Untitled

Albert Bertran Teresa Ricciardi

2023-05-28

In this laboratory session we are going to analyze three cases of data protection that can be used to compute the mean of a certain attribute and evaluate the accuracy of the resulting mean. After installing the R package called "LaplacesDemon" and loading the package into the R session, we read the CSV file "adult_train.csv" and store the data in a variable called "adult_train". During this laboratory we want to focus only on the attribute "age", so we create a new data frame called "adult" by selecting only the 'age' column from the adult_train data frame. We can also display the mean age of the original dataset, being 38.58.

```
install.packages("LaplacesDemon")
```

```
adult_train <- read.csv("./adult_train.csv")
library(LaplacesDemon)
```

```
## Warning: package 'LaplacesDemon' was built under R version 4.2.3
```

```
adult <- adult_train[c('age')]
mean(adult$age)
```

```
## [1] 38.58165
```

A Naïve DP microdata

After this we can start to perform the Laplace mechanism, using the function "laplace_mechanism1" that takes two parameters, 'x' and 'epsilon', where 'x' represents the input data and 'epsilon' represents the amount of privacy protection applied by the Laplace mechanism. Inside the function we define the sensitivity of the data, that is a measure of how much a function's output can change when the input database changes by a single entry. In this case we set the sensitivity to 125, because the oldest person to ever lived is 122 y.o., so a reasonable upper bound is 125. We also define the scale that is given by the ratio between the sensitivity and epsilon and determine the amount of noise to be added to the data. Then we draw a random sample from a Laplace distribution using the rlaplace() function from the "LaplacesDemon' ' package. The 'n' parameter is set to 1 to generate a single sample, 'loc' is set to 0 to center the distribution at 0, and 'scale' is set to the calculated scale factor. The resulting value represents the noise to be added to the data. The function returns the original input x plus the generated noise. This is the result of applying the Laplace mechanism to the data, which adds privacy protection by injecting noise to the output.

```
laplace_mechanism1 <- function(x, epsilon) {
  sensitivity <- 125
  scale <- sensitivity / epsilon
  noise <- rlaplace(n = 1, loc = 0, scale = scale)
  return(x + noise)
}
```

At this point we can focus on the first case of data protection, the Naive DP microdata. We start defining the dp_naive function that takes two parameters: adult and epsilon. Then we calculate the number of rows and columns in the adult data frame and assign them to the variables 'n' and 'd'. We create a matrix called "dp_data" with the same dimensions as the adult data frame. The matrix is initialized with zeros, and it will store the results of applying the Laplace mechanism to each row of the "adult" data frame. Finally we convert the "dp_data" matrix into a data frame using the "as.data.frame()" function and return it as the result of the "dp_naive" function. The new data frame contains the noisy versions of the rows.

