Mathematics and Programming for Machine Learning with R for Business and Engineering

William B. Claster

Contents

Table of Figures

original variable	newVar1	newVar2	newVar3	newVar4	newVar5	newVar6	newVar7	newVar8	newVar9	newVar10
5	0	0	0	0	0	1	0	0	0	0
0	1	0	0	0	0	0	0	0	0	0
4	0	0	0	0	1	0	0	0	0	0
1	0	1	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	1

1 -Functions Tutorial

- Functions with and without arguments
- Defining versus running
- Arguments
- Parameter versus Argument
- Argument Order
- Environments
- Scope

Replicating the results shown here

Note that this text uses the following version of R: version 3.4.2 (2017-09-28), To obtain the same results as this text you may install this version at CRAN.

Knowledge of R

It is assumed that the reader has gone through the tutorial offered through an R package called Swirl. You can find out more information at https://swirlstats.com/. The tutorial can be run inside RStudio and gives a very good course in R. It is expected that the reader has completed the tutorial through the section on functions.
According to the website you can run the tutorial in RStudio by following steps below.

Step 1: Get R. In order to run swirl, you must have R 3.1.0 or later installed on your computer. ...
Step 2 (recommended): Get RStudio. ...
Step 3: Install swirl. ...
Step 4: Start swirl. ...
Step 5: Install an interactive course. ...
Step 6: Have fun!

Getting set up

To get things going you should install R and install the free version of RStudio. Here are the steps to follow.

1. Download and install R from https://cran.r-project.org/. On Windows, if you don't mind living dangerously, I recommend that you use the "install as administrator" option because sometimes R wants to create folders, for example when new packages are installed, and if it is not installed and run as administrator, Windows may block the creation of those folders.
2. Download and install the free version of RStudio from https://www.rstudio.com. Same remarks as above regarding "install as administrator".
3. Run RStudio
4. Configure RStudio to look the way you want. You can adjust the appearance in the Global Options menu which is under Tools. When you open RStudio it may look something like this.

