HW2

CS208

Lipika Ramaswamy

https://github.com/lipikaramaswamy/cs208_lr/tree/master/homework/HW3
(https://github.com/lipikaramaswamy/cs208_lr/tree/master/homework/HW3)

Collaborators: Bhaven Patel, Karina Huang and Anthony Rentsch

R imports

In [1]:
```
## Setup
# rm(list=ls())
library(ggplot2)
library(repr)
```

In [2]:
```
### Using James's functions

rlap = function(mu=0, b=1, size=1) {
    p <- runif(size) - 0.5
    draws <- mu - b * sgn(p) * log(1 - 2 * abs(p))
    return(draws)
}


dlap <- function(x, mu=0, b=1) {
    dens <- 0.5 * b * exp(-1 * abs(x - mu) / b)
    return(dens)
}


plap <- function(x, mu=0, b=1) {
    cdf <- 0.5 + 0.5 * sgn(x - mu) * (1 - exp(-1 * (abs(x - mu) / b)))
    return(cdf)
}

qlap <- function(p, mu=0, b=1) {
    q <- ifelse(p < 0.5, mu + b * log(2 * p), mu - b * log(2 - 2 * p))
    return(q)
}

sgn <- function(x) {
    return(ifelse(x < 0, -1, 1))
}
```

Problem 1:

(a)

1. **Tails, Trimming, and Winsorization:** In all of the parts below, the dataset is $x \in \{0, 1, \ldots, D\}^n$. In all of the implementation parts, you should write code that takes as input $D \in \mathbb{N}$, $n \in \mathbb{N}$, $x \in \{0, 1, \ldots, D\}^n$, and $\varepsilon > 0$.

   (a) Prove that the following algorithm for estimating a Trimmed mean is ε-DP and implement it in code:

$$M(x) = \frac{1}{.9n} \cdot \left(\sum_{P_{.05} \le x_i \le P_{.95}} x_i \right) + \mathrm{Lap}\left(\frac{D}{\varepsilon n \underset{0.9}{\times}} \right),$$

   where $P_{.05}$ and $P_{.95}$ are the 5th and 95th percentile of the dataset. That is, we are applying the Laplace mechanism after removing the bottom and top 5% of the dataset. (Hint: Think about Lipschitz constants.)

First, we will show that the transformation of trimming data is 1-Lipschitz. Then, we will use the fact that the mechanism of Laplace noise added to the mean is ϵ-DP and that the composition of this mechanism and the transformation is $(1 \times \epsilon)$-DP.

1:

Consider the ordered datasets: $x = \{x_1, x_2, \ldots, x_{n-1}, x_n\}$ of size n and $x = \{x_2, x_3, \ldots, x_n, x_{n+1}\}$ of size n, where x and x' difer on one element only corresponding to the most extreme change in the mean. So, $d(x, x') = 1$.

Let T be the transformation of trimming the data to the middle 90 percentile. Let x_{p^5} be the data point at the 5th percentile and $x_{p^{95}}$ be the data point at the 95th percentile. Then, $T(x) = \{x_{p^5}, \ldots, x_{p^{95}}\}$ and $T(x') = \{x_{p^5+1}, \ldots, x_{p^{95}+1}\}$. These two datasets only differ on one element, so $d(T(x), T(x')) = 1$.

By definition, a mapping from datasets to datasets is c-Lipshitz iff: $\forall x, x', d(T(x), T(x')) \leq c \cdot d(x, x')$.

For arbitrary x, x', we showed that $c = 1$. Hence, by definition, the mapping T is 1-Lipschitz.

2:

Consider the mechanism below:

$$M' = \frac{1}{n'} \cdot \left(\sum_{i=1}^{n'} x_i \right) + Lap\left(\frac{D}{\varepsilon n'} \right)$$

Adding Laplace noise with scale equal to the global sensitivity of the mechanism divided by the privacy loss, i.e. $\frac{GS}{\varepsilon}$, to a mean is ε-DP, as shown in class and referred to in the last homework. Here, the global sensitivity for the mean is $\frac{D-0}{n'}$, where n' is the size of the dataset. The noise is being drawn from a Laplace distribution with scale $\frac{D}{\varepsilon n'}$. Thus, this mechanism is ε-DP.

3:

By a lemma, we know that if a mechanism M' is ε-DP and a transformation T is c-Lipschitz, then M'∘T is a $c\varepsilon$-DP mechanism. Note that $n' = 0.9 \cdot n$.

Here, we have the above mechanism M' and transformation T, which is 1-Lipschitz, so the composition is ε-DP, i.e.:

$$M(x) = M' \circ T(x) = \frac{1}{.9n} \cdot \left(\sum_{P_{.05} \leq x_i \leq P_{.95}} x_i \right) + Lap\left(\frac{D}{.9\varepsilon n} \right) \text{ is } \varepsilon\text{-DP.}$$

In [3]:
```
## trim variable to a range

# Trimming helper function
#
# Determines the probability a draw from a LaPlace distribution is less
 than
#    or equal to the specified value.
#
# x - vector/column/list to be trimmed
# lower -  Numeric, the center of the LaPlace distribution, defaults to
 0.
# b Numeric, the spread of the LaPlace distribution, defaults to 1.
#
# return Probability the LaPlace draw is less than or equal to \code{x}.
# example:
#
# x <- 0
# plap(x)

trim <- function(x, lower, upper){
    x.trim <- x
    x.trim <- x.trim[x.trim<upper]
    x.trim <- x.trim[x.trim>lower]
    return(x.trim)
}

## DP trimmed mean release function
trimmedMeanRelease <- function(x, epsilon, D, n=0){

    # populate length of dataset if not specified
    if(n == 0){
        n <- length(x)
    }

    # calculate true upper and lower quantiles
    upper <- quantile(x, probs = 0.95)
    lower <- quantile(x, probs = 0.05)

    # trim the data, i.e. cutoff points below and above the 5th and 95th
percentile
    trimmed.x <- trim(x, lower, upper)

    # calculate sensitivity - note that we use 0.9 * n instead of length
(trimmed.data)
    # to ensure we don't reveal anything about the trimmed dataset
    sensitivity <- D / (0.9 * n)
    scale <- sensitivity / epsilon

    # calculate true mean of trimmed data
    sensitiveValue <- mean(trimmed.x)

    # add laplace noise to the true mean to get DP release
    DPrelease <- sensitiveValue + rlap(mu=0, b=scale, size=1)

    return(list(release=DPrelease, true=sensitiveValue))
}
```

A quick example below uses some randomly sampled data with range between 1 and 100, and returns the trimmed mean. As seen below, the released trimmed mean is different from the true value.

```
In [4]: ## Testing trimmed mean

        # make some fake natural number valued data of size 100
        x.testing <- sample.int(100, 100, replace = TRUE)

        # check that fn works
        trimmedMeanRelease(x.testing, 1, 100 , 100)
```

```
$release
47.7811275041238
$true
47.8333333333333
```

(b)

(b) Prove that for large enough n, the analogous algorithm for the *Winsorized* mean is *not* ε-DP:

$$M(x) = \frac{1}{n} \cdot \sum_{i=1}^{n} [x_i]_{P_{.05}}^{P_{.95}} + \text{Lap}\left(\frac{D}{\varepsilon n}\right),$$

where $[x]_a^b$ is defined as in Problem Set 2. In Winsorization, we clamp points rather than dropping them. (In class on 3/11, we incorrectly referred to dropping points as Winsorization.) Again, it may be useful to first think in terms of Lipschitz constants.

First, we will show that the transformation of trimming data is 0.1n-Lipschitz. Then, we will use the fact that the mechanism of Laplace noise added to the mean is ε-DP and that the composition of this mechanism and the transformation is $(0.1n \times \varepsilon)$-DP, which is not ε-DP.

1:

Consider the ordered datasets: $x = \{x_1, x_2, \ldots, x_{n-1}, x_n\}$ of size n and x' to be another dataset of size n, where x and x' difer on one element, where, in x', the value of the 5th percentile has been changed to a value above the 95th percentile. So, $d(x, x') = 1$.

