UC Davis STA 242 2015 Spring Assignment 5 [1]

Wenhao Wu, 998587583

May 31, 2015

1 Algorithm Design

1.1 Compute the Deciles

In order to compute the deciles of the total fare less the tolls, denoted as f_{net}, we count the occurence of each value of f_{net}. The benefits are:

- There are much more records than the possible values of f_{net} in the original data. Consequently, it **saves tremendous memory usage** by counting the occurence.
- This algorithm is highly **compatible with parallel processing**. We can keep multiple tables to count the occurence of each value of f_{net} for different data files, update these tables fully in parallel, and then merge these tables to compute the deciles.

In both of our implementations, we build a table to count the occurence for each pair of data files by updating it seqentially as we read in a new piece/bulk of record(s), then combine the 12 tables to compute the deciles.

1.2 Solve The Linear Regression

Denote the trip time as t and the surcharge as f_s, respectively. In the two regression tasks, the responses are denoted as $\mathbf{y}$, a n-by-1 vector of f_{net} in all records. In the first regresssion tasks, the predictors are denoted as $\mathbf{X}_1$, a n-by-2 matrix where the first column represents the t from all records and the second column is an all 1 vector. In the second regresssion tasks, the predictors are denoted as $\mathbf{X}_2$, a n-by-3 matrix where the first and the second columns represent the t, and f_s from all records and the third column is an all 1 vector. Theoretically, the coefficients of the linear model can be computed as [2]

$$\boldsymbol{\beta}_i = (\mathbf{X}_i^H \mathbf{X}_i)^{-1} \mathbf{X}_i^H \mathbf{y},\ i = 1, 2. \tag{1}$$

Apparently, the sufficient statistic for the linear regression tasks are $\mathbf{X}_i^H \mathbf{X}_i$ and $\mathbf{X}_i^H \mathbf{y}$, $i = 1, 2$ which has very low dimension. Moreover, these sufficient statistics can be updated sequentially as we read in a new piece/bulk of record(s), and are again highly compatible with parallel processing.
In both of our implementations, we update $\mathbf{X}_i^H \mathbf{X}_i$ and $\mathbf{X}_i^H \mathbf{y}$, $i = 1, 2$ sequentially for each pair of data files, then combine the 12 set of statistics by summing them up and solve the linear problem as in (1) to get the coefficients for the regression models.

2 Data Inspection, Pre-Processing and Extraction

Due to the limited hard drive space available on my workstation, I keep the original .zip files without decompressing them. Firstly we check that the "data" and "fare" files match each other row by row in the 3 index fields "medallion", " hack_license" and " pickup_datetime". To do so, we primarily make use of a combination of shell commands `unzip`, `cut`, `diff` , IO redirection and pipe commands to compare the 3 fields in each pair of files. (See **checkmatch.sh** in the Appendices.) We verified that the files indeed match in pairs.
During the inspection, we also notice that "trip_fare_8.csv.zip", "trip_data_9.csv.zip" and "trip_fare_9.csv.zip" contains duplicated .csv files, whcih are removed manually.

In both of our implementations, we build a "connection" to read in the output of shell pipe commands to extract the data. The shell command to extract " surcharge", "tolls_amount" and "total_amount" from the "fare" files is

```
unzip -cq ../data/trip_fare_n.csv.zip | cut -d , -f 7,10,11
```

According to the data file description [3], roughly 7.5% of all trips' "trip_time" is wrong so we take a safe approach to extract "pickup_datetime" and "dropoff_datetime" from the "data" files. The corresponding shell command is

```
unzip -cq ../data/trip_data_n.csv.zip | cut -d , -f 6,7
```

Later we take the differences between them as the actual trip time. Fortunately, both these two fields have a very neat format as "%Y-%m-%d %H:%M:%S" which can be easily processed.

