Intro to R for Decision Modeling

Data Manipulation

SickKids and DARTH

11/2/2020

Change `eval` to `TRUE` if you want to knit this document.

This session presents the key commands needed to manipulate a dataset in `R`. This worksheet is split into the following sections:

1. Data input and output
2. Subselecting data
3. Identifying and replacing data
4. Calculating useful statistics
5. Data manipulation with `dplyr`
6. Data manipulation with `pipe`

Throughout the course, we will demonstrate code and leave some empty *code chunks* for you to fill in. We will also provide solutions after the session.

Feel free to modify this document with your own comments and clarifications.

1. Data input and output

Input

There are several ways of loading your data in `R`. A convenient way of importing data into `R` is through tab-delimited text (.txt) or comma separated (.csv) files. For this course, we have provided a `.txt` file that contains the data we are going to be using.

If you have a dataset to analyse in `R`, you can save the file as a tab-delimited or as a comma separated file by opening it in Excel and clicking **File -> Save As -> Save as type** and selecting either the tab delimited or comma separated file type.

Once you have saved your file, it is time to load it in `R`:

- `read.table()` is used to load `.txt` data
- `read.csv()` is used to load `.csv` data.

The following example loads everything from a tab-delimited file called `course_data.txt` and creates an `R` object called `mydata` that contains all the data from our file:

```
mydata <- read.table('C:/Users/UserName/Documents/course_data.txt', header = TRUE)
```

To ensure that `R` can find your file, you need to specify the *filepath* of your dataset (C:/Users/UserName/Documents/course_da
The easiest way to find this filepath in Windows is to navtigate to the file in the file explorer, right click on the file, select Properties and copy the **Location:** into `R` (`C:/Users/UserName/Documents` in the above example). On a Mac, you can hold down the command button, right click and choose the option **Copy File Path**.

- When you copy directly, you need to change the backslashes in the filepath to forward slashes!
- You then need to add the name of the file and the file extension following another forward slash
- You need to enter your *filepath* between two quotation marks ("").

The code above contains the statement `header = TRUE` to specify that the first row of the data contains the variable names for your dataset. The data is also loaded and saved as an *object* so it can be used later for our data manipulation and analysis.

If your data file and the `R` file that loads the data file are in the same folder, you can avoid typing your filepath by navigating to the top bar of RStudio and click **Session -> Set Working Directory -> To Source File Location**. This sets the default filepath in `R`, often referred as the working directory, to the filepath of your `R` or R Mardkown file that is currently open.

EXERCISE 1 Use the function `read.csv()` to read in `framingham.csv`, the dataset provided for the exercises of this workshop. Store the dataset as a variable named `data`. You can use the *code chunk* below.

```
# Your turn
data <- read.csv("framingham.csv", header = TRUE)
```

Output

When you are done with your statistical analysis and you want to export the output of your work, you can use the `write.table()` function to export the output as a tab delimited file or the `write.csv()` function to export the output as a `.csv` file.

`write.table(mydata, file = 'course_data_export.txt')` exports the `R` object `mydata` back to a tab delimited text file.

`write.csv(data, file = 'framingham_export.csv')` exports the `R` object `mydata` back to a `.csv` file.

To export to a specific file, you should use the entire *filepath* as described above. In the code above, the data would be saved to `R`'s *working directory*. This is default folder where `R` will save all your results unless you specify the filepath. You can find out the current working directory using:

```
getwd()
```

It is possible to change the *working directory*, using the command `setwd()` and specifying the *path* to a specific folder. You can also set the working directory to your current file location using **Session -> Set Working Directory -> To Source File Location**.

2. Subselecting data

Most data manipulation in R involves comparing two values using a *relational operator*.

R's relational operators compare two values and return either TRUE or FALSE depending on the value and the comparison. The simplest relational operator is the "equal to" operator (==) which tests whether two values are the same.

The following table provides a list of R's relational operators and examples of possible logical expressions.

