R for Data Science

Table of Contents

Introduction

This document is meant as an introduction to programming in R for data sciences. For this purpose we discuss concepts that address simple utilities to discover questions and begin exploring advanced R. The format of this document is as follows:

- Function names are written in *italics* - *function*(x)
- General asides/notes are tabbed italicized section:

  This is an aside to discuss a concept or issue which although rare can possibly occur. Alternatively, this may be a note to discuss a relevant note to add to a discussion.

- we will focus on data frames for the majority of our work, to simplify the discussion of data frames any such object will be referenced as **df**
- We will indicate additional parameters that may be added or are required - (. . .)

Installation

We will be using R-Studio and the Samples of Code in this document will refer to R version 4.1.1 (Kick Things). Additionally, we will be liberally using packages to do many tasks that, in base R, would be very abstract. Where R, R-Studio, and the packages can be found are as follows:

- Base R: https://www.r-project.org/
- R-Studio: https://www.rstudio.com/
- Libraries for R: https://cran.r-project.org/

Additionally, tidyverse must be loaded from R-Studio after installing everything it can be installed from Tools > Install Packages ... then enter tidyverse in the package name to find and download it.

1 Variables

Variables in R work similarly to other programming languages, you may work with integers, floats, strings, and other data structures. The difference is R focuses on working with Data Structures by default. So while Python has access to similar structures R has optimized functions for working on large amounts of data that nears the territory of SQL.

1.1 Variable Assignment

A variable in R may be assigned one of 3 ways “=”, “<-”, “<<-”. The operators “=” and “<-” work in a similar way, with one notable exception. “=” may only be used at the top level, and as such, although in scripts it is valid, it is bad form to use “=” to assign variables in functions and in scripts. This type of assignment should be left to “<-”. The final operator “<<-” is special, when used it checks for any variables with that name in the global environment and assigns values to that variable.

> *Variables assigned in R that are currently being used, are in the current local or global environment, are stored in the RAM. As such large data sets can slow down or even crash the client.*
>
> *R does however store data frames in common storage, any added data frames refer to specific indices it needs to access the data. So even as a programmer may add many variables that refer to the same data frame, and different columns or rows, the amount of active memory in use will remain the same.*

1.1 Data Structures

Larger data structures than integers, strings, etc.. are what you will be working with in R almost all of the time. Notable default data structures include:

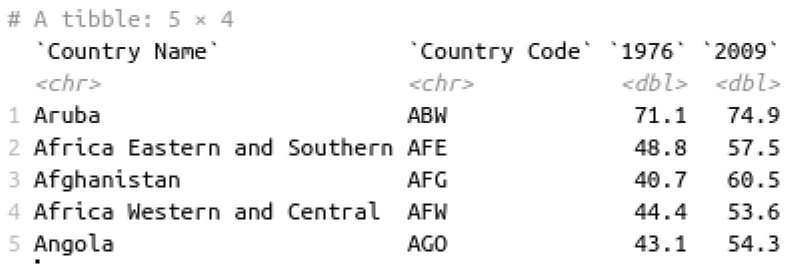
- Lists (L)
- Vectors (**v**)
- Matrices (**M**)
- Data Frames (**<u>df</u>**)

Lists in R work similarly to their traditional counterparts with the exception that a list in R can be called in either of two ways: [[index]] and list$named.element. Lists have order, even for named elements, depending on when they were declared they can be called in that order. Additionally, new indices may be called and assigned but attempting to get an undefined value will cause an error. Although all the given data structures can be modified at location the list is the only of which can grow without reassigning the variable.

Vectors in R can be treated like rows in a database, similarly we can refer to them as tuples and they may contain any combination of simple data types within them, however, when a larger type is added to a vector all values are coerced to that type which can lead to information loss in some cases. Vectors can be accessed and treated like lists in other programming languages, **v**[index]. The underlying structures that compose the rows and columns of matrices are vectors and data frames in R. As such data frames and matrices can be referenced and modified by their underlying data structures and are similarly mutable.

