DASS: data manipulation with dplyr

Alla Tambovtseva

Data manipulation with `dplyr`

The library `dplyr` is a library for convenient data handling. With the help of this library we can quickly get summary of a data set, save the results obtained and group data by a selected feature. It is not difficult to work with this library, the only crucial thing is to understand the logic of data handling and to practise.

Let us install this library first:

```
install.packages("dplyr")
```

And load it:

```
library(dplyr)
```

Now we can load data on Marketing Campaign, Promotion Effectiveness in the Fast Food Chain taken from IBM website.

```
dat <- read.csv("http://math-info.hse.ru/f/2018-19/comm-math/marketing.csv")
```

Variables:

- `MarketID`: unique identifier for market (1 – 10).
- `MarketSize`: size of market area by sales (Small, Medium, Large).
- `LocationID`: unique identifier for store location.
- `AgeOfStore`: age of store in years (1 – 28).
- `Promotion`: one of three promotions that was tested (1, 2, 3).
- `week`: one of four weeks when the promotions were run (1 – 4).
- `SalesInThousands`: sales amount for a specific LocationID, Promotion and week.

Basic `dplyr` functions and a pipe operator `%>%`

Let us start from basic functions of `dplyr` that do not differ significantly from standard R functions. For example, `select()` that chooses columns from a data frame:

```
# choose MarketID, Promotion and SalesInThousands
dat_small <- dplyr::select(dat, MarketID, Promotion, SalesInThousands)
```

We can exclude columns via `select()` as well. To do this we should add a minus before a vector of column names:

```
# choose all except Week and LocationID
dat_small2 <- dplyr::select(dat, -c(Week, LocationID))
```

If columns are subsequent, we can use a slice for choosing them:

```
# from AgeOfStore to Week
dat_small3 <- dplyr::select(dat, AgeOfStore:Week)
```

If we want to select particular rows (observations) based on conditions, we need `filter()`:

```
# choose medium-size companies
medium <- filter(dat, MarketSize == "Medium")
```

Conditions can be complex:

```
# choose medium-size companies with AgeOfStore > 5
dat2 <- filter(dat, MarketSize == "Medium" & AgeOfStore > 5)
```

Well, you can ask: why do we need `dplyr`? The results obtained do not differ dramatically from those we got using standard R functions. However, actually it makes sense.

The library `dplyr` has a special operator `%>%` (called a *pipe operator*) that allows to perform operations step by step. It works as follows: take an object that is left to `%>%` and pass it to the function on the right of `%>%`. Consider a simple example:

```
df %>% View
```

What does it mean? Take the data frame `dat` and pass it to the `View` function. As we can see, in `View` there are no brackets and no name of the data frame since R already knows what it has to work with.

Consider another example. Take `dat`, choose columns `MarketID`, `Promotion` and `SalesInThousands`, and then ask for the first rows in a resulting data frame:

```
dat %>% select(MarketID, Promotion, SalesInThousands) %>%
  head %>% View
```

The library `dplyr` has a lot of useful functions. For example, `arrange()`, a function that sorts rows of a data frame based on a particular column (or columns). Let us sort our data frame based on `SalesInThousands` and look at the first rows:

```
dat %>% arrange(SalesInThousands) %>% head
```

```
##   MarketID MarketSize LocationID AgeOfStore Promotion Week
## 1        6     Medium        507          5         2    2
## 2        6     Medium        506         12         2    4
## 3        1     Medium          6         10         3    2
## 4        1     Medium          5         10         2    4
## 5        6     Medium        507          5         2    4
## 6        1     Medium         10          5         2    3
##   SalesInThousands
## 1            17.34
## 2            19.26
## 3            22.18
## 4            23.35
## 5            23.44
## 6            23.93
```

If we want to sort rows in a descending order, we can add `desc()` inside `arrange()` (from *descending*):

```
dat %>% arrange(desc(SalesInThousands)) %>% head
```

```
##   MarketID MarketSize LocationID AgeOfStore Promotion Week
## 1        3      Large        218          2         1    1
## 2        3      Large        220          3         1    3
## 3        3      Large        209          1         1    4
## 4        3      Large        208          1         3    1
## 5        3      Large        209          1         1    2
## 6        3      Large        216          4         3    1
##   SalesInThousands
## 1            99.65
## 2            99.12
## 3            97.61
## 4            96.48
```

```
## 5            96.01
## 6            94.89
```

