HW 6 Final

Austin Miles, Diane Phan

12/6/2019

Problem 1. Logistic regression for toxicity data

Logistic regression for pesticide toxicity data (part 2). As a reminder from homework 5, an environmental agency is testing the effects of a pesticide that can cause acute poisoning in bees. In the last homework assignment, 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.

a. Complete the specification for the log posterior for the data `x` and `y` by filling in the missing pieces of the function below. Before proceeding verify that your code works by checking that `log_posterior(-1, 1)` evaluates to `-11.49`.

```
## 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
  exp_reg <- exp(alpha+beta*x)

    prob <- prod(((exp_reg/(1+exp_reg))^y)*(1-(exp_reg/(1+exp_reg)))^(1-y))
  if(any(prob == 0) | any(prob == 1))
```

```
    -Inf ## log likelihood is -Inf is prob=0 or 1
  else
    log(prob)
}

test <- c(-1,1)
log_posterior(test)
```

```
## [1] -11.49705
```

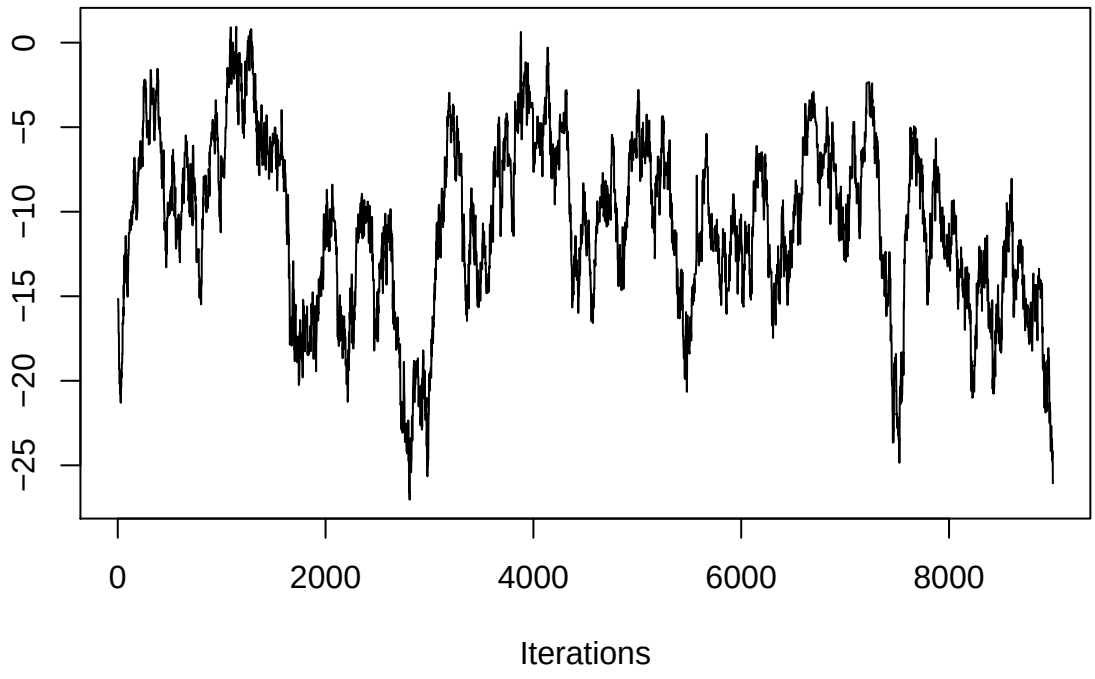
b. You will now complete the Metropolis-Hastings sampler by filling in the missing pieces of the algorithm below. `theta_0` is a vector of length 2, with the first argument as the initial alpha value and the second argument as the initial beta value. As your proposal, use $J(\theta * |\theta_t) \sim Normal(\theta_t, \Sigma)$. 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.

```
################################################
## 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, mean = theta_t, sigma = 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 <- min(0, log_posterior(theta_p) - log_posterior(theta_t))

        ## Update theta_t based on whether the proposal is accepted or not
        theta_t <- if (log(runif(1,0,1)) < logr){
                theta_p
        }
```

```
        else {
          theta_t
        }

        ## 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(1,2), 1000, 10000)

coda::effectiveSize(as.mcmc(samples))
```

```
##    alpha     beta
## 24.03390 24.72531
```

```
coda::traceplot(as.mcmc(samples[,1]))
```