Let T be the transformation of clamping/Winsorizing the data to the middle 90 percentile. Let x_{p^5} be the data point at the 5th percentile and $x_{p^{95}}$ be the data point at the 95th percentile in dataset x and x'_{p^5} be the data point at the 5th percentile and $x'_{p^{95}}$ be the data point at the 95th percentile in dataset x'. Then, $T(x) = \{x_{p^5}, \ldots, x_{p^5}, x_{p^5}, \ldots, x_{p^{95}}, x_{p^{95}}, x_{p^{95}}, \ldots, x_{p^{95}}\}$ where the values below the 5th percentile get clamped to x_{p^5} and the values above the 9th percentile get clamped to $x_{p^{95}}$. There are $0.05 \times n$ such values below the 5th percentile and $0.05 \times n$ values above the 95th percentile. Similarly, $T(x') = \{x'_{p^5}, \ldots, x'_{p^5}, x'_{p^5}, \ldots, x'_{p^{95}}, x'_{p^{95}}, x'_{p^{95}}, \ldots, x'_{p^{95}}\}$ where the values below the 5th percentile get clamped to x'_{p^5} and the values above the 9th percentile get clamped to $x'_{p^{95}}$. There are $0.05 \times n$ such values below the 5th percentile and $0.05 \times n$ values above the 95th percentile.

These two datasets (T(x) and T(x')) now differ on $0.05n + 0.05n = 0.1n$ rows, so $d(T(x), T(x')) = 0.1n$.

By definition, a mapping from datasets to datasets is c-Lipshitz iff: $\forall x, x', d(T(x), T(x')) \leq c \cdot d(x, x')$.

For arbitrary x, x', we showed that $c = 0.1n$. Hence, by definition, the mapping T is $0.1n$-Lipschitz.

2:
This is the same as in part (a).

3:

By a lemma, we know that if a mechanism M' is ε-DP and a transformation T is c-Lipschitz, then M'∘T is a $c\varepsilon$-DP mechanism.

Here, we have the above mechanism M' and transformation T, which is 0.1n-Lipschitz. The scale of noise that would be added to the Windsorized mean would thus increase for larger datasets (higher n, ≥ 10). Therefore, the mechanism is **not** ε-DP.

(c)

(c) In class, we saw how to use the exponential mechanism to an estimate of the median, $P_{.5}$. Describe and implement a version of the exponential mechanism that releases an estimate of the tth percentile P_t of a dataset $x \in \{0, \ldots, D\}^n$ any desired $t \in [0, 100]$. (A direct implementation of the exponential mechanism would require explicitly calculating weights for each of the $D+1$ possible outputs, which can be too slow for large values of D such as in the parts below. One way to solve this is to bin the elements into fixed, coarser intervals. Alternatively, you can sample more quickly from the output distribution of the exponential mechanism by noting that if you sort the elements of the dataset $x_{i_1} \leq x_{i_2} \leq \cdots \leq x_{i_n}$, then all elements of each interval between x_{i_j} and $x_{i_{j+1}}$ have the same weight, so you can sample by choosing an interval with probability proportional to the sum of weights within it and then sampling uniformly from that interval. Feel free to use either solution below.)

An exponential mechanism to release the median is one that releases an estimate of the 50th percentile of the dataset. We now alter this mechanism to release an estimate of the t^{th} percentile. We will use the following utility function:

$$u(x, y) = -|n \cdot t - \#\{\text{ i} : x_i \leq y\}|$$

In this mechanism, the elements of the dataset are first sorted and this sorted set serves as the bins. To ensure that the mechanism is DP, we add an additional bin, which is the upper bound of the data, i.e. D. For a given percentile, t, the utility of each of the bins is calculated. This utility function, shown above, is 0 for bins at the tth percentile, and becomes negative for bins that get further from the tth percentile. The likelihood of the bins is then calculated using $exp\left(\frac{\varepsilon \cdot u(x,y)}{2 \cdot GS}\right)$. The likelihood of bins is also weighted by the distance of this bin from the next bin (to account for values between two bins that may not be present in the data). Using these likelihoods, probabilities are calculated for each bin, and these serve as probabilities for a multinomial distribution. Once we sample from this multinomial distribution (in a roundabout fashion using `runif` in R, as shown below and straight from lecture). Once we have selected a bin to release, we do a uniform random draw from this bin and release this value.

In [5]:
```
## 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 [6]: percentileRelease <- function(x, epsilon, t, D, n=0){
            # get length of data
            if (n == 0){
                n <- length(x)
            }

            # populate nbins
            bins <- sort(x)
            bins <- append(bins, D, after = length(bins))
            nbins <- length(bins)

            # get the t-th quantile
            sensitiveValue <- quantile(x, probs = t)

            # empty vector to store utility scores
            likelihoods <- rep(NA, nbins)

            # calculate utility scores
            for(i in 1:length(likelihoods)){

                # multiply through by n to make GS_u = 1
                utility <- - abs(n*t - i)

                # for weighting the bins propotional to the number of elements i
        n it
                dist <- 1
                if (i < n){

                    # accounting for values that aren't present in the data and
         weighting the bins
                    dist = abs(x[i] - x[i+1]) + 1
                }
                likelihoods[i] <-  dist * exp(epsilon * utility/ 2)
            }

            # calculate probability
            probabilities <- likelihoods/sum(likelihoods)

            # compare probability vector to a random uniform draw between 0 and
         1
            flag <- runif(n=1, min=0, max=1) < cumsum(probabilities)

            ## sample uniformly from the bin
            indices.of.bins <- which(flag == 1)

            if(sum(flag) > 1){
                min.runif <- bins[indices.of.bins[1]]
                max.runif <- bins[indices.of.bins[1] + 1]
                DPrelease <- runif(n=1, min=min.runif, max=max.runif)
            }
            # in the case that we're in the last bin
            else{
                DPrelease <- bins[indices.of.bins[1]]
            }
            return(list(release=DPrelease, true=sensitiveValue))
        }
```

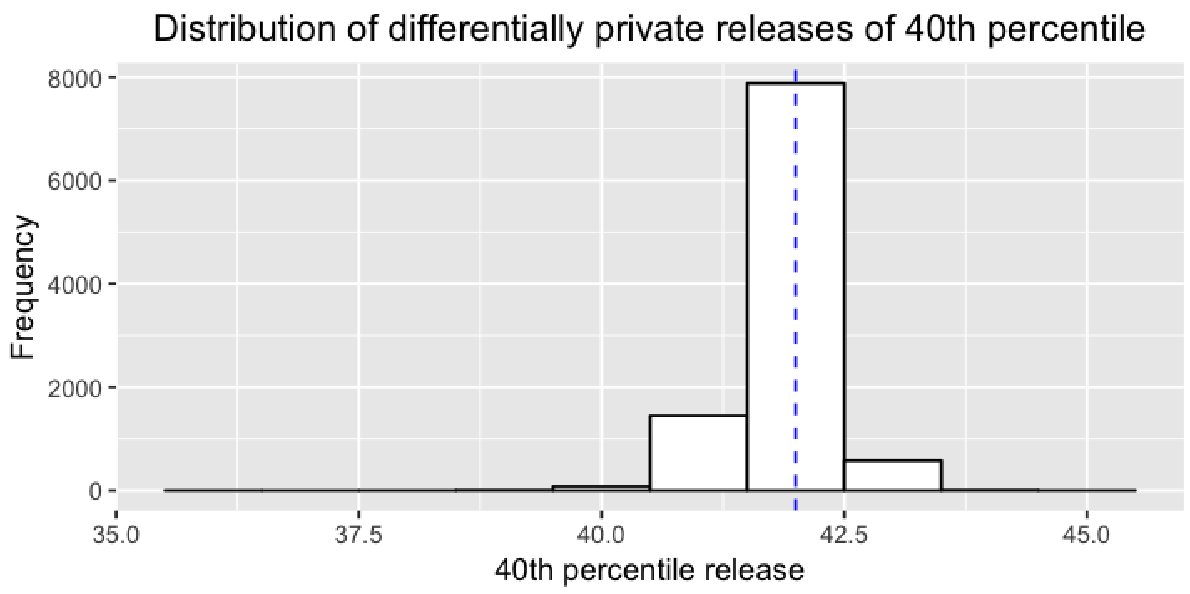
Example of release for 10th percentile - distribution of released value

In the simulation below, integers in the range of 1 to 100 are randomly sampled as a dataset and the 40th percentile is released using a DP mechanism. This DP release is performed for 10,000 simulations. The true values are also saved and the true value is plotted as the blue line. The histogram shows that the DP release of the 40th percentile peaks around the true 40th percentile, and the distribution is symmetric, as expected from the construction of the mechanism.