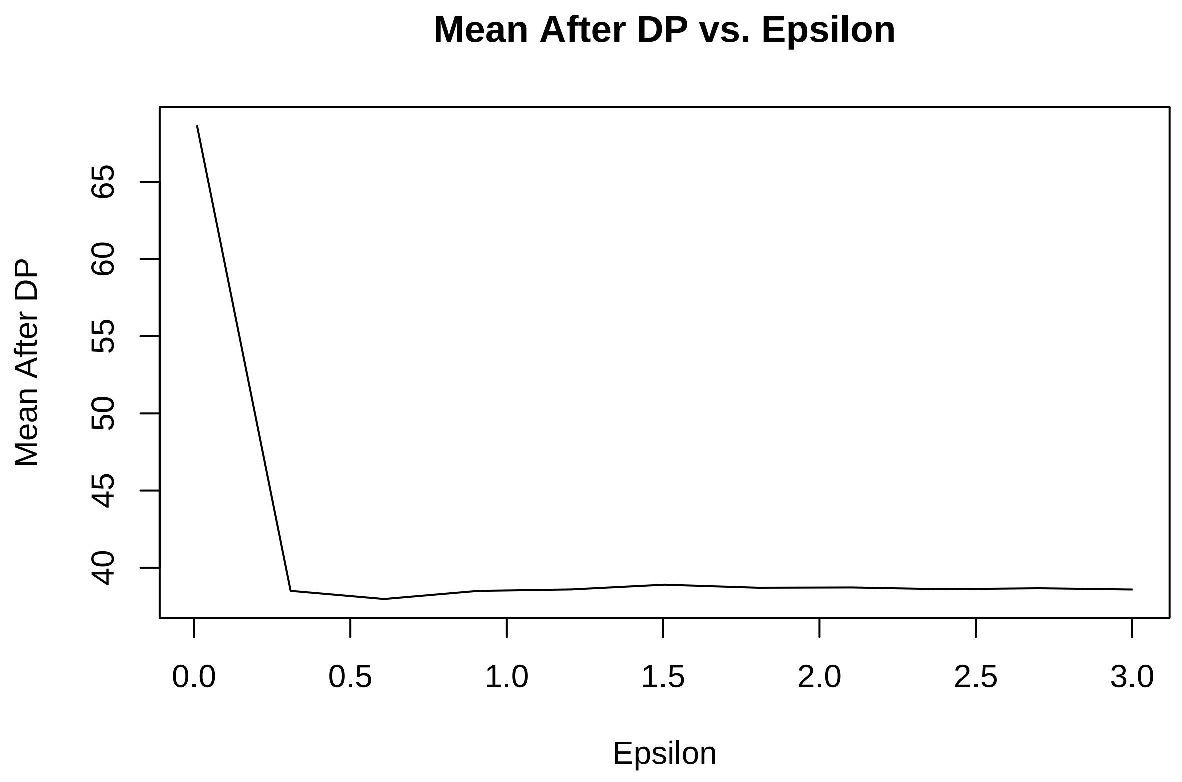
```r
dp_naive <- function(adult, epsilon) {
  n <- nrow(adult)
  d <- ncol(adult)
  dp_data <- matrix(0, nrow = n, ncol = d)
  for (i in 1:n) {
    dp_data[i] <- laplace_mechanism1(adult[i,], epsilon)
  }
  return(as.data.frame(dp_data))
}
```

In the following part of the code we start calculating the errors introduced by the Laplace mechanism when applied to the "adult" dataset with increasing values of epsilon (0.01-3). For each value of epsilon we take 25 repetitions. For each repetition we calculate the mean value of the 'V1' column in the "dp_data" data frame, representing the mean value after applying the Laplace mechanism and the mean value of the 'age' column in the original "adult" data frame, representing the mean value before applying the Laplace mechanism. We calculate the absolute difference between the mean values before and after applying the Laplace mechanism and store it in the "histogramE" vector. Instead, the "histogramM" vector stores the mean value after applying the Laplace mechanism.