Relational Operators	Description	Example logical expression
==	equal to	`gender == "male"`
!=	not equal	`gender != "male"`
<	less than	`age < 40`
>	greater than	`age > 10`
<=	less than or equal to	`age <= 40`
>=	greater than or equal to	`age >= 40`
%in%	in	`age %in% c(30,31,32)`

EXERCISE 2 Create two R objects called `gender` and `age`. In the *code chunk* below, run a few example logical expressions from the table above.

```
gender <- "male"
gender <- c("male","female","other")
age <- 80

# Your turn
gender == "male"
gender != "male"
age <= 40
age %in% c(30, 31, 32)
```

When comparing a vector to a single item, the logical expression returns TRUE or FALSE for each item in the vector depending on the logical comparison.

EXERCISE 3 Run the following *code chunk* to understand vectors and logical comparisons:

```
# Vectors and Relational Operators
genders <- c("male","female","male","female")
genders == "male"
c(1,2,3,4) > 2
```

Comparing vectors of the same length compares the i^{th} element in the first vector to the i^{th} element in the second vector.

EXERCISE 4 Run the following *code chunk* to understand vectors and logical comparisons:

```
# Vectors and Relational Operators
genders == c("Bridge", "Building", "male", "Road")
```

We can combine the relational operators with R's logical operators of 'and' (&) and 'or' (|) to create more complex logical expressions.

- & returns TRUE when both expressions on the left and right side of the ampersand return TRUE.

- `|` returns TRUE when either of the expressions on the left or right side of the `|` returns TRUE.

EXERCISE 5 Use to code in the following *code chunk* to understand `&` and `|`:

```
# Your turn
TRUE | FALSE
TRUE & FALSE
TRUE | TRUE
FALSE & FALSE
```

Subselecting rows or columns in a dataset

There are many cases where you will need to select specific rows or columns of your data after you load it in `R`, for example:

- You may import more variables into `R` than you would like to use in the analysis.
- Subjects may need to be excluded from further analysis, e.g. for incomplete data or outlier testing.
- You might want to stratify your data into two or more categories before proceeding with the analysis.

This is called subsetting or subselecting and can be easily performed in `R`.

If you want to create a new dataset `data_new` that only contains the first five columns of the Framingham dataset, we type the following code:

```
data_new <- data[ , 1:5]
head(data_new)
```

Remember, the Framingham dataset is stored in an *object* called `data` that we loaded earlier.

In `R`, the statement `1:5` is read as 'from 1 to 5' and specifies that we should select columns, 1, 2, 3, 4, 5.

EXERCISE 6 Create a new dataset named `data_new` that only contains the first 10 columns of the Framingham dataset.

```
# Your turn
data_new <- data[, 1:10]
head(data_new)
```

The `data` *object* is stored by `R` as a data frame. A data frame has columns that contains the different variables and rows that contain information about each subject. In the above code, we selected a specific subset of the *columns*. We can also select a specific subset of *rows* by placing numbers before the comma, e.g.

```
data1 <- data[1:5, 1:5]
head(data1)
```

Subselects the first five columns from the first five subjects only and stores them in a variable called `data1`.

You can also subselect columns and rows by their name, e.g.

```
data2 <- data[, c("SEX", "AGE", "BMI", "HEARTRTE", "GLUCOSE")]
head(data2)
```

EXERCISE 7 Create a new dataset named `data2_new` that contains the first 10 subjects and the subject ID, sex, age, bmi, heart rate and glucose columns from the Framingham dataset. Then, take a quick glance of the new dataset.

- You can see the names of the columns of a dataset using the command `colnames(data)`, you can also look at the data dictionary.
- The function `head()` takes a dataset and returns the first 6 rows and all columns. It is a very useful function to get a quick glace of your data.

```
# Your turn
data2_new <- data[1:10,
                  c("RANDID", "SEX", "AGE", "BMI",
                    "HEARTRTE", "GLUCOSE")]
head(data2_new)
```

It is also possible to subset with logical expressions and select the rows or columns of a data frame that fulfils a certain condition. For example, we can extract and present all rows of our `data` where the participant is older than 30.

```
head(data[data$AGE > 30,])
```

- `data$AGE` is shorthand in `R` for `data[,"AGE"]`. In general, you can use the $ sign followed by the name of the variable to select a specific column of a data set. You can only use this shorthand to select a single column from a data frame.