Matrices in R act exactly as described previously. They are mutable and composed of vectors, and can be multiplied together and with vectors by matrix multiplication (%*%). Regular multiplication can still be done and will be done by like terms. However, this structure is not relevant for introductory data science so we will leave the discussion here.

Finally there are data frames, which we will focus on the most. These structures may be composed of many different types of columns and unlike matrices information will not be lost. Columns are all named and can be referenced either by their order or by their names and rows may be referenced by order. For our purposes we will use a version of the data frame from a package tidyverse, this package gives us access to tibbles which are data frames that give us extra information about data types and a cleaner representation than the base data frame.



```
# A tibble: 5 × 4
  `Country Name`              `Country Code` `1976` `2009`
  <chr>                       <chr>           <dbl>  <dbl>
1 Aruba                       ABW              71.1   74.9
2 Africa Eastern and Southern AFE              48.8   57.5
3 Afghanistan                 AFG              40.7   60.5
4 Africa Western and Central  AFW              44.4   53.6
5 Angola                      AGO              43.1   54.3
```

2 Workflow

The R Studio environment provides with it many tools to improve workflow. Here by section I list these tools for your reference. By working and using these tools consciously eventually they become habit.

2.1 Troubleshooting

- help(*function*) or ?function : when called from the console it shows the manual page of the *function.*

2.2 Inline Workflow Tools

- Tab : tabbing when entering a name R Studio will show results for completions of the term in context.
- Ctrl + “R”: Look through and choose previous commands.
- ^ : like the command line pressing up may be used to retrieve previous commands made in the console.
- Alt + “-” : shorthand for variable assignment <-

2.3 Scripts and Functions

Scripts in R are used as a way to reproduce commands. When sourced R Studio runs all commands in the script and as such variables are stored in the local environment. Scripts can be used to reproduce many intricate commands to produce graphs and modify data. With the highest level variables being accessible to the user for evaluation or to further test changes on the data.

Within any environment users may declare functions which are separately contained local environments. Using these we may execute longer scripts which produce results which need only be stored temporarily. After a function has finished executing all local variables are wiped, which frees up space but local variables may be accessed during debugging.

2.4 Environment Controls

R Studio offers utilities for saving the local environment to later continue working on even after closing the program. This utility can be accessed through Session > Save Workspace As... this will save us time when we want to continue working on our environment after powering off the computer. It will also give us a fallback if a script we execute fails. It is however recommended that for regular use cases avoid restoring the environment. The purpose of scripts are to save the steps for reproducing our data if need be.

While working with many data sets we may need to move between multiple directories and change where we store files. To do so R has built in functions to set the working directory, this is *setwd(*“path”*).* Additionally, we may check which directory we are currently in by running *getwd().*

3 Packages

Through this tutorial we will be using the package tidyverse which comes with many other convenient packages for our purposes. For the sake of knowing what tools we are using better these packages and their purpose are listed below:

- tidyverse: In general, it is a collection of R packages for data science that all share a common philosophy, grammar, and data structures.
- dplyr: Common data manipulation tasks in R.
- ggplot2: Graphing package that provides common grammar for producing informing graphics.
- readr: Fast and transparent methods for reading rectangular data sets.
- magrittr: Piping data sets between functions to eliminate repeated variable re-assignment and renaming in between operations.
- lubridate: Working with date times in R.
- tidyr: Functions for tidying.
- Packagefinder (not from tidyverse): Package for finding packages on CRAN using keywords from R-Studio.

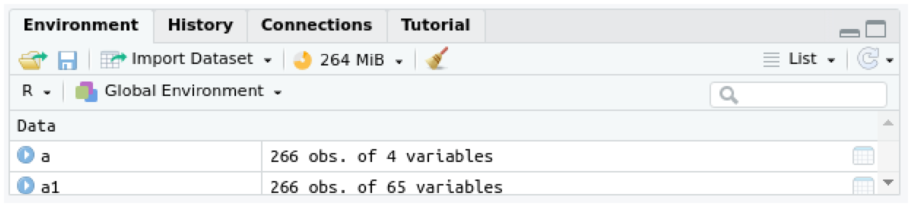
  Additional documentation is available at https://www.tidyverse.org/packages/ along with packages for importing from different sources and working with different data types.

