HW4b: Stochastic Gradient Descent and Lipschitz Extensions

Lipika Ramaswamy
April 30th, 2019

https://github.com/lipikaramaswamy/cs208_lr/tree/master/homework/HW4b
(https://github.com/lipikaramaswamy/cs208_lr/tree/master/homework/HW4b)

Problem 1

1a.

$G = \mathbb{R}^n$
$x \sim x'$ if they differ on one row
$H = [a,b]^n, a Mean function: $f(x) = \frac{1}{n}\sum_{i=1}^{n} x_i$

1. **Global Sensitivity** of f, GS_f

   Let the neighboring datasets be $x = [0, \ldots, 0, 0]$ and $x' = [0, \ldots, 0, \infty]$.
   So, $f(x) = 0$ and $f(x') = \infty$. Thus, $GS_f = \max\limits_{x \sim x'} |f(x') - f(x)| = \infty$.

2. **Minimum Local Sensitivity** of f, $\min\limits_{x \in G} LS_f$

   By definition, $\min\limits_{x \in G} LS_f = \min\limits_{x \in G} \max\limits_{x' : x \sim x'} |f(x') - f(x)|$, i.e. we want to calculate the sensitivity for the worst case/furthest neighbor for every dataset, $x \in G$, and determine the best case over all datasets $x \in G$. Since the space $\mathbb{R}^n$ is unbounded, the worst case/furthest neighbor for all datasets in G will be the same, or $\max\limits_{x' : x \sim x'} |f(x') - f(x)| = \infty, \forall x \in G$, as for any dataset, there is always a neighbor with one element that could be flipped to an infinite value.
   Thus, $\min\limits_{x \in G} LS_f = \infty$.

3. **Restricted Sensitivity** of f, $RS_f^H = \max\limits_{x, x' \in H} \dfrac{|f(x) - f(x')|}{d(x, x')}$

   To calculate the restricted sensitivity, it suffices to consider datasets at distance 1 from each other, i.e. differing on one row. Consider worst case neighboring size n datasets $x = [a, \ldots, a]$ and $x' = [a, \ldots, a, b]$. Here, $f(x) = \dfrac{na}{n}$ and $f(x') = \dfrac{(n-1)a + b}{n}$. The restricted sensitivity on this set H will thus be, $RS_f^H = \frac{b-a}{n}$.

   Lipschitz Extension, f': Here, we are looking for a function f' such that:

   - f' agrees with f on H
   - $GS_{f'} = RS_f^H$
     Consider the following function $f'(x) = \frac{1}{n}\sum_{i=1}^{n} [x_i]_a^b$, i.e. the winsorized mean, with global sensitivity $GS_{f'} = \dfrac{b-a}{n} = RS_f^H$ and $\forall x \in [a, b], f(x) = f(x')$. This serves as a Lipschitz extension for the mean.

1b.

$G = \mathbb{R}^n$
$x \sim x'$ if they differ on one row
$H = [a,b]^n, a Median function: $f(x) = median(x_1, \dots, x_n)$

1. **Global Sensitivity** of f, GS_f

   Let the neighboring datasets be $x = [0, \dots, 0, 0, \infty, \dots, \infty]$ (the number of elements that are 0 is one more than the number of elements that are ∞) and $x' = [0, \dots, 0, \infty, \infty, \dots, \infty]$ (flip one 0 so that the number of elements that are ∞ is one more than the number of elements that are 0).
   So, $f(x) = 0$ and $f(x') = \infty$. Thus, $GS_f = \max\limits_{x \sim x'} |f(x') - f(x)| = \infty$.

2. **Minimum Local Sensitivity** of f, $\min\limits_{x \in G} LS_f$

   By definition, $\min\limits_{x \in G} LS_f = \min\limits_{x \in G} \max\limits_{x' : x \sim x'} |f(x') - f(x)|$, i.e. we want to calculate the sensitivity for the worst case/furthest neighbor for every dataset, $x \in G$, and then determine the best case over all datasets $x \in G$. The best case dataset $x \in G$ will be of the form $x = [0, \dots, 0]$ whose worst neighbor is $x' = [0, \dots, \infty]$. In this case, $f(x) = 0$ and $f(x') = 0$, so the minimum local sensitivity, $\min\limits_{x \in G} LS_f = 0$.