EXERCISE 8 Create a new dataset called `data3` that contains all the subject ID, sex, age, BMI and heart rates for the subjects whose heart rate is greater than 100. Display the first 6 rows.

```
# Your turn
data3 <- data[data$HEARTRTE > 100,
              c("RANDID", "SEX", "AGE", "BMI", "HEARTRTE")]
head(data3)
```

3. Identifying and replacing data

Identifying missing values

Datasets are hardly ever complete. Missing values in data are very common and problematic in analysis procedures as they can cause issues with the analysis. It is useful to identify, and potentially, replace or remove missing values from a dataset. `R` assigns the value `NA` (Not Available) in every cell of a data frame that is missing.

To identify missing values, you can use the function `is.na()`. This function returns a logical vector or matrix (depending on the input) with the value TRUE where there is an NA and FALSE where the data are complete. For example:

```
b <- c(1, 2, 10, NA, 9, NA)
is.na(b)
```

If you want to subselect subjects without missing data for heart rate. You could use the function `is.na()` to identify the subjects that have no missing values in their heart rate variable and exclude them:

```
No_missing_heartrate_data <- data[is.na(data$HEARTRTE) == FALSE, ]
```

EXERCISE 9 Subselect subjects without any glucose information. You do not need to store the new dataset, just display the first 6 subjects and the `RANDID` and `GLUCOSE` columns.

```
# Your turn
head(data[is.na(data$GLUCOSE) == TRUE, c('RANDID', 'GLUCOSE')])
```

`R` also has a function `na.omit()`, that removes all observations with missing values in any of the columns.

EXERCISE 10 Calculate the number of observations in our data that does not have any missing values for the heart rate and glucose columns. The `nrow()` and `ncol()` functions give the total number of rows and columns of a dataset, respectively, when used on a dataset.

```
# Your turn
# 1st way, using & and is.na()
data3 <- data[(is.na(data$HEARTRTE) == FALSE) &
                (is.na(data$GLUCOSE) == FALSE),
              c("HEARTRTE", "GLUCOSE")]
nrow(data3)
# 2nd way, using na.omit()
# na.omit function omits NAs from your dataset.
data_no_missing <- data[, c('HEARTRTE', 'GLUCOSE')]
data_no_missing <- na.omit(data_no_missing)
nrow(data_no_missing)
```

Identifying and replacing values in a dataset

It is also possible to find and replace values in a dataset. For example, we may want to create a new variable named `SEX_char` for the existing Framingham dataset. We want our new varible to be equal to "male" when `SEX` is equal to 1 and "female" when `SEX` equals 2. This is achieved using the following commands.

```
# Creating a new variable
data$SEX_char[data$SEX == 1] <- 'male'
data$SEX_char[data$SEX == 2] <- 'female'
```

EXERCISE 11 Construct a new variable named `DIABETES_char` within the Framingham dataset. In this variable, all `DIABETES` elements equal to '0' should be given the character string 'Not a diabetic', and all elements equal to '1' should be given the character string 'Diabetic'.

```
# Your turn
data$DIABETES_char[data$DIABETES == 0] <- 'Not a diabetic'
data$DIABETES_char[data$DIABETES == 1] <- 'Diabetic'
```

You can also replace `NA` values with a numerical value. For example, if all subjects with missing values are not diabetic, we can correct this using the `R` code below

```
# Updated missingness
data$DIABETES[is.na(data$DIABETES)] <- 0
```

Factors

If some of the predictors have a categorical structure, we need to ensure that `R` recognizes these as *factors* as opposed to numerical values. When we create *factors*, we can also decide which category is going to be used as the reference level.

- The function `factor()` turns a character variable into a factor variable.
- The function `levels()` shows the categories, or *levels*, of a factor, in ascending order. The first level by default is the reference category when inserted into a regression model.

