Foundation of Machine Learning

CSE4032

Lecture 00: Concise tutorial for R scripts

Dr. Kundan Kumar
Associate Professor
Department of ECE

Faculty of Engineering (ITER)
S'O'A Deemed to be University, Bhubaneswar, India-751030

Outline

Introduction

- R is an open source language for doing statistics. It is available from http://www.R-project.org.
- An open source IDE, RStudio, is available from https://www.rstudio.com/.
- Thousands of packages are available from CRAN (the Comprehensive R Archive Network).
- To download packages from CRAN into your library, execute the R command `install.packages(quotedNames)` or `update.packages(quotedNames)`.
- To use a package, enter `library(unquotedNames)`. Also, `library()` will tell you which packages you have installed, and `search()` will say which packages are currently loaded and available.
- Names in R are case-sensitive. A dot (.) is often used as part of a name; it isn't an operator.

Types of objects

- The simple object types are:

`numeric`	:	The default type for numeric literals, equivalent to "real," "float, or "double" in other languages.
`integer`	:	To indicate a numeric literal is an integer, an "L" suffix is required, e.g. 5L.
`complex`	:	Numbers with an imaginary part, denoted by an "i" suffix, e.g. $4 + 5i$.
`character`	:	Zero or more characters enclosed in single or double quotes.
`logical`	:	(boolean) Legal literal values are `TRUE` (or T) and `FALSE` (or F).
`factor`	:	A vector of "levels" or category names, converted to small integers.
`function`	:	In R, functions are another type of object.

Types of objects

- The simple object types are:

  `NA` : "Not available," with the literal value `NA`. Any operation involving an NA value (for example, 5 + NA) results in NA.

  `NULL` : is the null object. It has strange properties.

- Data structures:

  `vector` : A sequence of one or more values.

  `array` : A multidimensional vector.

  `matrix` : A rectangular arrangement of values; that is, a two-dimensional array.

  `list` : The values in a list may be of different types, and may have names.

  `data.frame` : A list of vectors.

Operators

$+$	:	add	$<$	:	less than
$-$	:	subtract	$\leq$	:	less or equal
$*$	:	multiply	$==$	:	equal
$/$	:	divide	$!=$	:	unequal
$\hat{}$	:	exponentiate	$>=$	:	greater or equal
$\%\%$	:	modulus	$>$	:	greater
$\%/\%$	:	integer divide	$+$	:	add

&&	:	logical and
&	:	vectorized and
||	:	logical or
|	:	vectorized or
!	:	not

Vectors

- Simple objects are considered to be a vector containing one object. For example, `3[1]` (where `[1]` denotes the first element of the vector) is the same as 3.
- The `c` function constructs a vector of its arguments, for example, `c(obj1, obj2, ..., objN)`.
- Arguments which are vectors are "flattened" into a single vector, for example, `c(1, c(2, 3))` is the same as `c(1, 2, 3)`.
- c coerces all objects in the vector to be of the same type, where NULL $<$ raw $<$ logical $<$ integer $<$ double $<$ complex $<$ character $<$ list $<$ expression. It also removes all attributes, except names, from the objects.
- The `class(type)` of a vector is the class of its components, for example, `class(c(1, 2, 3))` and `class(2)` as `"numeric"`.

Vectors

- A vector of numbers can be generated by `from:to`.
- The difference between adjacent numbers in the vector is 1 (if `from < to`) or -1 (if `to < from`). `from` and `to` are not restricted to integers; `from` is always included in the vector, while `to` is included if `last-from` is an integer.
- In an expression, the colon (:) operator has the next highest precedence, after subscripting.
- The `seq` function has arguments `from`, `to`, `by`, `length.out`, and `along.with`.
- Arguments may be given in order or referenced by name. It is an error to give possibly incompatible arguments.
  - `seq(from=1, to=5, by=1)` is equivalent to `1:5`.
  - `length.out` specifies how many numbers should be in the resultant vector.
  - `along.with=vector` is equivalent to `1:length(vector)`.