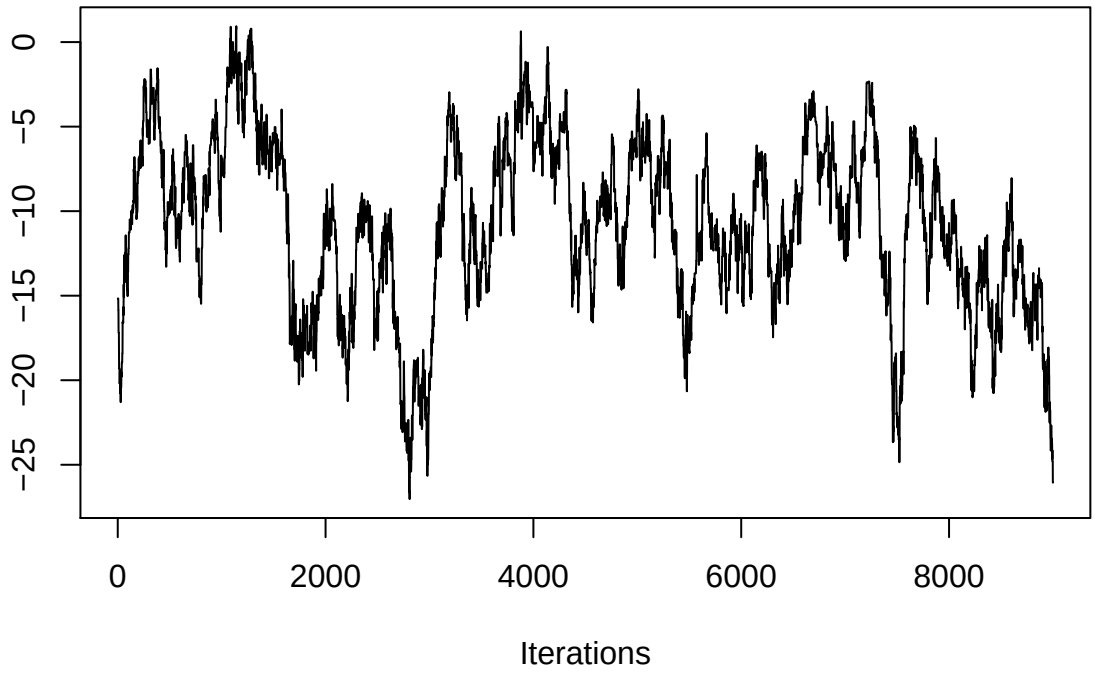
c. Report the effective sample size for the alpha parameter using the `coda::effectiveSize` function. Make a traceplot of alpha. If `alpha_samples` were the name of the samples of the alpha parameter, then you can plot the traceplot using `coda::traceplot(as.mcmc(alpha_samples))`. Improve upon

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 paraemters. 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`.

```
coda::effectiveSize(as.mcmc(samples))
```

```
##    alpha     beta
## 24.03390 24.72531
```

```
coda::traceplot(as.mcmc(samples[,1]))
```



Problem 2. 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: 10 x 4
##    name                 y     n   val
##    <chr>            <dbl> <dbl> <dbl>
##  1 Austin Barnes       18    86 0.206
##  2 Chase Utley         22   106 0.208
##  3 Chris Taylor        52   210 0.255
##  4 Cody Bellinger      48   199 0.265
##  5 Corey Seager        27    94 0.287
##  6 Enrique Hernandez   26   122 0.257
##  7 Joc Pederson        32   129 0.249
##  8 Matt Kemp           57   163 0.292
##  9 Yasiel Puig         36   137 0.274
## 10 Yasmani Grandal     39   155 0.24
```

```
## 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
```

a. 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?

```
sd(theta_mle)
```

```
## [1] 0.04264024
```

```
sd(val)
```

```
## [1] 0.02925007
```

The end of the year batting average has a smaller variance because there are more samples and players tend to shift toward the mean over time.

b. 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?

```r
RMSE = function(m, o){
  sqrt(mean((m - o)^2))
}
#No Pooling
RMSE(val, theta_mle)
```

```
## [1] 0.02479514
```

```r
p_comp <- sum(y)/sum(n)

#Complete Pooling
RMSE(val, p_comp)
```

```
## [1] 0.02779054
```

No pooling gives us a better estimate of the end-of-year batting averages since the RMSE is lower.

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:

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

c. 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.

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 is squared, then it will obviously become very large and the true skill differences between players is assumed to be large, thus our estimates will be close to the "no pooling" estimator. As τ goes to infinity, the posterior mean estiate of pi in this model will approach the no pooling estimator. As τ goes to zero, the posterior mean estimate of pi will approach the complete pooling estimator.

d. 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_lpmf(y | n, theta)`.
   - Define a vector `p_i` which is the inferred batting average (p_i) for each player. It is the inverse logit transformation of the thetas. Add to the "transformed parameters" block `vector[J] p_i`

and set it to the inverse-logit of `theta`. You can simply use the `inv_logit` function.

- 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 `p_i`. Report the RMSE for hiearchical estimator. How does this compare to the RMSE of the complete pooling and no pooling estimators? Which estimator had the lowest error?