The following code makes `SEX` into a factor variable and display the level.

```
data$SEX_char <- factor(data$SEX_char)
levels(data$SEX_char)
```

4. Calculating useful statistics

Assume you want to calculate the arithmetic mean and variance of the age of your Framingham sample. You can easily do this by typing:

```
mean(data1$AGE)
var(data1$AGE)
```

EXERCISE 12 Calculate the mean and variance of BMI. Hint: most statistical functions in `R` does not automatically exclude missing values. If your variable has missing values, you need to specify `na.rm = TRUE` or the result will always be `NA`.

```
# Your turn
mean(data$BMI, na.rm = TRUE)
var(data$BMI, na.rm = TRUE)
```

There are several functions in `R` to compute different descriptive statistics:

- `median()`: Calculates the median for the data
- `min()`: Finds the minimum value for the data
- `max()`: Finds the maximum value for the data
- `sd()`: Calculates the standard deviation of the data
- `quantile()`: Find data quantiles, by default the min, 25% quantile, median, 75% quantile and the maximum.

`summary()` is a useful function that produces various common statistics in one go.

EXERCISE 13 Call the `summary()` function on BMI and see what types of statistics it gives.

```
# Your turn
summary(data$BMI)
```

We can use subsetting to calculate summary statistics for different subgroups in our data. For example, we can compute the mean and variance of age for men and women:

```
# Your turn
mean(data$AGE[data$SEX_char == "male"])
mean(data$AGE[data$SEX_char == "female"])
var(data$AGE[data$SEX_char == "male"])
var(data$AGE[data$SEX_char == "female"])
```

EXERCISE 14 Calculate the median and max of BMI for those who are over 60 years old with a heart rate of over 100 beats/min.

```
# Your turn
BMI_short <- data$BMI[data1$AGE > 60 & data$HEARTRTE > 100]
median(BMI_short, na.rm = TRUE)
max(BMI_short, na.rm = TRUE)
summary(BMI_short)
```

We can calculate the correlation between two or more variables using the `cor()` function. There are three methods for calculating correlation supported within `R` (Pearson, Kendal or Spearman method), more information can be found by running `?cor`.

To calculate the Pearson correlation coefficient for BMI and systolic blood pressure, we use:

```
cor(data$AGE, data$SYSBP)
```

If you want to calculate the correlations across several continuous variables in your dataset, the `cor()` function can be called on a data frame.

EXERCISE 15 Create a new dataset that only contains age, BMI, heart rate, glucose and systolic blood pressure. Use the `cor()` function to find the pairwise correlations between all these variables.

Note that `cor()` does not automatically exclude missing values. This can be achieved using the option `use = "pairwise.complete.obs"` in the `cor()` function.

```
# Your turn
cont_vars <- dplyr::select(data, AGE, BMI, HEARTRTE, GLUCOSE, SYSBP)
cor(cont_vars, use = "pairwise.complete.obs")
```

5. Data manipulation with `dplyr`

When manipulating larger datasets the `dplyr` package provides a set of functions that are easier to read, modify, and computationally faster than the code we showed above.

We will be focusing on these five functions:

- `filter()` for selecting rows based on observational characteristics
- `select()` for selecting variables based on their names
- `mutate()` for creating new variables
- `summarise()` for summarizing data
- `group_by()` for sub-population analysis

Let's start by loading the `dplyr` package that you installed in the previous session.

```
library(dplyr)
```

An important thing to note is that all `dplyr` functions use the dataset (e.g. `data`) as the first argument.

The function `filter()` returns a new dataset with all the observations that satisfy a set of logical expressions.

