Introduction of efficient feature engineering with data.table in R

Sheng Ming

April 4, 2018

```r
knitr::opts_chunk$set(cache = TRUE, message = FALSE, warning = FALSE)
```

This tutorial will briefly introduce how to use the `data.table` package in R to implement efficient feature engineering functions for a dataset sampled from a Kaggle competition about online advertising fraud detection (https://www.kaggle.com/c/talkingdata-adtracking-fraud-detection), while the equivalent sql code with sqldf package is used as the benchmark. The tutorial is written by USC Marshall Masters of Business Analytics (MBSA) alumnus Sheng Ming (https://www.linkedin.com/in/ShengMing26) for Dr. Xin Tong's (https://sites.google.com/site/xintonghomepage/) DSO 530 class Machine Learning Methods.

As implemented in the C language with the optimized operations and memory usage, `data.table` provides with a new class for tabular data (like `data.frame`) as well as a full set of data manipulation functions (like `dplyr`). Depending on the volume of data you have, data.table operations could be 10 to 100 times faster than those on a data.frame, even you're using popular packages like dplyr or sqldf. To get familiar with the syntax of `data.table`, I strongly recommend you to study the package description in R carefully by typing `?data.table`. Besides, there're many online tutorials you could refer to. Here is one of those: https://cran.r-project.org/web/packages/data.table/vignettes/datatable-intro.html.

In this tutorial, I subset 100k consecutive rows from the full TalkingData dataset which you could download from the kaggle website. You could download this 100k sample data from BB to reproduce all the results below.

Loading the data

The **fread** provided by `data.table` is an efficient function to load large data files. On my 2015 Macbook pro, it takes around 5 minutes to read the entire 185M rows of data into my R workspace, and the data is already a data.table object once loaded.

```r
library(data.table)
library(lubridate)
train = fread('train100k.csv', stringsAsFactors = F)
# See it has a data.table class
str(train)
```

```
## Classes 'data.table' and 'data.frame':   100000 obs. of  8 variables:
##  $ ip             : int  2698 5729 122891 105433 191894 66120 81356 63883 50027 2698 ...
##  $ app            : int  25 2 3 15 20 2 64 15 2 9 ...
##  $ device         : int  1 1 1 2 2 1 1 2 1 1 ...
##  $ os             : int  30 19 35 25 15 13 19 19 32 30 ...
##  $ channel        : int  259 237 280 245 259 477 459 3 477 442 ...
##  $ click_time     : chr  "2017-11-06 16:21:51" "2017-11-06 16:21:51" "2017-11-06 16:21:51" "2017-11-0
##  $ attributed_time: chr  "" "" "" "" ...
##  $ is_attributed  : int  0 0 0 0 0 0 0 0 0 0 ...
##  - attr(*, ".internal.selfref")=<externalptr>
```

```r
# We'll not use the attribute_time column
train = train[, c(1:6,8)]
# Convert click_time to the right timestamp format
```

```r
train$click_time = ymd_hms(train$click_time)
# Add a record column which serves as the unique identifier for each row
# and indicates the chronological order, which is very important
train$record = as.integer(row.names(train))
str(train)
```

```
## Classes 'data.table' and 'data.frame':   100000 obs. of  8 variables:
##  $ ip           : int  2698 5729 122891 105433 191894 66120 81356 63883 50027 2698 ...
##  $ app          : int  25 2 3 15 20 2 64 15 2 9 ...
##  $ device       : int  1 1 1 2 2 1 1 2 1 1 ...
##  $ os           : int  30 19 35 25 15 13 19 19 32 30 ...
##  $ channel      : int  259 237 280 245 259 477 459 3 477 442 ...
##  $ click_time   : POSIXct, format: "2017-11-06 16:21:51" "2017-11-06 16:21:51" ...
##  $ is_attributed: int  0 0 0 0 0 0 0 0 0 0 ...
##  $ record       : int  1 2 3 4 5 6 7 8 9 10 ...
##  - attr(*, ".internal.selfref")=<externalptr>
```

As we see, all the variables are categorical and are given in the format of IDs (app ID, device ID, etc. see variable definitions at https://www.kaggle.com/c/talkingdata-adtracking-fraud-detection/data). Some types of the features we could build based on these categories are as below:

1. **Mean Encoding**: 1-to-1 mapping between each category and the target variable based on historical data (i.e. the average value of the target).
2. **Simple time-window counts**: something like how many times this IP appeared in the last 60 seconds.
3. **Interactive time-window counts**: something like how many different devices associated with this IP appeared in the last 60 seconds.

```
p.s. "simple time-window counts" and "interactive time-window counts" are just what I
call such variables but not formal terminologies.
```

In this tutorial, I'll focus on how to create the time-window variables with data.table in an efficient way.

