R Training Notes

Lesson 5

Stefanie Molin

April 18, 2017

I. Other Useful Packages for Data Processing

Now, we are going to cover a few more useful packages, however, it goes without saying that this is not an exhaustive list. R is open source and there are many packages out there for varying use-cases. To see how these packages work, we are going to use some daily data by client and by vertical for all Tier 1 clients.

```
# QueryVertica is already loaded along with username/password
query <- "
SELECT
    *
FROM
    (SELECT
        client_name
        , vertical_name
        , client_id
    FROM
      [redacted]
    WHERE
        ranking = 'TIER 1'
    GROUP BY
        client_name
        , vertical_name
        , client_id) clients
JOIN
    (SELECT
        *
    FROM
      [redacted]
    WHERE
        day >= CURRENT_DATE - 30) stats
ON
    clients.client_id = stats.client_id
"

# query data into a dataframe
df <- QueryVertica(username, query, password)

# see dimensions of result
dim(df)
```

```
## [1] 1042211      21
```

```r
# see what columns we have
colnames(df)
```

```
##  [1] [redacted]
##  [2]
##  [3]
##  [4]
##  [5]
##  [6]
##  [7]
##  [8]
##  [9]
## [10]
## [11]
## [12]
## [13]
## [14]
## [15]
## [16]
## [17]
## [18]
## [19]
## [20]
## [21]
```

Looks like we're dealing with quite a bit of data!

A. `data.table`

Dataframes are great, but they have their limits. When they get very large, they slow down considerably. This is where `data.table` comes in. `data.table` *inherits* from `data.frame` meaning we can work with objects of the `data.table` class as we would with dataframes, however, they also have additional functionality that dataframes do not; in other words, `data.table` is an enhanced `data.frame`. `data.table` provides fast operations for subsetting, grouping, updating values, etc. You can turn `data.frame` objects into `data.table` objects using `data.table()`

```r
# load the data.table package
library(data.table)

# turn df into a data.table
DT <- data.table(df)

# see what type of object we have
class(DT)
```

```
## [1] "data.table" "data.frame"
```

1. **Syntax**

Selecting rows and columns from `data.table` objects is similar to a SQL statement; there are 3 parts: `i` (`WHERE`), `j` (`SELECT`), `by` (`GROUP BY`) where `by` can be a list.

- **General Form**:
  - `DT[i, j, by]`
  - Take `DT`, subset rows using `i`, then calculate `j` grouped by `by`
- **Selecting Rows (`i`)**:
  - `DT[2:4,]`
  - `DT[2:4]`
  - Note the second option here will throw an error if you try it on a `data.frame`
- **Selecting Columns (`j`)**:
  - `DT[, .(col2, col4)]`
  - *`.()` is the same as `list()`*; you will need this notation anytime you choose more than one column.
  - You can also run computations and recycle columns (this should remind you of `dplyr`):
    * `DT[, .(Total = sum(A), C = mean(C))]`
  - To *apply* the same function across multiple columns use `lapply(list_of_cols, function)`:
    * `DT[, lapply(.SD, median), by = B]`
    * `.SD` = Subset of Data (this will use all data selected by `i`)
    * `lapply()` returns a list so there is no need to use `.()`
- **Grouping (`by`)**:
  - `DT[3:5, .(sum(B)), by = .(A)]`
  - Note that the `by` argument also uses the `.()` for making lists and that you can just provide the value if it isn't more than one
- **Update Column Values**:
  - `DT[, LHS := RHS]`
  - Values of `RHS` (right-hand side) will be assigned to `LHS` (left-hand side) variables.
    * `LHS` should be a vector if more than one
    * `RHS` should be a list if more than one
  - Set `RHS` to NULL to remove columns
- **Update Values by Row**:
  - Rather than use a `for` loop with `data.table`, you should use `set(DT, i, j, value)`
    * `for(i in 1:5) set(DT, i, 3L, i + 1)`
    * For each row in DT, set the 3rd column's value to the row number plus 1.
- **Update Column Names**:
  - `setnames(DT, "old", "new")`
- **Modify Column Order**:
  - `setcolorder(DT, newOrderVector)`
- **Indexing**:
  - `DT[A == "a"]`
    * The filter should be placed in the `i`; if it is in the `j`, it will return a logical vector of whether or not each row met that criteria.
  - `DT[A %in% c("a", "c")]`
    * Filter can use an `in` statement.
  - Selecting data is faster than on `data.frame` since `data.table` creates an index automatically (by default) on the `A` column (in this case) the first time we use it, so that it is faster the next time.