For example, we can select all the male participants (represented as 1) in the Framingham dataset who are under 40 years old and have a heart rate of over 100 beats/min:

```
head(filter(data, SEX == 1, AGE > 40, HEARTRTE > 100))
```

EXERCISE 16 Select all female participants (represented as a 2 in the `SEX` variable) in the Framingham dataset who are over 50 years old and have a glucose measurement of over 100 mg/dL. Only display the first 6 observations.

```
# Your turn
head(filter(data, SEX == 2, AGE > 50, GLUCOSE > 100))
```

`select()` allows you to select variables from a dataset based on the variable names. To select `SEX`, `AGE`, `BMI` `HEARTRTE` and `GLUCCOSE` from the Framingham dataset, the following command is used:

```
head(select(data, SEX, AGE, BMI, HEARTRTE, GLUCOSE))
```

It is also possible to remove variables from a dataset by including `-`. To remove our newly created `DIABETES` variable from `data` you can type:

```
head(select(data, -DIABETES))
```

Be careful, once you remove a variable from a dataset you cannot recover it. Therefore, you should usually make a copy of the original dataset and manipulate the copy.

`mutate()` creates new variables in a dataset. The following command creates a new variable that is the logarithmic transformation of `AGE`.

```
head(mutate(data, log_AGE = log(AGE)))
```

EXERCISE 17 Add an additional column to the dataset that is a logarithmic transformation of `HEARTRTE`. Display the age, log of age, heart rate, and log of heart rate columns for the first 6 observations. If you want to save the manipulated dataset, you have to store it as a variable using `<-`. You can also overwrite the original dataset the same way. Without using `<-` the manipulated dataset will not be saved.

```
#Your turn
data_e13 <- mutate(data,
                   log_AGE = log(AGE),
                   log_HEARTRTE = log(HEARTRTE))

data_e13 <- select(data_e13, AGE, log_AGE, HEARTRTE, log_HEARTRTE)

head(data_e13)
```

When a variable is created, it becomes instantly available to use in the same `mutate()` function to create new variables. The following command takes the `log()` of `AGE` and then taking the `exp()` of the new variable to return the original `AGE` value.

```
head(mutate(data, log_AGE = log(AGE), AGE_new = exp(log_AGE)))
```

EXERCISE 18 Multiply `BMI` by 2 and revert the muliplication to return the orginal `BMI` value. Only display the first 6 observations and the two BMI variables.

```
# Your turn
head(select(mutate(data, BMI_twice = BMI*2,
                   BMI = BMI_twice / 2), BMI_twice, BMI))
```

`summarise()` creates a summary of a dataset based on a set of functions provided. It returns a new dataset with columns for each summary.

If we wanted to compute the mean and variance of `AGE` for the Framingham dataset, we would use the following command:

```
summarise(data, mean_AGE = mean(AGE), variance_AGE = var(AGE))
```

EXERCISE 19 Compute the mean and variance of `BMI` for the Framingham dataset.

```
# Your turn
summarise(data, mean_BMI = mean(BMI, na.rm = TRUE), variance_BMI = var(BMI, na.rm = TRUE))
```

EXERCISE 20 Calculate the mean and variance of `BMI` whilst removing the missing values.

```
# Your turn
summarise(data, mean_BMI = mean(BMI, na.rm = TRUE),
          variance_BMI = var(BMI, na.rm = TRUE))
```

We can create subgroups in our datasets, to calculate summary measures for different patient subgroups. Firstly, we create the groups using the `group_by()` function, e.g. to group by `SEX_char`:

```
data_grouped <- group_by(data, SEX_char)
```

We can then use the `summarise()` function to return a summary for each group in our new dataset. For example, we can estimate the mean and variance of `AGE` for men and women.

```
summarise(data_grouped, mean_AGE = mean(AGE), variance_AGE = var(AGE))
```

EXERCISE 21 Group the data by `DIABETES_char` and estimate the mean, median, variance and standard deviation of 'BMI' for diabetic and non-diabetic groups. Display the summary.