3. **Restricted Sensitivity** of f, $RS_f^H = \max\limits_{x, x' \in H} \dfrac{|f(x) - f(x')|}{d(x, x')}$

   To calculate the restricted sensitivity, it suffices to consider datasets at distance 1 from each other, i.e. differing on one row. Consider worst case neighboring datasets $x = [a, \dots, a, a, b, \dots, b]$ and $x' = [a, \dots, a, b, b, \dots, b]$ (flipping the middle element from a to b). Here, $f(x) = a$ and $f(x') = b$. The restricted sensitivity on this set H will thus be, $RS_f^H = b - a$.

1c.

G = set of undirected graphs with no self loops on the vertex set $\{1, \ldots, n\}$
x and x' are identical except for the neighborhood of a single vertex (node privacy)
H = set of graphs in G where every vertex has degree at most d for $2 \leq d \leq n-1$
function: $f(x)$ = number of isolated (degree 0) vertices in x

By definition of node privacy, two graphs are neighbors if one can be ontained from the other by deleting a node and its adjacent edges.

1. **Global Sensitivity** of f, GS_f

   Let x be a graph with $n-1$ isolated vertices, with $f(x) = n-1$. Let it's node neighbor x' (with n vertices) be the graph with a vertex added that now connects to each of the other $n-1$ vertices, ensuring that there are no isolated vertices. So, $f(x') = 0$. This corresponds to the worst case that would arise in this vertex set. Thus, $GS_f = \max_{x \sim x'} |f(x') - f(x)| = n-1$.

2. **Minimum Local Sensitivity** of f, $\min_{x \in G} LS_f$

   By definition, $\min_{x \in G} LS_f = \min_{x \in G} \max_{x' : x \sim x'} |f(x') - f(x)|$, i.e. we want to calculate the sensitivity for the worst case/furthest neighbor for every graph, $x \in G$, and then determine the best case over all graphs $x \in G$. The best case graph $x \in G$ will be one that is fully connected, with $f(x) = 0$, whose worst case neighbor is a graph x' with one node added that's not connected to any other vertex. So, $f(x') = 1$. Thus, the minimum local sensitivity, $\min_{x \in G} LS_f = 1$.

3. **Restricted Sensitivity** of f, $RS_f^H = \max_{x, x' \in H} \frac{|f(x) - f(x')|}{d(x, x')}$

   To calculate the restricted sensitivity, it suffices to consider graphs at distance 1 from each other, i.e. differing on one row. Consider worst case neighboring graphs, x with d vertices, none of which are connected, and x' with a vertex added that connects to each of the d vertices by an edge (this holds as the upper bound on d is $n-1$. Thus, $f(x) = 0$ and $f(x') = d$. The restricted sensitivity on this set H will thus be, $RS_f^H = d$.

Problem 2

SGD under local DP

To release an estimated local differentially private logistic regression, noise is added to each individual's gradient in a given batch and then the average gradient is estimated for the batch, which provides the direction of update for the coefficients, β_0 and β_1. Specifically, given the following,

- a dataset of size N
- batch size $L<
- learning rate η
- clipping parameter C
- noise variance τ
- number of batches, T
- privacy parameters, ϵ and δ

After an initial point $\overset{\wedge}{\beta_0} = (0,0)$ for the coefficients is chosen and the dataset is shuffled, we iterate through $t = 1$ to T with the following:

1. Select the first (and then successive) L rows as the batch
2. Add noise to the gradient of each individual, i's, negative log likelihood, l, after clipping
$$\hat{g}_{t,i} = [\nabla l(\hat{\theta}_{t-1}|x_i, y_i)]_{-C}^{C} + \mathcal{N}(0, \tau^2 I)$$
3. Get an estimate of the gradient for the batch,
$$\hat{g}_t = \frac{1}{L}\sum_{t=1}^{L} \hat{g}_{t,i}$$
4. Update the coefficients, β for this step: $\hat{\beta}_t = \hat{\beta}_{t-1} - \eta \cdot \hat{g}_t$.