4 Working with Data

From this point onward we will focus on the data frame, or **df**, when discussing data. However, we are going to coerce any data frame we import into another data type, the tibble. To explore these concepts, throughout this document we will be exploring a data set of world life expectancy from Our World in Data: https://ourworldindata.org/life-expectancy (Accessed 10/26/2021).

4.1 Importing Data

Data may be imported into R Studio using two methods: manually through the UI or by using any number of functions to import data. By using the I*mport Dataset* tab then selecting the type a user may manually import data. Though this only applies the base R reading function and cannot be customized.



```
Data
a     266 obs. of 4 variables
a1    266 obs. of 65 variables
```

We will use read_<type> in this case *read_csv*("filename/path") to import data. By default this will import data as a tibble and is much more flexible and transparent. Progress will be shown for importing the CSV and *read_csv()* will work better for non-standard formatting.

4.2 Data Operations

dplyr provides us with many functions that cover basic operations in relational algebra below listed are the functions we will make use of:

- Projection Π *select(**df**,* c(attr$_1$, . . . , attr$_n$)*)*: allows us to select specific columns to keep in a data frame. This can be done in 3 ways: passing a vector of attributes to keep, passing a vector of attributes to **remove,** *select(**df**, !*c(attr$_1$, . . . , attr$_n$)*)*, and by using supplied functions to logically match strings from the columns, we will go over this in more detail in examples.
- Selection σ *filter(**df**,* <logical statement>*)*: keep only rows that match certain criteria, these can use column/attribute names for comparison (e.g. *filter(**df**,* attr==value | `string attr` == value*)*).
- Joins X *inner_join(**df1**, **df2**) or (full_join, right_join, left_join)*: These classes of functions will join data frames on values that match between shared columns in the data frames. If the names of the columns are different or we only want to join on one shared column we can manually declare these names by passing (by.x, by.y, or by).

We can also transform data by either adding columns, mutating existing attributes, or summarizing data by grouping attributes. The functions we can use are as follows:

- *mutate(*__df__, attr = <attr or function wrapping attr> . . .*):* By re-declaring an attribute we can replace the values with adjusted or transformed values. We may also use many attributes and put them in one if we need a derived value. *transmute(*__df__, . . .*)* also operates on the same principle but it does not maintain all attributes from the data frame being modified.

  For the next case I will be using R with piping, this allows us to avoid storing inter operational steps like grouped data frames that we will not need for more operations. ***df*** *%>% function() replaces function(****df****).*

- __df__ %>% *group_by*(attr$_1$, . . . , attr$_n$) %>% *summarize*(attr = <function applied on attributes>): This will apply a function across a vector on the attribute(s) then put the final value(s) into attribute(s) representing the entire group. This function may be any which transforms a vector into a single value (e.g. *max(), sum(), mean(), . . .*). There is also a special function *n()* which counts the number of members in the group and returns that.

  Adding rows to a tibble is simple ***df*** *%>% add_row(attr$_1$=val$_1$, . . .) this will return the tibble with an added row where missing values are filled with NA.*

4.3 Tidying Data

Data does not often come in a form that lets us properly discriminate or express information for data science. In some cases data is stored in the name of attributes, or attributes are stored in another attribute. To extract this information tidyr provides us with four functions:

- __df__ %>% *gather(*attr1, . . . , key = <attr to store attr1, . . . in>, value = <attr to store observations in>*)*: This will take all the declared attributes and store their names and values as new attributes in the data frame.

- __df__ %>% spread*(*key=<attribute to turn into multiple attributes>, value=<attribute that stores value for that attribute>*)*: *spread* will turn a single attribute that stores many different observations for one row into many attributes.