- **Using Keys**:
  - Create a key for easy lookup without having to use the index method above:
    - `setkey(DT, A)` to set column A as the index
    - `DT["a"]` now selects all rows where column A is "a"
  - If there are duplicates in the key column, use the `mult` argument during selection to specify which one you want:
    - `DT["a", mult = "first"]` – select the first occurence of "a" in column A
    - `DT["a", mult = "last"]` – select the last occurence of "a" in column A
  - Handling keys with no value for the selection:
    - `DT[c("a", "7"), nomatch = NA]` – default; adds a row with the missing key along with NA for all other columns
    - `DT[c("a", "7"), nomatch = 0]` – don't show any values for keys that aren't present in the data
  - Multiple keys:
    - `setkey(DT, A, B)` – set both columns A and B as keys
    - `DT[.("b", 6)]` – select values with "b" in column A **and** 6 in column B
    - `DT[.("b")]` – select based on only 1 of the keys

2. Examples

You can find more about the above in the package documentation. Below are a few ways to use these with Criteo data, however, we won't cover all of the above details.

```r
# drop the duplicated client_id column (column 5)
DT <- DT[, -5, with = FALSE]

# remove some other columns by name
DT <- DT[, c("zone_currency_id", "cumulated_p_ctr", "cumulated_p_cr") := NULL]

# rename some columns
setnames(DT, c("post_click_conversions", "post_view_conversions"),
         c("pc_conv", "pv_conv"))

# properly format the day column as a date
DT <- DT[, day := as.Date(day)]

# select spend by vertical over the last 30 days
spend_by_vertical <- DT[,.(sum(revenue, na.rm = TRUE)), by = vertical_name]
head(spend_by_vertical, 5)
```

```
##                vertical_name          V1
## 1:                   APPAREL [redacted]
## 2:            HOTELS/RESORTS
## 3: TELECOMMUNICATIONS (L2)
## 4:                OTHER (L2)
## 5:          REAL ESTATE (L2)
```

```r
# select data above but only for travel verticals
DT[vertical_name %in% c("TRAVEL (L2)", "ONLINE TRAVEL AGENTS", "AIRLINES"),
   .(Spend = sum(revenue, na.rm = TRUE)), by = vertical_name]
```

```
##          vertical_name      Spend
## 1: ONLINE TRAVEL AGENTS [redacted]
## 2:          TRAVEL (L2)
## 3:             AIRLINES
```

```r
# select sum by client for all [redacted] accounts
head(DT[client_name %like% "[redacted], .(Spend = sum(revenue, na.rm = TRUE)),
        by = client_name])
```

```
##                        client_name    Spend
## 1: [redacted]
## 2:
## 3:
## 4:
## 5:
## 6:
```

```r
# create keys on client, vertical, and day for easier lookup
setkey(DT, client_name, vertical_name, day)

# lookup [redacted] peformance yesterday
DT[.("[redacted]", "DEPARTMENT STORES", Sys.Date() - 1),
   lapply(.(displays, clicks, revenue, pc_conv), sum, na.rm = TRUE),
   by = .(client_name, vertical_name, day)]
```

```
##    client_name      vertical_name        day       V1     V2       V3    V4
## 1: [redacted] DEPARTMENT STORES 2017-04-17[redacted]
```

```r
# lookup [redacted] spend total
DT[[redacted]', .(Spend = sum(revenue, na.rm = TRUE)), by = .(client_name, vertical_name)]
```

```
##    client_name      vertical_name    Spend
## 1:  [redacted]DEPARTMENT STORES [redacted]
```

```r
# lookup spending on department store advertisers in the last 15 days by day
DT[(day >= Sys.Date() - 15 & vertical_name == "DEPARTMENT STORES"),
   .(Spend = round(sum(revenue, na.rm = TRUE), 2)), by = day][order(day)]
```

```
##            day    Spend
##  1: 2017-04-03[redacted]
##  2: 2017-04-04
##  3: 2017-04-05
##  4: 2017-04-06
##  5: 2017-04-07
##  6: 2017-04-08
##  7: 2017-04-09
##  8: 2017-04-10
##  9: 2017-04-11
## 10: 2017-04-12
## 11: 2017-04-13
## 12: 2017-04-14
## 13: 2017-04-15
## 14: 2017-04-16
## 15: 2017-04-17
```

```r
# pull out the client list
client_list <- DT[, 1, by = client_name][, -2, with = FALSE]
```