Creating time-window variables with sqldf

Before we introduce the data.table method, let's first start with the sql method as it's more readable and understandable. The sql code mainly serves two purposes here:

1. The sqldf results are used to verify the data.table results and help to debug the data.table code if anything goes wrong;
2. The sqldf performance are used to benchmark the data.table performance so you can see how the data.table method improves the process.

For the first example, I'll apply a 60-second time window on the attribute IP and calculate four metrics for each row:

1. How many times this IP appeared in the last 60 seconds;
2. How many times this IP appeared in the last 30 seconds;
3. How many different apps associated with this IP in the last 60 seconds;
4. How many different apps associated with this IP in the last 30 seconds.

```r
library(sqldf)
# Create a copy of the training data
train1 = train
```

```
system.time(ipVars60s_sql <- sqldf('
SELECT t1.record,
-- This is the variable 1
COUNT(*) AS ip60s,
-- This is the variable 2, you can also use COUNT instead of SUM
SUM(CASE WHEN t1.click_time - t2.click_time <= 30 THEN 1 ELSE NULL END) AS ip30s,
-- This is the variable 3
COUNT(DISTINCT t2.app) AS ipApp60s,
-- This is the variable 4
COUNT(DISTINCT CASE WHEN t1.click_time - t2.click_time <= 30 THEN t2.app ELSE NULL END) AS ipApp30s
-- A self join is applied, in this case either left or inner join is fine
FROM train1 t1 LEFT JOIN train1 t2
-- Match the same ip
ON t1.ip = t2.ip
-- Match the records within the time interval
AND t1.click_time - t2.click_time BETWEEN 0 AND 60
-- If an ip has multiple records in the same timestamp,
-- only match the records before it
AND t1.record >= t2.record
-- Now each record has matched all the corresponding records in the specified time window,
-- we can group by the record id to calculate the four aggregated metrics
GROUP BY t1.record
'))
```

```
##    user  system elapsed
##   3.244   0.315   3.600
```

It only takes about 4 seconds for the sqldf code to run, which is pretty efficient.

Creating time-window variables with data.table

Now let's see the equivalent data.table code to achieve the same result. To make the functions more flexible and reusable, I split them to two steps: a function **timeWinJoin** to perform the time-interval join and the codes to create new features.

Step 1

```
# Create another new copy
train2 = train
# Define a function to implement time-interval join in data.table
timeWinJoin = function(dt, n, byVar){
  dt1 = dt
  # Generate duplicated copy for the columns we'll join on
  # as they'll disappear after running the data.table join method,
  # n is the length of the time window
  dt1$join_ts1 = dt1$click_time
  dt1$join_ts2 = dt1$click_time + n
  dt1$join_rec = dt1$record
  # The join conditions below are equivalent to what in the sqldf code
  keys = c(byVar, 'join_ts1<=click_time', 'join_ts2>=click_time', 'record<=record')
  # Pass the join conditions to On and set allow.cartesian=T
  # to allow large multiplicative return.
  # Note that dt1[dt] is equivalent to FROM dt LEFT JOIN dt1, which is a little
  # counter-intuitive. If you want to do an inner join, specify nomatch=0 in the brackets.
  dt2 = dt1[dt, on=keys, allow.cartesian=T]
  return(dt2)
}
system.time(ipJoin60s_dt <- timeWinJoin(train2, 60, 'ip'))
```

```
##    user  system elapsed
##   0.434   0.042   0.478
```