- __df__ %>% *separate(*attr, into=c(attr$_1$, . . .), sep=<device used to separate parts or number of characters to separate from front>*)*: splits the value from attr into multiple attributes.

- __df__ %>% *unite*(attr, attr$_1$, . . . , sep=<string to add between portions being united into attr>): Take all the values from attr$_1$, . . . and join them into attr with the given separator.

5 Plotting Data

ggplot2 is a unified language for plotting in R that offers similarly named functions and a format for plotting data that remains standard across all graph types. It also cuts down on repeat information by storing the data early and only referring to attributes to reference from this.

5.1 ggplot2 Standard Format

By default any ggplot has the format:

ggplot(data=**df**) +

geom_<plot type>(mapping=aes(. . .), . . .)

In this format we clearly leave open that more may be added to these features but first lets discuss these in parts starting with geom. A geom is what ggplot uses to refer to a type of plot. A few we will focus on are:

- *geom_point()*: we use this for a scatter plot, we should declare an x and y for this clearly to plot properly. This can be used to examine the relationship between x and y and to a lesser extent examine if clusters exist in the data.
- *geom_bar()*: denotes a bar chart, we must denote an x for it to plot, in this case the x must be some discrete value. By default it counts all occurrences of the each case in the discrete attribute given to it, but we may give it a value to use by changing the stat. Bar charts give us a better idea about how much different features of the data are expressed in the data set we have. For example we may note that most of our cases fall in one discrete category or, with more splitting, that a certain features appear more commonly within other categories. We will discuss this further in section 4.2.
- *geom_histogram():* plots similarly to a bar graph but instead on continuous values and it will bin together similar values. As such it requires an x but similarly we can give it a value to use as identity. A histogram gives us an idea of whether the data is normally distributed or not.
- *geom_boxplot()*: takes a y value then plots a box plot for this data. A box plot gives us insight into the skew and potential outliers in the data.

5.2 Graphs Split on Discrete Features

ggplot2 includes 2 ways of graphing along different features, these are color or fill and facets. The difference is where they are expressed and how they change the plot, of course they can be used together. Note that combining these we may express up to three features simultaneously in a single plot.

First considering color, in any plot we may express color or fill this will change the color of the points or fill of the bars we plot. Color can take a value for one colour if we are plotting entirely separate features but the most common use case we encounter is coloring based on different discrete values our data exhibits. To this end any geom may simply take a discrete value as color:

```
ggplot(data=df) +
    geom_point(mapping=aes(x=..., y=..., color=attr))
ggplot(data=df) +
    geom_bar(mapping=aes(x=..., y=..., fill=attr))
```

For using continuous data on geom_point() and specific plots like geom_tile() color may be set to a continuous variable and will darken or lighten with an increase in value.

Our other option for expressing the features of our data is using facets, these group the data on features that we choose then use only the different groups for each plot. There are two types of facets that we use:

- *facet_wrap(*~attr*)*: creates different plots based on grouping the data by the one given attribute.
- *facet_grid(*attr$_1$ ~ attr$_2$*)*: creates a grid of each combination of values in the two attributes and plots rows and columns which express the two values.

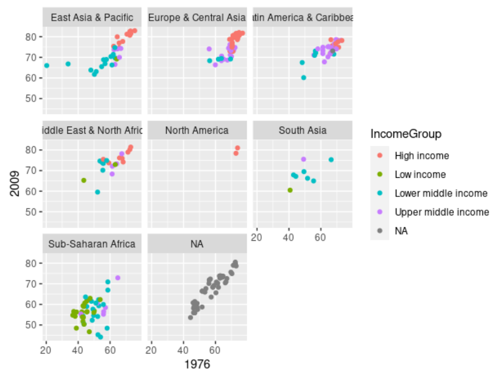
Given both of these we may express a full plot in the following format:

```
ggplot(data=df) +
    <geom function>(mapping=aes(. . .)) +
    <facet function>(. . .)
```

When we begin modelling functions where many geoms will be used on the same plot we can declare the mapping in the first statement, ggplot(. . .). By using this all graphs we plot will inherit the mapping we chose.

