Advanced Analytics in R

– SEMINAR WINTER SEMESTER 2014/2015 –

Data Manipulation with dplyr

– SEMINAR PAPER –

Submitted by:
Anton Mosin

Advisor:
Prof. Dr. Dirk Neumann

Contents

1 Introduction

dplyr is an R package for dataset manipulation released in January 2014 by Hadley Wickham, who is also the author of popular R package ggplot2. dplyr is the offspring of his previous plyr package, and focuses exclusively on the data frames. dplyr provides a set of tools to facilitate data manipulation and formulate a unifying framework for conducting it. Its key features are as follows:

- Any problem can be reduced to five basic methods (verbs), with each of them performing only one task.
- Writing in natural language can be mimicked by %>% pipeline operator that allows putting verbs one after another on a single line, greatly facilitating code comprehension.
- Data frame is the unit of analysis; it is the only possible input and output type for all functions.
- Databases like MySQL, SQLite, etc. are supported.
- Remote data bases are treated as local data; same methods work in both cases.
- dplyr functions do not produce side effects, e.g. they never modify existing data frames, but rather create copies of them.
- dplyr displays significant performance improvements over basic R functions. It is mainly achieved by having parts of R code evaluated by C++ libraries, where certain routines are implemented more efficiently.

The purpose of the paper is to introduce reader to the dplyr by explaining the use of the main package methods on concrete examples. Section 2 discusses methods of loading data into dplyr, followed by Section 3 introducing five essential methods for the analysis with dplyr. Their capability is greatly augmented when used with the grouping operator, as Section 4 demonstrates. An integral part of the dplyr grammar is pipeline operator that overhauls the way functions are written (Section 5). Specific types of methods for aggregating information and dealing with two data tables are discussed in Sections 6 and 7. Finally, the use of a general purpose method for defining one's own functions is explained in Section 8. [1]

2 Loading data

Before performing any operations on data, it must be loaded and examined. Though dplyr can work with default data frames, it defines its own class for these objects called tbl – local data frame.[2] Its main purpose is to control the way a data frame is printed to the screen. In base R, when we call a data frame, the whole data is displayed; if its size is large, it may be inconvenient. By contrast, the tbl object reveals only its first 10 entries. Moreover, the columns that do not fit the screen are masked, with the corresponding message printed.
There are two ways of creating local data frame. One can call data_frame() method that is

1 The present work serves as introduction to dplyr and strongly encourages consulting official documentation, where more details on, for example, window functions and working with remote databases can be found [6].
2 tbl class does not affect the data export; such base R functions as write.table() work the same.

almost identical to data.frame() core R function, to which data must be supplied. Alternatively, if standard data frame already exists, tbl_df() shall be used to convert it to the tbl class. To examine the data frame, an improved version of str() method called glimpse() exists that prints out data more tidy.

To demonstrate the functions, in this and subsequent sections the nycflights13 library is used. It contains information on all flights that departed or arrived to three New York airports on each day in 2013, as well as weather conditions and plane specifications. The flights data frame can be accessed by calling flights:

```
Source: local data frame [336,776 x 16]
   year month day dep_time dep_delay arr_time arr_delay carrier tailnum
1  2013     1   1      517         2      830        11      UA  N14228
2  2013     1   1      533         4      850        20      UA  N24211
3  2013     1   1      542         2      923        33      AA  N619AA
4  2013     1   1      544        -1     1004       -18      B6  N804JB
5  2013     1   1      554        -6      812       -25      DL  N668DN
6  2013     1   1      554        -4      740        12      UA  N39463
7  2013     1   1      555        -5      913        19      B6  N516JB
8  2013     1   1      557        -3      709       -14      EV  N829AS
9  2013     1   1      557        -3      838        -8      B6  N593JB
10 2013     1   1      558        -2      753         8      AA  N3ALAA
..  ...   ... ...      ...       ...      ...       ...     ...     ...
Variables not shown: flight (int), origin (chr), dest (chr), air_time (dbl),
  distance (dbl), hour (dbl), minute (dbl)
```

Here, the data was already stored in local data frame format, so no additional operations are required.

3 Primitive dataframe operations

dplyr is a language for data manipulation. As in all languages, nouns and verbs are combined to construct a sentence. Here, nouns are data frames, verbs are methods, and sentence is operation to perform. In this section, primitive dataframe functions are explained. They are primitive in a sense that all complex tasks can be reduced to them, since they refer to the most common operations performed on data table.