B. `stringr`

`stringr` makes working with strings in R easier by providing consistent functions and simplicity of use over base R string operations; these can also be utilized with regular expressions, just note that any regular expresions involving “\” must be escaped (i.e. “\w” becomes “\\w”). Here are some useful functions and their implementations:

- `str_trim()` – remove leading/trailing whitespace
- `str_pad()` – pad a string with extra whitespace
- `str_length()` – returns number of characters in string with improved handling of NAs and factors
- `str_sub()` – get a substring
- `str_c()` – equivalent to `paste0()`, but also removes NULLs
- `str_detect()` – checks for presence of a pattern
- `str_locate()` – returns location (start and end index) of the pattern
- `str_extract()` – extracts the first match of the text (`str_extract_all()` returns all matches)
- `str_match()` – extracts matches and returns a matrix with the first column being the full match, and the remaining columns, the individual capture groups (very useful with regular expressions)
- `str_replace()` – replaces the first match (`str_replace_all()` replaces all matches)

Note that you can use negative indices without any issues. Negative indices start from the end of the item (i.e. -1 is the last index).

```r
# load stringr package
library(stringr)

# find all client names with apostrophes and their verticals
# (use a dummy column for grouping then remove)
head(apostrophes <- DT[str_detect(DT[,client_name], "'"), 1,
                       by = .(client_name, vertical_name)][, -3, with = FALSE], 5)
```

```
##                client_name     vertical_name
## 1: [redacted]     SPORTS/OUTDOOR
## 2: [redacted] DEPARTMENT STORES
## 3: [redacted]        TRAVEL (L2)
## 4: [redacted] DEPARTMENT STORES
## 5: [redacted]     CONSUMER GOODS
```

```r
# replace all the apostrophes with an empty string, and
# calculate the string length (before apostrophe removal)
head(apostrophes[, .(client_name = str_replace_all(apostrophes$client_name, "'", ""),
                 client_name_length_pre_removal = str_length(client_name))], 5)
```

```
##               client_name client_name_length_pre_removal
## 1: [redacted]                             24
## 2: [redacted]                             12
## 3: [redacted]                             17
## 4: [redacted]                              9
## 5: [redacted]                             11
```

```r
# find total number of clients with the word "GOOD" in them (using lapply)
# TRUE is evaluated as 1 and FALSE as 0 when summing
sum(lapply(client_list, FUN = str_detect, pattern = "GOOD")[["client_name"]])
```

```
## [1] 16
```

```r
# what were they?
DT[str_detect(client_name, "GOOD"), 1, by = client_name][, -2, with = FALSE]
```

```
##                  client_name
##  1: [redacted]
##  2:
##  3:
##  4:
##  5:
##  6:
##  7:
##  8:
##  9:
## 10:
## 11:
## 12:
## 13:
## 14:
## 15:
## 16:
```

```r
# extract 1st word from each client name using regex and look at a few
lapply(client_list, str_extract, "[A-Z]+")[["client_name"]][1150:1155]
```

```
## [1] [redacted]
## [5]
```

```r
# get last 2 letters from client names (locations)
tail(client_list[, .(location = str_sub(client_name, start = -2, end = -1))])
```

```
##    location
## 1:       TW
## 2:       CN
## 3:       CN
## 4:       FR
## 5:       FR
## 6:       FR
```