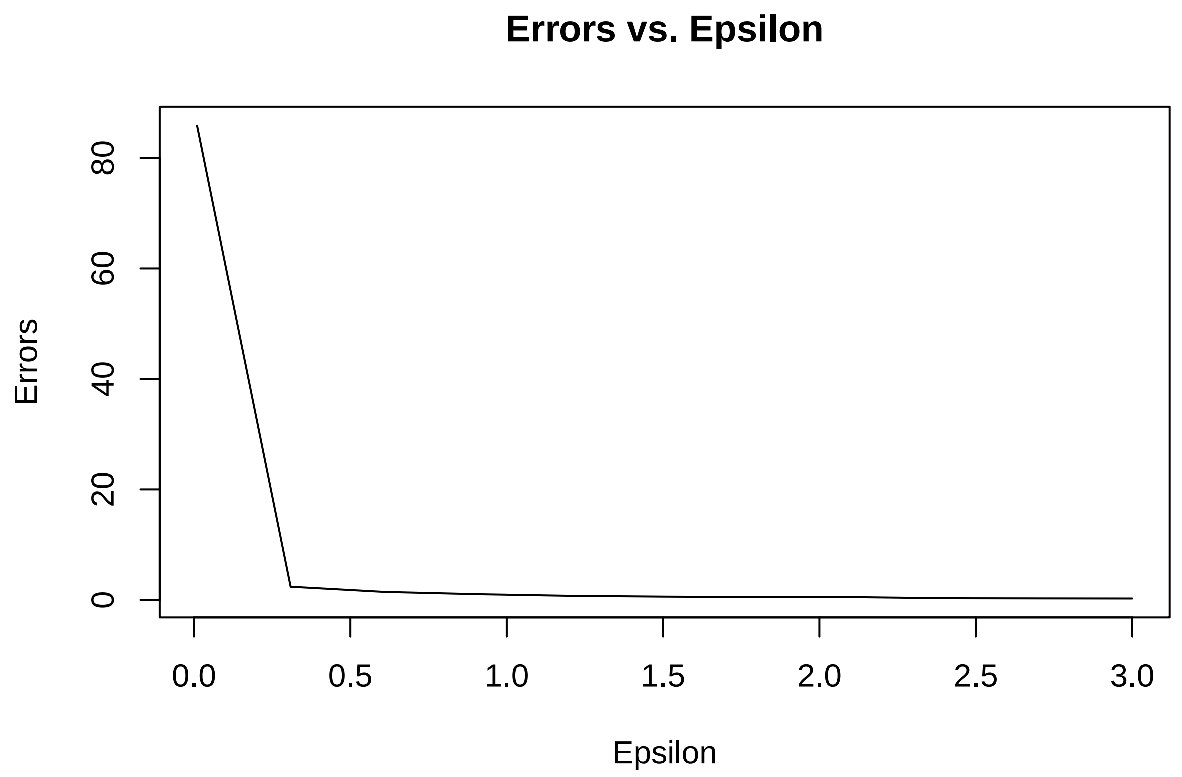
```r
epsilon <- 0.01
errors = c()
meanAfterDp = c()
for (i in 1:11) {
  histogramE = c()
  histogramM = c()
  for (j in 1:25) {
    dp_data <- dp_naive(adult, epsilon)
    mdp <- mean(dp_data$V1)
    morig <- mean(adult$age)
    histogramE[j] <- abs(mdp - morig)
    histogramM[j] <- mdp
  }
  e <- mean(histogramE)
  errors[i] <- e
  m <- mean(histogramM)
  meanAfterDp[i] <- m
  epsilon <- epsilon + 0.299
}

epsilon_values <- seq(0.01, 3, by = 0.299)
```

Now we can plot the results after applying the DP mechanism.

```r
plot(epsilon_values, meanAfterDp, type = "l", xlab = "Epsilon", ylab = "Mean After DP",
     main = "Mean After DP vs. Epsilon")
```



```
# Plot for errors
plot(epsilon_values, errors, type = "l", xlab = "Epsilon", ylab = "Errors",
     main = "Errors vs. Epsilon")
```



We can see in the plots above that as we increase the value of epsilon, the mean adjusts to the real value and the error decreases significantly. However note that since there is a part of randomization involved in the computation we had some different values in another executions, like that till epsilon 1 the mean value was not as adjusted as the one shown above. Even though when we keep increasing the epsilon the values adjust to the real ones