In [7]:
```
## Testing distribution of the 40th percentile releases
history <- rep(NA, 10000)
trueHistory <- rep(NA, 10000)

# Make dataset
x <- sample.int(100, 1000, replace = TRUE)

for(i in 1:10000){
    # Do percentile release - 10th percentile
    release <- percentileRelease(x, epsilon = 0.5, t = 0.4, D = 100)
    history[i] <- release$release
    trueHistory[i] <- release$true
}
```

In [8]:
```
# plotting this histogram
for.percentile <- data.frame(history, trueHistory)
p1c <- ggplot(for.percentile, aes(x=history)) +
    geom_histogram(color="black", fill="white", binwidth = 1) +
    geom_vline(aes(xintercept=mean(trueHistory)),
            color="blue", linetype="dashed", size=0.5)+
    labs(x = "40th percentile release", y = 'Frequency', title =
         'Distribution of differentially private releases of 40th percen
tile') +
    theme(plot.title = element_text(hjust = 0.5))
```

```
In [9]: options(repr.plot.width=6, repr.plot.height=3)
        p1c
```



(d)

(d) Implement the following ε-DP algorithm for estimating a Trimmed mean of a dataset: use your algorithm from Part 1c to get $\varepsilon/3$-DP estimates $\hat{P}_{.05}$ and $\hat{P}_{.95}$ of the 5th and 95th percentiles, drop all datapoints that lie outside the range $[\hat{P}_{.05}, \hat{P}_{.95}]$, and then use the Laplace mechanism to compute an $(\varepsilon/3)$-DP mean of the trimmed data. That is, your code should compute

$$M(x) = \frac{1}{.9n} \cdot \left(\sum_{i:\hat{P}_{.05} \le x_i \le \hat{P}_{.95}} x_i \right) + \text{Lap}\left(\frac{3(\hat{P}_{.95} - \hat{P}_{.05})}{0.9\varepsilon n} \right).$$

In [10]:
```
## DP trimmed mean release function

trimmedMeanReleaseNew <- function(x, epsilon, D, n=0,
                                  lowerPercentileCutoff = 0.05, upperPercen
tileCutoff = 0.95){
    # populate n if not given
    if(n == 0){
        n <- length(x)
    }

    # Get DP releases for upper and lower bound
    upper <- percentileRelease(x, epsilon = epsilon/3, t = upperPercenti
leCutoff, D = D, n)
    lower <- percentileRelease(x, epsilon = epsilon/3, t = lowerPercenti
leCutoff, D = D, n)

    # trim data based on DP upper and lower bounds
    trimmed.x <- trim(x, lower$release, upper$release)

    # calculate scale
    sensitivity <- (upper$release - lower$release ) / (0.9 * n)
    scale <- sensitivity / (epsilon/3)

    # calculate sensitive mean of trimmed data
    sensitiveValue <- mean(trimmed.x)

    # calculate differentially private mean by adding laplace noise
    DPrelease <- sensitiveValue + rlap(mu=0, b=scale, size=1)

    return(list(release=DPrelease, true=sensitiveValue))
}
```

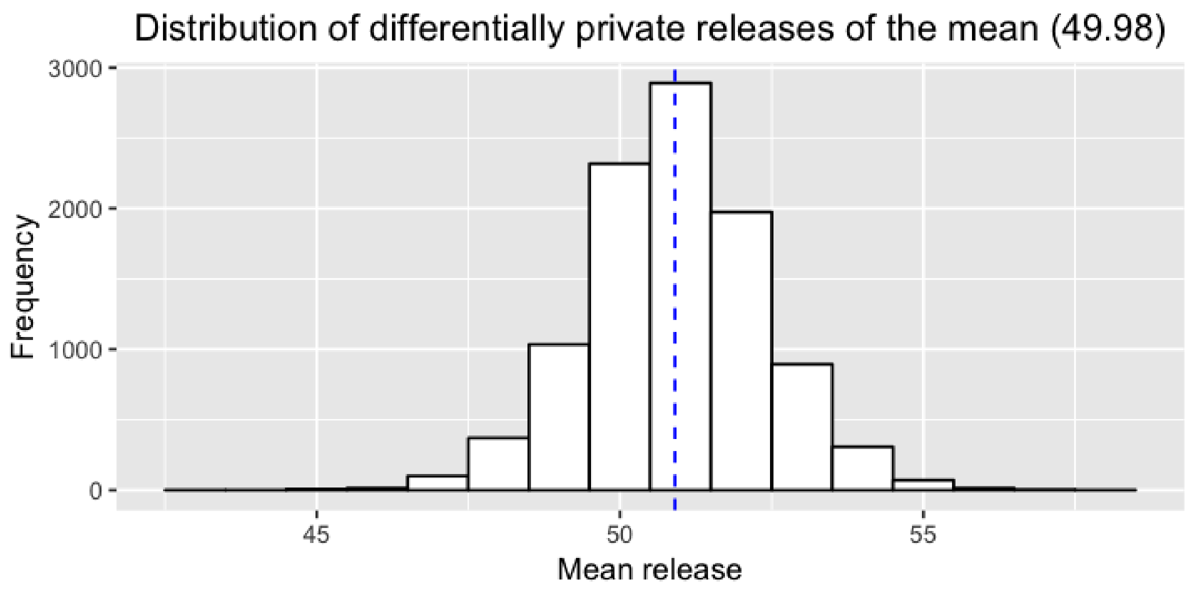
Example of release for trimmed mean - distribution of released value

In the simulation below, 1000 integers in the range of 1 to 100 are randomly sampled as a dataset and the trimmed mean (between 5th and 95th DP percentiles) is released using a DP mechanism. This DP release is performed for 10,000 simulations. The true values are also saved and the true value is plotted as the blue line. The histogram shows that the DP release of the trimmed mean around the true trimmed mean averaged across all simulations, and the distribution is symmetric, as expected from the construction of the mechanism.

In [11]:
```
## Testing distribution of the 50th percentile releases
history.d <- rep(NA, 10000)
trueHistory.d <- rep(NA, 10000)

# Make dataset
x.d <- sample.int(100, 1000, replace = TRUE)

for(i in 1:10000){
    # Do percentile release - 10th percentile
    release.d <- trimmedMeanReleaseNew(x.d, epsilon = 0.5, D = 100)
    history.d[i] <- release.d$release
    trueHistory.d[i] <- release.d$true
}
```

In [12]:
```
# plotting this histogram
for.percentile.d <- data.frame(history.d, trueHistory.d)
p1d <- ggplot(for.percentile.d, aes(x=history.d)) +
    geom_histogram(color="black", fill="white", binwidth = 1) +
    geom_vline(aes(xintercept=mean(trueHistory.d)),
            color="blue", linetype="dashed", size=0.5)+
    labs(x = "Mean release", y = 'Frequency', title =
        'Distribution of differentially private releases of the mean (4
9.98)') +
    theme(plot.title = element_text(hjust = 0.5))
```

In [13]:
```
options(repr.plot.width=6, repr.plot.height=3)
p1d
```



(e)

(e) Determine whether or not the following analogue for a Winsorized mean is ε-DP: use Part 1c to get $\varepsilon/3$-DP estimates $\hat{P}_{.05}$ and $\hat{P}_{.95}$ of the 5th and 95th percentiles, and output

$$M(x) = \frac{1}{n} \cdot \left(\sum_{i=1}^{n} [x_i]_{\hat{P}_{.05}}^{\hat{P}_{.95}} \right) + \text{Lap}\left(\frac{3(\hat{P}_{.95} - \hat{P}_{.05})}{\varepsilon n} \right).$$

You do not need to formally prove your answer, but you should at least provide an informal explanation.

