## Chain 1:
## Chain 1:  Elapsed Time: 0.059005 seconds (Warm-up)
## Chain 1:                0.053981 seconds (Sampling)
## Chain 1:                0.112986 seconds (Total)
## Chain 1:
##
## SAMPLING FOR MODEL 'beehive' NOW (CHAIN 2).
## Chain 2:
## Chain 2: Gradient evaluation took 5e-06 seconds
## Chain 2: 1000 transitions using 10 leapfrog steps per transition would take 0.05 seconds.
## Chain 2: Adjust your expectations accordingly!
## Chain 2:
## Chain 2:
## Chain 2: Iteration:    1 / 2000 [  0%]  (Warmup)
## Chain 2: Iteration:  200 / 2000 [ 10%]  (Warmup)
## Chain 2: Iteration:  400 / 2000 [ 20%]  (Warmup)
## Chain 2: Iteration:  600 / 2000 [ 30%]  (Warmup)
## Chain 2: Iteration:  800 / 2000 [ 40%]  (Warmup)
## Chain 2: Iteration: 1000 / 2000 [ 50%]  (Warmup)
## Chain 2: Iteration: 1001 / 2000 [ 50%]  (Sampling)
## Chain 2: Iteration: 1200 / 2000 [ 60%]  (Sampling)
## Chain 2: Iteration: 1400 / 2000 [ 70%]  (Sampling)
## Chain 2: Iteration: 1600 / 2000 [ 80%]  (Sampling)
## Chain 2: Iteration: 1800 / 2000 [ 90%]  (Sampling)
## Chain 2: Iteration: 2000 / 2000 [100%]  (Sampling)
## Chain 2:
## Chain 2:  Elapsed Time: 0.057401 seconds (Warm-up)
## Chain 2:                0.049367 seconds (Sampling)
## Chain 2:                0.106768 seconds (Total)
## Chain 2:
##
## SAMPLING FOR MODEL 'beehive' NOW (CHAIN 3).
## Chain 3:
## Chain 3: Gradient evaluation took 8e-06 seconds
```

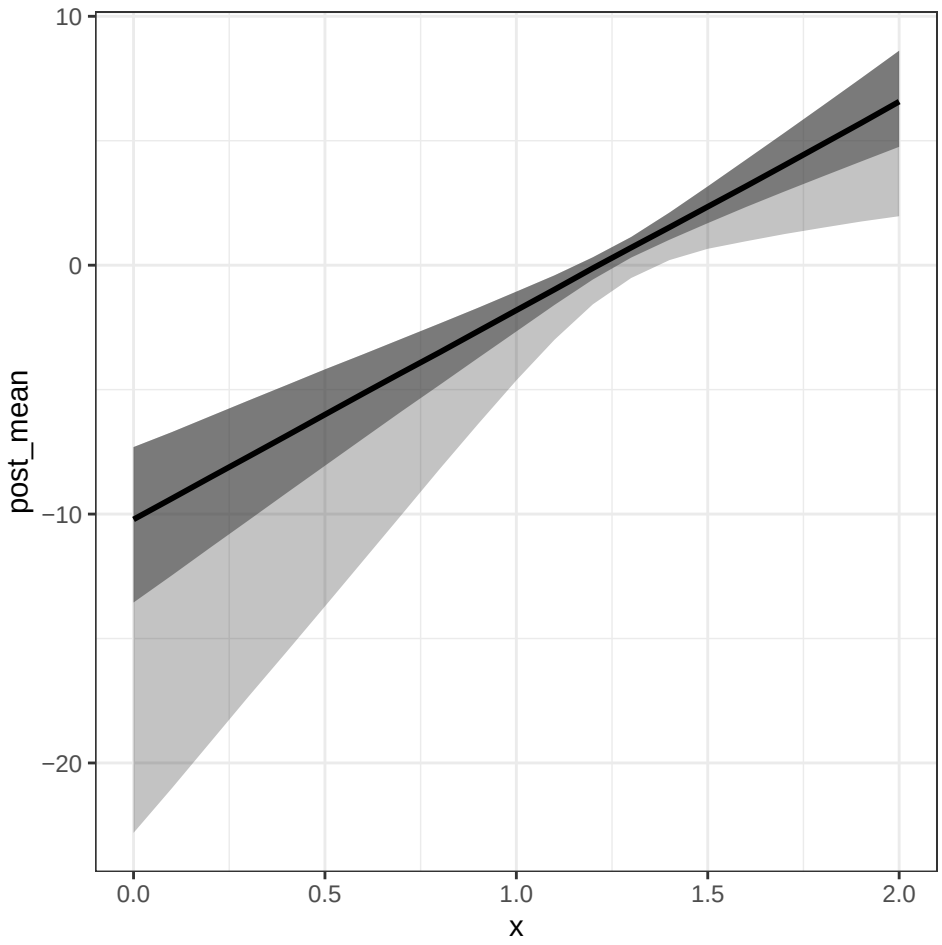
```
## Chain 3: 1000 transitions using 10 leapfrog steps per transition would take 0.08 seconds.
## Chain 3: Adjust your expectations accordingly!
## Chain 3:
## Chain 3:
## Chain 3: Iteration:    1 / 2000 [  0%]  (Warmup)
## Chain 3: Iteration:  200 / 2000 [ 10%]  (Warmup)
## Chain 3: Iteration:  400 / 2000 [ 20%]  (Warmup)
## Chain 3: Iteration:  600 / 2000 [ 30%]  (Warmup)
## Chain 3: Iteration:  800 / 2000 [ 40%]  (Warmup)
## Chain 3: Iteration: 1000 / 2000 [ 50%]  (Warmup)
## Chain 3: Iteration: 1001 / 2000 [ 50%]  (Sampling)
## Chain 3: Iteration: 1200 / 2000 [ 60%]  (Sampling)
## Chain 3: Iteration: 1400 / 2000 [ 70%]  (Sampling)
## Chain 3: Iteration: 1600 / 2000 [ 80%]  (Sampling)
## Chain 3: Iteration: 1800 / 2000 [ 90%]  (Sampling)
## Chain 3: Iteration: 2000 / 2000 [100%]  (Sampling)
## Chain 3:
## Chain 3:  Elapsed Time: 0.054788 seconds (Warm-up)
## Chain 3:                0.059687 seconds (Sampling)
## Chain 3:                0.114475 seconds (Total)
## Chain 3:
##
## SAMPLING FOR MODEL 'beehive' NOW (CHAIN 4).
## Chain 4:
## Chain 4: Gradient evaluation took 5e-06 seconds
## Chain 4: 1000 transitions using 10 leapfrog steps per transition would take 0.05 seconds.
## Chain 4: Adjust your expectations accordingly!
## Chain 4:
## Chain 4:
## Chain 4: Iteration:    1 / 2000 [  0%]  (Warmup)
## Chain 4: Iteration:  200 / 2000 [ 10%]  (Warmup)
## Chain 4: Iteration:  400 / 2000 [ 20%]  (Warmup)
## Chain 4: Iteration:  600 / 2000 [ 30%]  (Warmup)
## Chain 4: Iteration:  800 / 2000 [ 40%]  (Warmup)
## Chain 4: Iteration: 1000 / 2000 [ 50%]  (Warmup)
## Chain 4: Iteration: 1001 / 2000 [ 50%]  (Sampling)
## Chain 4: Iteration: 1200 / 2000 [ 60%]  (Sampling)
## Chain 4: Iteration: 1400 / 2000 [ 70%]  (Sampling)
## Chain 4: Iteration: 1600 / 2000 [ 80%]  (Sampling)
## Chain 4: Iteration: 1800 / 2000 [ 90%]  (Sampling)
## Chain 4: Iteration: 2000 / 2000 [100%]  (Sampling)
## Chain 4:
## Chain 4:  Elapsed Time: 0.058222 seconds (Warm-up)
## Chain 4:                0.066032 seconds (Sampling)
## Chain 4:                0.124254 seconds (Total)
## Chain 4:
```