B Histograms

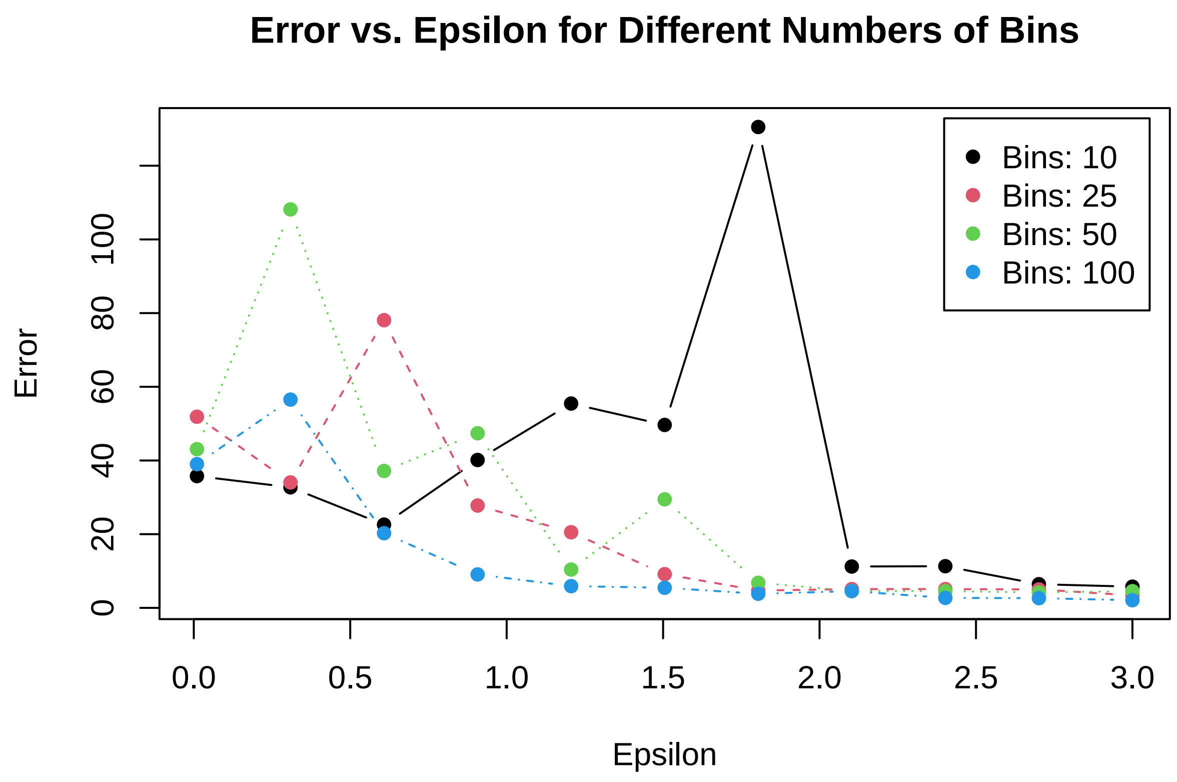
In this second part of the laboratory we define a function called "dp_histogram_mean" that calculates the differentially private mean using a histogram-based approach. The function takes three parameters: 'adult', 'epsilon', and 'num_bins', where 'num_bins' represents the number of bins to divide the data for the histogram.

```r
dp_histogram_mean <- function(adult, epsilon, num_bins) {
  n <- length(adult$age)

  #These lines calculate the minimum and maximum values of the age column in the adult data frame and a

  min_age <- min(adult$age)
  max_age <- max(adult$age)
  bin_width <- (max_age - min_age) / num_bins

  #In the following part of the code we implement different iterations to construct a histogram of the

  max_count <- 0
  histogram_counts <- rep(0, num_bins)

  for (i in 1:n) {
    individual_age <- adult$age[i]
    bin_index <- pmin(pmax(floor((individual_age - min_age) / bin_width) + 1, 1), num_bins)
    histogram_counts[bin_index] <- histogram_counts[bin_index] + 1
    max_count <- max(max_count, histogram_counts[bin_index])
  }

  noisy_counts <- histogram_counts + rlaplace(num_bins, scale = max_count / epsilon)

  sum_age <- 0
  total_count <- sum(noisy_counts)

  for (i in 1:num_bins) {
    bin_age <- min_age + (i - 0.5) * bin_width
    sum_age <- sum_age + noisy_counts[i] * bin_age
  }

  dp_mean <- sum_age / total_count

  return(dp_mean)
}
```