In part (b), the mechanism failed to be ε-DP because of the reliance on n (the mechanism was $0.1n\epsilon$-DP for values of n less than or equal to 10, but not for any datasets larger than 10). Here, if the mechanism is ϵ-DP, we would have to show that it's not reliant on the size of the dataset. My intuition is that the DP releases of the 5th and 95th percentile aren't directly reliant on the size of the dataset, as those use the exponential mechanism, and the noise added to the Winsorized mean is has the correct scale(GS/ε), so perhaps we can use composition to show that it's ε-DP.

(f)

(f) The dataset `MaPUMS5full.csv` provides the 5% PUMS Census file for Massachusetts. For $\varepsilon = 1$ and $D = 1,000,000$, compare the RMSE between DP means and the actual means for each PUMA in Massachusetts,[1] for DP means calculated using (i) the ordinary Laplace mechanism for a mean (remembering to clamp your data to the range!) and (ii) the algorithm from Part 1d. Also show box-and-whisker plots of the DP released means for each PUMA by these algorithms, noting the true means. You should probably order these by mean income, or perhaps skew of income, or anything you think reveals an interesting pattern. Give an intuitive explanation of the kinds of datasets on which algorithm (i) is likely to perform better than algorithm (ii) and vice-versa. Describe any modifications you might propose would increase the utility (at the same level of privacy preservation) for data similar to this income example.

In [14]:
```
# import data
library("foreign")
PUMSdata <- read.csv(file=
                     "https://raw.githubusercontent.com/privacytoolsproj
ect/cs208/master/data/MaPUMS5full.csv")
data.inc <- (PUMSdata$income)

# get true mean of income
true.mean<- mean(data.inc)
```

In [15]:
```
## (i) SIMPLE LAPLACE NOISE

## Bound/Censor/Clip/Clamp 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)
}

## Differentially private mean release
basicMeanRelease <- function(x, lower, upper, epsilon, n = 0){
    if(n == 0){
        n <- length(x)
    }
    sensitivity <- (upper - lower)/n
    scale <- sensitivity / epsilon
    x.clipped <- clip(x, lower, upper)
    sensitiveValue <- mean(x.clipped)
    DPrelease <- sensitiveValue + rlap(mu=0, b=scale, size=1)
    return(list(release=DPrelease, true=sensitiveValue))
}
```

In [16]:
```
## Get all puma regions
all.pumas <- unique(PUMSdata$puma)
n.iter <- 100

## Set parameters for the mean releases
myeps <- 1
myD <- 1e+06

history <- matrix(NA, nrow = length(all.pumas)*n.iter * 2, ncol = 5)
## COLS structure
# puma  iter  method          DPmean  truemean
#             1 = basic
#             2 = trimmed


## For loop for 100 iterations of getting DP releases (lap and exp)
counter = 1
for(one.puma in all.pumas){

    ## subset df to the puma region
    puma.df <- PUMSdata[PUMSdata$puma == one.puma,]
    puma.inc <- (puma.df$income)

    ## Real mean
    true.mean <- mean(puma.inc)

    for(i in 1:n.iter){

        ## Do DP mean releases
        basic.release <- basicMeanRelease(x = puma.inc, lower = 0, upper
= myD, epsilon = myeps)
        trimmed.release <- trimmedMeanReleaseNew(x = puma.inc, epsilon =
myeps, D = myD)

        # store vals
        history[counter,] = c(one.puma, i, 1, basic.release$release, tru
e.mean)
        history[counter+1,] = c(one.puma, i, 2, trimmed.release$release,
true.mean)

        # update counter
        counter = counter + 2
    }
}
```

In [17]:
```
# make a df of the matrix
history.df <- data.frame(history)
names(history.df) <- c("puma", "iteration", "method", "DPmean", "truemea
n")
# history.df
```

In [18]:
```r
# utility function to calculate RMSE
calcRMSE <- function(true.col, pred.col){
    error = (true.col-pred.col)
    errorsq = error ** 2
    RMSE = sqrt(sum(errorsq) / length(true.col))
    return(RMSE)
}
```

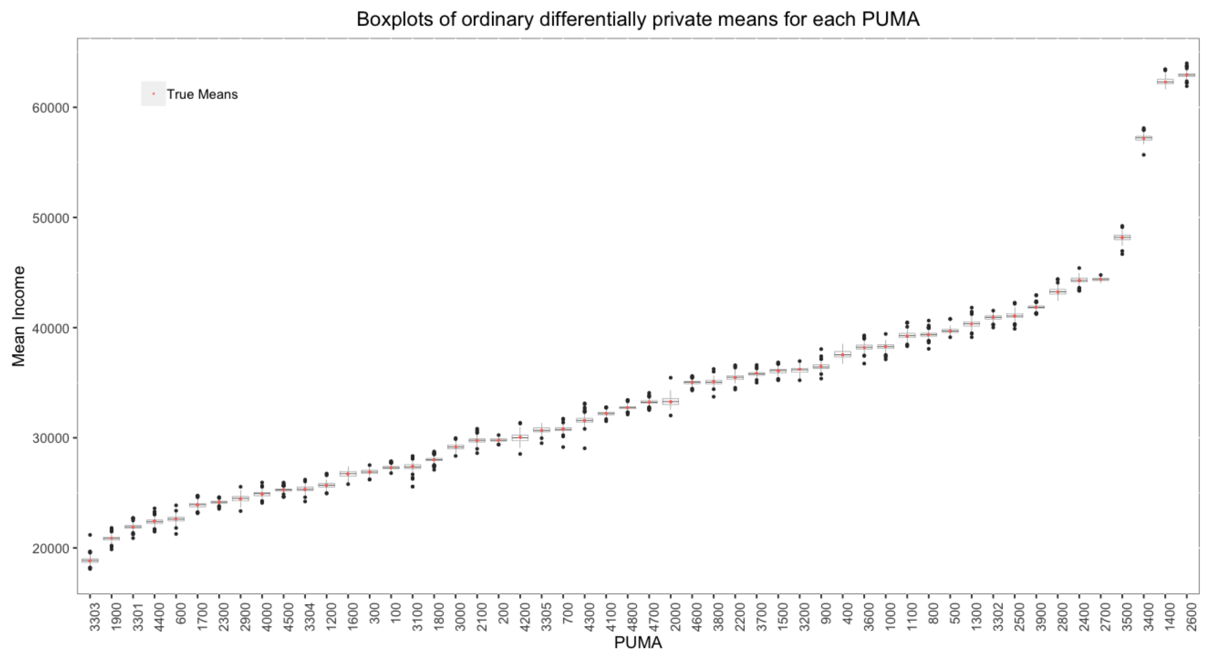
In [21]:
```
## calculate RMSE
store.RMSE.lap <- rep(NA, length(all.pumas))
store.RMSE.exp <- rep(NA, length(all.pumas))

counter = 1
for(one.puma in all.pumas){
    puma.specific.df.lap <- history.df[(history.df$method == 1)
                                               &(history.df$puma == one.
puma), 4:5 ]
    puma.specific.df.exp <- history.df[(history.df$method == 2)
                                               &(history.df$puma == one.
puma), 4:5 ]
    store.RMSE.lap[counter] <- calcRMSE(puma.specific.df.lap[,1], puma.s
pecific.df.lap[,2])
    store.RMSE.exp[counter] <- calcRMSE(puma.specific.df.exp[,1], puma.s
pecific.df.exp[,2])
    counter = counter + 1
}

display.RMSE <- data.frame(all.pumas, store.RMSE.lap, store.RMSE.exp, st
ore.RMSE.exp /store.RMSE.lap )
names(display.RMSE) <- c("PUMA", "Ordinary Mechanism RMSE", "Trimmed Mec
hanism RMSE" ,
                         "RMSE - Ratio of Trimmed to Ordinary")
display.RMSE <- display.RMSE[order(display.RMSE$PUMA, decreasing=FALSE
),]
rownames(display.RMSE) <- 1:nrow(display.RMSE)
display.RMSE
```