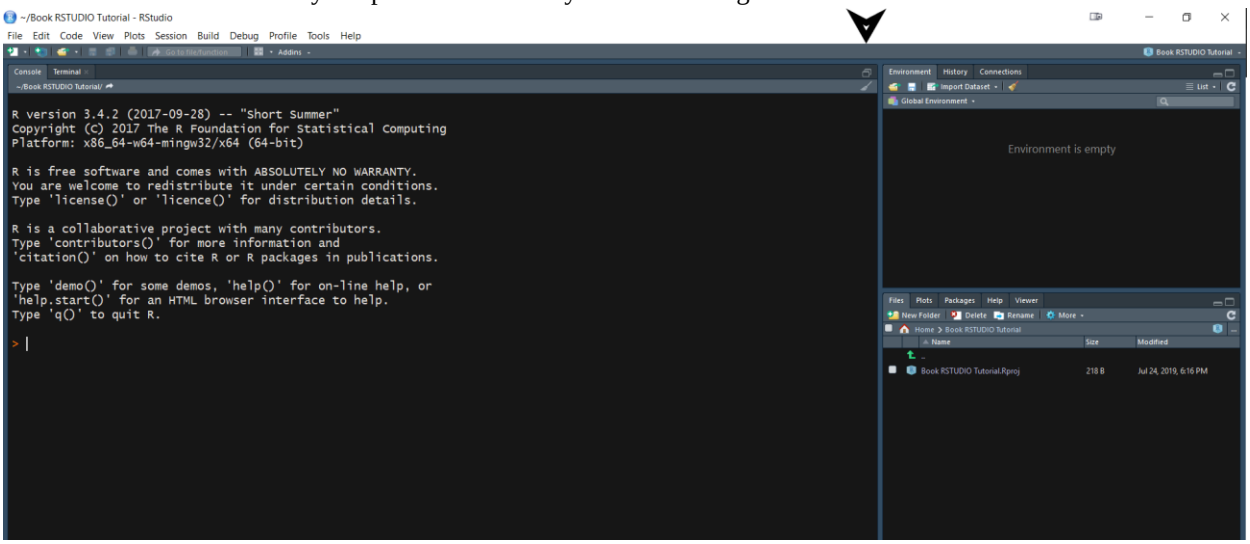

Figure 1-1 Initial configuration of RStudio

5. The so-called console window is the large window on the left. On the right are two other windows. Each window has various tabs that you can select. The first tab on the top right-hand window is the Environment tab and it will show any variables you have created, functions you have created, datasets that have been loaded and so on. The bottom window currently is showing the first tab which is just a file browser for the current directory that you are working in. You can see that the current directory I have open is Home>Book RSTUDIO Tutorial. There is one window which is not showing now that is important to have. It is called the source window. The console allows you to type single commands into R and when you hit the enter key, R will run that code. However, most of the time, when I am writing code, I don't want to evaluate each time I write a new line of code. I want to be able to write several lines of code first and then run it. This can't be done in the console window. The source window is used for this. To show the source window you can either click on the icon of two little boxes where the arrow is in the figure above or go to Tools<Global Options<Pane Layout. I prefer to have the console window above and the source below, but you can arrange as you wish. Also, as you can see, my background is black. This can also be adjusted in Global Options, as can many other options. Once you have the source window visible you can type code in it and then if you want to run a particular set of lines, select those lines, and then press the button on the upper right-hand side of the source window that says Run. Instead of pressing the button you can also select the code to be run and press the control key (on Windows) and simultaneously press the enter key.

To run the tutorial, you need to install the Swirl package. The simplist way to install a package is to go to Tools>Install Packages. Then type in the name of the package you are looking for, in this case swirl. Then click the install button. RStudio will download and install the package and when it is finished you will see the prompt symbol (looks like '>') in the console. According to the website on Swirl, next type `library("swirl")` and hit the enter key. Then type `swirl()` and hit the enter key. Next according to the website it says, "the first time you start swirl, you'll be prompted to install a course. You can either install one of the recommended courses or visit our course repository for more options. There are even more courses available from the Swirl Course Network. If you

follow the instructions it will bring you to a screen where you can choose the course you want to study". The recommended course for this text is called R Programming: The basics of programming in R and it is expected that you have completed up to and including the section on functions.

Functions Tutorial

It is assumed that you have gone through the above tutorial at this point. Some of the following is a review of the Swirl course.

Our tutorial will concentrate on building functions since this is a critical skill in the material that follows.

Functions coded in this chapter
myFirstFunction
addFive
addThem
divid
outerFunction
embeddedFunction

Figure 1-2 Functions coded in chapter 1

1.2 Functions

If you have used R at all you have already worked with many functions. A function we often encounter is `mean(x)`

```
x=1:5
mean(x)
```

```
## [1] 3
```

Another easy function is the function to get the date on the computer system you are using, `Sys.Date()`.

```
Sys.Date()
```

```
## [1] "2018-10-02"
```

There is an important difference between these two functions. They both have names and they both have parentheses but there is nothing inside the parentheses when we run Sys.Date(), however there is something inside the parentheses when we run `mean(x)`. The `mean()` function requires an input. In fact, if we run it without one, we get an error message.

```
mean()
```

```
Error in mean.default() : argument "x" is missing, with no default
```

Admittedly, R has error messages that are not so easy to figure out but if you skim them you can sometimes figure out what kind of problem has occurred.

Write our first R function

Let's make this simple. To write a function is as simple as

```
myFirstFunction <- function(){
  print("Hi There!")
}
```

This shows the syntax of creating a function. We can divide the construction of this function into 3 parts. Before reading what these 3 parts are, look at the above statement and make a guess as to where the 3 parts might be (I am not referring to the <- symbol as one of the parts but of course that needs to be there).

1. It needs a name[1]:

```
myFirstFunction
```

2. You need to tell R that this name is the name of a function (as opposed to a text (character) variable, or a vector, or a list, or whatever else it might be).

```
function()
```

3. You want the function to do something and you put that inside the curly brackets `{}`

```
{
  print("Hi There!")
}
```

This function will just print out a string: `"Hi There!"`. Note that what is written between the curly brackets is referred to as the body of the function. The body of our function contains

```
print("Hi There!")
```

Defining the function versus running it

Although we define a function, nothing happens until we run it. That is done by typing its name along with the succeeding parentheses.

```
myFirstFunction()
```

Note that in RStudio, if the definition is typed into the console and the enter key is hit then the function is defined. However, if the definition is typed into the Source panel, it won't be defined until the *code is run* in the Source panel.

However, even after the function is defined, there will be no response from R until the function is run (this is usually referred to as "calling the function"). So, Hi There! will not appear in the console until myFirstFunction() is run, either by typing myFirstFunction() into the console and hitting enter, or typing myFirstFunction() into the Source

[1] There are cases where it is not necessary to name a function. That topic is called anonymous functions. We will not discuss it here.

panel and running it. If either of these is performed, the function myFirstFunction() will be called, and then it will perform the code within the curly brackets, which in this case is to print out Hi There!