Here, we choose batch size, $L = \sqrt{N}$ and number of steps, $T = \sqrt{N}$ so that we use each individual's information only once in the computation of the gradient, which ensures that the algorithm meets the definition of differential privacy. $\left(\lceil T \cdot B/n \rceil = \frac{\sqrt{n}\cdot\sqrt{n}}{n} = 1\right)$.

Also assume we have a total privacy budget of ε for the release. Based on this, the formula for the scale of the Gaussian noise added is given by:

$$\tau^2 = \left(\frac{C}{\varepsilon/2}\right)^2 \left(T \cdot \frac{L}{N} \cdot \log\left(\frac{1}{\delta}\right)\right) = \left(\frac{C}{\varepsilon/2}\right)^2 \left(\sqrt{N} \cdot \frac{\sqrt{N}}{N} \cdot \log\left(\frac{1}{\delta}\right)\right) = \left(\frac{C}{\varepsilon/2}\right)^2 \cdot \log\left(\frac{1}{\delta}\right)$$

This algorithm is implemented below for ϵ ranging from 0.1 to 5.

Read in data and select columns

```
In [1]: library(ggplot2)
        require(gridExtra)
        library("foreign")
        PUMSfull <- read.csv(file="../../data/MaPUMS5full.csv")
        PUMSuse<-PUMSfull[c("married","educ")]
```

Non-privacy preserving estimates: logistic regression

In [2]:
```
output <- glm(married ~ educ, family="binomial", data=PUMSuse)
true.beta0 <- output$coefficients[1]
true.beta1 <- output$coefficients[2]
true.preds <- output$fitted.values
cat('The true estimates for coefficients for the logistic regression are
\n Intercept (beta0): ', round(true.beta0,3),
   '\n Education (beta1)', round(true.beta1,3))
```

```
The true estimates for coefficients for the logistic regression are
 Intercept (beta0):  -0.62
 Education (beta1) 0.078
```

Functions for the LDP release

In [3]:
```
## LOSS FUNCTION:
## Likelihood function to calculate the negative log likelihood
calcllik<-function(betas,data){
    y<-data[,1]
    x<-data[,2]
    # calculate probability based on logit
    pi<- 1/(1+exp(-betas[1] - betas[2]*x))

    # to avoid exploding log likelihoods
    if (pi == 1){
        pi <- 0.999999
    }
    else if (pi == 0){
        pi <- 0.000001
    }

    llik<-y * log(pi) + (1-y) * log(1-pi)
    return(-llik)
}
```

In [4]:
```
## HELPER FUNCTION:
## Bound/Censor/Clip a variable to a range
clip <- function(x, lower, upper){
    x.clipped <- x
    x.clipped[x.clipped<lower] <- lower
    x.clipped[x.clipped>upper] <- upper
    return(x.clipped)
}
```