PUMA	Ordinary Mechanism RMSE	Trimmed Mechanism RMSE	RMSE - Ratio of Trimmed to Ordinary
100	208.8370	4593.1909	21.9941419
200	160.4150	4552.9316	28.3822081
300	249.7880	2865.3620	11.4711743
400	367.8808	4615.5938	12.5464375
500	259.0095	4685.2317	18.0890347
600	338.8985	1574.0438	4.6445872
700	327.2044	5545.1399	16.9470197
800	354.9651	6768.4476	19.0679229
900	364.1774	5441.6906	14.9424165
1000	336.0814	5986.2555	17.8119215
1100	397.8716	7455.9724	18.7396453
1200	304.1696	2206.2662	7.2534086
1300	390.4201	5499.0584	14.0849763
1400	360.7336	10385.8836	28.7910105
1500	307.3117	5364.4213	17.4559619
1600	344.9837	5106.8902	14.8032803
1700	336.5842	2397.7741	7.1238457
1800	274.6155	3311.9775	12.0604173
1900	306.8886	1694.9859	5.5231317
2000	426.2673	5165.8106	12.1187132
2100	313.5553	2799.7785	8.9291366
2200	342.3741	4986.3827	14.5641345
2300	183.9921	2524.8283	13.7224791
2400	378.1734	7239.5507	19.1434697
2500	404.7310	6902.4755	17.0544755
2600	330.1341	14361.4565	43.5018866
2700	150.4890	7977.5598	53.0109120
2800	341.8431	8001.2299	23.4061466
2900	363.3379	1394.1085	3.8369474
3000	291.3666	2384.5949	8.1841739
3100	442.9709	2805.9906	6.3344807
3200	301.4722	7015.8796	23.2720649
3301	278.3968	2569.0321	9.2279508
3302	266.2024	8581.1068	32.2352727
3303	379.5143	175.1224	0.4614382

PUMA	Ordinary Mechanism RMSE	Trimmed Mechanism RMSE	RMSE - Ratio of Trimmed to Ordinary
3304	293.7557	1011.7057	3.4440374
3305	296.7386	2990.7721	10.0788109
3400	310.6263	10690.9088	34.4172673
3500	357.9191	8670.7992	24.2255837
3600	390.4911	5032.6851	12.8880932
3700	233.2445	4721.4615	20.2425458
3800	329.7311	5639.9955	17.1048323
3900	270.5271	7584.9540	28.0376863
4000	296.0419	1274.5718	4.3053757
4100	234.7951	3432.8863	14.6207753
4200	427.4231	2968.5893	6.9453187
4300	483.3018	3899.0291	8.0674827
4400	335.1479	2162.8128	6.4533082
4500	220.6227	3678.3997	16.6728066
4600	225.4540	5288.7589	23.4582666
4700	261.8731	5814.9682	22.2052937
4800	228.7611	5664.2386	24.7604987

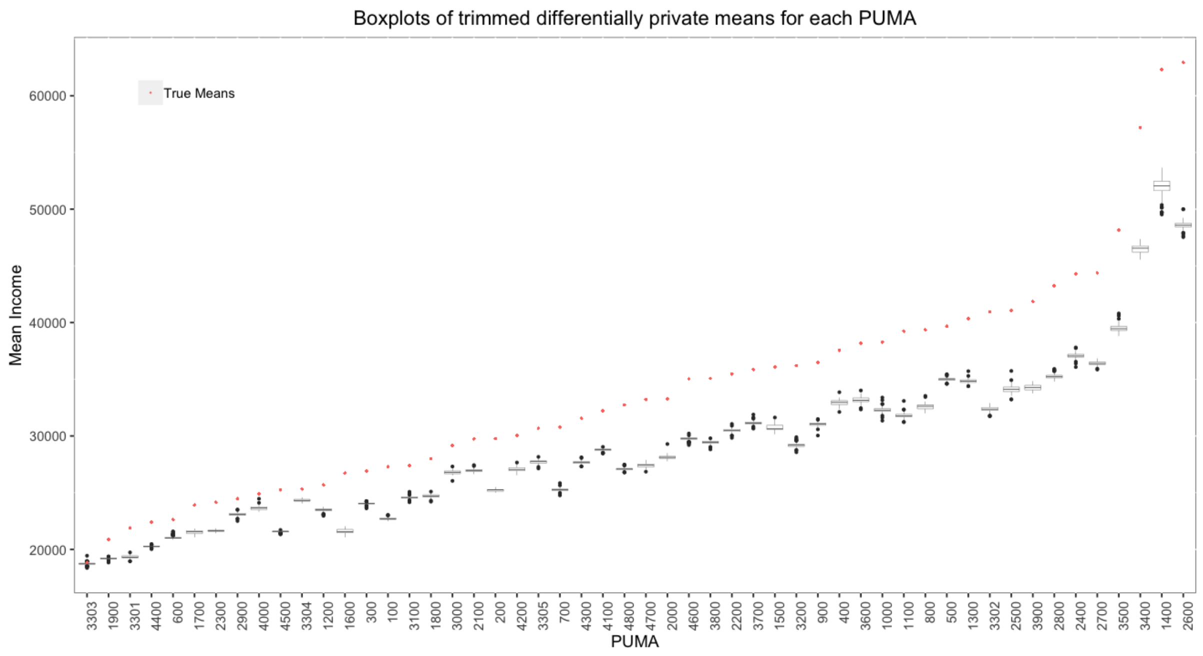
In [22]:
```
pp1 <- ggplot(data = history.df[history.df$method == 1, ] , ## this corr
esponds to ordinary mechanism
            aes(x=reorder(factor(puma), truemean), y=DPmean)) +
    geom_boxplot(lwd = 0.1, outlier.size = 0.5)+
    theme(plot.title = element_text(hjust = 0.5), axis.text.x = element_
text(angle = 90, hjust = 1),
          panel.background = element_rect(fill = "white", colour = "grey
50"),legend.title = element_blank(),
          legend.position = c(.1, .9)) +
    geom_point( aes(x=reorder(factor(puma), truemean), y=truemean, color
= 'True Means'), shape=18, size = 0.5)+
    labs(x = "PUMA", y = 'Mean Income', title =
         'Boxplots of ordinary differentially private means for each PUM
A')
```

In [23]:
```
options(repr.plot.width=11, repr.plot.height=6)
pp1
```



In [24]:
```
pp2 <- ggplot(data = history.df[history.df$method == 2, ],
            aes(x=reorder(factor(puma), truemean), y=DPmean)) +
    geom_boxplot(lwd = 0.1, outlier.size = 0.5)+
    theme(plot.title = element_text(hjust = 0.5), axis.text.x = element_
text(angle = 90, hjust = 1),
          panel.background = element_rect(fill = "white", colour = "grey
50"), legend.title = element_blank(),
         legend.position = c(.1, .9)) +
    geom_point( aes(x=reorder(factor(puma), truemean), y=truemean, color
= 'True Means'), shape=18, size = 0.5)+
    labs(x = "PUMA", y = 'Mean Income', title =
         'Boxplots of trimmed differentially private means for each PUM
A')
```

In [25]:
```
pp2
```


Boxplots of trimmed differentially private means for each PUMA
True Means
Mean Income
60000
50000
40000
30000
20000
PUMA
3303
1900
3301
4400
600
1700
2300
2900
4000
4500
3304
1200
1600
300
100
3100
1800
3000
2100
200
4200
3305
700
4300
4100
4800
4700
2000
4600
3800
2200
3700
1500
3200
900
400
3600
1000
1100
800
500
1300
3302
2500
3900
2800
2400
2700
3500
3400
1400
2600

In this problem, we performed two mean releases for each PUMA, (i) that *clamps* the data to a range specified by the user and then adds laplace noise, and (ii) that calculates differentially private 5th and 95th percentiles, *trims* the data based on these percentiles, and then adds laplace noise. 100 iterations were performed for each PUMA to obtain a distribution of mean releases from both mechanisms.

The root mean square error measures how different the true means are from the released means. For all PUMAs, the RMSE of the of the trimmed mean mechanism is orders of magnitude larger than the RMSE of the ordinary mechanism, indicating that the trimmed mean mechanism is doing a poorer job of giving us a true value for the mean.