```
baseball_stan <- rstan::stan_model("baseball.stan")
stan_fit <- sampling(baseball_stan ,data = list(J = J, y = y, n = n))
```

```
##
## SAMPLING FOR MODEL 'baseball' NOW (CHAIN 1).
## Chain 1:
## Chain 1: Gradient evaluation took 7.3e-05 seconds
## Chain 1: 1000 transitions using 10 leapfrog steps per transition would take 0.73 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: 0.127942 seconds (Warm-up)
## Chain 1:                0.092381 seconds (Sampling)
## Chain 1:                0.220323 seconds (Total)
## Chain 1:
##
## SAMPLING FOR MODEL 'baseball' NOW (CHAIN 2).
## Chain 2:
## Chain 2: Gradient evaluation took 1.1e-05 seconds
## Chain 2: 1000 transitions using 10 leapfrog steps per transition would take 0.11 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: 0.101927 seconds (Warm-up)
## Chain 2:                0.088689 seconds (Sampling)
## Chain 2:                0.190616 seconds (Total)
## Chain 2:
##
## SAMPLING FOR MODEL 'baseball' NOW (CHAIN 3).
## Chain 3:
## Chain 3: Gradient evaluation took 1.1e-05 seconds
## Chain 3: 1000 transitions using 10 leapfrog steps per transition would take 0.11 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: 0.112988 seconds (Warm-up)
## Chain 3:                0.148917 seconds (Sampling)
## Chain 3:                0.261905 seconds (Total)
## Chain 3:
##
## SAMPLING FOR MODEL 'baseball' NOW (CHAIN 4).
## Chain 4:
## Chain 4: Gradient evaluation took 1.2e-05 seconds
## Chain 4: 1000 transitions using 10 leapfrog steps per transition would take 0.12 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: 0.107279 seconds (Warm-up)
## Chain 4:                0.083462 seconds (Sampling)
## Chain 4:                0.190741 seconds (Total)
## Chain 4:
```

```
samples <- rstan::extract(stan_fit)
```

```
colMeans(samples$p_i)
```

```
##  [1] 0.2431866 0.2416841 0.2518597 0.2493091 0.2628147 0.2415701 0.2524255
##  [8] 0.2880048 0.2568486 0.2530429
```

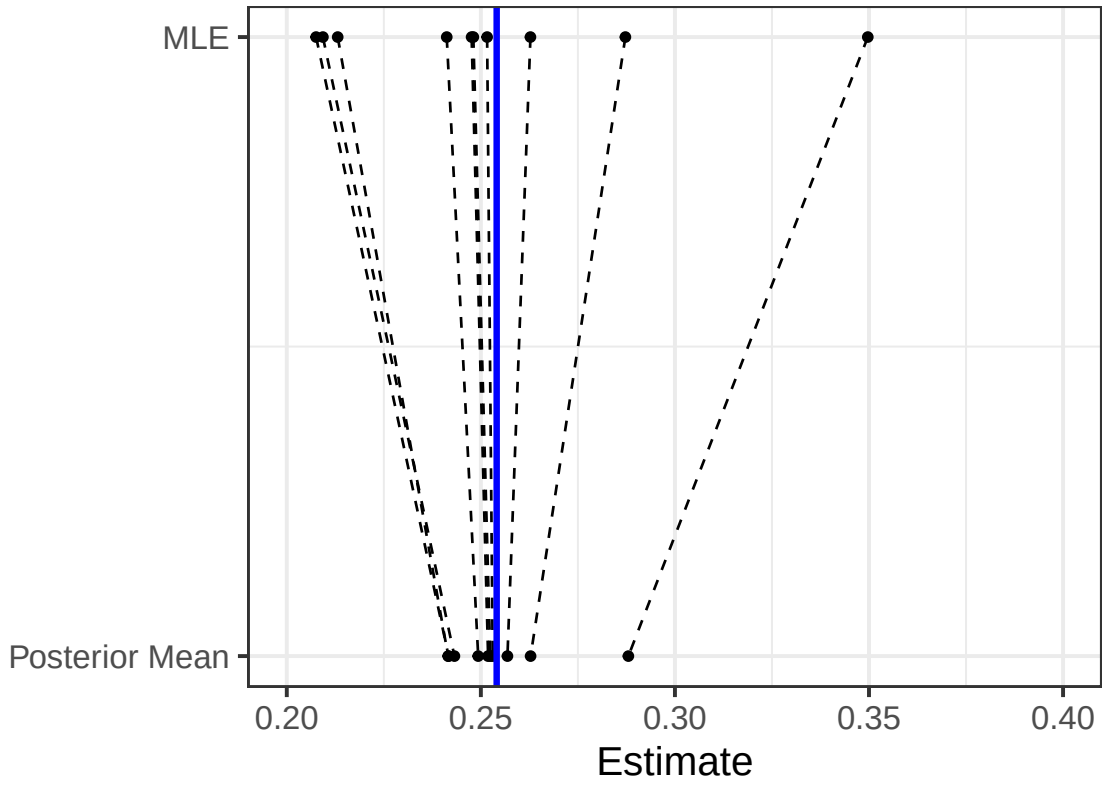
```
RMSE(samples$p_i, samples$theta)
```