The join step is very swift. What `timeWinJoin(train2, 60, 'ip')` does is almost equivalent to:

```
system.time(ipJoin60s_sql <- sqldf('
SELECT *
FROM train2 t1 LEFT JOIN train2 t2
ON t1.ip = t2.ip
AND t1.click_time - t2.click_time BETWEEN 0 AND 60
AND t1.record >= t2.record
'))
```

```
##    user  system elapsed
##   2.212   0.262   2.483
```

We see the time-interval join performed by the data.table method is more efficient. Let's verify the correctness of the join step:

```
# The joined table contains 1,264,927 rows
nrow(ipJoin60s_dt)
```

```
## [1] 1264927
```

```
# Check whether the two methods get equivalent results.
# Here I just check whether the numbers of rows are the same
# because there are some different columns in the two results.
nrow(ipJoin60s_dt) == nrow(ipJoin60s_sql)
```

```
## [1] TRUE
```

Step 2

Now let's create the four variables with the data.table implementation:

```
system.time(
  ipVars60s_dt <- ipJoin60s_dt[, .(
  # This is the variable 1
  ip60s=.N,
  # This is the variable 2
  ip30s=sum(join_ts1-click_time<=30),
  # This is the variable 3
  ipApp60s=sum(!duplicated(app)),
  # This is the variable 4
  ipApp30s=sum(!duplicated(app[join_ts1-click_time<=30]))), by=record])
```

```
##    user  system elapsed
##  11.735   1.036  12.775
```

```
# Check whether all the results are the same
# the all.equal function requires everything to be identical
all.equal(ipVars60s_sql, data.frame(ipVars60s_dt[order(record)]))
```

```
## [1] TRUE
```

The data.table method seems to be slower in creating the variables although it's fast in the join step, But let's look at another example, where we apply a 2-second time window for the app variable and create 4 similar time-window variables as the above example.

```r
# The data.table method, now I combine the two steps together
system.time(appVars2s_dt <- timeWinJoin(train2, 2, 'app')[, .(
  app2s=.N,
  app1s=sum(join_ts1-click_time<=1),
  appDevice2s=sum(!duplicated(device)),
  appDevice1s=sum(!duplicated(device[join_ts1-click_time<=1]))), by=record])
```

```
##    user  system elapsed
##  13.924   1.554  15.547
```

```r
# The sqldf method
system.time(appVars2s_sql <- sqldf('
SELECT t1.record,
COUNT(*) AS app2s,
SUM(CASE WHEN t1.click_time - t2.click_time <= 1 THEN 1 ELSE NULL END) AS app1s,
COUNT(DISTINCT t2.device) AS appDevice2s,
COUNT(DISTINCT CASE WHEN t1.click_time - t2.click_time <= 1
                    THEN t2.device ELSE NULL END) AS appDevice1s
FROM train1 t1 LEFT JOIN train1 t2
ON t1.app = t2.app
AND t1.click_time - t2.click_time BETWEEN 0 AND 2
AND t1.record >= t2.record
GROUP BY t1.record
'))
```

```
##    user  system elapsed
## 157.582   5.478 164.250
```

```r
# Check whether the two methods get equivalent results
all.equal(appVars2s_sql, data.frame(appVars2s_dt[order(record)]))
```

```
## [1] TRUE
```

```r
# Define a function to create time-window variables
createTwVars = function(dt, tws, byVar, vars){
  # Keep a count of the number of newly created variables
  new_cols = 0
  for (tw in tws){
    # Create one simple time window count in each iteration
    dt[join_ts1-click_time <= tw, paste0(byVar, tw, 's') := .N, by=record]
    new_cols = new_cols + 1
    for (var in vars){
      # If not the base variable, interactive tw variables are built
      if (var != byVar){
        # Let R parse a string and evaluate it as a variable
        var = parse(text=var)
        # Create one interactive time window count in each iteration
        dt[join_ts1-click_time <= tw, paste0(byVar, '_', var, tw, 's') := sum(!duplicated(eval(var))), 
        new_cols = new_cols + 1
      }
    }
  }
  # Only keep the newly created variables
  cols = (ncol(dt)-new_cols+1):ncol(dt)
  # Aggregate all the built variables to generate the final table
  # .SD and .SDcols are used to specify the subset of columns from dt and
  # lapply is used to apply a function through the subset of columns
  # All the values for each created tw variable within each group are the same
  # so mean is just used to fetch the value
  return(dt[, lapply(.SD, mean, na.rm=T), by=record, .SDcols=cols])
}

system.time(appVars2s_dt2<-createTwVars(timeWinJoin(train2, 2, 'app'), c(1, 2), 'app', c('device')))
```

```
##    user  system elapsed
##   6.898   2.061   9.174
```

```r
str(appVars2s_sql)
```

```
## 'data.frame':    100000 obs. of  5 variables:
##  $ record     : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ app2s      : int  1 1 1 1 1 2 1 2 3 1 ...
##  $ app1s      : int  1 1 1 1 1 2 1 2 3 1 ...
##  $ appDevice2s: int  1 1 1 1 1 1 1 1 1 1 ...
##  $ appDevice1s: int  1 1 1 1 1 1 1 1 1 1 ...
```

```r
str(data.frame(appVars2s_dt2[order(record), c(1,4,2,5,3)]))
```

```
## 'data.frame':    100000 obs. of  5 variables:
##  $ record      : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ app2s       : num  1 1 1 1 1 2 1 2 3 1 ...
##  $ app1s       : num  1 1 1 1 1 2 1 2 3 1 ...
##  $ app_device2s: num  1 1 1 1 1 1 1 1 1 1 ...
##  $ app_device1s: num  1 1 1 1 1 1 1 1 1 1 ...
```

```r
# Verify the result
all.equal(appVars2s_sql, data.frame(appVars2s_dt2[order(record), c(1,4,2,5,3)]))
```

```
## [1] "Names: 2 string mismatches"
```