```
samples = extract(stan_fit)
alpha_samples = samples$alpha
beta_samples = samples$beta

# estimate of posterior mean of the LD50
mean(-alpha_samples/beta_samples)
```

```
## [1] 1.189261
```

1d. Make a plot showing both 50% and 95% confidence band for the probability of a hive collapse as a function of pollutant exposure, $\Pr(y = 1 \mid \alpha, \beta, x)$. Plot your data on a grid of x-values from $x = 0$ to 2. *Hint:* see lab 7 for a similar example.

```
# YOUR CODE HERE
x <- seq(0, 2, by=0.1)

point_estimates <- function(sample){
  alpha <- sample[1]
  beta <- sample[2]
  y_values <- alpha + beta*x
}

results <- apply(cbind(alpha_samples, beta_samples),1, point_estimates )
sample_quantiles <- apply(results, 1, function(x) quantile(x, c(0.025,0.25, 0.75, 0.975)))

post_mean <- rowMeans(results)
post_mean <- apply(results, 1, median)

data <- tibble(x=x, q1 = sample_quantiles[1,],
       q2 = sample_quantiles[2,],
       q3 = sample_quantiles[3,],
       q4 = sample_quantiles[4,],
       mean = post_mean)

ggplot(data)+
    geom_ribbon(aes(x = x, ymin = q1, ymax = q3), alpha=0.3) +
    geom_ribbon(aes(x = x, ymin = q2, ymax = q3), alpha = 0.5) +
    geom_line(aes(x = x, y= post_mean), size = 1 ) +
    theme_bw()
```



Problem 2. Logistic regression for toxicity data (part 2)

In the problem 1, we inferred the effects of the pesticide by fitting a model in Stan. In order to develope a deeper understanding of MCMC, in this problem we will implement our own Metropolis-Hastings algoritihm. To do so, we need to first write a function to compute the *log* posterior density. Why the log posterior? In practice, the posterior density may have *extremely* small values, especially when we initialize the sampler and may be far from the high posterior mode areas. As such, computing the

For example, computing the ratio of a normal density 1000 standard deviations from the mean to a normal density 1001 standard deviations from the mean fails because in both cases `dnorm` evalutes to 0 due to numerical underflow and 0/0 returns NaN. However, we can compute the log ratio of densities:

```
dnorm(1000) / dnorm(1001)
```

```
## [1] NaN
```

```
dnorm(1000, log=TRUE) - dnorm(1001, log=TRUE)
```

```
## [1] 1000.5
```

Let $r = \min(1, \frac{p(\theta^*|y)}{p(\theta_t|y)})$. In the accept/reject step of the your implementation of the MH algorithm, rather than checking whether $u < r$, it is equivalent to check whether $log(u) < log(r)$. Doing the accept/reject on the log scale will avoid any underflow issues and prevent our code from crashing.

2a. Complete the specification for the log posterior for the data `x` and `y` by filling in the missing pieces of the function below.