3 Implementation in Python

Our first implementation is based on Python3. The pararllel processing is implemented with package "multiprocessing": we define a worker function `analyze_file()` to compute the count of occurence table and the sufficient statistics for the two linear regression tasks for a single pair of data/fair files. A total of 12 copies of this worker function are mapped to a pool of multiple processes and run in parallel. The results are then combined, from which the deciles are computed and the 2 linear regression problems are solved.
The worker function `analyze_file()` has a coroutine structure [4]: it is mainly composed of a "source" function `parse_file()` which read in one line from a pair of data/fare files, process it, and send the result to a "sink" function `accumulate_lines()`, which is in charge of updating the count of occurence table for the total amount less the toll and the sufficient statistics for the regressions.
In terms of data structure, the count of occurence table is updated as a python `dict` object and later converted to a pandas `Series` object to enable easy combination. The suffficient statistics are represented as numpy `ndarray` objects.

4 Implementation in R

Our second implementation is based on R. Similar to our first implementation, we define a worker function `analyzeFile()` to compute the count of occurence table and the sufficient statistics for the two linear regression tasks for a single pair of data/fair files, then use `parLapply()` to run it on a "cluster" for different files. After the 12 pairs of fare/data files are all processed. The results are then combined with function `reduceListSummaryNYCTaxi()` where the deciles are computed and the 2 linear regression problems are solved.
In the worker function `analyzeFile()`, instead of reading in 1 line from a pair of fare/data files at a time as in our first implementation, we use function `read.csv()` to read in a bulk of 500000 records as a data frame, and then update the count of occurence table and the sufficient statistics for regression. The benefits are twofold. Firstly, we can use R's function `table()` to count the occurence for this bulk of record and then update the overall count of occurence table implemented with package "hash", which prove to be more efficient than updating the table directly one line at a time. Secondly, this bulk-style update would result in more accuracy in computing the sufficient statistics theoretically.
In order to further speed up function `analyzeFile()`, we implement the function to update the sufficient statistics, `updateSuffStat()`, in C++.

5 Results

5.1 Deciles and Linear Regression Results

The deciles and the two linear regression results computed using our Python and R implementation are presented in Table 1 and Table 2, respectively. As we can see, the results of the two implementaions are very close to each other. The difference is likely due to numerical accuracy issues. We believe that the R implementation is more precise.

Table 1: Deciles of the total fare less the tolls. Implementation 1, 2 are based on python and R, respectively.

	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
1	-1430.0	6.0	7.5	8.5	9.75	11.0	13.0	15.0	18.5	26.120000000000001	685908.09999999998
2	-1430.00	6.00	7.50	8.50	9.75	11.00	13.00	15.00	18.50	26.12	685908.10

Table 2: Linear regression results

	Linear model 1		Linear model 2		
	trip time	intercept	trip time	surcharge	intercept
Python	2.02064511e-03	1.30134743e+01	2.02196225e-03	3.04104587e-01	1.29153642e+01
R	0.00202051	13.01345450	0.002021825	0.303964991	12.915390308

5.2 Running Time Comparison

We test the running time of both of our implementations using different number of processes. In there tests, we use a Dell Precision T1700 workstation equipped with 16GB DDR3 RAM, a Core i7-4790K CPU and a Samsung 850 PRO SSD in Ubuntu 14.04 OS. The elapsed time are plotted in Fig. 1. Without parallel processing, the Python and R implementaions take 3861 and 1231 seconds to go through all the 12 pairs of files. With a pool/cluster of size 12, the elased times for both implementations reduce to 1144 and 344 seconds, corresponding to a maximum speed-up of 3.58 and 3.34, respectively. We note that, however, with a pool/cluster of size 6, we can achieve approximately the same speed-up.