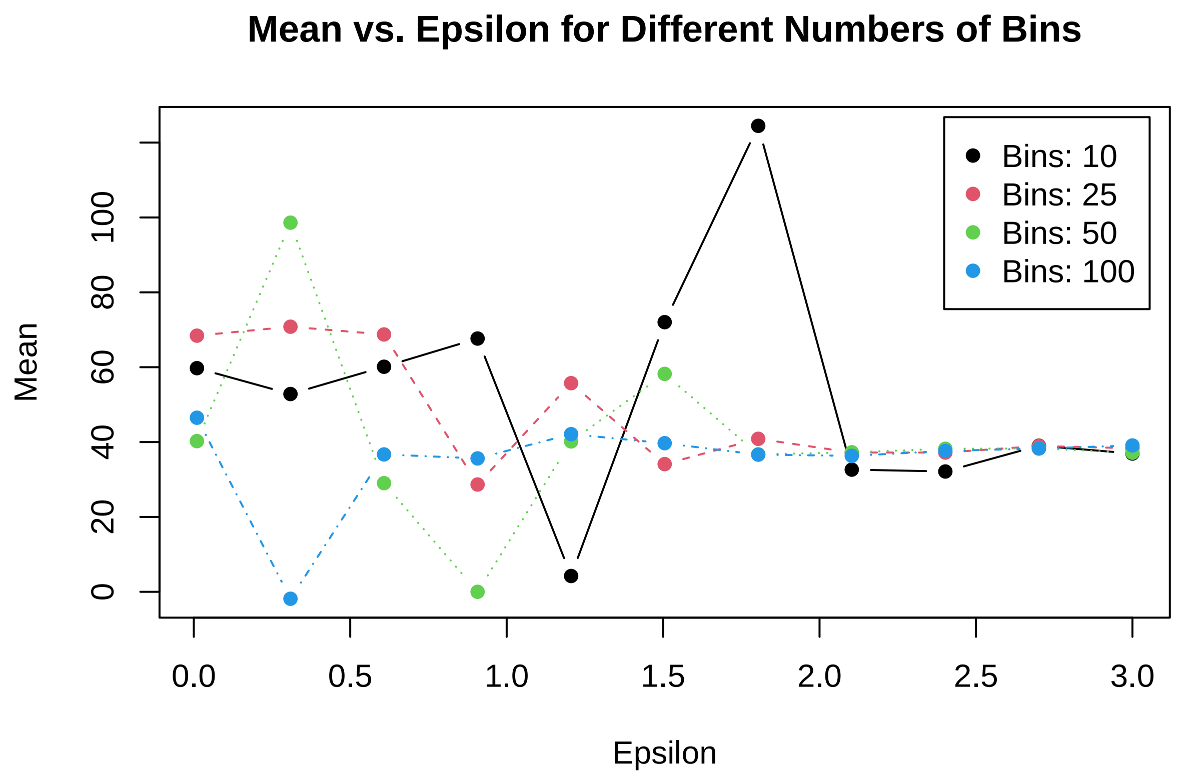
Then, in the main code, we construct the sequence with the possible values of epsilon, we set the number of iterations, as 25 and we set the list of bins, in our case, [10, 25, 50, 100]. Then we compute for each epsilon value, the mean value and error for each possible bin size and we then store the error and the mean value to be displayed in a plot.

```r
# Main
epsilon_values <- seq(0.01, 3, by = 0.299)
num_bins_list <- c(10, 25, 50, 100)
num_iterations <- 25

mean_values <- matrix(0, nrow = length(epsilon_values), ncol = length(num_bins_list))
error_values <- matrix(0, nrow = length(epsilon_values), ncol = length(num_bins_list))

for (i in 1:length(epsilon_values)) {
  epsilon <- epsilon_values[i]

  for (j in 1:length(num_bins_list)) {
    num_bins <- num_bins_list[j]

    histogram_errors <- matrix(0, nrow = num_iterations, ncol = 1)
    mean_after_dp_list <- matrix(0, nrow = num_iterations, ncol = 1)

    for (iter in 1:num_iterations) {
      dp_mean <- dp_histogram_mean(adult, epsilon, num_bins)
      mean_after_dp_list[iter] <- dp_mean
      orig_mean <- mean(adult$age)
      histogram_errors[iter] <- abs(dp_mean - orig_mean)
    }
    error_values[i, j] <- mean(histogram_errors)
    mean_values[i, j] <- mean(mean_after_dp_list)
  }
}
```

```
# Plot results
plot_labels <- paste("Bins:", num_bins_list)

matplot(epsilon_values, error_values, type = "b", pch = 16, xlab = "Epsilon", ylab = "Error",
        main = "Error vs. Epsilon for Different Numbers of Bins", col = 1:length(num_bins_list))
legend("topright", legend = plot_labels, col = 1:length(num_bins_list), pch = 16, inset = 0.02)
```



```
matplot(epsilon_values, mean_values, type = "b", pch = 16, xlab = "Epsilon", ylab = "Mean",
        main = "Mean vs. Epsilon for Different Numbers of Bins", col = 1:length(num_bins_list))
legend("topright", legend = plot_labels, col = 1:length(num_bins_list), pch = 16, inset = 0.02)
```



In the plots above, we can see the mean and the errors when increasing the epsilon value depending on the number of bins. Each number of bins is represented by a different color. As we can see, when we use 10 bins the results fluctuate a lot in the first values of epsilon, showing that when the bins are smaller the result is less accurate to the original representation. We can also see that 25 and 50 take the correct curve of values earlier than the 10 case. And we can see that when we use 100 bins the results are a lot more accurate even when epsilon is still quite small. It is also worth noting that since we added the computation of the bins, this was the mechanism that takes most time to compute.