C. `lapply` and the `apply` family

You probably noticed me using a new function in the last 2 sections: `lapply()` (this function is part of base R). This allows you to take a `list` (dataframes and data tables are also lists) and a apply a function to all its elements returning a list of the same size as the input. (Lists can be subsetted with `$name` or `[["name"]]`). There are several other members to the `apply` family that have slightly different behavior (and syntax) based on the type of the object input/output. *This is more efficient than a for loop.*

Syntax: `lapply(X = list, FUN = function_to_apply, ... = other_arguments_to_function)`

> `lapply()` will use the list `X` as the *first* argument to the function supplied (`FUN`); however, you may need additional arguments–those can be provided right after the required elements in the function call (`X` and `FUN`). The `...` denotes optional arguments that are usually passed to the underlying functions inside the function in question.

D. reshape2

You can restructure your data with the **reshape2** package. There are 2 main functions:

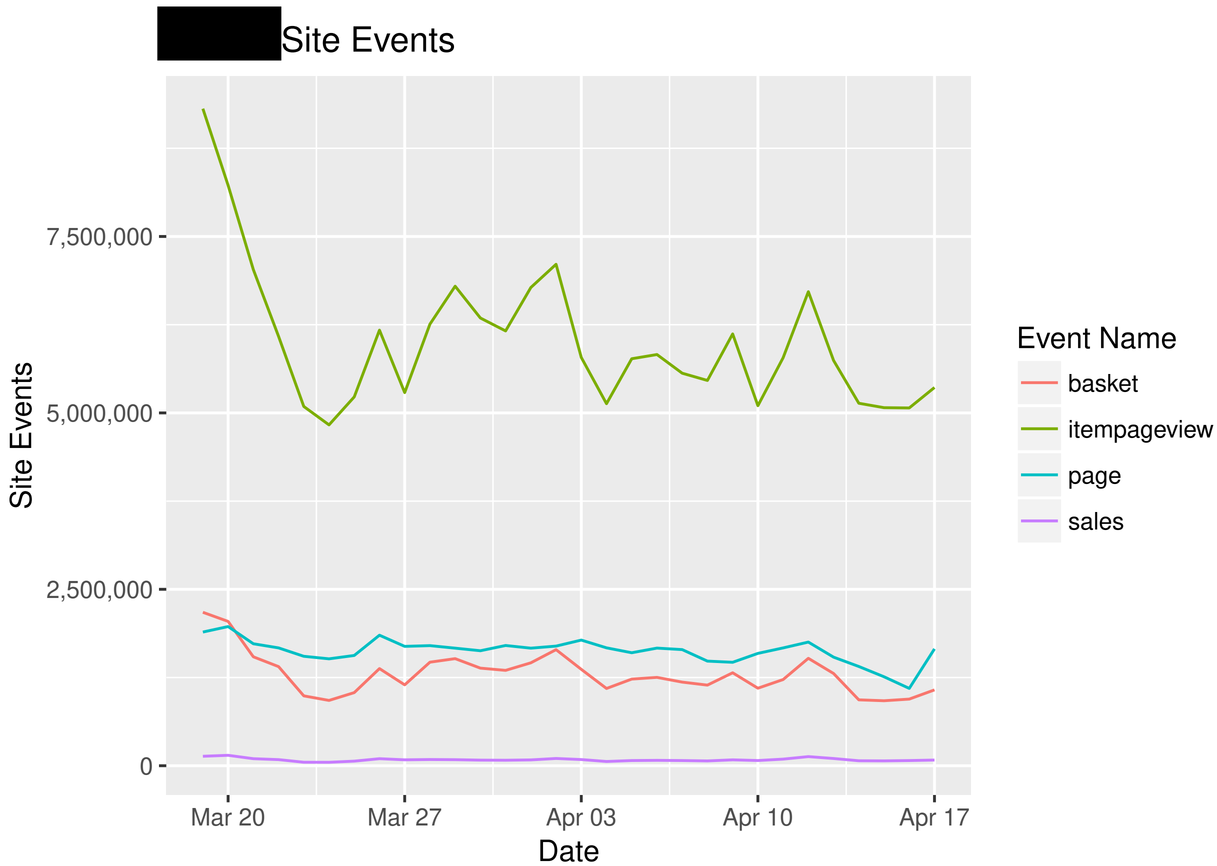
- `melt()` – turn a dataframe into a form allowing it to be reshaped (i.e. splitting columns into a variable and a value column)
- `dcast()` – takes the results from `melt()` and aggregates according to specified columns and functions

```
# load reshape2 package
library(reshape2)

# query for [redacted] site events data
query <- "
SELECT
    day
    , eventname
    , SUM(events) AS events
FROM
    [redacted]
WHERE
    partner_id = 5535
    AND day >= CURRENT_DATE() - 30
GROUP BY
    day
    , eventname
"

[redacted]_events <- QueryVertica(username, query, password)

# look at data
head([redacted]_events)
```

