JEM221 Data Science with R I
Week #5-#6

Loading data
&
Managing and cleaning data
&
Sampling

Ladislav Krištoufek

Outline

1. Loading data
   - Well-structured data
   - Less-structured data
   - Some other useful tricks
2. Exploring data
3. Managing data
   - Cleaning data
   - Sampling for modeling

Outline

1. Loading data
   - Well-structured data
   - Less-structured data
   - Some other useful tricks
2. Exploring data
3. Managing data
   - Cleaning data
   - Sampling for modeling

Working directory

- Unless loading data from URLs (HTML links) or working with large datasets (many variables) that need repeated reformatting, it is more practical to work from your specified working directory.
- Files are then loaded from and saved into the predefined directory.
- For the purposes of working with a dataset repeatedly, it is more practical to have even the data loading procedure written in code.

Outline

Well-structured data from files and URLs

- There are more and more ready files to be used for analysis. If we are so lucky to have our dataset ready like a table-structured data with headers, loading is almost effortless.
- The standard way of organizing data is to have variables as columns and observations as rows (ideally with the first row as a header, i.e. variables names).

read.table()

- To load a plain-text file.
- The first argument should be the file name (in quotes) followed by other arguments (options):
  - *sep*: separator specification
  - *header*: logical for the first row being a row of names
  - *na.strings*: specification of the *NA* notation
  - *stringsAsFactors*: forces character strings to be loaded as character strings and not factors
- Even though the *read.table()* function is the most commonly used one, there are several others, usually built for specific file types, e.g. *read.csv()*, *read.delim()*. Each of these has a different default setting. It is usually safer to check the help.
- There is an *Import Dataset* button in RStudio as well.

HTML links

- Loading HTML links works the same way as the read.table and connected commands. A complete link (in quotes) must be provided.

Saving data files

- Via *write.csv()* and *write.table()* commands. Arguments:
  - object to be written
  - file name
  - *sep*: separator specification (for write.table)
  - *row.names* usually set to FALSE
- Practical when you let your code run on another computer.

Outline

Adding names to the variables

- If not given in the datafile, R will automatically give names to the variables, usually X1, X2, etc.
- As the column names of a data frame are an attribute, this can be rewritten using the *colnames()* command (and assigning an atomic vector of strings to it).

Mapping

- The observations can be given under codenames while a dictionary is only attached to the datafile.
- Using the *list()* function, we can create a mapping list which can be later used to rewrite parts of the dataset with desirable values/strings.
- The actual rewriting is usually done via a combination of a *for* loop and *if* conditioning.

Example data

- German bank credit dataset from the USI Machine Learning Repository.
- 1000 observations of 20 variables from 1994.

Outline

Subsetting

- We might be interested only in a part of the whole dataset.
- We can create an additional object as a subset of the original dataset using the *subset()* function.
- Standard conditioning notation holds here.

Working with factors

- Variables might be given as strings. If there is only a handful of possible levels, it pays off to transform the observations into factors using the *as.factor()* function.
- The factors are in fact defined via a base level (similarly to the dummy variables in econometrics). If needed, we can change the base/reference variable using the *relevel()* function.

Example data

- A subset of the United States Census 2011 national PUMS American Community Survey data.
- The complete dataset contains information about 3 million individuals and 1.5 million households.
- 200 variables for each – income, employment, education, etc.
- Anonymized.

Outline

Example data

- A subset of health insurance data.
- Focused on predicting probability of being insured given a set of explanatory variables, e.g. employment, gender, marital status, and income.

Explore before analyzing

- Exploring the dataset might seem as a waste of time and a ’low-tech’ approach towards data.
- However, it is in a way similar to first cleaning and restructuring your dataset before using it.
- It is usually better to spend some time exploring rather than running the whole analysis only to find that there is a problematic observation at the very end.

summary()

- The *summary()* function gives the very basic information about a variable. As such, it can uncover several issues:
  - missing values
  - invalid values and outliers
  - ranges

Exploring via visualization – one variable

- Histogram
- Density plot
- Bar chart

Exploring via visualization – two variables

- Most of the time, a simple *plot()* is enough for the basic data inspection.
- We will get to the more complicated ones when necessary.
- If you want to get more into visualization, check the *ggplot* package. There is also a DataCamp course for it.

Outline

Outline

Treating missing values

- If you have missing values (*NA*s), there are two things you can do: drop them or convert them into something meaningful.
- If the missing values are only a small fraction, it is usually safe to drop them.
- If the fraction is large, it could be reasonable to create a new level, e.g. called *missing*. The fact that the observations are missing can carry information.
- Mind that this is pretty much the same thing but, as noted several times, some functions do not work with *NA*s.
- If the observations are missing randomly, the standard is to substitute with a mean (unless you believe that the *NA* has some specific meaning).
- You can also substitute the *NA*s with zeros and treat them as dummy variables.

Treating missing values – useful functions

- *is.na()* for detecting missing values
- *ifelse()* for a quick version of *if() {} else {}* conditioning
- *cut()* for splitting the variable with respect to given ranges/values.

Data transformations

- You transform data to make the analysis more understandable. Specific transformations:
  - normalization
  - conversion to discrete variables (ranges)
  - rescaling
  - taking logarithms

Outline

Test and training splits

- When building a model, we are not only interested in being able to fit the data but (and in data science especially) being able to predict accurately.
- We thus need to split the data into a training set and a testing set (sometimes also a calibration set as an in-between step).
- In practice, this is done by combining a random numbers draw (*runif()*) and subsetting (*subset()*).
- For grouped records, we do not want to split the groups during randomization. This is performed via the *merge()* function.

Next block

- Model evaluation
- Memorization methods