```r
## Pesticide toxicity data
x <- c(1.06, 1.41, 1.85, 1.5, 0.46, 1.21, 1.25, 1.09,
       1.76, 1.75, 1.47, 1.03, 1.1, 1.41, 1.83, 1.17,
       1.5, 1.64, 1.34, 1.31)

y <- c(0, 1, 1, 1, 0, 1, 1, 1, 1, 1,
       1, 0, 0, 1, 1, 0, 0, 1, 1, 0)

#Log posterior function.  Must incorporate x and y data above.
log_posterior <- function(theta) {

  alpha <- theta[1]
  beta <- theta[2]

  ## Compute the probabilities as a function of alpha and beta
  ## for the observed x, y data
  prob <- (exp(alpha + beta * x)) / (1+exp(alpha + beta*x))

  if(any(prob == 0) | any(prob == 1))
    -Inf ## log likelihood is -Inf is prob=0 or 1
  else
    sum(y*(log(prob)) + (1-y) * log((1-prob)))
}
```

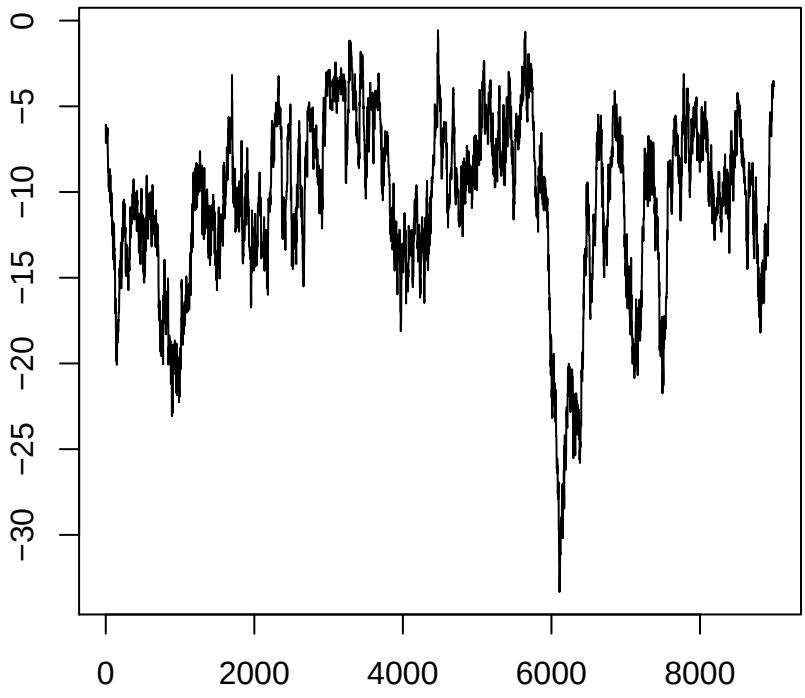
2b. You will now complete the Metropolis-Hastings sampler by filling in the missing pieces of the algorithm below. `theta_0` is a vector of length 2, with the first argument as the initial alpha value and the second argument as the initial beta value. As your proposal, use $J(\theta * |\theta_t) \sim Normal(\theta_t, \Sigma)$. You can sample from the multivariate normal using `mvtnorm::rmvnorm`. The effectiveness of your sampler will be determined by the tuning parameter, Σ, the covariance of the bivariate normal distribution. This determines the size / shape of the proposal. Σ is determined by the `cov` argument in your sampler. Run the sampler with `cov = diag(2)`, the default. In homework 5 you showed that the dose at which there is a 50% chance of hive collapse, the LD50, can be expressed as $-\alpha/\beta$. Run your sampler for 10000 iterations with a burnin of 1000 iterations. Verify that the posterior mean LD50 based on your sampler is close to 1.2, as it was with stan.

```r
###############################################
## Metropolis-Hastings for the Logistic Model
###############################################