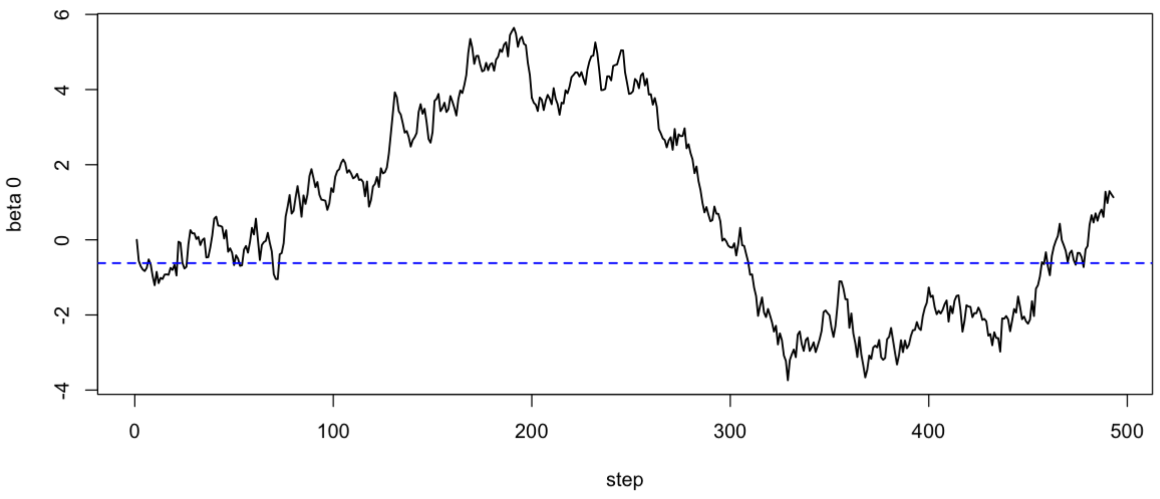
```
In [5]: ## LOCAL GRADIENT FUNCTION:
        ## Calculate the gradient for a given row at a point in the parameter sp
        ace and add noise using first principles
        calcLocalGradient <- function(row,                  #row of dataset
                                      C,                     #clipping parameter
                                      beta,                  #coefficients
                                      fun,                   #loss function
                                      L,                     #batch size
                                      epsilon, delta,        #privacy parameters
                                      N                      # size of dataset
                                     ){
            dx <- 0.0001

            # evaluate log likelihood at original point, and along
            out1 <- eval(fun(betas=beta, data=row))
            out2 <- eval(fun(betas=beta + c(0,dx), data=row))
            out3 <- eval(fun(betas=beta + c(dx,0), data=row))

            Del.1 <- (out3 - out1)/dx
            Del.1 <- clip(Del.1, lower=(-C), upper=C)

            Del.2 <- (out2 - out1)/dx
            Del.2 <- clip(Del.2, lower = (-C), upper = C)

            result <- c(Del.1,Del.2) + rnorm(n=2, mean=0, sd= ((2*C  / epsilon)
        * sqrt(steps * (L/N) * log(1/delta)) ) )

            return(result)
        }
```

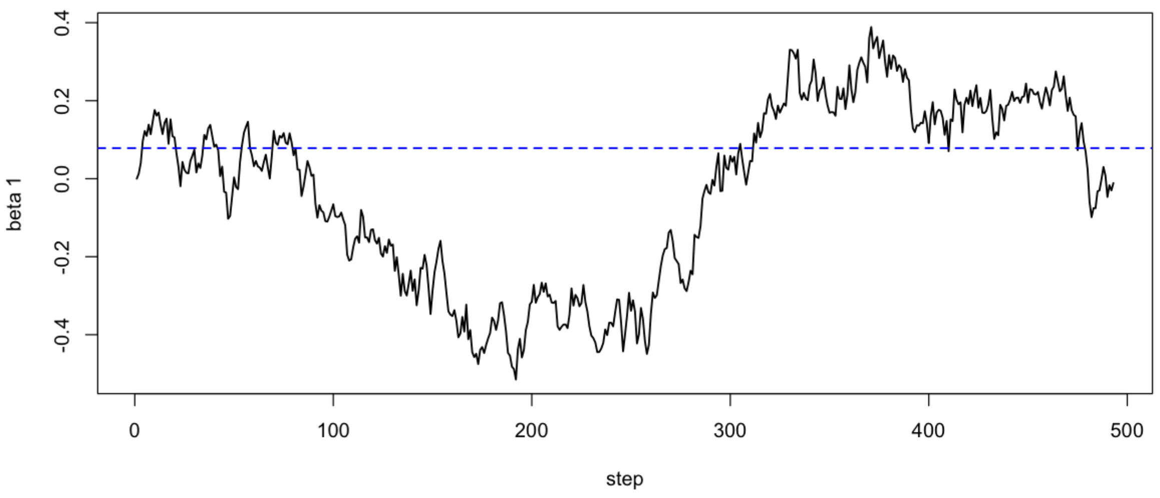
Set parameters for SGD

```
In [7]: ## SET PARAMETERS
        N <- nrow(PUMSuse)                    # size of dataset
        L <- round(sqrt(nrow(PUMSuse)))       # batch size
        steps <- L                            # number of steps, equivalent to T i
        n notation above
        epsilon <- 1                         # privacy parameter
        delta <- 1e-6                         # privacy parameter
        C <- 10                               # Interval to clip over
```