C Summation and counting

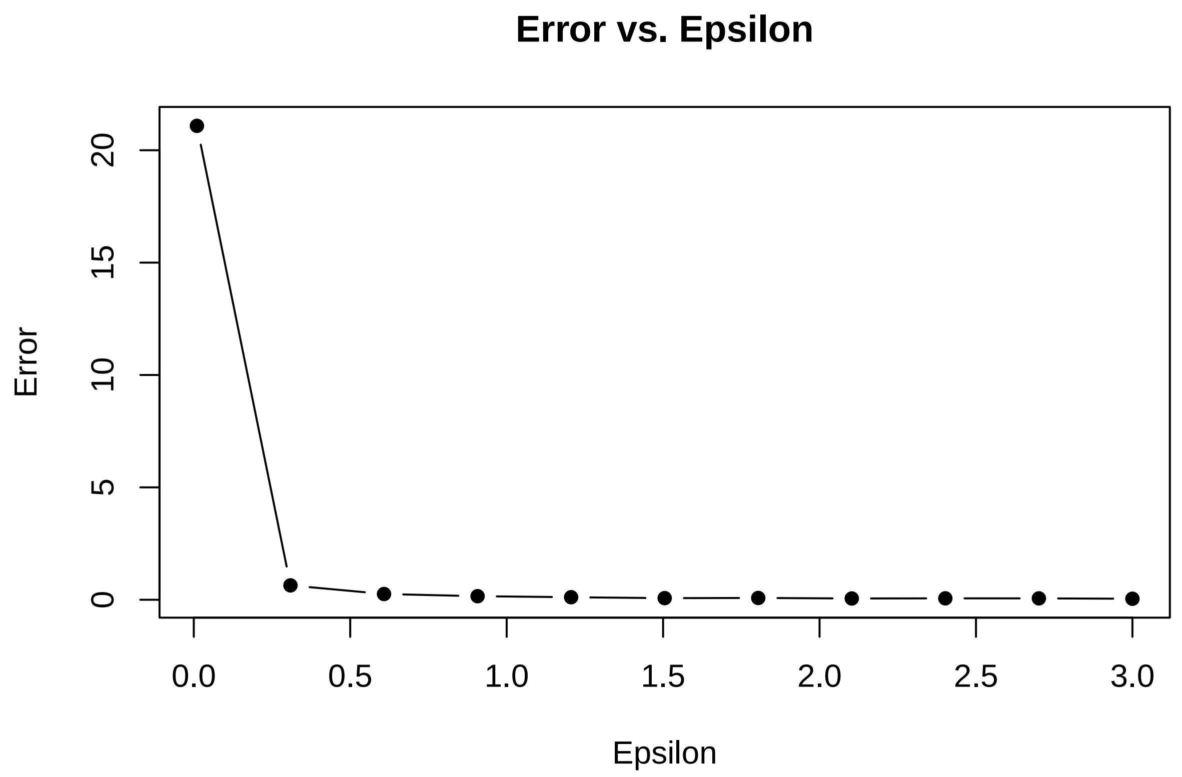
Finally, in this third exercise we will compute the DP for the summation and counting query. First we define the both functions corresponding to computing, first the DP of the summation query and second the one for the counting query.

```r
#Summation query
dp_summation <- function(data, epsilon) {
  sensitivity <- max(data) - min(data)
  scale <- sensitivity / epsilon
  noise <- rlaplace(n = 1, loc = 0, scale = scale)
  return(sum(data) + noise)
}

#Counting query
dp_counting <- function(data, epsilon) {
  sensitivity <- 125
  scale <- sensitivity / epsilon
  noise <- rlaplace(n = 1, loc = 0, scale = scale)
  return(length(data) + noise)
}
```