Verb	Meaning
filter()	Keep rows that match criteria
arrange()	Order rows
select()	Select columns by name
mutate()	Add new columns
summarise()	Aggregate information to single value

Table 1: Five primitive functions of dplyr.

All of them have the same interface: the first argument is the data frame, the subsequent arguments specify the task to perform. They always output a data frame. To get a sense of dplyr functions, in the first examples we contrast them with core R functions.

3.1 filter()

As the name suggests, it filters the rows keeping only those that match certain condition. For example, we want to see all the flights that took place on January 1. In base R, one writes:

```
> flights[flights$month==1 & flights$day==1,]
```

One must use the square brackets to subset the data frame, and then write the names of the columns with $ and the logic operator &; moreover, the comma sign must never be forgotten to be put in the end indicating that filter works along rows.
dplyr does the same operation the following way:

```
> filter(flights, month==1, day==1)
```

The syntax is common to all primitive verbs. First, specify the data frame, then pass in the desired operations on it, which are filtering expressions in this case. In fact, one can also use any logic operators in the body of the filter method. The above function is equivalent to

```
> filter(flights, month==1 & day==1)
```

The OR operator is represented by | character in R. To find out information about all the flights by "‘UA"’ or "‘AA"’ carriers

```
> filter(flights, carrier=="AA" | carrier=="UA")
```

To avoid writing the column name twice, the match operator %in% can be used

```
> filter(flights, carrier %in% c("AA", "UA"))
```

Related to the filter function is verb slice(). This method returns the data frame cut at specified point. For instance, to obtain the first two rows, write slice(flights, 1:2) that returns g

```
Source: local data frame [2 x 16]
  year month day dep_time dep_delay arr_time arr_delay carrier tailnum
1 2013     1   1      517         2      830        11      UA  N14228
2 2013     1   1      533         4      850        20      UA  N24211
Variables not shown: flight (int), origin (chr), dest (chr), air_time (dbl),
  distance (dbl), hour (dbl), minute (dbl)
```

3.2 arrange()

arrange() is a simple method of ordering the columns. For example, to order the data by year, month, and the carrier's name, we must write the following code in base R:

```
> flights[order(flights$year, flights$month, flightscarrier), ]
```

In the meantime, dplyr performs it the following way:

```
> arrange(flights, year, month, carrier)
```

By default, sorting is performed in ascending order; desc(column name) must be used to reverse the ordering.

3.3 select()

While filter() function returned the specified rows, select() method returns the columns meeting specified requirements.

```
> select(flights, carrier, dep_time, dep_delay)
```

```
Source: local data frame [336,776 x 3]
   carrier dep_time dep_delay
1       UA      517         2
2       UA      533         4
3       AA      542         2
...
```

It prints only the carrier's information about departure time and departure delay.
One can also select the range of columns, or exclude them.

```
> select(flights, year:arr_delay)
> select(flights, - (year:arr_delay))
```

A number of auxiliary functions are specified that are working solely within the select() body

Verb	Meaning
starts_with()	Select column by first characters
ends_with()	Select column by last characters
contains()	Select column by any character
matches()	Select column matching a pattern
one_of()	Select column from vector of possible names
num_range(x, 1:n)	Select columns from specified range

Table 2: Special functions for select().

Suppose we want to see each flight's arrival and departure delay. This can be achieved by writing

```
> select(flights, carrier, contains("Delay"))
> select(flights, flight, matches("_Delay"))
```

Sometimes, it is necessary to see the list of unique values a variable can take on. This is the task for distinct(). For example, we can find all carriers that operated in 2013 by distinct(select(flights, carrier))

```
Source: local data frame [16 x 1]
   carrier
1       UA
2       AA
3       B6
4       DL
5       EV
6       MQ
...
```

3.4 mutate()

Apart from selecting variables, one also needs to add new ones. It can be done by mutate(). This method is similar to the R base function transform(), except for the ability to refer to the newly created variables. For example, we can find speed of each flight by

```
> mutate(flights, flight, speed = distance/air_time*60)
```

This function adds the new variable to the end of the table. To see only the features of interest, call the following command that prints only the newly created variables:

```
> transmute(flights, flight, speed = distance/air_time*60)
```

```
Source: local data frame [336,776 x 2]
   flight    speed
1    1545 370.0441
2    1714 374.2731
3    1141 408.3750
4     725 516.7213
...
```

Note that mutate() does not alter existing data frame, so if there is a need to store the new variable, a new data frame or vector must be created.