Functions can do far more than simply print a string, but the code above is the full implementation of a function.

Practice 1.2.2

1. Write a function named *fact* that will print the string “Tokyo is in Japan”.

Hopefully you will have functions do more than just print a string but nevertheless we have constructed a basic function.
What if you wanted to write a function to print the string "Tokyo is in Japan", how would you write that? Try it. And how would you run it (call it)? Try this in your Rstudio. If you can do this, you have created your own function.

Arguments

Now try something a little more interesting. Write a function to add 5 to a given input. In this case, we need to add one more piece to the syntax of writing a function. The function needs to accept an input. It needs to accept given number. We say the function will have an argument. The etymology of the word argument is fairly convoluted and comes from something in Old English. But for us, argument refers to the input that we put into the function. If we call the function addFive() we want it to take an input, say 7, and then output 7+5.

Ok, so now we have decided what we want addFive() to do. Now we have to write it. It needs the first 3 parts mentioned above plus a place to put the argument. Also of course we have to change what is put into the curly brackets {} because we don't want the function to print out Hi There. We want it to add 5 to the input.

```
addFive <- function(inp){
  outp=inp+5
  return(outp)
}
```

We need a symbol to hold the input. We have choosen inp. Then inside the curly brackets we add 5 to inp and assign that to the variable outp. Finally we use the function return(). return() at the end of our code will determine what the output of the function will be. In our case the output is contained in outp. If we run the function with an argument of 7 we should expect an output of 12.

```
addFive(7)

## [1] 12
```

1.3 Parameter versus argument

If you want to be able to read explanations on the web about programming issues you need to understand the difference between the two words *parameter* and *argument*. We have used the term argument. What is meant by parameter? We want to distinguish between what the user puts into our function (in the last case it was 7) versus the symbol that holds that value, and which is later used in the body of the function. The symbol we used to hold the argument is inp. This would be called a parameter. Thus, the argument entered into the function is 7 but the parameter is inp. It is very useful to be able to distinguish between these two when discussing our programming code and so the concept should be well understood.
We could certainly write a function which accepts 2 arguments. In that case we would use two parameters. For example if we want to take two inputs and get their sum we could write a function called addThem(). We hope that if we put 1 and 3 into addThem() it will return or output 4.

```
addThem <- function(inp1,inp2){
  outp=inp1 + inp2
  return(outp)
}
addThem(1,3)
```

Practice 1.3.3

1. What are the parameters in addThem()?
2. What were the arguments we used in the addThem() test above?
3. What was returned in the addThem() test above?

In the addThem() function we said there were two parameters inp1 and inp2. We will refer to the list of all the parameters in the definition of the function as the parameter list. So here we would say that the parameter list has two elements inp1 and inp2. A function could have any number of parameters in the parameter list. One way to get an idea of what a function is doing is to look at the body of the function (what is inside the curly brackets). If there is a variable inside the curly brackets, then check to see if that is in the parameter list. If it is, it means that this value will be supplied when the function is called and then used in the way that is written inside the curly brackets. Another good practice for understanding what a function does, is to check the output of the function. That is, check the final return() statement of the function. This will tell you what is output and the form of the output. So far, we have had single numbers as the output of our functions, but it is possible to output other types like lists or vectors2. Checking the return statement is a quick way to get an idea about the function. Note also that if the function does not have a return statement, it will not show the value of the output when you run the function but it is still assignable to a variable which when printed (or just typed at the command line), will show that value.

```
addThem <- function(inp1,inp2){
  outp=inp1 + inp2
  #Leave out the return statement
}
addThem(1,3)
k=addThem(1,3)
k
```

```
## [1] 4
```

Some readers may object to the sentence above where we wrote, “So far, we have had single numbers as the output of our functions”, saying we output the text string “Hi There!”. In programming we make a distinction between what a function prints out and what it returns. It seems like an odd distinction to make and actually in R it may not be an important distinction but typically in most programming languages we cannot assign the value of the function myFirstFunction() to another variable, whereas we can certainly assign the output of addThem(1,3) to a variable. However, in R it is actually possible to assign the myFirstFunction() to a variable. This is not typical for most programming languages.

```
y=addThem(1,3)
y
```

```
## [1] 4
```

```
y+3
```

```
## [1] 7
```

[2] Outputting a list is a convenient way to output more than one item.

```
#This actually works in R. Notice that even during the assignment, the functi
on actually runs and actually outputs "Hi There!".

x=myFirstFunction()

## [1] "Hi There!"

X

x

## [1] "Hi There!"
```