This time you see the data.table performance prevails over the sqldf performance. So why is that? Let's see some numbers below:

```r
# Number of unique ips & apps
length(unique(train$ip))
```

```
## [1] 15239
```

```r
length(unique(train$app))
```

```
## [1] 132
```

```r
# The join table for the second example contains 16,130,438 rows
nrow(timeWinJoin(train2, 2, 'app'))
```

```
## [1] 16130438
```

There are over 16 million rows in the join result in this example, which is much larger than the 1 million rows in the first example. Is there anything wrong here? Not really. Notice that there are 15239 unique IPs and only 132 unique apps in our training samples. When you create the time-window variables, you need to join all the occurrences of the given variable in the given time window to itself and you need to do this for **every single row**. What makes things worse is that the fewer unique values a categorical variable has, the larger the joined result would be as there would be more occurrences for the same value within the given time interval.

And as the number of rows increases, the advantage of data.table will become significant. Look at another more extreme example:

```r
# Create many variables in a 60s time window for the app variable
system.time(appJoin60s_dt <- timeWinJoin(train2, 60, 'app'))
```

```
##    user  system elapsed
##  19.078  22.087  46.674
```

```r
system.time(appVars60s_dt <- appJoin60s_dt[, .(
  app60s=.N,
  app30s=sum(join_ts1-click_time<=30),
  app1s=sum(join_ts1-click_time<=1),
  appIp60s=sum(!duplicated(ip)),
  appIp30s=sum(!duplicated(ip[join_ts1-click_time<=30])),
  appIp1s=sum(!duplicated(ip[join_ts1-click_time<=1])),
  appDevice60s=sum(!duplicated(device)),
  appDevice30s=sum(!duplicated(device[join_ts1-click_time<=30])),
  appDevice1s=sum(!duplicated(device[join_ts1-click_time<=1])),
  appOs60s=sum(!duplicated(os)),
  appOs30s=sum(!duplicated(os[join_ts1-click_time<=30])),
  appOs1s=sum(!duplicated(os[join_ts1-click_time<=1])),
  appChannel60s=sum(!duplicated(channel)),
  appChannel30s=sum(!duplicated(channel[join_ts1-click_time<=30])),
  appChannel1s=sum(!duplicated(channel[join_ts1-click_time<=1]))), by=record])
```

```
##    user  system elapsed
## 166.265  66.478 249.092
```

```r
# There are 308,355,893 rows!
nrow(appJoin60s_dt)
```

```
## [1] 308355893
```

```r
# We don't want to keep the large intermediate result in our memory
rm(appJoin60s_dt)
```

See that the intermediate table contains more than 300 million rows! If you use a data.frame method like dplyr or sqldf to implement this, it'll take you forever.