## Function to generate samples using the Metropolis-Hasting Sampler

## theta_0: initialization of the form c(alpha_init, beta_init) for some values alpha_init, beta_init
## burnin: amount of iterations to discard to reduce dependence on starting point
## iters: total number of iterations to run the algorithm (must be greater than `burnin`)

mh_logistic <- function(theta_0, burnin, iters, cov=diag(2)){

    # Initialize parameters.
    theta_t <- theta_0

    ## Create a matrix where we will store samples
    theta_out <- matrix(0, nrow=iters, ncol=2, dimnames=list(1:iters, c("alpha", "beta")))

    for(i in 1:iters){
```

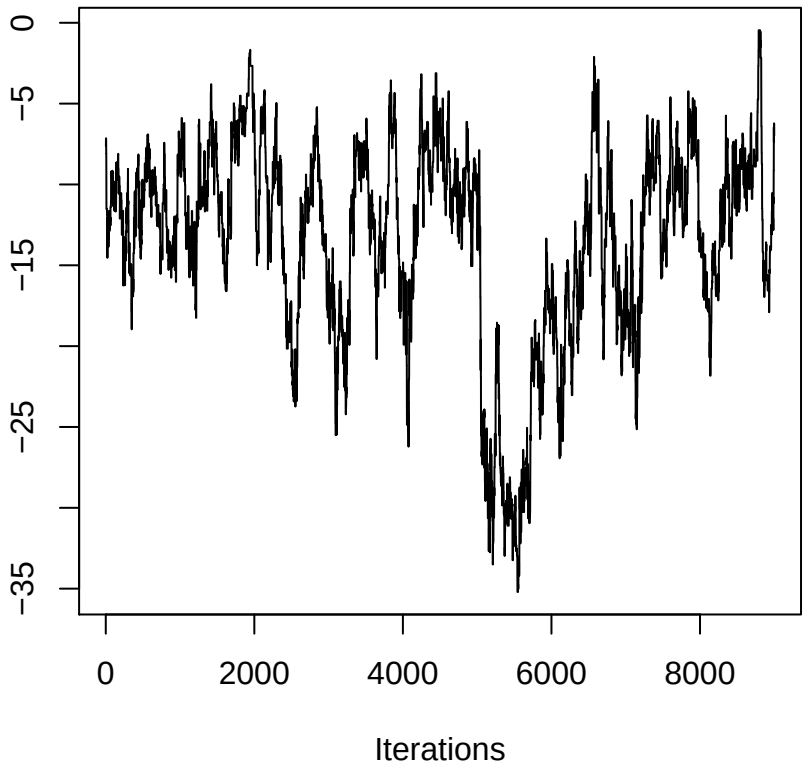
```
        ## Propose new theta = (alpha, beta)
        ## The proposal will be centered the current
        ## value theta_t.  Use mvtnorm::rmvnorm

        theta_p <- mvtnorm::rmvnorm(1, theta_t, cov)

        ## Accept/reject step.  Keep theta prev if reject, otherwise take theta_p
        ## Will require evaluting `log_posterior` function twice
        ## Log-rejection ratio for symmetric proposal
        logr <- log_posterior(theta_p)-log_posterior(theta_t)

        ## Update theta_t based on whether the proposal is accepted or not

        s <-log(runif (1, min=0, max=1))
        if(s<logr){theta_t=theta_p}

        ## Save the draw
        theta_out[i, ] <- theta_t
    }

    ## Chop off the first part of the chain -- this reduces dependence on the starting point.
    if(burnin == 0)
      theta_out
    else
      theta_out[-(1:burnin), ]
}

samples <- mh_logistic(c(0, 0), 1000, 10000)


ld50_posterior_mean <- -mean(samples[,1]) / mean(samples[,2])
```

2c. Report the effective sample size for the alpha samples using the `coda::effectiveSize` function. Make a traceplot of the samples of the alpha parameter. If `alpha_samples` were the name of the samples of the alpha parameter, then you can plot the traceplot using `coda::traceplot(as.mcmc(alpha_samples))`. Improve upon this effective sample size from your first run by finding a new setting for `cov`. *Hint:* try variants of `k*diag(2)` for various values of k to increase or decrease the proposal variance. If you are ambitious, try proposing using a covariance matrix with non-zero correlation between the two parameters. What effective sample size were you able to achieve? You should be able to at least double the effective sample size from your first run. Plot the traceplot based on the new value of `cov`.

```
library(coda)

samples <- samples
alpha_samples <- samples[,1]
alpha_ess <- coda::effectiveSize(alpha_samples)

# TRACEPLOT HERE
coda::traceplot(as.mcmc(alpha_samples))
```




```
## Re run the sampler using your new setting of cov
k=2
alpha_samples_new <- mh_logistic(c(0,0),1000,10000, k*diag(2))
alpha_samples_new <- alpha_samples_new[,1]
alpha_ess_new <- coda::effectiveSize(alpha_samples_new)

# TRACEPLOT HERE
coda::traceplot(as.mcmc(alpha_samples_new))
```



Problem 3. Estimating Skill In Baseball

In baseball, the batting average is defined as the fraction of base hits (successes) divided by "at bats" (attempts). We can conceptualize a player's "true" batting skill as $p_i = \lim_{n_i \to \infty} \frac{y_i}{n_i}$. In other words, if each at bat was independent (a simplifying assumption), p_i describes the total fraction of success for player i as the number of attempts gets very large. Our goal is to estimate the true skill of all player as best as possible using only a limited amount of data. As usual, for independent counts of success/fail data it is reasonable to assume that $Y_i \sim \text{Bin}(n_i, p_i)$. The file "lad.csv" includes the number of hits, `y` and the number of attempts `n` for $J = 10$ players on the Los Angeles Dodgers after the first month of the most recent baseball season. The variable `val` includes the end-of-season batting average and will be used to validate the quality of various estimates. If you are interested, at the end of the assignment we have included the code that was used to scrape the data.