Vectors

- A vector containing `n` repetitions of value `v` can be created with the function `rep(v, n)`.
- Arithmetic operations work on vectors. If one vector is shorter than another, and the length of the longer vector is a multiple of the length of the shorter vector, the elements of the shorter object are recycled. Examples:
  - `c(1, 2, 3) + c(100, 100, 100)` gives the vector `101 102 103`.
  - `100 - c(1, 2, 3)` gives the vector `99 98 97`.
  - `1:6 * c(10, 100)` gives the vector `10 200 30 400 50 600`.
  - But: `1:5 * c(10, 100)` gives the error message longer object length is not a multiple of shorter object length.

Vector subscripting

- Positive numbers, starting at `1`. For example, `(1:10)[3]` is `3`.
- Negative numbers. The value returned is a vector with the element at the positive value of the subscript removed. For example, `(1:5)[-4]` is the vector `1 2 3 5`.
- A vector of positive numbers. For example, `(101:150)[c(2, 3, 7)]` is `102 103 107`, while `(101:150)[5:8]` is `105 106 107 108`.
- A vector of negative integers. For example, `(1:7)[c(-1, -4, -6)]` is `2 3 5 7`. However, positive and negative subscripts cannot be mixed.
- A vector of logical values, where only the TRUE values are retained. For example, `(1:8)[c(FALSE, TRUE, TRUE, FALSE, TRUE, FALSE, TRUE, FALSE)` is `2 3 5 7`.
- A logical condition on the named vector. For example, if `s` is `c(3, -5, 7, 1, 4)`, then `s[s > 1]` is `3 7 4`.

Vector subscripting

- Simple objects are treated as vectors of length one, and may be subscripted. For example, `c(10, 20, 30)[1]` and `10[1]` both result in `10`. So does `10[1][1][1]`, as `10` and `c(10)` are equivalent.
- Brackets (subscripting) has higher precedence than the colon (:). Thus, `(10:20)[3]` is `12`, but `10:20[3]` is equivalent to `10:(20[3])`, which is illegal (`20` is a vector with only a single value).

Matrices

- A matrix is a two-dimensional array. An array can be constructed with the function `array(data, dim)`, where `data` is a vector of values and `dim` is a vector giving the length of each dimension; the leftmost subscript varies the fastest.
- A more direct way to construct a matrix is to arrange the elements of a vector into a specified number of rows and columns, using the function `matrix(data, nrow, ncol, yrow)`.
  - `matrix(1:6)`, or `matrix(data=1:6)`, gives a matrix consisting of 6 rows and 1 column, containing the numbers 1 through 6.
  - `matrix(1:6, nrow=3, ncol=2)` gives a matrix of 3 rows and 2 columns, containing 1 2 3 in the first column, and 4 5 6 in the second column. Either nrow or ncol may be omitted, with the same result.
  - `matrix(1:6, nrow=3, ncol=2, byrow=TRUE)` gives a matrix of 3 rows and 2 columns, containing 1 2 in the first row, 3 4 in the second row, and 5 6 in the third row.

Matrices

- The elements of a matrix can be accessed with `[rows, columns]`, where rows and columns are vectors (possibly single numbers). A comma (,) indicates an entire row or column. For example,

```
> m <- matrix(1:9, nrow=3)
> m
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

```
> m[2:3, c(1, 3)]
     [,1] [,2]
[1,]    2    8
[2,]    3    9
```

```
> m[,]
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

```
> m[2,]
[1] 2 5 8
```

```
> m[,2]
[1] 4 5 6
```

Matrices

- If the number of data elements is less than `nrow*ncol`, and `nrow*ncol` is a multiple of the number of data elements, then the data elements will be recycled. For example, `matrix(1:3, nrow=2, ncol=12)` yields the matrix

```
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12]
[1,]    1    3    2    1    3    2    1    3    2     1     3     2
[2,]    2    1    3    2    1    3    2    1    3     2     1     3
```

- Conversely, if the number of data elements is a multiple of `nrow*ncol`, only the first `nrow*ncol` elements will be used; the remainder are discarded.
- `dim(x)` will return a vector of dimensions, `c(nrow, ncol, ...)` of the matrix, while `dim(x) <- vector` can be used to change the dimensions.
- `rbind(matrix1, matrix2)` returns a new matrix containing the rows from `matrix1` followed by the rows from `matrix2`. `cbind(matrix1, matrix2)` returns a new matrix containing the columns from `matrix1` followed by the columns from `matrix2`.