A hybrid method?

Some of you might think since data.table is faster in the time-interval join and sqldf seems to be faster in creating the variables, what if we combine these two methods together, which is to create the variables in sqldf based on the table resulted from the `timeWinJoin`? Unfotunately it doesn't work very well (I've tried it). That's because the sqldf package is designed to operate on a data.frame. If we pass a data.table to a sqldf query, it will be converted to a data.frame automatically and as you have learned, the data.frame operations are much less efficient than those of the data.table class.

Caveats:

You see, as the data is with high frequency (100k rows only cover around 3 minutes), a 60-second time window is a too long time window for these categorical variables. There are generally two suggestions on dealing with such a dataset:

1. Do not use a too long time window.
2. You could build the variables on a rolling basis. That is, you divide your large data (which is sorted chronologically) to several subsets, then iteratively build the time-window variables on each subset and finally use a rbind() function to combine all the results. An illustration of this method is as below:

```
# Here I roll the data backwards, you could also do it forwards
end_time = max(train2$click_time)
# For example, we want to apply a 60s time window
tw = 60
# The time range of the data we want to operate for each iteration
step = 100
# Prepare an empty data.table for rbind()
ipVars60s_dt2 = data.table()
# If no rows left, it means we have rolled over all the records
while (nrow(train2[click_time<=end_time]) > 0) {
  # We'll need to use the preceding time window of data
  # to create variables for the next time window
  tw_subset = train2[click_time>=end_time-step-tw & click_time<=end_time, ]
  # Same function as what you've seen before
  # but here join_ts1(click_time) is added to the groupby key
  # because we'll need to subset the result again based on the timestamp
  subset_vars = timeWinJoin(tw_subset, tw, 'ip')[, .(
    ip60s=.N,
    ip30s=sum(join_ts1-click_time<=30),
    ipApp60s=sum(!duplicated(app)),
    ipApp30s=sum(!duplicated(app[join_ts1-click_time<=30]))), by=.(record,join_ts1)]
  # Exclude the preceding time window of data from the results
  ipVars60s_dt2 = rbind(ipVars60s_dt2, subset_vars[join_ts1>=end_time-step])
  # Roll the time window backward
  end_time = end_time - step - 1
}
# Verify the result with the one that has been proved to be correct
all.equal(ipVars60s_dt[order(record)], ipVars60s_dt2[order(record),-"join_ts1"])
```

```
## [1] TRUE
```

Creating simple time-window counts with data.table rolling join

You've seen that the drawback of the time-interval join method (no matter it's implemented in sqldf or data.table) is that, it'll first generate a huge intermediate table and then perform aggregation on it. Now I'll introduce a method which could avoid this process hence improving the efficiency a lot. But please notice that this method **only works for the simple time-window counts but not for the interactive ones.**

The logic behind this `data.table` method is similar to the time-interval join. However, instead of performing an expensive join and then counting the total number of rows in a certain time interval, the counts are directly computed by taking the difference of two cumulative counts. For example, to derive how many times a certain IP appeared in the last 60 seconds, I don't join all the occurrences of that IP in the last 60 seconds for each record. Instead, I first count how many times this IP has occurred so far and then try to find the 60-second-ago value of this count, and then take the difference of this two cumulative counts as the last-60-second time-window count.

Several conditions are handled here:

- If there is no occurrence just right at 60 seconds ago, the match is carried forward to search for the nearest preceding occurrence;

- If there is no occurrence before 60 seconds ago, that means the IP could be pretty new in the samples we have and the cumulative count itself is also the time-window count;
- If there are multiple occurrences in the target timestamp, the first occurrence in that timestamp is selected as the target to match.