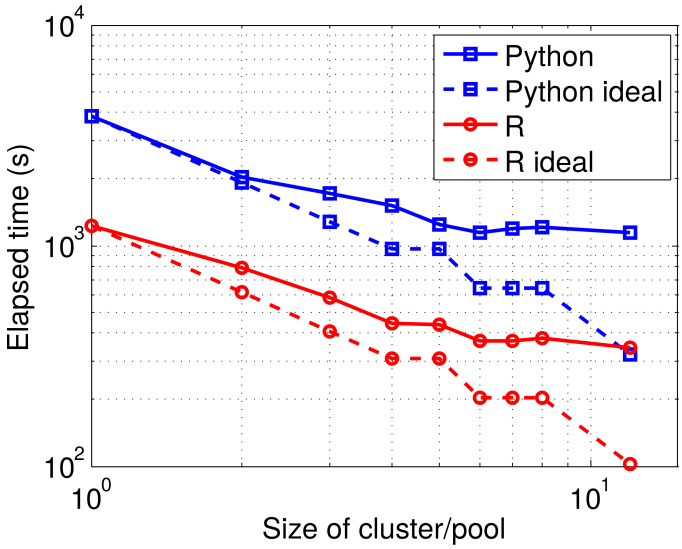

Figure 1: The elased time of both implementations vs different sizes of the pool/cluster N. The ideal times are computed as $t_{\text{ideal}}(N) = t_1/12 \times \lceil 12/N \rceil$, where t_1 is the elapsed time when $N = 1$. This equation is based on the assumption that there is no I/O bottleneck, all files take equal amount of time to process, and the time to setup the pool/cluster and combine results returned by multiple worker functions is negligible.

6 Comments

Here we make a brief comparison between the two implementations:

- In terms of performance, our second implementation based on R clearly wins against the first implementation based on Python, although both implementations result in a decent short running time considering we are only using a small form factor workstation.

- In terms of programming time, it turns out that our second implementation takes longer than the first implementation as shown in the commit history. In fact, I start working on implementation 2 first and then implmentation 1. Consequently, the programming time difference is mainly due to the fact that we ecountered some problem playing with the original data when firstly working with implementation 2 and the author is not familiar with R. Since both implementations are based on a similar, highly specific algorithm design, it is expected that in capable hands the programming time would not be so different.
- Despite that our first implementation takes much longer time, it uses very little amount of memory. Should we reduce the bulk size in the second implementation to reduce the memory usage to the amoung of the first implmentation, it is going to run even more slowly. Consequently, if we are on a highly memory-limited platform (which is rarely the case), the first implementation is more appropriate. Otherwise the second implementation will be more appropriate due to its advantage in speed.

References

[1] Wenhao Wu. STA 242 Assignment 5: Working with "Big Data". `git@bitbucket.org:shasqua/stat242_2015_assignment5.git`, 2015. [Online; accessed 30-May-2015].

[2] Trevor Hastie, Robert Tibshirani, Jerome Friedman, T Hastie, J Friedman, and R Tibshirani. *The elements of statistical learning*, volume 2. Springer, 2009.

[3] Dan Work. 2010-2013 New York City Taxi Data. `http://publish.illinois.edu/dbwork/open-data/`, 2014. [Online; accessed 30-May-2015].

[4] David Beazley. A Curious Course on Coroutines and Concurrency. `http://www.dabeaz.com/coroutines/`, 2009. [Online; accessed 30-May-2015].

Appendix: Source Files

Verify row-by-row matching between fare/data files: checkmatch.sh

```
#!/bin/bash

for idx in $(seq $1 $2)
do
  filename_data="trip_data_${idx}.csv.zip"
  filename_fare="trip_fare_${idx}.csv.zip"

  if [ ! -f $filename_data ]; then
    echo "${filename_data} does not exist!"
  elif [ ! -f $filename_fare ]; then
    echo "${filename_fare} does not exist!"
  else
    echo "Comparing ${filename_data} and ${filename_fare} ..."

    if diff -w <(unzip -cq $filename_data | cut -d , -f 1,2,6) <(unzip -cq
        $filename_fare | cut -d , -f 1,2,4) > /dev/null; then
      echo "${filename_data} and ${filename_fare} match!"
    else
      echo "${filename_data} and ${filename_fare} do not match!"
    fi
  fi
done
exit 0
```