The same is demonstrated by the boxplots. Note that both figures show one boxplot for each PUMA, which are in increasing order of true sample mean of income. The boxplots for (i), the ordinary mean release, show the spread of the released means, the box of most of which (i.e. 25th to 75th percentiles) always cover the true sample mean income for the PUMA. An aside is that there are several outliers for each of the boxplots, as seen by the dots. The underlying data, i.e. income in each of the PUMAs, is right skewed, with high outliers. The underlying mechanism in (i) is clamping the data based on some prior knowledge of what we think upper and lower bounds should be (here we use lower bound of 0 and upper bound of $D = 1,000,000$). Given that we know the largest value for income in MA is less than $850,000, we are not removing any outliers in our clamping. Our input of D affects the scale of the Laplace noise added, which increases proportional to D. I found that when I increase D to 10 million, the spread of the boxplots becomes wider, but the middle 50 percentile always covers the true mean.

In mechanism (ii), we are first calculating the 5th and 95th percentiles in a differentially private manner using the exponential mechanism. We then use these to trim our data, i.e. remove data points that lie outside these percentiles, and then add Laplace noise. It is clear that we will be removing some of the high outliers in each PUMA in each iteration, and thus always release an estimate of the mean that is lower than the true mean. This can be seen in the boxplots as the true means (red dots) never being covered by the boxes (except perhaps for PUMA 3303).

Our privacy budget in both mechanisms is the same, i.e. $\varepsilon = 1$, so we can make comparisons regarding utility. We see in this case where we set D to 1 million, we get higher utility, i.e. lower RMSEs, for the ordinary differentially private mean release, mechanism (i). I view this as a consequence of a) the distribution of income, which is right skewed, and b) the fact that our upper bound, D, is never larger than the largest value in the sample.

As mentioned before, we are using some information about what we think the distribution of income should be in mechanism (i). When this is somewhat accurate, then it is reasonable to expect that mechanism (i) will yield higher utility than mechanism (ii). However, suppose we make an inaccurate judgement about the lower and upper bounds for the data (for example the data is truly standard normally distributed, and we decide that upper and lower bounds should be 0 and 1 respectively), then the ordinary mean release (mechanism (i)) will perform poorly, however, mechanism (ii) will still perform reasonably, as the choice for upper bound is just affecting the probability distribution that we use in the exponential mechanism to calculate the 5th and 95th percentiles. The impact of the upper bound is somewhat more impactful for mechanism (i) as compared to mechanism (ii).

Problem 2:

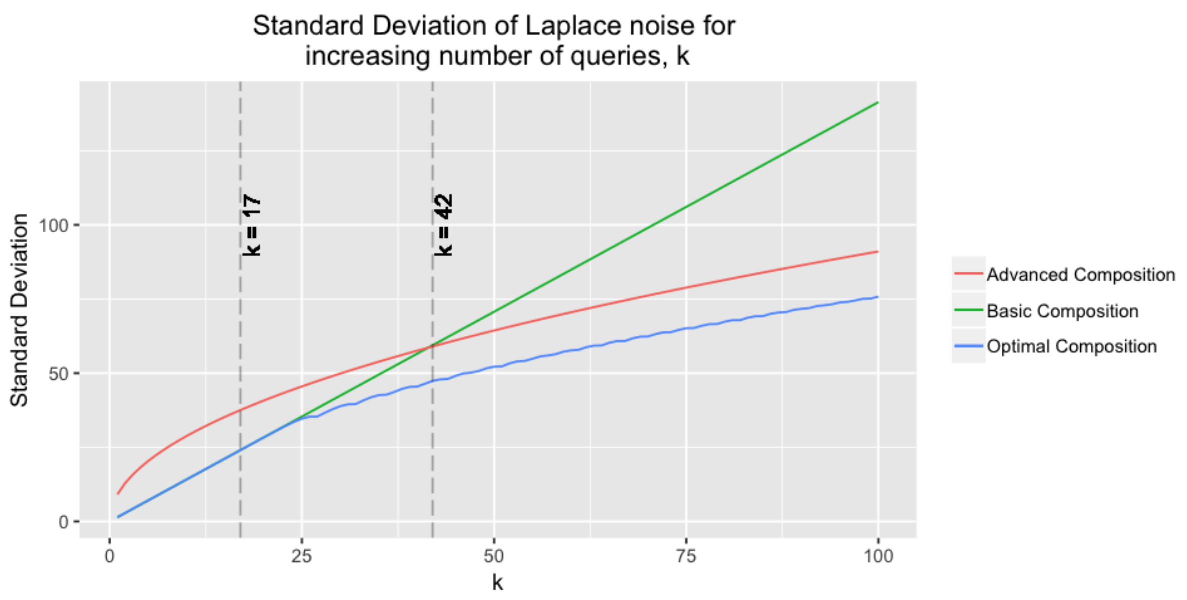
2. **Composition:** Suppose you have a global privacy budget of $\varepsilon = 1$ (and are willing to tolerate $\delta = 10^{-9}$) and you want to release k count queries (i.e. sums of Boolean predicates[2]) using the Laplace mechanism with an individual privacy loss of ε_0. By basic composition, you can set $\varepsilon_0 = \varepsilon/k$. Using the advanced composition theorem, you can set $\varepsilon_0 = \varepsilon/\sqrt{2k\ln(1/\delta)}$. We have provided you with code from PSI for the "optimal" composition theorem for differential privacy that calculates the largest value of ε_0 that ensures global (ε, δ)-DP as a function of ε, δ, and k.[3] For each of these choices, plot (on the same graph) the standard deviation of the Laplace noise added to each query as a function of k, and find the smallest values of k where the advanced and optimal composition theorems strictly improve upon the basic composition theorem.

The global sensitivity of the sum is 1 because changing 1 row in the dataset changes at most 1 in the Boolean predicates.

In [26]:
```
# Global Sensitivity of sums of Boolean predicates
#           - changing one row changes the sum by at most 1
gs = 1

# Set Params
epsilon = 1
delta = 1e-09
max_k = 100

store_sds = matrix(NA, nrow = max_k, ncol = 3)

for(k in 1:max_k){
    # calculate the epsilons for each release
    eps_0_comp = epsilon/k
    eps_0_adv = epsilon/sqrt(2 * k * log(1/delta))

    # calculate optimal epsilon using PSI Library
    init <- rep(c(1/k, 0), k )
    params <- matrix(init, nrow=k, ncol=2, byrow=TRUE)
    inverse <- PSIlence:::update_parameters(params=params, hold=0, eps=e
psilon, del=delta)
    eps_0_opt = inverse[1,1]

    # calculate standard deviation
    sds = sqrt(2) * c(gs/eps_0_comp, gs/eps_0_adv, gs/eps_0_opt)

    # store standard deviation
    store_sds[k, ] = sds
}
```

In [27]:
```
list_of_k = seq(1,100, 1)
sds.for.plot = data.frame(list_of_k, store_sds[,1], store_sds[,2], store
_sds[,3])
names(sds.for.plot) <- c("list_of_k", "basic_comp", "advanced_comp", "op
timal_comp")

## finding k where advanced and optimal comp yield lower sd for lap nois
e as cmopared to basic comp
k.where.adv.comp.wins = sds.for.plot[sds.for.plot$advanced_comp<sds.for.
plot$basic_comp,][1,1]
k.where.opt.comp.wins = sds.for.plot[sds.for.plot$optimal_comp<sds.for.p
lot$basic_comp,][1,1]

# plot stuff
p1 <- ggplot(sds.for.plot) +
    geom_line(aes(x=sds.for.plot$list_of_k, y=sds.for.plot$basic_comp, c
olor = "Basic Composition")) +
    geom_line(aes(x=sds.for.plot$list_of_k, y=sds.for.plot$advanced_comp
, color='Advanced Composition')) +
    geom_line(aes(x=sds.for.plot$list_of_k, y=sds.for.plot$optimal_comp,
color='Optimal Composition')) +
    labs(x = "k", y = 'Standard Deviation', title =
         'Standard Deviation of Laplace noise for \nincreasing number of
queries, k') +
    theme(plot.title = element_text(hjust = 0.5), legend.title = element
_blank()) +
    geom_vline(aes(xintercept=k.where.adv.comp.wins,),linetype = "longda
sh", alpha = 0.3) +
    geom_text(aes(x=k.where.adv.comp.wins, label="k = 42", y=100), angle
=90, vjust = 1.2)+

    geom_vline(aes(xintercept=k.where.opt.comp.wins,),linetype = "longda
sh", alpha = 0.3)+
    geom_text(aes(x=k.where.opt.comp.wins, label="k = 17", y=100), angle
=90, vjust = 1.2)
```

```
In [28]: options(repr.plot.width=8, repr.plot.height=4)
         p1
```



The plot of standard deviation of the Laplace noise added for increasing number of count queries released (k) shows that basic composition performs the worst, with standard deviation of the noise increasing linearly with the number of queries. Advanced composition performs worse than basic composition for a small number of queries (k < 42), but yields lower standard deviation needed for the Laplace noise, which results in released counts having higher utility. Similarly, optimal composition adds noise with the same standard deviation as basic composition up until k = 17, after which it decreases, yielding higher utility of the released counts.