In [7]:
```
nu <- c(0.09,0.01)                # Learning rate for beta0 and beta1

## Shuffle the data
# Note: dont want to sample with replacement, or choose the same person
 multiple times
# as then there would be no bound on the sensitivity

index <- sample(1:nrow(PUMSuse), replace = FALSE)
PUMSuse <- PUMSuse[index,]

# Starting parameters
beta <- c(0,0)
# matrix to store the progression of betas
history <- matrix(NA, nrow=steps+1, ncol=2)
history[1,] <- beta

## sqrt n steps of batch size sqrt n

# Iterate one step of SGD
for(i in 1:steps){

    startB <- ((i-1)*L+1)
    if(i<L){
        stopB <- i*L
    }else{
        stopB <- nrow(PUMSuse)
    }

    B <- PUMSuse[startB:stopB, ] # batch B

    # matrix to store gradients for each batch with respect to beta0 and
beta1
    storeGradients <- matrix(NA, nrow = nrow(B), ncol = ncol(B))

    for(row in 1:nrow(B)){

        rowGradient <- calcLocalGradient(B[row,], C, beta, fun=calcllik,
L, epsilon, delta, N)
        storeGradients[row,]<-rowGradient
    }

    Del <- colSums(storeGradients) / L
    beta <- beta - Del * nu
    history[i+1,] <- beta

}
options(repr.plot.width=10, repr.plot.height=10)

par(mfcol=c(2,1))

all.ylim<-c( min(c(history[,1],output$coef[1] )), max(c(history[,1],outp
ut$coef[1] )))
plot(history[,1], type="l", ylim=all.ylim, ylab="beta 0", xlab="step", l
wd=1.5)
abline(h=output$coef[1], lty=2, col="blue", lwd=1.5)
```

```
all.ylim<-c( min(c(history[,2],output$coef[2] )), max(c(history[,2],outp
ut$coef[2] )))
plot(history[,2], type="l", ylim=all.ylim, ylab="beta 1", xlab="step", l
wd=1.5)
abline(h=output$coef[2], lty=2, col="blue", lwd=1.5)
```





I used the above plots to determine the learning rates that work best for $\varepsilon = 1$ (I ran through the algorithm several times to ensure that plots always looked close to the above). A learning rate of 0.09 for β_0 and 0.01 for β_1 provided results that were closest to the true values of the coefficients. I also tried these learning rates with other extreme values of ε and found that it was converging to the non-private values.

Simulations:

10 simulations were perfomed for each value of epsilon, $\epsilon = [0.1, 0.3, 0.5, 0.7, 0.9, 2, 3, 4, 5]$

In [48]:
```
# different values of epsilon to explore
epsilon.options <- c(seq(from = 0.1, to = 1, by = 0.2), seq(from = 2, to
= 5, by = 1))

# total number of iterations
niter <- 10

# matrix to store betas from each run
LDPbetas <- matrix(NA, nrow=length(epsilon.options)*niter , ncol=3)

counter = 1
for(iter in 1:niter){

    # loop through all the possible epsilons
    for(epsilon in epsilon.options){
        # shuffle data
        index <- sample(1:nrow(PUMSuse), replace = FALSE)
        PUMSuse <- PUMSuse[index,]
        # Starting parameters
        beta <- c(0,0)
        # matrix to store the progression of betas in each step
        history <- matrix(NA, nrow=steps+1, ncol=2)
        history[1,] <- beta

        # Iterate one step of SGD
        for(i in 1:steps){
            # Get batch B
            startB <- ((i-1)*L+1)
            if(i<L){
                stopB <- i*L
            }else{
                stopB <- nrow(PUMSuse)
            }

            B <- PUMSuse[startB:stopB,] # batch B

            # matrix to store gradients for each batch with respect to b
eta0 and beta1
            storeGradients <- matrix(NA, nrow = nrow(B), ncol = ncol(B))

            for(row in 1:nrow(B)){
                rowGradient <- calcLocalGradient(B[row,], C, beta, fun=c
alcllik, L, epsilon, delta, N)
                storeGradients[row,]<-rowGradient
            }

            Del <- colSums(storeGradients) / L
            beta <- beta - Del * nu
            history[i+1,] <- beta
        }

        LDPbetas[counter,] <- c(epsilon, history[steps,])
        counter = counter + 1
    }
}
```