Implementation 1: implementation1.py

```
#!/usr/bin/env python3

from subprocess import Popen, PIPE
from datetime import datetime
import numpy as np
import pandas as pd
from pandas import Series
import time
from multiprocessing import Pool

def coroutine(func):
    def start(*args,**kwargs):
        cr = func(*args,**kwargs)
        next(cr)
        return cr
    return start

def parse_file(idx_file, path, accumulator):
    cmd_parse_data = "unzip -cq {0}/trip_data_{1}.csv.zip | cut -d , -f 6,7"
       .format(path, idx_file)
    cmd_parse_fare = "unzip -cq {0}/trip_fare_{1}.csv.zip | cut -d , -f
       7,10,11".format(path, idx_file)

    format_time = '%Y-%m-%d %H:%M:%S'

    pipe_data = Popen(cmd_parse_data, shell=True, stdout=PIPE)
```

```
    pipe_fare = Popen(cmd_parse_fare, shell=True, stdout=PIPE)

    for (line_data, line_fare) in zip(pipe_data.stdout, pipe_fare.stdout):
        try:
            pickup_dropoff = line_data.strip().decode("utf-8").split(',')
            fares_str = line_fare.strip().decode("utf-8").split(',')

            trip_time = datetime.strptime(pickup_dropoff[1], format_time) -
                datetime.strptime(pickup_dropoff[0], format_time)
            fares = [float(fare) for fare in fares_str]

            accumulator.send((trip_time.total_seconds(), fares[2] - fares
                [1], fares[0]))
        except:
            continue

    accumulator.close()
    return

@coroutine
def accumulate_lines(count_fare, mat_reg1_XX_XY, mat_reg2_XX_XY, idx_file,
    verbose):
    try:
        while True:
            time_fares = (yield)
            #print(time_fares)

            # Update the count of occurence for the total fare less toll
            count_fare[time_fares[1]] = count_fare.get(time_fares[1], 0) + 1

            # Update the sufficient statistics
            mat_reg1_XX_XY[0, 0] += time_fares[0] ** 2
            mat_reg1_XX_XY[0, 1] += time_fares[0]
            mat_reg1_XX_XY[1, 1] += 1
            mat_reg1_XX_XY[0, 2] += time_fares[0] * time_fares[1]
            mat_reg1_XX_XY[1, 2] += time_fares[1]

            mat_reg2_XX_XY[0, 1] += time_fares[0] * time_fares[2]
            mat_reg2_XX_XY[1, 1] += time_fares[2] ** 2
            mat_reg2_XX_XY[1, 2] += time_fares[2]
            mat_reg2_XX_XY[1, 3] += time_fares[2] * time_fares[1]

    except GeneratorExit:
        mat_reg1_XX_XY[1, 0] = mat_reg1_XX_XY[0, 1]

        mat_reg2_XX_XY[0, 0] = mat_reg1_XX_XY[0, 0]
        mat_reg2_XX_XY[0, 2] = mat_reg1_XX_XY[0, 1]
        mat_reg2_XX_XY[1, 0] = mat_reg2_XX_XY[0, 1]
        mat_reg2_XX_XY[2, 0] = mat_reg2_XX_XY[0, 2]
        mat_reg2_XX_XY[2, 1] = mat_reg2_XX_XY[1, 2]
        mat_reg2_XX_XY[2, 2] = mat_reg1_XX_XY[1, 1]

        mat_reg2_XX_XY[0, 3] = mat_reg1_XX_XY[0, 2]
        mat_reg2_XX_XY[2, 3] = mat_reg1_XX_XY[1, 2]
```

```
        if (verbose):
            print("Processing␣data/fare␣{0}␣completed!".format(idx_file))
    return

def analyze_file(arg_list):
    idx_file = arg_list[0]
    path = arg_list[1]
    verbose = arg_list[2]
    mat_reg1_XX_XY = np.zeros((2, 3))
    mat_reg2_XX_XY = np.zeros((3, 4))
    count_fare = dict()

    # Hook everything up
    parse_file(idx_file, path, accumulate_lines(count_fare, mat_reg1_XX_XY,
       mat_reg2_XX_XY, idx_file, verbose))

    # Return a pandas Series
    count_fare = Series(count_fare)

    return([count_fare, mat_reg1_XX_XY, mat_reg2_XX_XY])


if __name__ == "__main__":
    path = "../data"
    idxs_file = range(1, 13)
    n_file = len(idxs_file)
    n_process = 12

    arg_lists = list(zip(idxs_file, [path] * n_file, [True] * n_file))

    # Parallel run
    start_time = time.time()
    with Pool(processes = n_process) as pool:
        results = pool.map(analyze_file, arg_lists)

        # Reduce the results
        count_fare = Series()
        mat_reg1_XX_XY = np.zeros((2, 3))
        mat_reg2_XX_XY = np.zeros((3, 4))
        for result in results:
            count_fare = count_fare.add(result[0], fill_value = 0)
            mat_reg1_XX_XY += result[1]
            mat_reg2_XX_XY += result[2]

        # Compute the deciles
        cdf = np.cumsum(count_fare) / np.sum(count_fare)
        deciles = [cdf[cdf >= p].index[0] for p in np.arange(0, 1.05, 0.1)]

        # Solve the regressions
        coeff1 = np.linalg.solve(mat_reg1_XX_XY[:, 0:2], mat_reg1_XX_XY[:,
           2])
        coeff2 = np.linalg.solve(mat_reg2_XX_XY[:, 0:3], mat_reg2_XX_XY[:,
           3])
```

```
print("Deciles of the total amount less toll:")
print(deciles)
print("Linear model of the total amount less the tolls versus trip
    time:")
print(coeff1)
print("Linear model of the total amount less the tolls versus
    surcharge and trip time:")
print(coeff2)
print("--- %s seconds ---" % (time.time() - start_time))
```