Problem 3:

3. **Synthetic Data:** Expanding the template from class, and using again `MaPUMS5Full.csv`, create a DP three-way histogram[4] release of income, education and age. You do not need to graph this histogram, just compute the release for each binned combination of the variables. From this, you should be able to generate synthetic data of these three variables. Run a linear regression as a post-process on your synthetic data, predicting income from education and age[5] using the equation:

$$\text{Income}_i = \beta_0 + \beta_1 \text{Education}_i + \beta_2 \text{Age}_i + \nu_i; \qquad \nu_i \sim \mathcal{N}(0, \sigma^2) \tag{1}$$

Let $\beta^* = \{\beta_0^*, \beta_1^*, \beta_2^*\}$ be the coefficients in the full sensitive data, while $\tilde{\beta}$ the DP release we generate. The mean-squared error of a DP release of $\tilde{\beta}$ can be decomposed into the contributions of bias and variance as:

$$\text{MSE}(\tilde{\beta}) = \text{bias}(\tilde{\beta})^2 + \text{var}(\tilde{\beta}) = (\text{E}[\beta^* - \tilde{\beta}])^2 + \text{E}[(\bar{\tilde{\beta}} - \tilde{\beta})^2] \tag{2}$$

For this calculation, we are taking the (sensitive) regression coefficients β^* on the entire dataset as the true values of β. Show the contributions to MSE of the bias and variance of the DP-regression coefficients.[6]

As a baseline to decide if these squared bias and error terms are large, we can compute the MSE due simply to sampling, by bootstrapping with replacement new datasets in which we compute new (sensitive) regression estimates $\hat{\beta}$ on the bootstrapped data and compute $\text{MSE}(\hat{\beta})$. How do the bias and variance terms due to creating DP-releases compare to the this numerical estimate of the error introduced by sampling?

Note that the linear regression is modified to be of the form:

$$\log(\text{Income}_i) = \beta_0 + \beta_1 \cdot \text{Education}_i + \beta_2 \cdot \text{Age}_i + \nu_i$$

as income is heavily right skewed and the transformation ensures more normally distributed residuals, which is more in keeping with the assumption of the linear model.

In [29]:
```
# store variables of interest for easy access
data.educ <- PUMSdata$educ
data.inc <- PUMSdata$income
data.age <- PUMSdata$age
```

Calculate the "true" betas

First, a linear regression per the equation above is performed to get what is being considered as the "true values" for regression coefficients. The data is preprocessed by clipping it to reasonable bounds. Note that income is clipped between 1 and 1 million in order to avoid problems with taking the logarithm.

In [30]:
```
### CALCULATE "TRUE" BETAS
# coefficients of the full sensitive data

# first clip the data to the bounds
data.educ.lm <- clip(data.educ, lower=1, upper=16)
data.inc.lm <- clip(data.inc, lower=1, upper=1e+06)
data.age.lm <- clip(data.age, lower=18, upper=100)

# regress age and education on logged income
true.output <- lm(log(data.inc.lm) ~ data.educ.lm + data.age.lm)
true.slope <- coef(true.output)[2]
```

MSE due to sampling

The data is bootstrapped and regressions are performed to get bias, variance and MSE. This will give us an idea of sampling error.

In [31]:

```
# Function to calculate MSE and it's components
#
# Inputs:
# - true value, which could be a vector or float
# - predictions, in the form of a vector
#
# Returns:
# bias, variance and MSE

calcMSE <- function(true.val, pred.col){
    error = (true.val-pred.col)
    bias = (mean(error))
    var = var(pred.col)
    MSE = bias ^ 2 + var
    return(list(bias = bias, var = var, MSE = MSE))
}

# James's function to post-process a vector or matrix of DP count releas
es, into probabilities
normalize <- function(x){
    x[x<0] <- 0
    x <- x/sum(x)
    return(x)
}
```

In [32]:

```
### SAMPLING ERROR

## Bootstrap the data to get an idea of sampling error via RMSE
set.seed(1)
# Do 1000 bootstraps
n.boot = 100
store.real.data.betas <- matrix(NA, nrow = n.boot, ncol = 3)

for(boots in 1:n.boot){

    # get sample indices (sample with replacement)
    indices.to.samp <- sample(x = 1:length(data.educ.lm),
                              size = length(data.educ.lm),
                              replace = TRUE)
    # subset the vectors
    data.educ.lm.samp <- data.educ.lm[indices.to.samp]
    data.inc.lm.samp <- data.inc.lm[indices.to.samp]
    data.age.lm.samp <- data.age.lm[indices.to.samp]

    # perform linear regression with logged income as response
    lm.output <- lm(log(data.inc.lm.samp) ~ data.educ.lm.samp + data.ag
e.lm.samp)

    # store coefficients
    store.real.data.betas[boots, ] <- c(lm.output$coefficients[1],
                                        lm.output$coefficients[2],
                                        lm.output$coefficients[3])
}
```