- `median` is the `R` function to find the median
- `sd` is the `R` function to find the standard deviation.

```
# Your turn
data_grouped1 <- group_by(data, DIABETES_char)
summarise(data_grouped1,
          mean_BMI = mean(BMI, na.rm = TRUE),
          median_BMI = median(BMI, na.rm = TRUE),
          variance_BMI = var(BMI, na.rm = TRUE),
          sd_BMI = sd(BMI, na.rm = TRUE))
```

We can also group and summarize using multiple variables. This creates a summary for each unique subgroup combination of the variables. For example, we could group our dataset `data` by `SEX_char` and `DIABETES_char` creating 4 subgroups:

- Males who are diabetic
- Males who are not diabetic
- Females who are not diabetic
- Females who are diabetic

EXERCISE 22 Estimate the mean and variance of `AGE` for the above 4 subgroups.

```
# Your turn
data_grouped2 <- group_by(data, SEX_char, DIABETES_char)
summarise(data_grouped2, mean_AGE = mean(AGE),
          variance_AGE = var(AGE))
```

6. Data manipulation with `pipe`

The *pipe* operator (`%>%`) is a part of the `dplyr` package. This offers a convenient and concise way of performing a sequence of data manipulations.

The operator `%>%` *pipe* their value to the left-hand of the operator forward into expressions on the right-hand side,

i.e. one can replace `f(x)` with `x %>% f(.)`, where `f(.)` can be any function in `R`. For example, instead of writing `ncol(data)` we can have `data %>% ncol(.)`, where `.` is a place holder for the object on the left-hand side.

```
# calculate the number of columns in our dataset
data %>% ncol(.)
```

The benefits are more apparent when you have a bigger series of operations to be carried out sequentially.

- If you have multiple pipes, the output from the previous *pipe* operation is the input of the next *pipe* operation. An easy way to remember this is that whatever is on the left-hand side is always the input of the *pipe* that follows immediately.
- *Pipe* works very well with the functions in the `dplyr` package.
- Most of the time *pipe* is used on a dataframe object since the funcitons in the `dplyr` package can only be applied to dataframes.

In the following command first subselects subjects over 60 years old using *pipe* and then selects the variables `RANDID` and `AGE`.

dplyr`knows that you are using the data on the left-hand side of the *pipe* thus the.‘` is optional.

```
data %>%
     filter(AGE > 60) %>%
     select(RANDID, AGE) %>%
     head()
```

EXERCISE 23 Subselect subjects with a heart rate of over 80 beats/min, and among them, only keep the ones with a glucose measure of over 200 mg/dL. Then, only keep the ones with a BMI value. Finally, display the subject ID, heart rate, glucose and BMI columns for the first 6 subjects.

```
# Your turn
data %>%
     filter(HEARTRTE > 80) %>%
     filter(GLUCOSE > 200) %>%
     filter(is.na(BMI) == FALSE) %>%
     select(RANDID, HEARTRTE, GLUCOSE, BMI) %>%
     head()
```

The following code gives the mean BMI stratified by sex for those over 60 years old.

```
data %>%
     filter(AGE > 60) %>%
     group_by(SEX_char) %>%
     summarise(Avg_BMI = mean(BMI, na.rm = TRUE))
```

EXTENSION EXERCISE Give the mean, median, variance and standard deviation of systolic blood pressure, stratified by prevalent stroke (`PREVSTRK`), among those who are *either* a smoker or a diabetic.

The `ifelse()` function takes in an expression as its first argument, returns the 2nd argument when the expression is satisfied, and returns the 3rd argument when it is not. Try the following code:

```
ifelse(2 < 3, 'Correct', 'Incorrect')
ifelse(1 > 10, 'Right', 'Wrong')
```

```
# Your turn
data %>%
     mutate(DIABETES_SMOKER =
               ifelse(DIABETES == 1 | CURSMOKE == 1, 1, 0)) %>%
     mutate(PREVSTRK_char = ifelse(PREVSTRK == 1, 'Yes', 'No'),
            PREVSTRK_char = factor(PREVSTRK_char, levels = c("Yes", "No"))) %>%
     filter(DIABETES_SMOKER == 1) %>%
     group_by(PREVSTRK_char) %>%
     summarise(mean_SYSBP = mean(SYSBP, na.rm = TRUE),
               median_SYSBP = median(SYSBP, na.rm = TRUE),
               variance_SYSBP = var(SYSBP, na.rm = TRUE),
               sd_SYSBP = sd(SYSBP, na.rm = TRUE))
```