In [9]:
```
# store results after running loops:
# write.csv(LDPbetas, 'ldpbetas.csv')

# load results on new runs:
LDPbetas <- read.csv('ldpbetas.csv')
```

In [10]:
```
# make a dataframe object to easily access columns
dfbetas <- as.data.frame(LDPbetas)
keeps <- c("V1", "V2", "V3")
dfbetas <- dfbetas[keeps]
colnames(dfbetas) <- c('epsilon', 'beta0', 'beta1')
```

Functions to enable prediction using a model, calculation of accuracy and RMSE

In [12]:
```
## PREDICTION FUNCTION
# takes a vector of beta0 and beta1 along with the data as input
# returns predictions of 0 or 1 based on a threshold of 0.5
predictMarried <- function(betas, data, thresh = 0.5){
    y<-data[,1]
    x<-data[,2]
    pi<- 1/(1+exp(-betas[1] - betas[2]*x))
    preds <- ifelse(pi >= thresh, 1, 0)
    return(preds)
}
```

In [13]:
```
## ACCURACY FUNCTION
calcAcc <- function(true, pred){
    store.sum <- sum(true == pred)
    acc <- store.sum/length(true)
    return(acc)
}
```

In [14]:
```
## RMSE FUNCTION
RMSE <- function(est.betas, true.beta){
    diff <- est.betas - true.beta
    sum2 <- (sum(diff))^2
    N<-length(est.betas)
    rmse <- sqrt(sum2/N)
    return(rmse)
}
```

Calculation of Accuracy

In [15]:
```
# Calculate classification error for each epsilon each iteration
storeError <- matrix(NA, nrow = length(epsilon.options), ncol = niter) #
rows for each epsilon, cols for each iteration

# loop through each epsilon
counter = 1
for(e in epsilon.options){
    subdf <- dfbetas[which(dfbetas$epsilon == round(e, 2)),]
    rownames(subdf) <- 1:nrow(subdf)

    # now loop through each iteration
    for(iter in 1:10){
        predz <- predictMarried(as.double(subdf[iter,2:3]), PUMSuse)
        acc <- calcAcc(PUMSuse$married, predz)
        storeError[counter, iter] <- 1 - acc
    }
    counter = counter + 1
    copydf <- subdf
}
avgError <- rowMeans(storeError)
```

In [16]:
```
# Calculate baseline classification error
baseline.error <- 1 - calcAcc(PUMSuse$married, true.preds >= 0.5)
cat('The classification error with the non-private logistic regression m
odel is ', baseline.error)
```

```
The classification error with the non-private logistic regression model
is  0.4402804
```