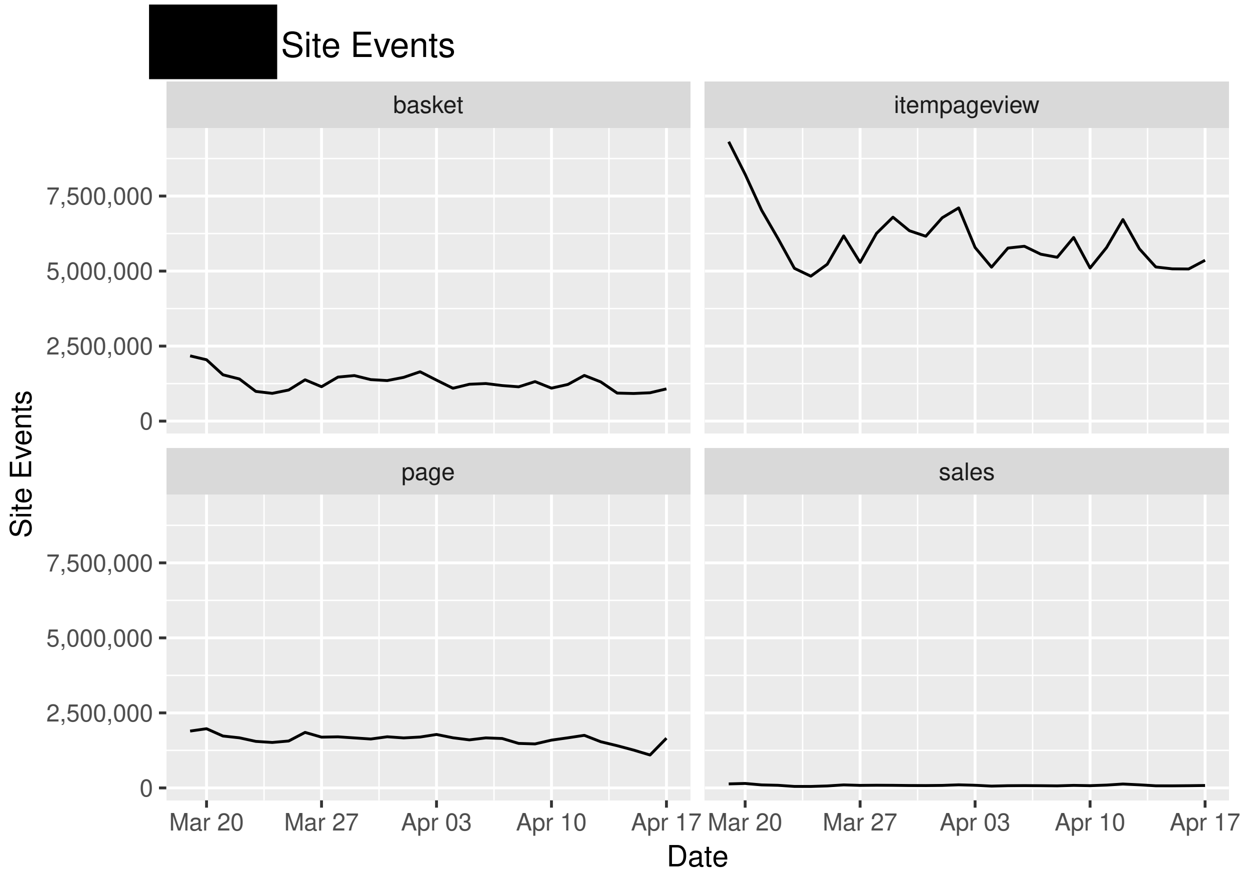
```
##          day eventname  events
## 1 2017-04-10     sales   73860
## 2 2017-03-22    basket 1404511
## 3 2017-04-11     sales   93732
## 4 2017-04-13      page 1538901
## 5 2017-04-08     sales   68124
## 6 2017-04-05     sales   73130
```