Implementation 2: implementation2.R

```
rm(list = ls())

library(NYCTaxi)
library(parallel)

idxs <- seq(12) # The indices of the files to be analyzed
path <- "../data"
size_bulk <- 500000L
size_cluster <- 12

# The serial version
t_serial <- system.time(list_sum_serial <- lapply(idxs, analyzeFile, path,
    size_bulk, TRUE))

# The parallel version
cl <- makeCluster(size_cluster)
t_parallel <- system.time(list_sum_parallel <- parLapply(cl, idxs,
    analyzeFile, path, size_bulk))
stopCluster(cl)

results_serial <- reduceListSummaryNYCTaxi(list_sum_serial)
print(results_serial)

results_parallel <- reduceListSummaryNYCTaxi(list_sum_parallel)
print(results_parallel)
print(t_serial)
print(t_parallel)
```

Implementation 2: NYCTaxi.R

```
#' The function to analyze a single pair of files
#'
#' Return a SummaryNYCTaxi S3 class object summarizing one pair of data and
    fare file, which contains the count of occurence for each value of fare
    amount less the tolls, and the sufficient statistics for the 2 regression
     tasks
#'
#' @import hash
#' @param idx the index of the file, ranging from 1 to 12
#' @param path the path of the directory holding the data and fare files
#' @param size_bulk the size of each bulk to process, limited by memory size
#' @param verbose whether to prompt an information when the processing is
    done
```

```
#' @param verbose_debug whether to print the error message when reaching the
    end of file (or potentially other error message)
#' @return a SummaryNYCTaxi S3 class object
#' @export
analyzeFile <- function(idx, path, size_bulk = 500000L, verbose = FALSE,
    verbose_debug = FALSE) {
  if (verbose) {
    message(paste("Processing␣data/fare", idx, "..."))
  }

  # Check whether the files exist
  filename_data <- paste(path, "/", "trip_data_", idx, ".csv.zip", sep = "")
  filename_fare <- paste(path, "/", "trip_fare_", idx, ".csv.zip", sep = "")

  if (!any(file.exists(filename_data, filename_fare))) {
    stop("Both␣files␣do␣not␣exist!")
  }

  cmd_parse_data <- paste("unzip␣-cq␣", filename_data, "␣|␣cut␣-d␣,␣-f␣6,7",
      sep = "")
  cmd_parse_fare <- paste("unzip␣-cq␣", filename_fare, "␣|␣cut␣-d␣,␣-f␣
     7,10,11", sep = "")

  connection_data = pipe(cmd_parse_data, 'r') # open a connection to the
     file
  connection_fare = pipe(cmd_parse_fare, 'r')
  readLines(connection_data, 1, skipNul = TRUE) # Skip the header line
  readLines(connection_fare, 1, skipNul = TRUE)

  pattern_time <- "%Y-%m-%d␣%H:%M:%OS" # The pattern of the pickup and
     dropoff time
  hist <- hash::hash() # The hash table to record the occurence of each fare
      less toll

  # The matrix to be updated iteratively based on bulks of records, the 2-by
     -2 (3-by-3) matrix X^HX and the 2-by-1 (3-by-1) vector X^HY, which is
     concatenated into a single 2-by-3 (3-by-4) matrix
  mat_reg1_XX_XY <- matrix(0, nrow = 2, ncol = 3) # Fare amount less the
     tolls vs trip time
  mat_reg2_XX_XY <- matrix(0, nrow = 3, ncol = 4) # Fare amount less the
     tolls vs trip time and surcharge

  # Start to go through the data file bulk by bulk
  tryCatch(
    {
      while (TRUE) {
        pickup_dropoff <- read.csv(connection_data, nrow=size_bulk, header=
           FALSE, stringsAsFactors=FALSE)
        fares <- read.csv(connection_fare, nrow=size_bulk, header=FALSE,
           stringsAsFactors=FALSE)

        # build a data frame and remove the rows with NAs
        pickup_dropoff <- data.frame(lapply(pickup_dropoff, strptime,