3.5 summarise()

Finally, the summarise() method aggregates the information to a single row. This function will be useful when dealing with different groups of variables explored in the next section. In the basic case, we can calculate the average duration of flights by

```
> summarise(flights, mean(air_time, na.rm = TRUE)/60)
```

As usual, the first argument is data frame, the second one is an aggregation function. Note the use of additional argument na.rm in mean() method to remove the missing values.
A number of summary functions can be used, both from the core R language and dplyr

Verb	Meaning
Base R	
median(x)	Finds median value of vector
max(x)	Finds maximum value
sd(x)	Calculates standard deviation
var(x)	Calculates variance
dplyr	
n()	Calculates number of elements in a group
n_distinct(x)	Calculates number of unique elements
tally(x)	Wrapper for summarise(df, n())

Table 3: Examples of aggregation functions.

All of them reduce variable vector to a single value. n() deserves special attention, as it is function of zero arguments, and can be used only within summarise(), mutate(), or filter(). tally(x) is a shortcut for the command that calculates the number of elements of observations for a given variable or specific group.

4 Grouped operations

All operations performed so far have been conducted on the whole data. However, identifying groups or clusters and conducting calculations on them is indispensable for data analysis. dplyr implements group_by() method, that can be thought of as a modal verb that augments other verbs and alters their meaning. It is group_by that makes primitive verbs truly powerful.
What group_by() does is grouping the data by specified variable or variables, so that subsequent operations are performed on each group separately. Let us revisit the summarise() verb, only this time, we will use it in conjunction with group_by. For instance, it might be interesting to know the average arrival delay for every carrier in the data set.

```
> summarise(
    group_by(flights, carrier), #data frame
    mean_delay = mean(arr_delay, na.rm=TRUE) # summary method
  )
```

The first argument to group_by is data frame, followed by grouping variables.

```
Source: local data frame [16 x 2]
   carrier mean_delay
1       9E  7.3796692
2       AA  0.3642909
3       AS -9.9308886
4       B6  9.4579733
...
```

In order to ensure that grouping was conducted correctly, unique values of the grouping variables can be found by the already mentioned distinct() function.

```
> distinct(select(flights, carrier))
```

which yields data frame with 16 rows, that is the same the group_by number.

```
Source: local data frame [16 x 1]
...
```

A modification of summarise() is summarise_each() that performs a sequence of operations on several columns in the data set at once. For example, we may wonder what is the mean departure and arrival delay for each airport

```
> summarise_each(group_by(flights, origin),
                 funs(mean(., na.rm = TRUE)),
                 matches("Delay")
  )
```

To specify function or functions to be applied, a list of functions must be passed to funs() argument. The last argument are the columns for the summary. The grouping variables are protected from modification.

```
Source: local data frame [3 x 3]
  origin dep_delay arr_delay
1    EWR  15.10795  9.107055
2    JFK  12.11216  5.551481
3    LGA  10.34688  5.783488
```

Related to summarise_each() is apply() from standard R library. This function takes an object in a form of vector or matrix (any other object type, e.g. data frame, are coerced to one of the two) and applies a specified function to each of its elements, either row-wise or column-wise. Thus, summarise() can be regarded as a special case of apply() that performs operations only column-wise, whereas summarise_each() is augmented version of summarise() allowing for multiple function applications at single apply() call.
For example, to calculate average departure and arrival delays across three airports, we can take the latest data frame and use well-known dplyr method:

```
> df %>%
  +  summarise(dep_delay = mean(dep_delay),
  +            arr_delay = mean(arr_delay))
```

Here, we have saved the result of previous data query to df data frame, applied mean() method to the two columns, and stored the result in new ones bearing the same name.

```
Source: local data frame [1 x 2]
  dep_delay arr_delay
1  12.52233  6.814008
```

Alternatively, apply() method can be used; the only difference is that the output object is of an array type.

```
> apply(df[,-1], 2, mean)
```

The second argument means that the specified function is applied to all columns in the data frame. In contrast, 1 would stand for rows. Since the first column contains non-numerical elements, we exclude it.

```
dep_delay arr_delay
12.522330  6.814008
```

The effect of group_by() on the primitive verbs is summarised in Table 4.

Verb	Change
filter()	None
arrange()	Grouping variables have precedence
select()	Grouping variables are always retained
mutate()	None
summarise()	Summary for each group

Table 4: Effect of group_by on primitive verbs.