Synthetic data generation from the histogram exponential mechanism

We will generate synthetic data using histograms

In [33]:
```
## Differentially private histogram
x1x2yHistogramRelease <- function(y, x1, x2,
                                ylower, yupper,
                                x1lower, x1upper,
                                x2lower, x2upper,
                                x1nbins = 0,
                                x2nbins = 0,
                                ynbins = 0,
                                epsilon){

    # get length of df essentially
    n <- nrow(x1)

    # clip data
    x1.clipped <- clip(x=x1, lower=x1lower, upper=x1upper)
    x2.clipped <- clip(x=x2, lower=x2lower, upper=x2upper)
    y.clipped <- log(clip(x=y, lower=1, upper=yupper))

    # preprocess the lower and upper bounds for y to convert to the log
 scale
    if(ylower == 0){
        ylower <- log(1)
    }
    yupper <- log(yupper)

    # populate x1 bins
    if(x1nbins==0){

        # get lower and upper values as integers
        x1lower <- floor(x1lower)
        x1upper <- ceiling(x1upper)

        # make vec of bins
        x1bins <- x1lower:(x1upper+1)

        # extract # bins
        x1nbins <- length(x1bins)-1

        # distance between bins is granularity
        x1granularity <- 1

        # just a codebook of bins used that will be output
        x1codebook <- x1bins[1:x1nbins]

    } else {

        # make a sequence of x1 values spaced out based on number of bin
s
        x1bins <- seq(from=x1lower, to=x1upper, length=x1nbins+1) # leng
th is nbins + 1 because
                                                                # we're
 looking at the number of endpoints of bins

        # get granularity
        x1granularity <- (x1upper-x1lower)/x1nbins
        # choose last elt of x1bins, which is the endpoint,
        # and add granularity to it to capture the endpoints of the data
        x1bins[x1nbins+1] <-  x1bins[x1nbins+1] + x1granularity

        # Release midpoints of bins
        x1codebook <- x1bins[1:x1nbins] + 0.5*x1granularity
    }

    # populate x2 bins
    if(x2nbins==0){

        # get lower and upper values as integers
        x2lower <- floor(x2lower)
        x2upper <- ceiling(x2upper)

        # make vec of bins
        x2bins <- x2lower:(x2upper+1)

        # extract # bins
        x2nbins <- length(x2bins)-1

        # distance between bins is granularity
        x2granularity <- 1

        # just a codebook of bins used that will be output
        x2codebook <- x2bins[1:x2nbins]

    } else {

        # make a sequence of x1 values spaced out based on number of bin
s
        x2bins <- seq(from=x2lower, to=x2upper, length=x2nbins+1) # leng
th is nbins + 1 because
                                                                # we're
 looking at the number of endpoints of bins

        # get granularity
        x2granularity <- (x2upper-x2lower)/x2nbins

        # choose last elt of x1bins, which is the endpoint,
        # and add granularity to it to capture the endpoints of the data
        x2bins[x2nbins+1] <-  x2bins[x2nbins+1] + x2granularity

        # Release midpoints of bins
        x2codebook <- x2bins[1:x2nbins] + 0.5*x2granularity
    }


    if(ynbins==0){

        # get lower and upper values as integers
        ylower <- floor(ylower)
        yupper <- ceiling(yupper)

        # make vec of bins
        ybins <- ylower:(yupper+1)
        # extract # bins
        ynbins <- length(ybins)-1

        # distance between bins is granularity
        ygranularity <- 1

        # just a codebook of bins used that will be output
        ycodebook <- ybins[1:ynbins]

    } else {

        # make a sequence of y values spaced out based on number of bins
        ybins <- seq(from=ylower, to=yupper, length=ynbins+1) # length i
s nbins + 1 because
                                                                    # we're
 looking at the number of endpoints of bins

        # get granularity
        ygranularity <- (yupper-ylower)/ynbins

        # choose last elt of ybins, which is the endpoint,
        # and add granularity to it to capture the endpoints of the data
        ybins[ynbins+1] <-  ybins[ynbins+1] + ygranularity

        # Release midpoints of bins
        ycodebook <- ybins[1:ynbins] + 0.5 * ygranularity

    }

    # sensitivity of histogram release as in the change model,
    #one person moves from one bin to another, changing both
    sensitivity <- 2
    scale <- sensitivity / epsilon

    sensitiveValue <- DPrelease <- matrix(NA, nrow=x1nbins * x2nbins * y
nbins, ncol=4)

    counter = 1
    for(i in 1:x1nbins){
        for (j in 1:x2nbins){
            for (k in 1:ynbins){

                # calculate sensitive count in bin
                sensitiveBinVal <-  sum(x1.clipped >= x1bins[i]
                                    & x1.clipped < x1bins[i+1]
                                    & x2.clipped >= x2bins[j]
                                    & x2.clipped < x2bins[j+1]
                                    & y.clipped >= ybins[k]
                                    & y.clipped < ybins[k+1])

                # add noise to sensitive val
                dpBinVal <- sensitiveBinVal + rlap(mu = 0, b = scale, si
ze = 1)

                # populate matrices (sensitive and DP) with the bin valu
e for x1, x2 and y
                sensitiveValue[counter,] <- c(x1codebook[i], x2codebook[
j], ycodebook[k], sensitiveBinVal)
                DPrelease[counter,] <- c(x1codebook[i], x2codebook[j], y
codebook[k], dpBinVal)

                # update counter
                counter <- counter + 1
            }
        }
    }

    return(list(release=DPrelease,
                true=sensitiveValue,
                x1codebook=x1codebook, x2codebook=x2codebook,
                ycodebook=ycodebook))
}
```

In [34]:
```
## build synthetic data
nsims <- 10

store.syn.betas <-matrix(NA, nrow=nsims, ncol=3)
Sys.time()
for(i in 1:nsims){
    out1 <- x1x2yHistogramRelease(y = data.inc, x1 = data.age, x2 = dat
a.educ,
                                   # cap income at 1 mil
                                    ylower = 0, yupper = 1e+06,
                                   # set age range b/w 18 and 100, based
on age range for Census data collection
                                    x1lower = 18, x1upper = 100,
                                   # education range based on codes
                                    x2lower = 1, x2upper = 16,
                                    x1nbins = 0,
                                    x2nbins = 0,
                                    ynbins = 10,
                                    epsilon = 1)

    # get probabilities
    syn.prob <- as.vector(normalize(out1$release[,4]))
    syn.sample.indices <- sample(x = 1:nrow(out1$release),
                                 size = nrow(out1$release),
                                 prob = syn.prob,
                                 replace = TRUE)

    # sample synthetic data
    syn.data <- out1$release[syn.sample.indices, ]

    output <- lm((syn.data[,3]) ~ syn.data[,1] + syn.data[,2])

    store.syn.betas[i,1] <- output$coef[1]
    store.syn.betas[i,2] <- output$coef[2]
    store.syn.betas[i,3] <- output$coef[3]
}
Sys.time()
```

```
[1] "2019-04-02 18:27:29 EDT"

[1] "2019-04-02 18:48:01 EDT"
```

In [39]:
```
# df.betas <- as.data.frame(as.table(store.syn.betas))
# write.csv(df.betas, file = "synthetic_betas.csv")
```

Calculate MSEs and compare

In [35]:
```
print("TRUE BETAS")
true.output$coefficients
```

```
[1] "TRUE BETAS"
```

(Intercept)	6.07885271515284
data.educ.lm	0.219804343118345
data.age.lm	0.0181769925701837

In [36]:
```
MSE.real.data.betas <- data.frame(matrix(NA, ncol = 4, nrow = 3))
names.for.betas <- c('Intercept', 'Education', 'Age')

for(i in 1:3){

    # run MSE
    runMSE = calcMSE(true.output$coefficients[i], store.real.data.betas
[,i])

    # store bias, variance, MSE
    MSE.real.data.betas[i,] = c(names.for.betas[i], runMSE$bias, runMSE$
var, runMSE$MSE)
}
names(MSE.real.data.betas) <- c('Variable', 'Bias', 'Variance', 'MSE')
print("Bias, Variance and MSE due to SAMPLING")
MSE.real.data.betas
```

```
[1] "Bias, Variance and MSE due to SAMPLING"
```

Variable	Bias	Variance	MSE
Intercept	-0.000201349114693565	0.00105776464606583	0.00105780518753182
Education	-6.04896234721952e-05	4.72112441807278e-06	4.72478341262059e-06
Age	2.37711057228804e-05	1.22741812063061e-07	1.23306877530349e-07

In [37]:
```
MSE.syn.data.betas <- data.frame(matrix(NA, ncol = 4, nrow = 3))
names.for.betas <- c('Intercept', 'Education', 'Age')
for(i in 1:3){
    runMSE = calcMSE(true.output$coefficients[i], store.syn.betas[,i])
    MSE.real.data.betas[i,] = c(names.for.betas[i], runMSE$bias, runMSE$
var, runMSE$MSE)
}
names(MSE.syn.data.betas) <- c('Variable', 'Bias', 'Variance', 'MSE')
print("Bias, Variance and MSE from DP release of betas")
MSE.real.data.betas
```

```
[1] "Bias, Variance and MSE from DP release of betas"
```

Variable	Bias	Variance	MSE
Intercept	-0.398576981894228	0.0112761810549042	0.170139791550816
Education	0.207981775932976	2.65944491668371e-06	0.0432590785651515
Age	-0.18749454333937	5.87154202540599e-05	0.0352129192022931

The MSE of all coefficients is lower in the case of the bootstrapped betas than it is for the DP released betas. This follows as we've randomly sampled from the sample at hand. The MSE can be decomposed into bias square and variance, and we can see that the majority of mean squared error comes from the variance of the betas as compared to the squared bias. This makes sense as we expect very low bias, i.e. deviation from the true value.

In the case of the DP betas, we find that the bias is a lot larger than the variance and the contribution of the bias square to MSE is larger than that of the variance. As we have introduced noise into counts that impacts how we sample from the dataset, our coefficients will be further from the true values.

Overall, it's worth noting though that the MSE is still less than 1 for all coefficients from the regression on synthetic data.