```
baseball_data <- read_csv("lad.csv", col_types=cols())
baseball_data
```

```
## # A tibble: 13 x 4
##    name                 y     n   val
##    <chr>            <dbl> <dbl> <dbl>
##  1 AJ Pollock          16    86 0.258
##  2 Alex Verdugo        22    65 0.293
##  3 Austin Barnes       10    66 0.186
##  4 Chris Taylor        12    65 0.269
##  5 Cody Bellinger      38    93 0.297
##  6 Corey Seager        23   100 0.274
##  7 David Freese         9    37 0.321
##  8 Enrique Hernandez   23    89 0.234
##  9 Joc Pederson        15    77 0.241
## 10 Justin Turner       26    93 0.291
## 11 Max Muncy           21    82 0.254
```

```
## 12 Rocky Gale            2    14 0.133
## 13 Russell Martin        4    16 0.217
```

```
## observed hits in the first month
y <- baseball_data$y

## observed at bats in the first month
n <- baseball_data$n

## observed batting average in the first month (same as MLE)
theta_mle <- y/n

## number of players
J <- nrow(baseball_data)

## end of the year batting average, used to evaluate estimates
val <- baseball_data$val
```

3a. Compute the standard deviation of the empirical batting average, y/n and then compute the sd of the "true skill", (the `val` variable representing the end of season batting average). Which is smaller? Why does this make sense? *Hint:* What sources of variation are present in the empirical batting average?

```
empirical_sd <- sd(y/n)

true_sd <- sd(val)

print(empirical_sd)
```

```
## [1] 0.07403955
```

```
print(true_sd)
```

```
## [1] 0.05081263
```

The true_sd is smaller than the empirical one. This makse sense as the variation for true skill is going to be small over long durations of time

3b. Consider two estimates for the true skill of player i, p_i: 1) $\hat{p}_i^{(\text{mle})} = \frac{y_i}{n_i}$ and 2) $\hat{p}_i^{(\text{comp})} = \frac{\sum_j y_j}{\sum n_j}$. Estimator 1) is the MLE for each player and ignores any commonalities between the observations. This is sometimes termed the "no pooling" estimator since each parameter is estimating separately without "pooling" information between them. Estimator 2) assumes all players have identical skill and is sometimes called the "complete pooling" estimator, because the data from each problem is completely "pooled" into one common set. In this problem, we'll treat the end-of-season batting average as a proxy for true skill, p_i. Compute the root mean squared error (RMSE), $\sqrt{\frac{1}{J}\sum_i (\hat{p}_i - p_i)^2}$ for the "no pooling" and "complete pooling" estimators using the variable `val` as a stand-in for the true p_i. Does "no pooling" or "complete pooling" give you a better estimate of the end-of-year batting averages in this specific case?

```
# Maximum likelihood estimate
phat_mle <- y/n

# Pooled estimate
phat_pooled <- rep(sum(y) / sum(n),J)

rmse_complete_pooling <- sqrt((1/J)*sum((phat_pooled-val)**2))
rmse_no_pooling <-sqrt((1/J)*sum((phat_mle-val)**2))
```

```
print(sprintf("MLE: %f", rmse_no_pooling))
```

```
## [1] "MLE: 0.055500"
```

```
print(sprintf("Pooled: %f", rmse_complete_pooling))
```

```
## [1] "Pooled: 0.048832"
```

No pooling gives a better estimate of the end-of-year batting averages as seen by its lower rmse value.

The no pooling and complete pooling estimators are at opposite ends of a spectrum. There is a more reasonable compromise: "partial pooling" of information between players. Although we assume the number of hits follow a binomial distribution. To complete this specification, we assume $\text{logit}(p_i) \sim N(\mu, \tau^2)$ for each player i. μ is the "global mean" (on the logit scale), $\exp(\mu)/(1+\exp(\mu)$ is the overall average batting average across all players. τ describes how much variability there is in the true skill of players. If $\tau = 0$ then all players are identical and the only difference in the observed hits is presumed to be due to chance. If τ^2 is very large then the true skill differences between players is assumed to be large and our estimates will be close to the "no pooling" estimator. How large should τ be? We don't know but we can put a prior distribution over the parameter and sample it along with the p_i's! Assume the following model:

$$
\begin{aligned}
&y_i \sim Bin(n_i, p_i) \\
&\theta_i = logit(p_i) \\
&\theta \sim N(\mu, \tau^2) \\
&p(\mu) \propto \text{const} \\
&p(\tau) \propto \text{Cauchy}(0,1)^+, \text{ (the Half-cauchy distribution, see part d.)}
\end{aligned}
$$

3c. State the correct answer in each case: as $\tau \to \infty$, the posterior mean estimate of p_i in this model will approach the (complete pooling / no pooling) estimator and as $\tau \to 0$ the posterior mean estimate of p_i will approach the (complete pooling / no pooling) estimator. Give a brief justification for your answer.

As tau approaches infinity, the post mean estimation will tend towards the no pooling estimator. This is because the skill of the players should increase over time, with varying amounts for each person

As tau approaches zero, the post mean estimation will tend towards the complete pooling estimator. This implied that the level of skill for the players is relatively the same at that level.

3d. Implement the hierarchical binomial model in Stan. As a starting point for your Stan file modify the `eight_schools.stan` file we have provided and save it as `baseball.stan`. To write the hierarchical binomial model, we need the following modifications to the normal hierarchical model:

- Since we are fitting a hierarchical binomial model, not a normal distribution, we no longer need sampling variance σ_i^2. Remove this from the data block.
- The outcomes `y` are now integers. Change `y` to an array of integer types in the data block.
- We need to include the number of at bats for each player (this is part of the binomial likelihood). Add an array of integers, `n` of length J to the data block.
- Replace the sampling model for y with the binomial-logit: `binomial_logit(n, theta)`. This is equivalent to `binomial(n, inv_logit(theta))`.
- The model line for `eta` makes $\theta_i \sim N(\mu, \tau^2)$. Leave this in the model.
- Add a half-cauchy prior distribution for τ: `tau ~ cauchy(0, 1);`. The half-cauchy has been suggested as a good default prior distribution for group-level standard deviations in hierarchical models. See http://www.stat.columbia.edu/~gelman/research/published/taumain.pdf.

Find the posterior means for each of the players batting averages by looking at the samples for `inv_logit(theta_samples)`. Report the RMSE for hierarchical estimator. How does this compare to the RMSE of the complete pooling and no pooling estimators? Which estimator had the lowest error?