A special rolling join function dedicated for the `data.table` package is built to implement the described logic. To understand how it works, you can refer to this article https://www.r-bloggers.com/understanding-data-table-rolling-joins/.

```
# Create a new copy of the training data
train3 = train
# Add auxiliary column "ones" for cumulative counts
train3$ones = 1

# Function to implement a rolling time-window join and calculate the counts.
# It's important to make sure the data is sorted in chronological order before
# applying this function
rollTwCnt = function(dt, n, var){
  # Create cumulative count so far
  dt[, paste0(var,'CumCnt') := cumsum(ones), by = var]
  # Create a copy of table to join
  dt1 = dt
  # Create a lagged timestamp to join on
  dt[, join_ts := (click_time - n)]
  dt1$join_ts = dt1$click_time
  # Set the join keys, the last key is the one we'll roll on
  setkeyv(dt, c(var, 'join_ts'))
  setkeyv(dt1, c(var, 'join_ts'))
  # This line is the key of the rolling join method,
  # check ?data.table to see the meaning of each argument below
  dt2 = dt1[dt, roll=T, mult="first", rollends=c(T,T)]
  # Different conditions are handled below
  dt2[[paste0(var, n, 's')]] =
    ifelse(dt2[['click_time']] > dt2[['join_ts']],
           dt2[[paste0('i.', var, 'CumCnt')]],
           ifelse(dt2[['click_time']] < dt2[['join_ts']],
           dt2[[paste0('i.', var, 'CumCnt')]] - dt2[[paste0(var, 'CumCnt')]],
           dt2[[paste0('i.', var, 'CumCnt')]] - dt2[[paste0(var, 'CumCnt')]] + 1))
  # After the join, new columns are created and column names become messy,
  # filter out the columns we don't want to keep with some regular expression
  cols = colnames(dt2)[grepl(paste0("i\\.|^", var, "$|", var, n, "s"), colnames(dt2))]
  dt3 = dt2[, cols, with=FALSE]
  colnames(dt3) = gsub("i\\.", "", colnames(dt3))
  return(dt3)
}

# Verify the result
ipVars60s_roll = rollTwCnt(train3, 60, 'ip')
all.equal(ipVars60s_roll[order(record)][, .(record, ip60s)],
          ipVars60s_dt[order(record)][, .(record, ip60s)])
```

```
## [1] TRUE
```

```
# Create simple time-window counts for multiple time windows
# and multiple variables at one shot
```

```r
tws = c(10, 30, 60)
vars = c('ip', 'app', 'device', 'os', 'channel')
system.time(
  for (var in vars){
    for (tw in tws){
      train3 = rollTwCnt(train3, tw, var)
      # Need to keep the data in chronological order
      train3 = train3[order(record)]
    }
  })
```

```
##    user  system elapsed
##   1.202   0.655   1.877
```

```r
# Verify the result again to check if anything goes wrong in the for loop
all.equal(train3[order(record)][, .(record, ip60s, ip30s)],
          ipVars60s_dt[order(record)][, .(record, ip60s, ip30s)])
```

```
## [1] TRUE
```