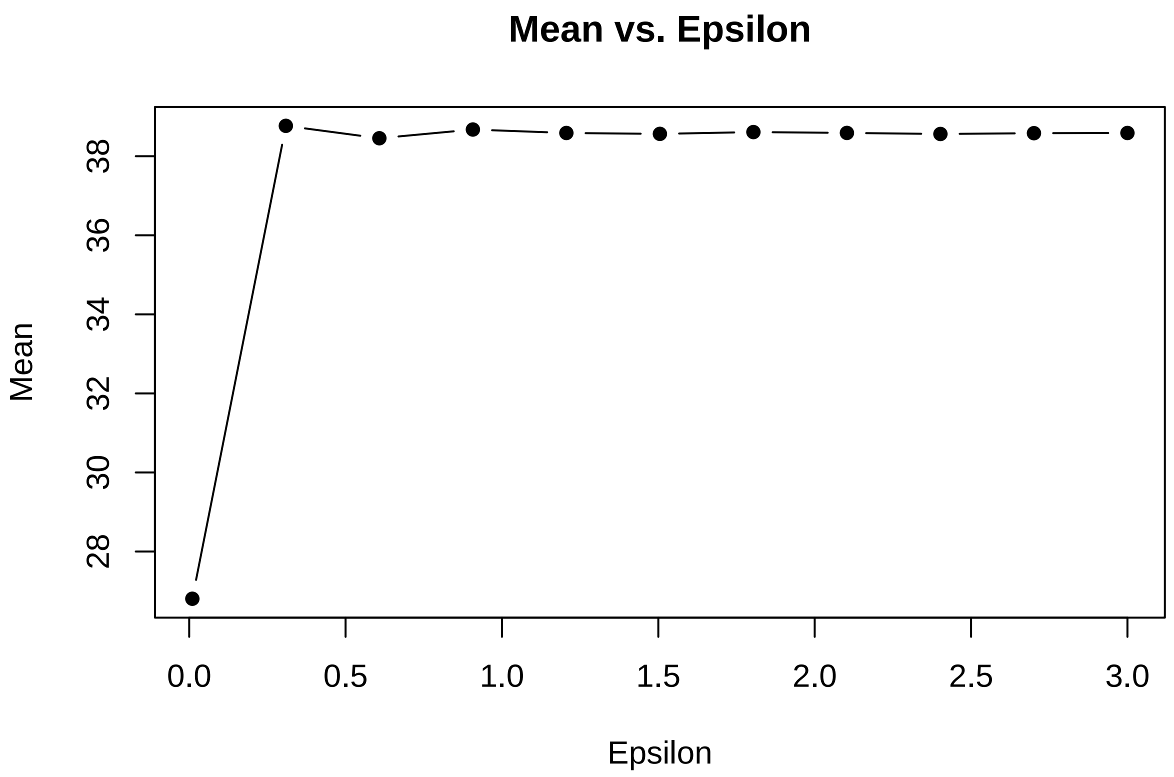
Finally, we initialize again the sequence for the epsilon value and we define the vectors that will store the mean values and the error. Then we iterate 25 times for each epsilon value and compute and store the mean to later on display it in the respective plots.

```
epsilon_values <- seq(0.01, 3, by = 0.299)
error_values <- rep(0, length(epsilon_values))
mean_values <- rep(0, length(epsilon_values))

for (i in 1:length(epsilon_values)) {
  epsilon <- epsilon_values[i]
  mean_results <- numeric(25)
  error_results <- numeric(25)

  for (j in 1:25) {
    dp_sum <- dp_summation(adult$age, epsilon)
    dp_count <- dp_counting(adult$age, epsilon)
    orig_mean <- mean(adult$age)

    dp_mean <- dp_sum / dp_count
    mean_results[j] <- dp_mean
    error_results[j] <- abs(dp_mean - orig_mean)
  }

  mean_values[i] <- mean(mean_results)
  error_values[i] <- mean(error_results)
}
```

```
# Plot results
plot(epsilon_values, error_values, type = "b", pch = 16, xlab = "Epsilon", ylab = "Error", main = "Erro
```



```
plot(epsilon_values, mean_values, type = "b", pch = 16, xlab = "Epsilon", ylab = "Mean", main = "Mean v
```



In this final exercise, we can see that the plots are quite similar with the first ones, as we increase the epsilon, the mean adjusts and the error reduces itself, however is worth noting that this last mechanism was the one that was achieving the most constant values over the several executions we did. Also note that this mechanism was the fastest one, while the others might last some minutes, this one is last a couple of seconds.