```
# Run Stan and compute the posterior mean

baseball_stan_model <- stan_model("baseball.stan")
results <- rstan::sampling(baseball_stan_model,data=list(J=J,y=y,n=n),refresh=0)

# Theta samples are logit scale
theta_samples <- extract(results)$theta
# Get batting averages by inverting with this function
inv_logit <- function(x) {
  exp(x) / (1+exp(x))
}
# and compute the posterior mean for each theta
pm <- colMeans(inv_logit(theta_samples))

# RMSE From Stan posterior means
rmse_partial_pooling <- sqrt((1/J)*sum((pm-val)**2))

print(c(rmse_complete_pooling, rmse_no_pooling, rmse_partial_pooling))
```

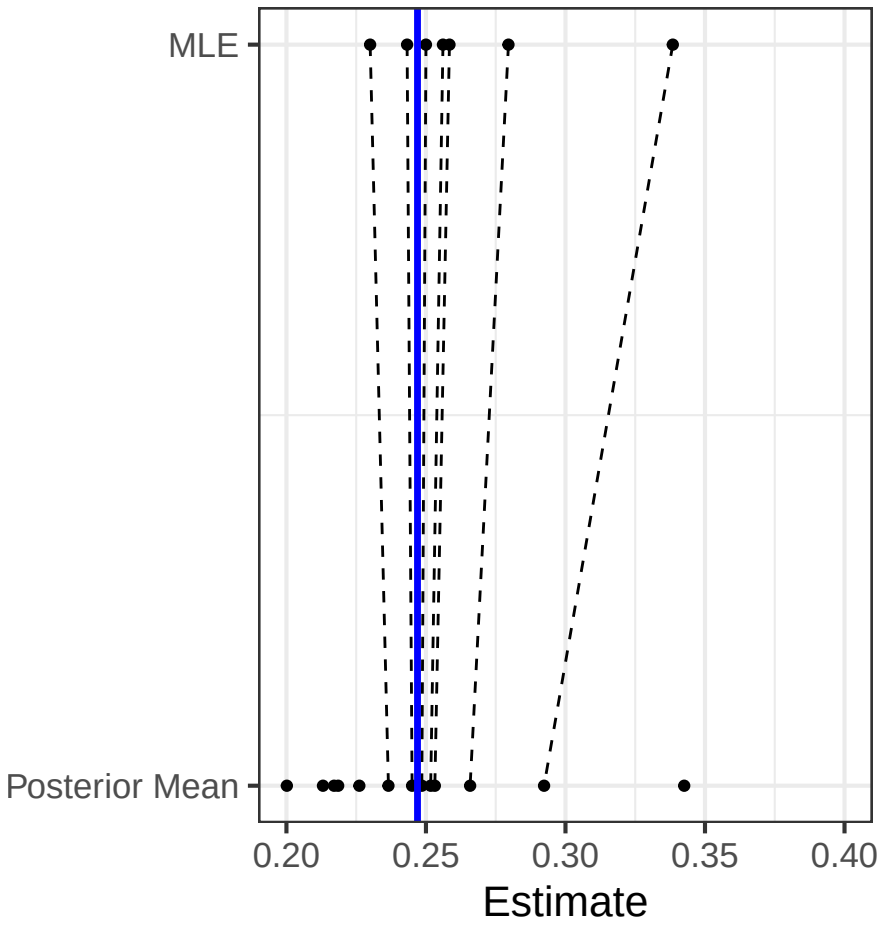
```
## [1] 0.04883162 0.05549984 0.04415029
```

3e. Use the `shrinkage_plot` function provided below to show how the posterior means shrink the empirical batting averages. Pass in `y/n` and the posterior means of p_i as arguments.