           pattern_time))
        fares <- data.frame(lapply(fares, as.numeric))
```

```
        bulk <- na.omit(cbind(pickup_dropoff, fares))

        # process data here
        trip_time <- difftime(bulk[,2], bulk[,1], units="secs")
        fare_less_toll <- bulk[,5] - bulk[,4]

        # Update the decile using a histogram (package "hash")
        hist_bulk <- table(fare_less_toll)
        value <- names(hist_bulk)
        count <- as.vector(hist_bulk)
        for (i in seq_along(hist_bulk)) {
          key <- value[i]
          (hash::has.key(key, hist)) || (hist[[key]] <- 0)
          hist[[key]] <- hist[[key]] + count[i]
        }

        # Update the 2 matrices for regression
        mat_reg1_XX_XY <- updateSuffStat(mat_reg1_XX_XY, fare_less_toll,
            matrix(trip_time))
        mat_reg2_XX_XY <- updateSuffStat(mat_reg2_XX_XY, fare_less_toll,
            cbind(trip_time, bulk[,3]))
      }

    },
    error=function(cond) {
      if (verbose_debug) {
        message("Appears to be at the end of file")
        message("Here's the original error message:")
        message(paste(cond))
      }
      return()
    },
    finally={
      close(connection_data)
      close(connection_fare)
    }
  )

  if (verbose) {
    message(paste("Processing data/fare", idx, "completed!"))
  }

  summary_NYCTaxi <- structure(list(count_occurence = hist, mat_reg1_XX_XY =
      mat_reg1_XX_XY, mat_reg2_XX_XY = mat_reg2_XX_XY), class = "
     SummaryNYCTaxi")
  return(summary_NYCTaxi)
}

#' The function to reduce a list of SummaryNYCTaxi class object
#'
#' Combine the count_occurence hash tables from the list and evaluate the
    deciles. Sum the sufficient statistics for the 2 regression tasks and
    solve the linear regression problems
#' @import hash
#' @import Hmisc
```

```
#' @param list_SummmaryNYCTaxi a list of SummaryNYCTaxi class object
    computed by apply the analyzeFile() function
#' @return a list containing the deciles and 2 sets of linear regression
    coefficients: total amount less the fee vs trip_time (and surcharge)
#' @export
reduceListSummaryNYCTaxi <- function(list_SummmaryNYCTaxi) {
  # Reduce the hash tables and compute the decile
  count_occurence_all <- hash::hash()
  for (i in seq_along(list_SummmaryNYCTaxi)) {
    for (key in hash::keys(list_SummmaryNYCTaxi[[i]]$count_occurence)) {
      (hash::has.key(key, count_occurence_all)) || (count_occurence_all[[key
          ]] <- 0)
      count_occurence_all[[key]] <- count_occurence_all[[key]] + list_
          SummmaryNYCTaxi[[i]]$count_occurence[[key]]
    }
  }
  deciles <- Hmisc::wtd.quantile(as.numeric(hash::keys(count_occurence_all))
      , weights = hash::values(count_occurence_all), probs=seq(0, 1, by=0.1))

  # Reduce the sufficient statistics
  mat_reg1_XX_XY_all <- matrix(0, nrow = 2, ncol = 3)
  mat_reg2_XX_XY_all <- matrix(0, nrow = 3, ncol = 4)
  for (i in seq_along(list_SummmaryNYCTaxi)) {
    mat_reg1_XX_XY_all <- mat_reg1_XX_XY_all + list_SummmaryNYCTaxi[[i]]$mat
        _reg1_XX_XY
    mat_reg2_XX_XY_all <- mat_reg2_XX_XY_all + list_SummmaryNYCTaxi[[i]]$mat
        _reg2_XX_XY
  }
  coeff1 <- solve(mat_reg1_XX_XY_all[, 1 : 2], mat_reg1_XX_XY_all[, 3])
  coeff2 <- solve(mat_reg2_XX_XY_all[, 1 : 3], mat_reg2_XX_XY_all[, 4])
  return(list(deciles = deciles, coeff_triptime = coeff1, coeff_triptime_
      surcharge = coeff2))
}

.onUnload <- function (libpath) {
  library.dynam.unload("NYCTaxi", libpath)
}
```