This is good for graphing…

```
# sample plot
library(ggplot2)

# multi-series line graph
ggplot([redacted]_events,
       aes(x = as.Date(day), y = events, col = eventname)) +
  geom_line() +
  ggtitle("[redacted]Site Events") +
  labs(x = "Date", y = "Site Events", col = "Event Name") +
  scale_y_continuous(labels = scales::comma)
```



```r
# facet wrapped
ggplot([redacted]_events,
       aes(x = as.Date(day), y = events)) +
  geom_line() +
  facet_wrap(~ eventname) +
  ggtitle("[redacted] Site Events") +
  labs(x = "Date", y = "Site Events") +
  scale_y_continuous(labels = scales::comma)
```



. . . but it's not too useful for reporting since people will need to pivot the results. We need to reshape the data and go from the long dataframe we queried for to a wide dataframe with each eventname as a column.

```r
# melt the dataframe
melted <- melt([redacted]_events)
```

```
## Using day, eventname as id variables
```

```r
head(melted)
```

```
##          day eventname variable   value
## 1 2017-04-10     sales   events   73860
## 2 2017-03-22    basket   events 1404511
## 3 2017-04-11     sales   events   93732
## 4 2017-04-13      page   events 1538901
## 5 2017-04-08     sales   events   68124
## 6 2017-04-05     sales   events   73130
```

```
# use dcast to reshape the data
reshaped <- dcast(melted, day ~ eventname + variable, sum)
head(reshaped)
```

```
##          day basket_events itempageview_events page_events sales_events
## 1 2017-03-19       2174482             9310799     1894374       133992
## 2 2017-03-20       2044991             8224969     1972082       147907
## 3 2017-03-21       1543560             7030633     1728515        99683
## 4 2017-03-22       1404511             6084602     1669751        86616
## 5 2017-03-23        990079             5090934     1550746        49615
## 6 2017-03-24        925404             4830268     1514893        48579
```

`dcast()` sorted our data for us, and this wide format is much better for human consumption! How about if we received our data in this format though? How do we go from wide to long?

```
# turn reshaped into the original (define the column names in this step)
reverted <- melt(reshaped, variable.name = "eventname", value.name = "events")
head(reverted)
```

```
##          day     eventname  events
## 1 2017-03-19 basket_events 2174482
## 2 2017-03-20 basket_events 2044991
## 3 2017-03-21 basket_events 1543560
## 4 2017-03-22 basket_events 1404511
## 5 2017-03-23 basket_events  990079
## 6 2017-03-24 basket_events  925404
```

```
# let's prove this is the same as the original using setequal from dplyr
library(dplyr)

# we need to rename the values in eventname since they got appended with "_events"
# let's use stringr and lapply to do this!
reverted$eventname <- lapply(reverted, str_replace,
                             pattern = "_events", replacement = "")[["eventname"]]

# check for equality
setequal(reverted, [redacted]_events)
```

```
## TRUE
```

Note that `reverted` is sorted while the original data was not; base R's `setequal()` function will wrongly declare they aren't equal but `dplyr`'s version gets it right!