```
## [1] 1.341756
```

The RMSE for the hiearchical estimator is much higher than the complete and no pooling estimators. The no pooling estimator had the lowest error.

e. 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.

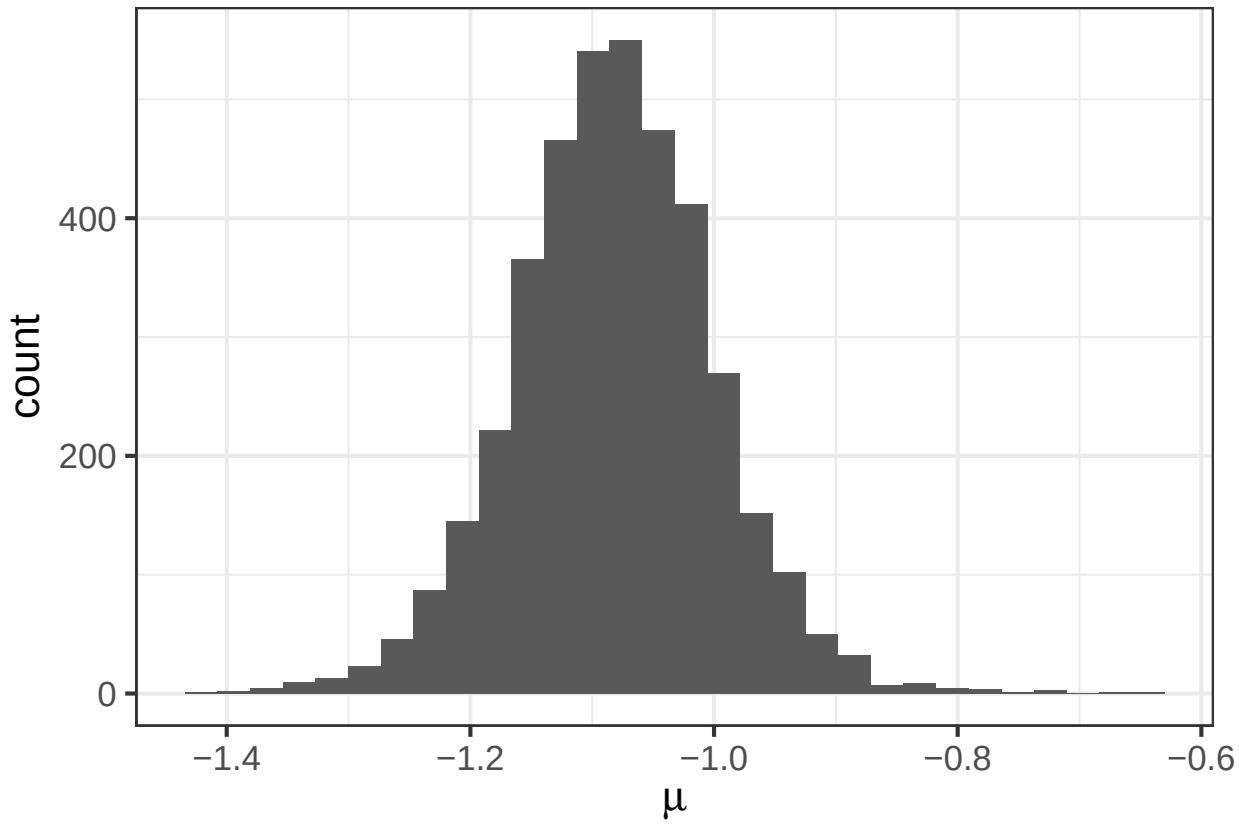
```
shrinkage_plot <- function(empirical, posterior_mean,
                           shrink_point=mean(posterior_mean)) {

  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(theta_mle, colMeans(samples$p_i))
```



f. 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?

```
mu_post = tibble(mu = samples$mu)
ggplot(mu_post, aes(mu)) +
geom_histogram() +
theme_bw(base_size = 16) +
labs(x = expression(mu))
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



True because the more samples we get, the less weight the prior has for the posterior.