```
shrinkage_plot <- function(empirical, posterior_mean,
                           shrink_point=mean(posterior_mean)) {

  tibble(y=empirical, pm=posterior_mean) %>%
    ggplot() +
    geom_segment(aes(x=y, xend=pm, y=1, yend=0), linetype="dashed") +
    geom_point(aes(x=y, y=1)) +
    geom_point(aes(x=pm, y=0)) +
    theme_bw(base_size=16) +
    geom_vline(xintercept=shrink_point, color="blue", size=1.2) +
    ylab("") + xlab("Estimate") +
    xlim(c(0.2, 0.4)) +
    scale_y_continuous(breaks=c(0, 1),
                       labels=c("Posterior Mean", "MLE"),
                       limits=c(0,1))

}
shrinkage_plot(y/n, pm)
```

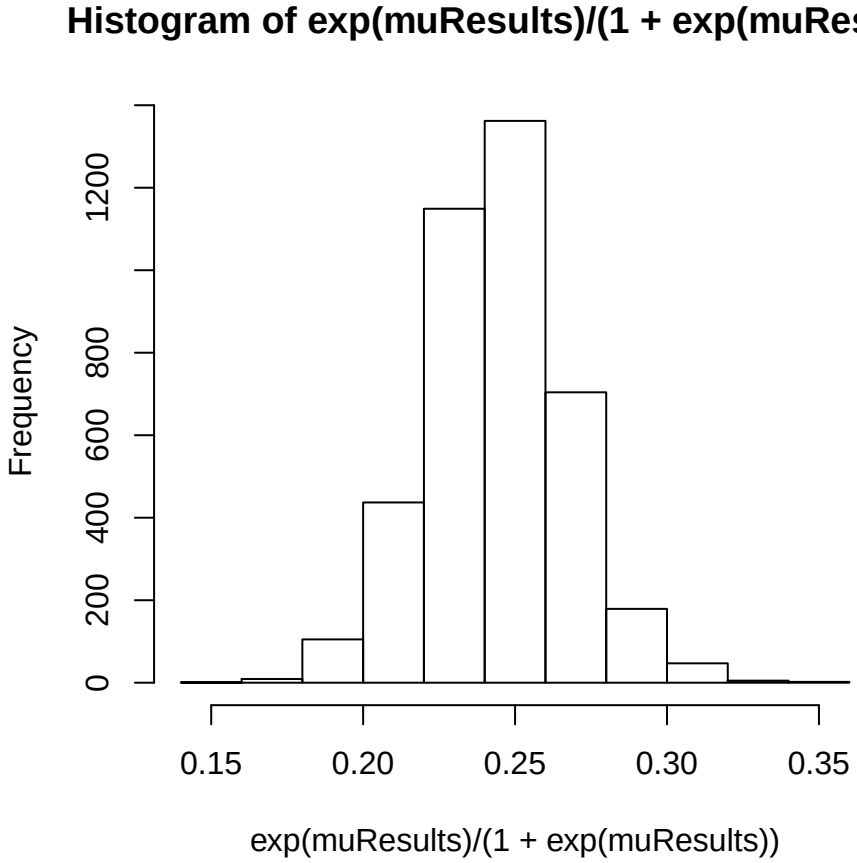
```
## Warning: Removed 6 rows containing missing values (geom_segment).
```

```
## Warning: Removed 6 rows containing missing values (geom_point).
```



3f. Make a histogram of the posterior distribution for the global batting average, $\frac{e^{\mu}}{1+e^{\mu}}$, based on the LAD data. True or false: as the observed at bats for each of the 10 LAD batters $n_i \to \infty$, our estimate of the global batting average converges to a constant. Why or why not?

```
muResults <- extract(results)$mu

hist(exp(muResults) / (1+exp(muResults)))
```



This is False. This is because the batting average converging to a constant would not be representative of the level of skill of each player

Appendix: Code for scraping Dodgers baseball data

http://billpetti.github.io/baseballr/

```
## Install the baseballr package
devtools::install_github("BillPetti/baseballr")
```

```
## Skipping install of 'baseballr' from a github remote, the SHA1 (f4c94205) has not changed since last
##   Use `force = TRUE` to force installation
```

```
library(baseballr)
```

```
## Registered S3 method overwritten by 'quantmod':
##   method            from
##   as.zoo.data.frame zoo
```

```
library(tidyverse)

## Download data from the chosen year
year <- 2019

one_month <- daily_batter_bref(t1 = sprintf("%i-04-01", year), t2 = sprintf("%i-05-01", year))
```

```
## Data courtesy of Baseball-Reference.com. Please consider supporting Baseball-Reference by signing up
```

```
## Warning in lapply(df[, c(2, 5:26)], as.numeric): NAs introduced by coercion
```

```
## Warning in lapply(df[, c(2, 5:26)], as.numeric): NAs introduced by coercion