5 Data pipilines

Due to the function wrapping, the last two summary functions written are rather hard to comprehend at first glance, as one have to read from the inside out. As the data queries become more sophisticated, this problem will be only exacerbated. Fortunately, there is a way of overcoming this difficulty called *chaining*. This method allows one to write a sequence of operations from left to right as a coherent narrative, thereby facilitating comprehension and mimicking the natural process of writing and reading. For this, special forward-pipe operator

%>% is borrowed from magrittr package, where it had been originally developed. Formally, the crust of the pipe operator is defined as

$$f(x,y) = x\%>\%f(y), \tag{1}$$

where x is an argument passed to the function f of argument y. The last summary function

```
> summarise_each(group_by(flights, origin),
                 funs(mean(., na.rm = TRUE)),
                 matches("Delay")
  )
```

can be rewritten as

```
> flights %>%
    group_by(origin) %>%
    summarise_each(funs(mean(., na.rm=TRUE)),
                   matches("Delay")
  )
```

The pipeline operator %>% should be read as *then*. First, we specify the data frame to use, *then* we group it by destination, *then* we summarize each column matching Delay pattern, i.e. arrival_delay and departure_delay, by the mean() function.

6 Window functions

Window functions are a type of aggregation functions. Whereas classic aggregators return 1 output when given n inputs (e.g. mean(x)), window functions return n outputs. The result of a window function depends on the whole data, so methods working element-wise are not included in this category.
The main types of window functions:

(1) ranking and ordering functions;

(2) offset functions.

6.1 Ranking functions

A number of functions are implemented for ranking the variables. row_number(x), min_rank(x) and dense_rank(x) are essentionnly the same methods that handle ties differently, i.e. the entries with the same value. They trace their roots back to SQL where no notion of row numbers exist, which requires defining ranking order of one variable in terms of another variable.
To illustrate the difference betwen them, we create vector x, on which three methods are applied. Their output is summarized in Table 5.

```
> x <- c(1, 1, 2, 2, 25)
```

row_number() gives standard ranking without considering ties. min_rank(x) assigns the minimum of the ranks' ties. In our example, the two ones have first and second ranking, the lowest is the first, so the other one gets it too. The dense_rank(x) is similar to min_rank(x), but disallows gaps in rankings.

Verb	Result
row_number(x)	[1] 1 2 3 4 5
min_rank(x)	[1] 1 1 3 3 5
dense_rank(x)	[1] 1 1 2 2 3

Table 5: Ranking functions comparison.

The ranking functions can help one to answer such questions as "“what are top xyz in this dataset"”? For example, the two most delayed flights in 2013 are found by

```
> flights %>%
    filter(min_rank(desc(arr_delay))<=2)
```

```
Source: local data frame [2 x 16]
  year month day dep_time dep_delay arr_time arr_delay carrier
1 2013     1   9      641      1301     1242      1272      HA
2 2013     6  15     1432      1137     1607      1127      MQ
Variables not shown: tailnum (chr), flight (int), origin (chr), dest
  (chr), air_time (dbl), distance (dbl), hour (dbl), minute (dbl)
```

By default, all ranking functions work in ascending order, so that the first rank would be assigned to the least delayed flight. This explains the use of desc() above.

6.2 Offset functions

The two offset functions are lag() and lead() that yield modified version of the original vector, with each entry value being either ahead or behind the original one (see Table 6).

```
> x <- c(1, 2, 3, 4)
```

Verb	Result
lead(x)	[1] 2 3 4 NA
lag(x)	[1] NA 1 2 3

Table 6: Comparison of the offset functions.

Apart from the obvious application in time series analysis, they are useful for calculating absolute or relative change. For example, we can calculate the absolute change in the average departure delay from the previous day.

```
> daily <- flights %>%
    + group_by(year, month, day) %>%
    + summarise(delay = mean(dep_delay, na.rm = TRUE))
```

First, the mean value is calculated. Then, additional column for the absolute difference is added.

```
> daily <- daily  %>% mutate(d.change = delay-lag(delay))
```

```
Source: local data frame [365 x 5]
Groups: year, month
   year month day     delay   d.change
```

```
1  2013     1   1 11.548926         NA
2  2013     1   2 13.858824  2.3098975
3  2013     1   3 10.987832 -2.8709917
...
```