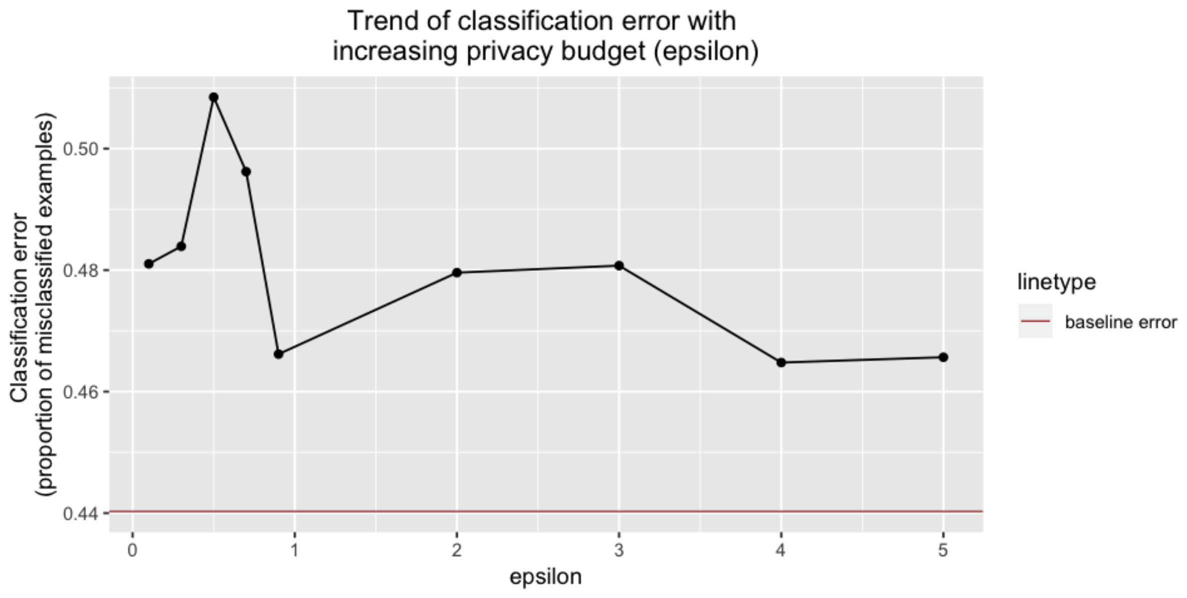
Calculation of RMSE

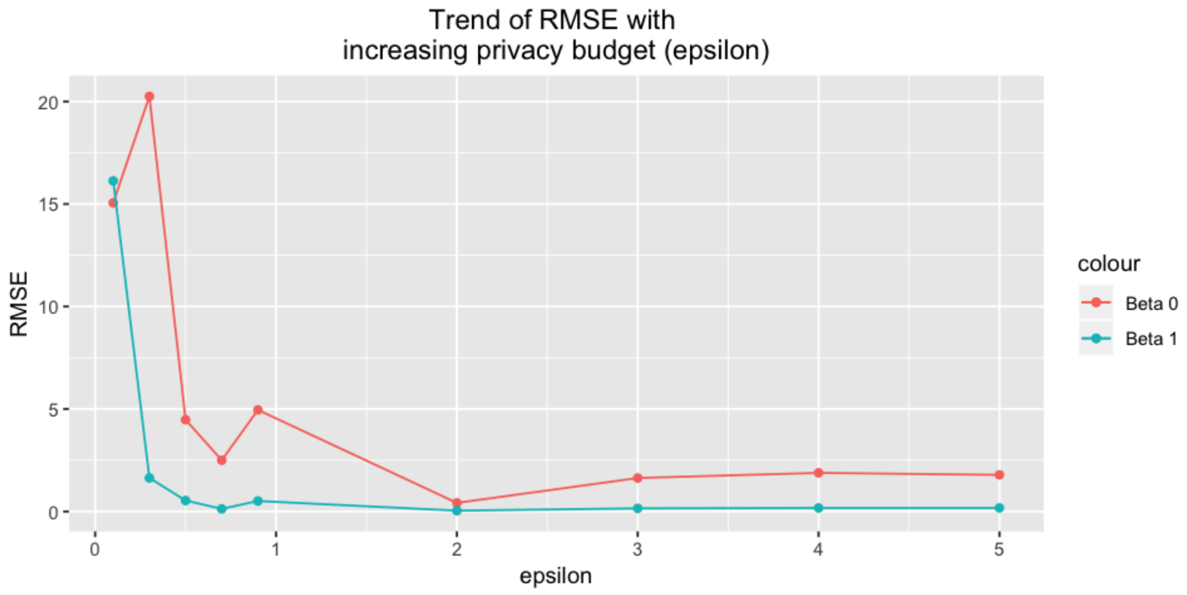
In [17]:
```
# Calculate RMSE for each beta

beta0RMSE <- matrix(NA, nrow = length(epsilon.options), ncol = 1) # each
row corresponds to an epsilon val
beta1RMSE <- matrix(NA, nrow = length(epsilon.options), ncol = 1) # each
row corresponds to an epsilon val

counter = 1
for(epsilon in epsilon.options){
    subsetdf = dfbetas[dfbetas$epsilon == round(epsilon,2), ]

    # work on beta 0
    store0 <- RMSE(subsetdf$beta0, true.beta0)
    beta0RMSE[counter,] <- store0

    # work on beta 1
    store1 <- RMSE(subsetdf$beta1, true.beta1)
    beta1RMSE[counter,] <- store1

    counter = counter + 1
}
```