## Warning in lapply(df[, c(2, 5:26)], as.numeric): NAs introduced by coercion

## Warning in lapply(df[, c(2, 5:26)], as.numeric): NAs introduced by coercion

## Warning in lapply(df[, c(2, 5:26)], as.numeric): NAs introduced by coercion

## Warning in lapply(df[, c(2, 5:26)], as.numeric): NAs introduced by coercion

## Warning in lapply(df[, c(2, 5:26)], as.numeric): NAs introduced by coercion

## Warning in lapply(df[, c(2, 5:26)], as.numeric): NAs introduced by coercion

## Warning in lapply(df[, c(2, 5:26)], as.numeric): NAs introduced by coercion

## Warning in lapply(df[, c(2, 5:26)], as.numeric): NAs introduced by coercion

## Warning in lapply(df[, c(2, 5:26)], as.numeric): NAs introduced by coercion

## Warning in lapply(df[, c(2, 5:26)], as.numeric): NAs introduced by coercion

## Warning in lapply(df[, c(2, 5:26)], as.numeric): NAs introduced by coercion

## Warning in lapply(df[, c(2, 5:26)], as.numeric): NAs introduced by coercion

## Warning in lapply(df[, c(2, 5:26)], as.numeric): NAs introduced by coercion

## Warning in lapply(df[, c(2, 5:26)], as.numeric): NAs introduced by coercion

## Warning in lapply(df[, c(2, 5:26)], as.numeric): NAs introduced by coercion

## Warning in lapply(df[, c(2, 5:26)], as.numeric): NAs introduced by coercion

## Warning in lapply(df[, c(2, 5:26)], as.numeric): NAs introduced by coercion

## Warning in lapply(df[, c(2, 5:26)], as.numeric): NAs introduced by coercion

## Warning in lapply(df[, c(2, 5:26)], as.numeric): NAs introduced by coercion

## Warning in lapply(df[, c(2, 5:26)], as.numeric): NAs introduced by coercion

## Warning in lapply(df[, c(2, 5:26)], as.numeric): NAs introduced by coercion
## Warning: `filter_()` was deprecated in dplyr 0.7.0.
## Please use `filter()` instead.
## See vignette('programming') for more help

## Warning: `arrange_()` was deprecated in dplyr 0.7.0.
## Please use `arrange()` instead.
## See vignette('programming') for more help
```

```r
one_year <- daily_batter_bref(t1 = sprintf("%i-04-01", year), t2 = sprintf("%i-10-01", year))
```

```
## Data courtesy of Baseball-Reference.com. Please consider supporting Baseball-Reference by signing up
```

```
## Warning in lapply(df[, c(2, 5:26)], as.numeric): NAs introduced by coercion
## Warning in lapply(df[, c(2, 5:26)], as.numeric): NAs introduced by coercion

## Warning in lapply(df[, c(2, 5:26)], as.numeric): NAs introduced by coercion

## Warning in lapply(df[, c(2, 5:26)], as.numeric): NAs introduced by coercion

## Warning in lapply(df[, c(2, 5:26)], as.numeric): NAs introduced by coercion

## Warning in lapply(df[, c(2, 5:26)], as.numeric): NAs introduced by coercion

## Warning in lapply(df[, c(2, 5:26)], as.numeric): NAs introduced by coercion

## Warning in lapply(df[, c(2, 5:26)], as.numeric): NAs introduced by coercion

## Warning in lapply(df[, c(2, 5:26)], as.numeric): NAs introduced by coercion

## Warning in lapply(df[, c(2, 5:26)], as.numeric): NAs introduced by coercion

## Warning in lapply(df[, c(2, 5:26)], as.numeric): NAs introduced by coercion

## Warning in lapply(df[, c(2, 5:26)], as.numeric): NAs introduced by coercion

## Warning in lapply(df[, c(2, 5:26)], as.numeric): NAs introduced by coercion

## Warning in lapply(df[, c(2, 5:26)], as.numeric): NAs introduced by coercion

## Warning in lapply(df[, c(2, 5:26)], as.numeric): NAs introduced by coercion

## Warning in lapply(df[, c(2, 5:26)], as.numeric): NAs introduced by coercion

## Warning in lapply(df[, c(2, 5:26)], as.numeric): NAs introduced by coercion

## Warning in lapply(df[, c(2, 5:26)], as.numeric): NAs introduced by coercion

## Warning in lapply(df[, c(2, 5:26)], as.numeric): NAs introduced by coercion

## Warning in lapply(df[, c(2, 5:26)], as.numeric): NAs introduced by coercion

## Warning in lapply(df[, c(2, 5:26)], as.numeric): NAs introduced by coercion

## Warning in lapply(df[, c(2, 5:26)], as.numeric): NAs introduced by coercion

## Warning in lapply(df[, c(2, 5:26)], as.numeric): NAs introduced by coercion
```

```r
## filter to only include players who hat at least 10 at bats in the first month
one_month <- one_month %>% filter(AB > 10)
one_year <- one_year %>% filter(Name %in% one_month$Name)

one_month <- one_month %>% arrange(Name)
one_year <- one_year %>% arrange(Name)

## Look at only the Dodgers
LAD <- one_year %>% filter(Team == "Los Angeles" & Level == "MLB-NL") %>% .$Name
```

```r
lad_month <- one_month %>% filter(Name %in% LAD)
lad_year <- one_year %>% filter(Name %in% LAD)

write_csv(tibble(name=lad_month$Name,
                 y=lad_month$H,
                 n=lad_month$AB,
                 val=lad_year$BA),
          file ="lad.csv")
```