Another way to use the offset functions is to check whether a variable has changed its value. Consider the following task: in the original flights data frame, we want to find the earliest flights for each day in 2013.

```
> flights %>%
    mutate(newday = day != lag(day)) %>%
    filter(newday == TRUE) %>%
    select(year, month, day, flight, hour, minute)
```

```
Source: local data frame [364 x 6]
   year month day flight hour minute
1  2013     1   2    707    0     42
2  2013     1   3    707    0     32
3  2013     1   4    707    0     25
4  2013     1   5    739    0     14
..  ...    ... ...    ...  ...    ...
```

As the first step, we mutate the data frame by adding new column holding Boolean TRUE if date changes. Then, we filter for entries satisfying this condition and select columns holding the flights number and its time.

7 Two-table functions

An analysis is rarely performed on a single data frame. Processing the data from two tables is another area where dplyr simplifies life by defining three categories of two-table verbs:

(1) mutating joins,

(2) filtering joins,

(3) set operators.

Mutating joins alter existing data frame by adding or removing rows and columns. Filtering joins select rows based on the other table's data. Finally, set operators treat data frames as sets, for which union, intersection and difference can be found.
For the demonstration purposes, three data frames are created. The first one holds six entries of student IDs with corresponding names,

```
> t1
```

```
Source: local data frame [6 x 2]
  student_id name
1        101    A
2        102    B
3        103    C
4        104    D
```

```
5         105    E
6         106    F
```

while the second keeps four entries of student IDs and their majors.

```
> t2
```

```
Source: local data frame [4 x 2]
  student_id major
1         101    CS
2         102   BIO
3         103   MED
4         104  ECON
```

The third data frame is modification of the first one, where data has been shuffled and changed.

```
> t3
```

```
Source: local data frame [4 x 2]
  student_id name
1         104    D
2         103    C
3         107    H
4         110    I
```

7.1 Mutating joins

There are several methods belonging to this group:

- left_join(x,y),
- right_join(x,y),
- inner_join(x,y),
- full_join(x,y).

left_join(x,y) is the most common type of join operation. It returns all rows from x that have matches in y, with all columns from x and y. If the match is not unique, it will return all combinations of them. For example, we can find major for each student ID via:

```
> t1 %>% left_join(t2) #OR
> t1 %>% left_join(t2, by = "student_id")
```

```
Source: local data frame [6 x 3]
  student_id name major
1         101    A    CS
2         102    B   BIO
3         103    C   MED
4         104    D  ECON
5         105    E    NA
6         106    F    NA
```

By default, dplyr guesses the column along which the data will be joined; to prevent ambiguity, it can be stated explicitly as additional argument. left_join(x,y) is safe operation since no

information is lost. If there are no matches, NA entry is made. Its twin brother is right_join(x,y) which is equivalent to left_join(y,x) except for the ordering of the columns.
To obtain exclusively the matching values, inner_join(x,y) is used.

```
> t1 %>% inner_join(t2)
```

```
Joining by: "student_id"
Source: local data frame [4 x 3]
  student_id name major
1        101    A    CS
2        102    B   BIO
3        103    C   MED
4        104    D  ECON
```

Finally, the full_join(x,y) includes all observations from x and y.

```
> full_join(t1,t2)
```

```
Joining by: "student_id"
Source: local data frame [6 x 3]
  student_id name major
1        101    A    CS
2        102    B   BIO
3        103    C   MED
4        104    D  ECON
5        105    E    NA
6        106    F    NA
```

7.2 Filtering joins

Filtering joins conduct similar operations as mutating joins; however, they affect observations (rows), not variables (columns):

- semi_join(x, y) selects all observations from x that are present in y.
- anti_join(x,y) selects all observations from x that are not present in y.

We can find names and IDs of all students with known majors as follows:

```
> semi_join(t1,t2)
```

```
Joining by: "student_id"
Source: local data frame [4 x 2]
  student_id name
1        101    A
2        102    B
3        103    C
4        104    D
```

Alternatively, we can find all student IDs for which there is no information about their majors:

```
> anti_join(t1,t2)
```

```
Joining by: "student_id"
Source: local data frame [2 x 2]
```

```
  student_id name
1        106    F
2        105    E
```

It is advisable to use filtering joins before executing left_join() or inner_join(), to check what observations are selected to ensure against mismatches.

7.3 Set operators

The set operators expect the x and y inputs to have the same variables, and treat the observations like sets. They find intersection of the two sets, their union and difference according to the mathematical definitions.