1.4 Argument Order and Parameter Names

In the calling of a function (when you actually use it) when you explicitly designate argument values by name using the corresponding parameter (in our last example, if you wrote inp2=1,input1=3), the ordering of the arguments becomes unimportant

Let's create a function called divid() which will take 2 arguments and divide the first by the second.

```
divid <- function(numerator,denominator){
  result=numerator/denominator
  return(result)
}
#The order of the arguments clearly matters.
divid(8,2)

## [1] 4

divid(2,8)

## [1] 0.25
```

```
#If we specify which parameter, we want a particular argument to be
assigned to, then the order does not matter. Here we are putting 2
first and then 8 but we have specified that the 2 is for the
denominator parameter not the numerator parameter and the 8 is for the
numerator parameter.
divid(denominator = 2, numerator = 8)
## [1] 4
```

Practice 1.4.4

1. Write a function called *poly*, with 2 arguments which adds the square of one of them to the cube of the second.
2. Run poly(2,3) then run poly(3,2)
3. Try switching the order of the arguments but using the parameter names poly(sq=2,cu=3) and poly(cu=3,sq=2). What is your conclusion?
4. What happens if you run poly(cu=3,2)?

Before we go on, we briefly discuss the concept of environment in R.

1.5 Environments

When we start up R and then create a variable, for example

```
x <- 7
```

then R needs to keep a record of the fact that a variable x has been created and that variable has been assigned the value 7. We say that the variable x has been stored in the global environment. An environment is just a place to store variables. It is like a piece a paper with the names of each variable and the values of those variables. It also keeps a record of the functions that have been created. In fact, there is a command we can run to see what is currently on that "sheet of paper", i.e., what is in the global environment. It looks like this.

```
ls(globalenv())
```

The `ls()` function will give a list of items and so when we put `globalenv()` into , we find out what is in `globalenv()` .

For example, if we created the function called `myFirstFunction()` and also created the variable x, then we would get

```
myFirstFunction <- function(){
  print("Hi There!")
}
myFirstFunction()

## [1] "Hi There!"

x <- 7
ls(globalenv())

## [1] "myFirstFunction" "x"
```

This tells us that myFirstFunction and x are in the global environment.
We can actually create separate environments. This is done with

```
my.env <- new.env()
#Now we have created a new environment. It is currently empty
ls(my.env)

## character(0)

#However, we can assign variables values in this new (separate) environment like this.
assign("y", 100, envir=my.env)
#We can also use the $ sign to assign variables to the new environment, like this.
my.env$thursday = "The weather is sunny"
#Now if we check the contents of the my.env we will see both of these variables.
ls(my.env)

## [1] "thursday" "y"

#So each environment is like a separate sheet of paper and they contain all the variable definitio
ns (and function names as well).
```

```
#In fact, we can have a variable x in the global environment and another, separate variable, x, in
 the my.env environment. They can have different values even though they both are called x.
assign("x", 20, envir=my.env)
#We can use the name of the environment, followed by a $ sign, followed by the name of the va
riable to see what the values of the variables are. (Note, the name of the global environment is .
Global)
.GlobalEnv$x

## [1] 7

my.env$x

## [1] 20
```

Generally, we don't create new environments for most of our programming tasks, but it is useful to see how once created, the items in the environment can be accessed. On the other hand, R will create a new environment every time any function is called. In our addthem() example

```
addThem <- function(inp1,inp2){
  outp=inp1 + inp2


}
```

if we run addThem(3,2), then a new environment is created (a new sheet of paper) which will contain the variables inp1, inp2, and outp along with the values each variable is assigned.

> By the way, a phrase often used in computer science is "bindings". This usually refers to the assignment of a value to a variable. Thus we could say that when we call addThem(3,2), R creates a new environment with 3 bindings[3].

We will see in a moment, that the concept of environments is tied to the important concept of scoping.

We saw above that we can have a variable named x in two separate environments and even though both variables are called x, they don't have the same values. That is, the fact that two variables have the same name is akin to two people having the same name. They may have the same name, but they certainly are not the same people. With regard to our 2 variables, if they reside in different environments, they are different variables. Now we said earlier, that when a function is called, it creates its own environment, its own piece of paper, where its variables and values are stored (where its bindings are stored).

Next, we consider what happens if we define a function inside another function. What do the R rules say about variables defined in the outer function. What do R rules say about variables defined in the inner function? If a variable is defined in the outer function, will R allow the inner function to access that variable? And visa-versa, if a variable is defined in the inner function, can the outer function access it? And also, if a variable in defined in a function, can we access that variable outside the function. These are often described as properties of the scope of a variable.