Visualizations

In [30]:
```
# make dataframe for plotting accuracy
df.plot <- data.frame(epsilon.options, beta0RMSE, beta1RMSE, avgError)

# setup plot for accuracy
p1 <- ggplot(data=df.plot, aes(x=epsilon.options, y=avgError, group=1))
+
    geom_line()+
    geom_point()+
    labs(x = "epsilon", y = 'Classification error \n(proportion of miscl
assified examples)', title =
     'Trend of classification error with \nincreasing privacy budget (ep
silon)') +
    geom_hline(aes(yintercept = baseline.error, linetype='baseline erro
r'), colour="#990000", size = 0.3 )+


    theme(plot.title = element_text(hjust = 0.5))

p2 <- ggplot(data=df.plot) +
    geom_line(aes(x=epsilon.options, y=beta0RMSE, color = 'Beta 0'))+
    geom_point(aes(x=epsilon.options, y=beta0RMSE, color = 'Beta 0'))+
    geom_line(aes(x=epsilon.options, y=beta1RMSE, color = 'Beta 1'))+
    geom_point(aes(x=epsilon.options, y=beta1RMSE, color = 'Beta 1'))+
    labs(x = "epsilon", y = 'RMSE', title =
         'Trend of root mean square error (RMSE) of coefficients with \n
increasing privacy budget (epsilon)') +
    theme(plot.title = element_text(hjust = 0.5))
```

In [31]:
```
options(repr.plot.width=8, repr.plot.height= 8)
grid.arrange(p1, p2, nrow=2)
```





The plots above demonstrate the effect of increasing the privacy budget, epsilon, holding δ constant on logistic regression under the local DP model.

- Classification error:
  The baseline classification error (i.e. that of the non-privacy preserving logistic regression model) is 0.44. When noise is added under the local DP model during stochastic gradient descent, the estimates result in higher classification error for all values of epsilon. Intuitively, we would expect that for lower values of epilon, as more noise is added to each observation's likelihood, the direction in which the SGD algorithm should direct betas is noisier, leading to worse convergence. The simulations show that the classification error is higher for values of epsilon lower than 0.9, with a high at $\epsilon = 0.5$. At $\epsilon = 0.9$, the classification error hits a low of 0.466. For higher values of epsilon, the classification accuracy stabilizes around 0.465. Note that for none of the epsilon values experimented with does the classification error fall to the baseline error.
- RMSE of coefficients:
  Similar to the argument for classification error, it would be expected that the distance of the privacy preserving coefficients from the true coefficients increases as more noise is added, i.e. for smaller epsilon values.
  - β_0:
    The RMSE for β_0 is highest for $\epsilon = 0.3$ and lowest for $\epsilon = 2$. In general, for higher values of epsilon, the RMSE of β_0 is lower.
  - β_1:
    The RMSE for β_1 is highest for $\epsilon = 0.1$ and lowest for $\epsilon = 5$. There is a clear trend of plummeting RMSE for higher values of ϵ, i.e. less noise added.

In general, we see that for more privacy (lower ϵ), the LDP algorithm provides us with more noisy estimates of the coefficients, and higher classification error. Currently we are using concentrated DP analysis to get the upper bound on noise that we use for the local DP algorithm. Perhaps we could think of better composition using the moments accountant, but I'm unsure about the details of how it would be implemented.