- intersect(x,y) finds values present in both x and y.
- union(x,y) determines unique values in x and y.
- setdiff(x,y) determines observations in x that y does not hold.

For the two data frames t1 and t3, the intersection are the rows present in both of them is found by:

```
> intersect(t1,t3)
Source: local data frame [2 x 2]
  student_id name
1        104    D
2        103    C
```

All unique values in t1 and t3:

```
> union(t1,t3)
Source: local data frame [8 x 2]
  student_id name
1        107    H
2        105    E
3        106    F
4        110    I
5        101    A
6        102    B
7        103    C
8        104    D
```

Observations in t1 not found in t3:

```
> setdiff(t1,t3)
Source: local data frame [4 x 2]
  student_id name
1        101    A
2        102    B
3        105    E
4        106    F
```

8 Do

Having one verb to perform one task is one of the philosophical pillars of dplyr. But, no matter how rich the library is, it is impossible to cater to everyone needs. For this reason, do() method comes with the package. As everything in dplyr, it always returns a data frame to ensure that the pipeline will not be interrupted by *invalid type* error. In the output data frame, the first column is a variable, and the others are calculated by specified method.
Suppose one wants to use linear regression to test the relationship between the distance of the flight and the arrival delay for each of the months in 2013. One way to approach the problem would be to extract the data for each month, and manually build twelve linear regression models. Luckily, all this can be done from within dplyr:

```
> models <- flights %>%
  group_by(month) %>%
  do(
    mod = lm(arr_delay ~ distance, data=.)
  )
```

The code groups the flight data by month, and then fits linear regression model group-wise for each month. The resulting data frame contains month column and mod column storing all information about the model. Unfortunately, there is no trivial way of extracting the information about coefficients, their estimates, t-values and p-values. To do this, an auxiliary function must be defined.

```
> coef.df <- function(x){
  sc <- coef(summary(x))
  colnames(sc) <- c("est", "se", "t-value", "P-value")
  data.frame(coef = rownames(sc), sc)
  }
```

The last step is to apply this function to the created dataframe via do()

```
> models %>% do(coef.df(.$mod))
Source: local data frame [24 x 5]
Groups: <by row>
          coef          est           se   t.value       p.value
1  (Intercept) 11.027244872 0.4272778616  25.80814 4.534656e-145
2     distance -0.004831834 0.0003433766 -14.07153  8.253662e-45
3  (Intercept) 13.482609359 0.4410197441  30.57144 2.455031e-201
4     distance -0.007810846 0.0003573554 -21.85736 7.254874e-105
...
```

After examining all data, no conclusions can be drawn, mainly due to the counter-intuitive sign of the distance coefficient, indicating the reverse relationship between the distance and arrival delay. Moreover, some of the p-values are borderline significant. All this hints at possible violation of the linear model assumptions, like omitted variable or presence of heteroskedasticity.

9 Conclusion

Unlike other R packages, dplyr provides not only the tools for data manipulation but also framework for thinking. By specifying five core functions and importing the pipeline %>% operator, it has formulated its own grammar of data manipulation that helps one structure own thoughts and easily translate them into R commands. Complemented by the integration with the remote data bases and focus on the computing performance, it has become one of the integral tools for an R user.

A References

[1] K. MARKHAM. *Hands-on dplyr tutorial for faster data manipulation in R*. 2015. URL: http://www.r-bloggers.com/hands-on-dplyr-tutorial-for-faster-data-manipulation-in-r/.

[2] H. WICKHAM. *Data Manipulation with dplyr*. 2014. URL: https://www.dropbox.com/sh/i8qnluwmuieicxc/AAAgt9tIKoIm7WZKIyK25lh6a.

[3] H. WICKHAM. *Introducing dplyr*. 2014. URL: http://blog.rstudio.org/2014/01/17/introducing-dplyr/.

[4] H. WICKHAM. *The Grammar and Graphics of Data Science*. 2015. URL: http://pages.rstudio.net/Webinar-Series-Recording-Essential-Tools-for-R.html.

[5] H. WICKHAM. *dplyr: A Grammar of Data Manipulation*. 2015. URL: http://cran.r-project.org/web/packages/dplyr/index.html.

[6] H. WICKHAM. *dplyr Vignettes*. 2015. URL: http://cran.r-project.org/web/packages/dplyr/vignettes/.

B List of Tables