List

- A list is a collection of tag/value pairs, similar to a dictionary or map in other languages, where the "tag" is the name or key for the value.
- Functions in R have named parameters; for example, `seq` has the parameters `from` and `to` (among others). These names may be used in the function call, as for instance `seq(from=5, to=10)`.
- The list function, which creates lists, uses the same syntax, but a different semantics. Arguments to list may be given tags (names) chosen by the user, for example, `list(employee="Mary", salary="60000")`. Unnamed values are tagged with their location in the list.

Lists

- If `m` has the value `list(employee="Mary", salary="60000")`, then the value of salary may be retrieved in three different ways:
  - `m $ salary` (or more commonly, `m$salary`).
  - `m[[2]]`. Double brackets are required to index into a list.
  - `m[["salary"]]`, or `m[[s]]` where `s="salary"`. Double brackets are required when the name is only known as a string.
- `names(x)` will return a vector of names of `x`. Unnamed elements of the list will result in empty strings in the vector.

Factors

- A factor is like a vector with additional information, called `levels`.
- There is a separate level for each unique value in the vector.
- A factor can be created by applying the factor function to a vector, for example, `factor(c("M", "F", "M", "M", "F"))` results in a factor containing not only the five values in the vector, but also the levels `"F"` and `"M"`.
- The function levels, applied to a factor, returns a vector of the level values. By default, the levels are in sorted order.

Data frames

- A data frame is a table in which the rows are numbered and the columns are named. (There may be other versions.)
- To construct a data frame, create one or more named vectors, then call the function `data.frame(vec1, vec2, ..., vecN)`, where the `veci` are the names of the component vectors. For example,
  `name <- c("Mary", "Sally", "Bob");`
  `salary <- c(60000, 55000, 80000);`
  `df <- data.frame(name, salary)` creates the data frame `df` as

```
   name salary
1  Mary  60000
2 Sally  55000
3   Bob  80000
```

Statement types

- Control structures in R are typical of the C family of languages, hence will not be described in detail.
- Assignment is `var <- expression` but `=` also works).
  - You can also use `expression -> variable`.
  - Assignment of a structure makes a copy of the structure, not just a reference to it.

```
if (condition) {
    statements
}
```

```
if (condition) {
    statements
} else {
    statements
}
```

```
for (variable in sequence) {
    statements
}
```

Statement types

- while (*condition*) {
  statements
  }
- repeat {
  statements # infinite loop, so should contain if...break
  }
- break
- next
- return # used only in functions; does not return a meaningful value
- return(*value*) # return is a function, hence parentheses are required

Statement types

- Statements are terminated by the end of the line.
- Multiple statements may be put on a single line, if separated by semicolons.
- The # character starts a comment, which extends to the end of the line.
- Assignment, `if/else`, `for`, `while`, and `repeat` are expressions, whose value is the last expression evaluated within them.

Functions

- Functions in R are first class objects: They may be stored in variables, passed as arguments to other functions, returned as the value of the a function, and defined withing other functions.
- The value of a function is the last expression evaluated with the function, although the `return(value)` function may also be used.
- The syntax of a function literal is

```
function(formal_arguments) {
    statements
}
```

- Although to be useful, a function literal should typically be assigned to be variable or used as an argument to another function.