Combining the results

Finally, let's say we want to include the features we built in the train3 and appVars60s_dt tables, let's join them together. Notice that we need to eliminate rows in the first 60 seconds as this is the **largest** time window we have applied to build the variables in the tutorial, which means the values of a 60s time-window count are invalid for the observations in the earliest 60 seconds.

```r
start_time = min(train$click_time)
# There are two duplicate variables in the two tables
train_full = appVars60s_dt[train3, on='record'][click_time>start_time+60, -c('i.app60s', 'i.app30s')]
str(train_full)
```

```
## Classes 'data.table' and 'data.frame':   57055 obs. of  42 variables:
##  $ record       : int  42946 42947 42948 42949 42950 42951 42952 42953 42954 42955 ...
##  $ app60s       : int  6281 1537 4529 1538 6059 142 5190 6282 5191 5192 ...
##  $ app30s       : int  3004 741 2250 742 2989 67 2519 3005 2520 2521 ...
##  $ app1s        : int  111 34 70 35 93 2 83 112 84 85 ...
##  $ appIp60s     : int  3833 1306 2822 1307 3775 112 2957 3833 2958 2958 ...
##  $ appIp30s     : int  2112 660 1604 661 2170 59 1675 2113 1676 1676 ...
##  $ appIp1s      : int  103 32 62 33 89 2 79 104 80 80 ...
##  $ appDevice60s : int  3 2 4 2 2 24 3 3 3 3 ...
##  $ appDevice30s : int  3 2 4 2 2 14 3 3 3 3 ...
##  $ appDevice1s  : int  2 1 2 1 2 1 2 2 2 2 ...
##  $ appOs60s     : int  59 53 60 53 58 8 58 59 58 58 ...
##  $ appOs30s     : int  50 40 54 40 53 7 53 50 53 53 ...
##  $ appOs1s      : int  24 14 16 14 20 2 23 24 23 23 ...
##  $ appChannel60s: int  24 17 15 17 27 6 16 24 16 16 ...
##  $ appChannel30s: int  23 15 14 15 27 6 15 23 15 15 ...
##  $ appChannel1s : int  14 11 9 11 20 1 14 14 14 14 ...
##  $ channel      : int  328 379 134 467 110 213 205 178 122 205 ...
##  $ os           : int  22 19 19 13 19 38 9 16 22 15 ...
##  $ device       : int  1 1 1 1 1 0 2 1 1 1 ...
##  $ app          : int  12 14 9 14 3 19 2 12 2 2 ...
##  $ ip           : int  23306 44555 37515 118648 90758 84972 105649 154860 43050 36183 ...
##  $ click_time   : POSIXct, format: "2017-11-06 16:22:52" "2017-11-06 16:22:52" ...
##  $ is_attributed: int  0 0 0 0 0 0 0 0 0 0 ...
##  $ ones         : num  1 1 1 1 1 1 1 1 1 1 ...
##  $ ipCumCnt     : num  14 24 32 2 19 3 14 6 1 90 ...
##  $ ip10s        : num  1 5 7 1 4 1 2 1 1 24 ...
##  $ ip30s        : num  4 8 17 2 10 3 6 2 1 50 ...
##  $ ip60s        : num  14 24 32 2 19 3 14 6 1 90 ...
##  $ appCumCnt    : num  6354 1548 4574 1549 6113 ...
##  $ app10s       : num  987 273 745 274 992 24 850 988 851 852 ...
##  $ deviceCumCnt : num  40621 40622 40623 40624 40625 ...
##  $ device10s    : num  6648 6649 6650 6651 6652 ...
##  $ device30s    : num  19788 19789 19790 19791 19792 ...
##  $ device60s    : num  40229 40230 40231 40232 40233 ...
##  $ osCumCnt     : num  1535 10507 10508 9561 10509 ...
##  $ os10s        : num  258 1716 1717 1706 1718 ...
##  $ os30s        : num  732 5176 5177 4696 5178 ...
##  $ os60s        : num  1525 10401 10402 9470 10403 ...
##  $ channelCumCnt: num  398 449 3190 32 145 ...
##  $ channel10s   : num  66 85 508 11 28 14 192 284 94 193 ...
##  $ channel30s   : num  197 225 1571 15 81 ...
##  $ channel60s   : num  394 446 3161 32 145 ...
##  - attr(*, ".internal.selfref")=<externalptr>
```

Down-sampling the "bad guys"

Assume that you have all the new features built. There is one thing you need to do before you build your models. For this dataset, the two y classes (attributed or not attributed) are highly imbalanced (the class ratio is around 1:400). So actually, the "good guys" are just a very small portion of the data. If you directly model on all the training data, the trained model will hopefully classify every sample to "not attributed". To avoid this, there are different resampling methods you could apply (i.e. down-sampling, up-sampling, hybrid methods, etc.). Here are some resampling options provided by the `caret` package: http://topepo.github.io/caret/subsampling-for-class-imbalances.html. But you may want to explore more other packages (i.e. `SMOTE`, `ROSE`, etc.) to find the best way to do this.

Below is a very naive illustration on how to do a down-sampling on the "bad guys".

```r
# See the two classes in our 100k sample is highly imbalanced
table(train_full$is_attributed)
```

```
##
##     0     1
## 56961    94
```

```r
train_good = train_full[is_attributed==1,]
train_bad = train_full[is_attributed==0,]
# Randomly sample 1% of the bad guys
perc1 = sample(nrow(train_bad), size = nrow(train_bad)/100)
# combine the down-sampled data and shuffle it
train_down = rbind(train_good, train_bad[perc1])
train_shuffled = train_down[sample(nrow(train_down))]
table(train_shuffled$is_attributed)
```

```
##
##   0   1
## 569  94
```