What is the scope of a variable defined within a function? Does it extend to functions that are defined within this function? Does the scope extend outside the function? Suppose we have a function called outerFunction(), and then

[3] More generally, binding refers to a mapping of one item to another.

inside that function we define another function called embeddedFunction() -yes, we are allowed to define functions inside other functions. What does the notion of environment imply about this situation?

```
#We define a function within another function. Then we call the outer functio
n. The outer function also calls the embedded function (by way of a print sta
tement)
outerFunction<- function(){
  myVar=2
  embeddedFunction <- function(){

    myVar=5
    return(myVar)
  }

  print(embeddedFunction())
  return(myVar)

}
```

What will happen when we call outerfunction()? In particular, when it returns myVar, which myVar will it return, the myVar defined in outerFunction (and thus return a 2) or the myVar defined in the embeddedfunction (and thus return a 5). Further, what will print(embeddedFunction) do? These are decisions that the designers of R had to make. Lets see what they decided R should do.

```
#Call outer function, which will return its myVar value but also print the output of
the embedded function
outerFunction()
```

```
## [1] 5

## [1] 2
```

What is the explanation for the above *output* of 5 and then 2? Note that at the end of the definition of `outerFunction()` we have both a `print()` statement and a `return()` statement. This is why there are 2 lines of output when we run `outerFunction()`. Why are the outputs not the same? They are both returning (or printing) `myVar`. `embeddedFunction()` returns `myVar` and so does `outerFunction()`, so why don't they both return the same value?

First notice that the first line of the body of `outerFunction()` defines a variable called `myVar`. Next in the body of `outerFunction()` there is a new function being defined called `embeddedFunction()`. That is, `embeddedFunction()` is being defined within `outerFunction()`. Now examine

the body of `embeddedFunction()`. It also defines a variable called `myVar`. Are these two variables the same or do they just have the same name. Remember that when a function is called it creates its own piece of paper to write its variables and values. At this point in the code, we have not called either function yet. But after defining the functions above, we do call the function `outerFunction()`. When `outerFunction()` runs, it will call `embeddedFunction()`, within the `print()` statement. When `print(embeddedFunction())` is run, this causes `embeddedFunction()` to be called and a new environment is created where the variable `myVar` is written with a value of 5. At the end of the execution of `embeddedFunction()`, the return statement with `myVar` will give 5. However, after `embeddedFunction()` returns 5, it completes its work. We say that then "control" passes back to `outerFunction()`. This will mean that we are switching to the environment of `outerFunction()`. When, finally, the last statement, `return(myVar)`, in `outerFunction()` is run, since this occurs within the environment of `outerFunction()` `myVar` will be 2.

There is one more concept associated with environments that we should discuss. Environments are said to be "nested". That is to say, in our example, if `myVar` was only defined in `outerFunction()` and not defined within `embeddedFunction()`, then we might expect an error to occur when

we run our code since `embeddedFunction()` will run and try to return `myVar` but `myVar` does not exist within `embeddedFunction()'s` environment. However, that is not what happens. What R does is search for `myVar` in other enviroments. In particular, R searches in the calling function's environment[4]. The calling function in this case is `outerFunction()` and R will find a variable called `myVar` there and this is the variable that will be used in . In this sense the environments are nested. Here is the example where we do not define `myVar` within the embedded function but try to access it within the embedded function anyway.

```
outerFunction<- function(){
  myVar=2
  embeddedFunction <- function(){
    #In this example, we do not create a variable called myVar within the emb
edded function.
    #myVar=5
    return(myVar)
  }

  print(embeddedFunction())
  return(myVar)

}
#Call outer function, which will return its myVar value but also print the ou
tput of the embedded function
outerFunction()
```

```
## [1] 2
```

```
## [1] 2
```

[4] Actually, R searches even beyond the calling functions environment. To see all the environments that R will search in (the so-called search path), run the command `search()`.

We see that the value of `myVar` is 2 in both cases, but in particular it is 2 in the embedded function even though it was not defined there. R finds `myVar` in the calling function `outerFunction()` and uses that binding.

$$\frac{\partial J}{\partial W^{(2)}} = \begin{bmatrix} \frac{\partial J}{\partial w_1^{(2)}} \\ \frac{\partial J}{\partial w_2^{(2)}} \\ \frac{\partial J}{\partial w_3^{(2)}} \end{bmatrix} \tag{1.1}$$