Functions

- The formal arguments may be
  - Simple names
  - Names with default values: `name = default_value`
  - `...` to indicate arguments not used by this function, but may be passed on to other functions

    ```
    addup <- function() { sum() }
    ```
  - Objects passed as arguments to a function are copied, not passed by reference. Hence,

```
> a <- matrix(1:6, c(2, 3))
> f <- function(x) { x[1,1] <- 99; x[1,]}
> f(a)
[1] 99  3  5
> a
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

Scope rules

- When looking up the value of a variable, R first looks in the "global environment" (the user's workspace). If not found, it looks through the packages in the search list, and uses the first variable of that name that it finds.
- The function `search()` will return this search list.
- The `library(package)` will add the package to the search list, just below the global environment.
- Functions may use "free variables," that is, variables that are not defined within the function, but in the environment in which the function is defined. (An "environment" is a collection of name/value pairs.)

Scope rules

- R uses "lexical scoping," which means that when the function is called, the values of any free variables are taken, not from the calling environment, but from the original defining environment.
- A "closure," or "function closure," is a function plus the environment in which it was defined. In the absence of any free variables, a function is not normally referred to as a closure.

Calling functions

- When calling a function, the arguments may be given in the same order as in the function definition, or they may be given by name with the syntax name = value.
- In the latter case, the name may be abbreviated to the first letter or letters, if that uniquely determines a formal argument.
- Positional and named arguments may be mixed; the position of the corresponding formal argument is determined after the supplied named arguments have been removed.
- Any arguments appearing after ... in the function definition must be called by name, and the name may not be abbreviated.

Calling functions

- Arguments to a function are evaluated lazily, that is, not until they are needed.
- Functions may be called with more or fewer arguments than given in the function definition. Missing arguments, if needed, are given their default value; it is an error if no default value has been supplied. Extra arguments are simply ignored.
- R makes extensive use of long lists of named formal arguments. Functions described in this document mention only the most common arguments, using this font for code that should be entered as shown and this font to indicate that appropriate values should be inserted.

"Apply functions

- An "apply" function applies another function to every element of a list.
- In these functions, the first argument is the list, the second argument is the function to be applied to each element of the list, and any additional arguments are passed into the function given as the second argument.
- For example, if `f` is a function, then `sapply(1:3, FUN)` returns the vector `c(FUN(1),FUN(2),FUN(3))`. If `FUN` takes more than one argument, those may be given as additional arguments to the apply function:
  `sapply(1:3,FUN, 6)` returns the vector
  `c(FUN(1, 6),FUN(2, 6),FUN(3, 6))`.
- By providing additional named arguments, the values from the list may be given to a formal argument other than the first, for example,

"Apply" functions