Implementation 2: NYCTaxi.h

```
#ifndef NYCTAXI
#define NYCTAXI

#include <Rcpp.h>
#include <string>

using namespace Rcpp;

//' Function to update the sufficient statistics of linear regression based
    on a bulk of data
//'
//' The sufficient statistics of the linear regression is recorded as a p-by
    -(p+1) matrix which is the row concatenation of x^Hx and x^Hy where p is
    the number of predictors (including constant 1).
//' @param xx_xy a p-by-(p+1) matrix, [1:p, 1:p] represents the current
    value of x^Hx and [p+1, 1:p] represents the current value of x^Hy
```

```
//' @param y a n-by-1 vector, the bulk of observatons
//' @param x_less_ones a n-by-(p-1) matrix, the bulk of predictors excluding
    1
//' @return the updated sufficient statistic xx_xy
//' @export
// [[Rcpp::export]]
NumericMatrix updateSuffStat(NumericMatrix xx_xy, NumericVector y,
   NumericMatrix x_less_ones);

#endif
```

Implementation 2: NYCTaxi.cpp

```
#include "NYCTaxi.h"
using namespace Rcpp;

NumericMatrix updateSuffStat(NumericMatrix xx_xy_input, NumericVector y,
   NumericMatrix x_less_ones)
{

  int p = xx_xy_input.nrow(); // Number of predictors
  int n = y.size(); // Number of observations
  NumericMatrix xx_xy(p, p + 1);

  // Check the dimension of the input arguments
  if ((xx_xy.ncol() != p + 1) || (x_less_ones.ncol() != p - 1) || (
     x_less_ones.nrow() != n))
  {
    stop("Size␣of␣input␣arguments␣must␣be␣consistent");
  }

  // Update x^Hx
  for (int r = 0; r < p - 1; r++)
  {
    for (int c = r; c < p - 1; c++)
    {
      for (int i = 0; i < n; i++)
      {
        xx_xy(r, c) += x_less_ones(i, r) * x_less_ones(i, c);
      }
    }
    for (int i = 0; i < n; i++)
    {
      xx_xy(r, p - 1) += x_less_ones(i, r);
    }
  }
  xx_xy(p - 1, p - 1) += n;
  for (int r = 1; r < p; r++)
  {
    for (int c = 0; c < r; c++)
    {
      xx_xy(r, c) = xx_xy(c, r);
    }
  }

  // Update x^Hy
```

```
  for (int r = 0; r < p - 1; r++)
  {
    for (int i = 0; i < n; i++)
    {
      xx_xy(r, p) += x_less_ones(i, r) * y[i];
    }
  }
  for (int i = 0; i < n; i++)
  {
    xx_xy(p - 1, p) += y[i];
  }

  // Update the cumulative x^Hx and x^Hy in this way for precision reason
  NumericVector::iterator itr_bulk;
  NumericVector::iterator itr_cum = xx_xy_input.begin();
  for (itr_bulk = xx_xy.begin(); itr_bulk != xx_xy.end(); itr_bulk++,
      itr_cum++)
  {
    *itr_bulk += *itr_cum;
  }
  return(xx_xy);
}
```