Adding additional plots onto the first may be done by simply "adding" the new plot to the original. This can be done exactly how we added the facet function as above, just replace it with another geom. These may even represent different data if we define data in the new geom function.

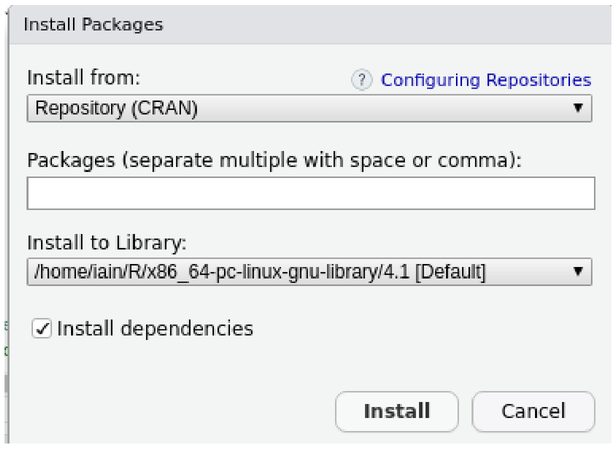
```
ggplot(data=c)+
  geom_point(mapping=aes(x=`1976`, y=`2009`, color=IncomeGroup))+
  facet_wrap(~Region)
```



6 Adding Modules

Adding modules to R in most cases only requires two steps: installing and loading. Installing can and should be done through menus if just for your purposes, installation can be accessed through tools>install packages... In this menu a user can name any number of packages they want to install, R will then find and download the packages. After this to load and use packages a user need only run: library(<package name>) this will load the package into the local environment. This should be done in the head of any R-Script for all packages used in that script.

Although packages allow the use of any of their functions without referencing the package they come from in larger projects it is recommended that functions from outside libraries are called <package name>::<function>. This gets around masking which can occur when two packages name a function the same way and will save headache finding where an error occurs or a function behaves differently in large scripts.



Install from: Repository (CRAN)

Packages (separate multiple with space or comma):

Install to Library: /home/iain/R/x86_64-pc-linux-gnu-library/4.1 [Default]

✓ Install dependencies

For Linux users you may encounter that R-Studio cannot find and download a package. Especially in larger packages with many requirements this can cause a lot of waiting which just leads to failure. In this case, the final few lines will say which required packages could not be found. Then the packages can be manually downloaded as tars from CRAN's website then loaded as such by choosing install from package repository in the installation menu.

Another alternative is changing the mirror to install packages from since different locations have access to alternative versions and sets of packages.

7. Exporting Information

After recombining and tidying data is done, which may take a while, it is better to save and reuse data that is in its final format rather than reload that data every time. Additionally, you may store results in a data frame or matrix and need to store these for more tests later or to communicate results. Maybe you have already calculated results and formed many graphs automatically to communicate them. No matter the final purpose we must store our results to present them when the time comes, so R and tidyverse include built in functions to save data.

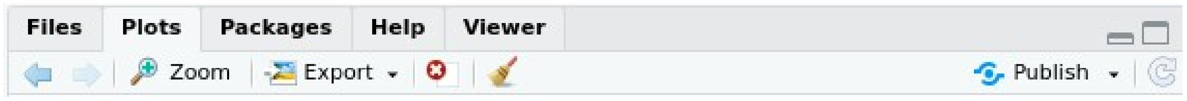
7.1 Exporting Data

To export data readr includes the function family *write_<type>()* specifically we will use *write_csv(***df**, "filename.csv"*)*. This should be paired with *paste0()* to make unique, specific names for any table you need to save, especially for clustering results.

paste0() takes any number of arguments of any type that can be coerced to a flat string, combines them and returns the result.

7.2 Exporting Graphs

Saving plots can be done in two ways, either manually through the export button on the select tab or through a script. Since we use ggplot2 we have access to a convenient function *ggsave(*"filename.format"*)* this will save the most recent plot made by ggplot2.