```
> f = function(n, del) { 100*n + del }
> sapply(1:3, f, 5)
[1] 105 205 305
> sapply(1:3, f, n=5)
[1] 501 502 503
```

- `lapply(X, FUN, additional arguments)` – Returns a list of results of applying function FUN to each element of `X`.
- `sapply(X, FUN, additional arguments)` – Attempts to simplify the result.
  - If the result is a list of elements of length 1, a vector is returned.
  - If the result is a list of vectors, all the same length, the vectors are used as the columns of a matrix.

"Apply" functions

- `apply(X, 1, FUN)` – Applies the FUNction to each row of the matrix `X`, returning a vector of results.
- `apply(X, 2, FUN)` – Applies the FUNction to each column of the matrix `X`, returning a vector of results.
- `split(x, f)` – Puts each element of vector `x` into the group specified by the corresponding element of factor `f`, and returns a list of the groups tagged by the element of `f`. If `length(x)` is a multiple of `length(f)`, the elements of `f` are recycled.
  - Example: `split(11:15, c("F", "F", "M", "M", "F"))` results in `list(F=c(11, 12, 15), M=c(13, 14))`.

"Apply" functions

- `split(x, f)` – Returns a list whose elements are "sub-dataframes" of the given dataframe `x`, with each value of the factor or factors f is in a sub-dataframe by itself.
- If the additional argument `drop=TRUE` is provided, levels in factor that do not occur in the vector are omitted.
- `tapply(X, INDEX, FUN)` – Groups the elements of vector X according to the tags in the factors INDEX and applies the FUNction to each group, returning a (tagged) list of the results.
- `mapply(FUN, vectors)` – Calls the FUNction with multiple arguments (the first argument from the first vector, the second argument from the second vector, etc.), returning a vector of results. The function must have at least as many arguments as there are vectors, and the vectors must be of equal lengths (or able to be recycled to be equal lengths).

Assorted functions

- `as.numeric(x)`, `as.logical(x)`, `as.character(x)`, `as.logical(x)`, `as.complex(x)` returns `x` converted to the named type.
- `class(x)` returns the class (type) of `x`. If `x` is a vector, the type of its elements is returned.
- `colMeans(x, na.rm=FALSE)` returns a vector of column means; default is not to remove `NA` values.
- `colSums(x, na.rm=FALSE)` returns a vector of column sums; default is not to remove `NA` values.
- `dim(x)` returns `c(number of rows in matrix x, number of columns in matrix x)`.
- `head(x, n=FALSE)` returns the first `n` parts (default 6) of `x`, where `x` is a vector, matrix, table, data frame or function.
- `interaction(factors)` returns a factor which represents all combinations of the given factors.

Assorted functions

- `invisible(x)`, used as the return value of a function, returns `x` but causes the REPL not to print `x`.
- `is.na(x)` tests if `x` is the value `NA` ("Not Available").
- `is.nan(x)` tests if `x` is the value `NaN` ("Not a Number," for example, the result of 0/0).
- `names(x)` returns the names in `x`, where `x` is a list, matrix, or data frame.
- `rowMeans(x, na.rm=FALSE)` returns a vector of row means; default is not to remove `NA` values.
- `rowSums(x, na.rm=FALSE)` returns a vector of row sums; default is not to remove `NA` values.
- `tail(x, n=6L)` returns the last `n` parts (default 6) of `x`, where `x` is a vector, matrix, table, data frame or function.
- `vector(mode=s, length=n)` creates a vector of class `s` (where `s` is a string such as "numeric") containing `n` default values.

Access to files

- `dir(path)` returns a list of the files in the given directory (default is "."), the current working directory).

```
read.table(file, header=FALSE, sep="",colClasses=NA,
           nrows=-1, skip=0,comment.char="#",
           stringsAsFactors=default.stringsAsFactors())
```

  - `file` is the name of a file, or a connection and `header` is a logical indicating if the file has a header line.
  - `sep` is a string indicating how the columns are separated.
  - `colClasses` is a character vector indicating the class of each column in the dataset.
  - `nrows` is the number of rows in the dataset.
  - `comment.char` is a character string indicating the comment character.
  - `skip` is the number of lines to skip from the beginning.
  - `stringsAsFactors` tells if character variables should be coded as factors.

Access to files

- `getwd()` returns the current working directory.
- `setwd(dir)` sets the current working directory.
- `write.table(x, file="")` writes object `x` to the file ("" indicates the console).
- `read.csv(file, header=TRUE)` reads a comma-separated file.
- `source(file)` reads R code in from the named file (or from a URL or connection).
- `dump(list of names, file)` writes the named objects to the file, which may later be read in by source.
- `dput(x, file)` writes the object `x` to the file.
- `dget(file)` reads an object in from the file.

Access to files

- `file(description, open)` returns a connection to a file.
  - description is a path to a file, a `URL`, or `"clipboard"`.
  - open is one of `r, w, a, rb, wb, ab,` denoting `read`, `write`, or `append`, with `b` for binary.
- `gzfile(description, open)` returns a connection to a file compressed with `gzip`.
- `bzfile(description, open)` returns a connection to a file compressed with `bzip2`.
- `url(description, open)` returns a connection to a webpage.
- `readLines(con, n)` (not `read.lines`) reads in up to `n` lines from a connection, or all of them if `n` is `-1L`.
- `writeLines(text, con=stdout(), sep="\n ")` (not `write.lines`) writes the character vector text to the connection.

References

- Introduction to RStudio, https://dss.princeton.edu/training/RStudio101.pdf
- R Tutorial, https://data-flair.training/blogs/r-tutorial/
- A Concise Guide to R, https://www.cis.upenn.edu/~matuszek/Concise%20Guides/Concise%20R.html
- R-Overview, https://www.tutorialspoint.com/r/r_overview.htm

Thank you!