On more useful function is `mutate()`. It is used for creating new columns in a data frame. Let's create a column `log_sales`, a natural logarithm of `SalesInThousands`.

```
dat %>% mutate(log_sales = log(SalesInThousands)) %>% head
```

```
##   MarketID MarketSize LocationID AgeOfStore Promotion Week
## 1        1     Medium          1          4         3    1
## 2        1     Medium          1          4         3    2
## 3        1     Medium          1          4         3    3
## 4        1     Medium          1          4         3    4
## 5        1     Medium          2          5         2    1
## 6        1     Medium          2          5         2    2
##   SalesInThousands log_sales
## 1            33.73  3.518388
## 2            35.67  3.574310
## 3            29.03  3.368330
## 4            39.25  3.669951
## 5            27.81  3.325396
## 6            34.67  3.545875
```

If we look at `dat`, we will be surprised:

```
View(df)
```

There is no `log_income` in our data set! Why? When we do something with a data frame via `dplyr` and do not save results, changes are not saved. How to save changes? As usual, reassign the value of `dat`:

```
dat <- dat %>% mutate(log_sales = log(SalesInThousands))
```

We can add more than one variable at once, you can list them inside `mutate()`.

Functions `summarize()`, `group_by()` and `tally()`

Now we will discuss the most helpful functions!

While manipulating data we often need to get aggregated information on variables. To do this we can use `summarise()`. Let us get the total number of rows in a data frame:

```
dat %>% summarise(total = n())
```

```
##   total
## 1   548
```

The function `n()` is universal, it is used for counting elements.

Now let us do something more interesting: find a minimum, a maximum and an average of the number of sales.

```
dat %>% summarise(avg_sales = mean(SalesInThousands),
          min_sales = min(SalesInThousands),
          max_sales = max(SalesInThousands))
```

```
##   avg_sales min_sales max_sales
## 1        NA        NA        NA
```

Why R does not want to count anything? There are missing values in this column! So we can add the argument `na.rm` that allows us to exclude NA's (`rm` - from *remove*) from consideration.

```
dat %>% summarise(avg_sales = mean(SalesInThousands, na.rm = TRUE),
          min_sales = min(SalesInThousands, na.rm = TRUE),
          max_sales = max(SalesInThousands, na.rm = TRUE))
```

```
##   avg_sales min_sales max_sales
## 1  53.46596     17.34     99.65
```

Often we have to get a summary not for all observations in a data set, but for a certain group. To do it we have to group data based on values of a variable chosen. Let us use the function `group_by()` and see how many companies of different size there are in `dat`:

```
dat %>% group_by(MarketSize) %>% summarise(count = n())
```

```
## # A tibble: 4 x 2
##   MarketSize count
##   <fct>      <int>
## 1 ""             1
## 2 Large        167
## 3 Medium       320
## 4 Small         60
```

Since there is a company of size "" that is not regarded as a true NAmissing value (NA), we got four groups instead of three. Let's correct it:

```
# delete rows with MarketSize = ""
# recall that != stands for 'not equal'
dat <- filter(dat, MarketSize != "")

dat %>% group_by(MarketSize) %>% summarise(count = n())
```

```
## # A tibble: 3 x 2
##   MarketSize count
##   <fct>      <int>
## 1 Large        167
## 2 Medium       320
## 3 Small         60
```

And now let's look at the average number of sales (in thousands) computed for every market size:

```
dat %>% group_by(MarketSize) %>% summarise(avg_sales = mean(SalesInThousands, na.rm = TRUE))
```

```
## # A tibble: 3 x 2
##   MarketSize avg_sales
##   <fct>          <dbl>
## 1 Large           70.0
## 2 Medium          44.0
## 3 Small           57.4
```

One more fact: a number of rows can be computed via `tally()` instead of `n()`:

```
dat %>% group_by(MarketSize) %>% tally()
```

```
## # A tibble: 3 x 2
##   MarketSize     n
##   <fct>      <int>
## 1 Large        167
## 2 Medium       320
## 3 Small         60
```