8 Intelligent Tools

We have access to many intelligent tools through CRAN, it is important to keep in mind that some of these tools need different formats of the data to be used. Tibbles, although excellent for clarity and communication, do not work with every function, so sometimes we will need to return to the base data frame or change the format to something new. Because of this for the listed functions we will include the data type it must take.

8.1 Association Rules

Association rules can be used to discover relationships between combinations of values that can occur in transactions. Although important support for association rule mining is only available in one package from CRAN:

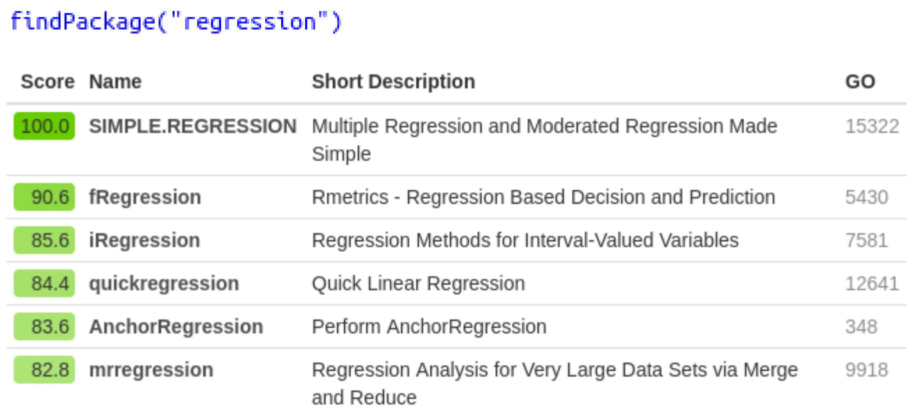
- arules: contains *apriori()* which takes a data set of type transaction (which must be formatted this way) and mines association rules from it.

8.2 Classification and Regression

Classification and regression are methods used to predict the outcome of an attribute or multiple attributes by analyzing results found in a data set.

- dtree: package that contains regression and decision tree functions for classifying discrete and continuous values. Takes a data frame and a method to produce the decision tree or regression model. Also optionally can quantify instability of the decision tree to indicate when results can be trusted.

- arulesCBA: package of methods that use rules generated by arules for classification.
- Many other packages are available for classification on CRAN these can be found with a search using *findPackage()*.



```
findPackage("regression")
```

Score	Name	Short Description	GO
100.0	SIMPLE.REGRESSION	Multiple Regression and Moderated Regression Made Simple	15322
90.6	fRegression	Rmetrics - Regression Based Decision and Prediction	5430
85.6	iRegression	Regression Methods for Interval-Valued Variables	7581
84.4	quickregression	Quick Linear Regression	12641
83.6	AnchorRegression	Perform AnchorRegression	348
82.8	mrregression	Regression Analysis for Very Large Data Sets via Merge and Reduce	9918

8.3 Clustering

Clustering is a method of splitting information into groups based on resemblance of their features. Unlike regression and classification it does not require a pre-existing label to make groups.

- fastcluster: provides a function *hclust()* which applies hierarchical clustering to quickly cluster data.
- HDclassif: package that provides function *hddc()* for clustering high dimensional data. Takes a matrix with rows denoting observations and columns which are dimensions of observations. Applies an expectation maximization algorithm to cluster results.
- funHDDC: a package which provides the function *funHDDC()* for clustering multivariate functional data using an expectation maximization algortihm.
- More methods are available to: https://cran.r-project.org/web/views/Cluster.html

  If exploring functional data is of interest then the package fda provides all the tools necessary to fit and plot functional data.

9 Conclusion

I hope this handout has provided a clear set of tools and references for data science in R. More resources are available online from tidyverse including cheat sheets for each package we use. As with all programming languages using them and finding new resources will lead to the most growth, applying troubleshooting techniques and packages we import will help with finding options and packages to fulfill any task.