STATISTICS 4CI3/6CI3

Computational Methods for Inference

Angelo J. Canty

Course Information

Lectures : Monday, Wednesday and Thursday at 1:30–2:20 in HH-104.

Office : Hamilton Hall 209

Office Hours : Monday, Wednesday and Thursday at 2:30-3:30. Appointments for other times can be made by e-mail.

Phone : (905) 525-9140 extn 27079

E-mail : cantya@mcmaster.ca

Website : All lecture notes and announcements about the course will be made on the website

http://www.math.mcmaster.ca/canty/teaching/4ci3

Marking Scheme

Assignments 4 assignments will be worth 40% of your final grade (30% for S6CI3).

Midterm Test An in-class midterm test (tentatively scheduled for March 7) will be worth 30% of your final grade (25% for S6CI3).

Final Exam A 2.5 hour final exam will be scheduled during the April exam period and will be worth 30% of your final grade (25% for S6CI3).

Final Project (S6CI3 only) You will be required to complete a simulation study in R and write a 10 page report in LaTeX to be submitted by April 19. A short proposal will be required before March 21, 2019.

Rough Timetable

Weeks 1–3 Fundamentals of R programming

Week 4 Generating Random Variates

Week 5 The Monte Carlo Method

Week 6 Simulation Studies

Weeks 7–8 Markov Chain Monte Carlo

Weeks 9–10 The Jackknife and Bootstrap

Weeks 11–12 Bootstrap Confidence Intervals

Week 13 Bootstrap and Permutation Tests

Fundamentals of Programming in R

* R is a free open-source computing environment for statistics.

* R has many in-built functions for basic (and some advanced) statistical techniques.

* A major advantage of R is that it is a programming language rather than a statistical analysis package.

* This means that users can write their own code to implement any methods not available in R.

* Many authors have used this to write code for new or more advanced statistical methods and made them available to the R community as packages.

The R Language

* R is an interpreted language which means that once a complete command has been submitted it is executed immediately.

* R works through functions; every command takes input arguments and returns a result.

* Many functions also have side effects (such as creating a plot) but they still return a value which can be accessed.

* Many functions are predefined in R when you download it, many more are available in additional user-written packages.

* We can also write our own functions as we shall see later.

The R Workspace

* When you start R, it will open a Workspace.
* This is a file usually called `.RData`.
* The workspace contains all of the user defined variables and functions that have been saved to the hard disk.
* While R is active, it works on a copy of the workspace in the computer's RAM.
* To save any changes to the permanent workspace you must save the temporary workspace.

Saving the Workspace

* `save.image()` will save the entire workspace. By default it saves in a file called `.RData` but this can be changed using `save.image( file="<filename>.RData")`.

* You are also asked if you want to save the workspace when you quit R (using `q()`). This always saves to `.RData`.

* When you save a workspace the old `.RData` file is overwritten by the new one.

* You can have different `.RData` files in different folders to keep projects separate.

* Usually double clicking on an R workspace will open an R session with that workspace as the initial temporary workspace.

Data Types

Numeric Any real numbers, `Inf` or -Inf.

Logical `TRUE` (`T`) or `FALSE` (`F`)

Character A string of alpha-numeric characters enclosed by double quotes.

Factor Level of a categorical variable.

The special value `NA` denotes missing values of any data type.

Factors

* A vector of class `factor` is treated specially by many functions.

* A factor is designed to hold a categorical variable.

* A factor is usually constructed by first entering a numeric or character vector and then declaring it to be a factor.

* It is impossible to do arithmetic (or character string manipulation) on a factor even if the levels are numbers (character strings).

Data Structures

Vectors One-dimensional collections of data all of the same type.

Matrices Two-dimensional collections of data all of the same type.

Dataframes Two-dimensional matrices for which different columns of data can be of different types. Note that character vectors are coerced to factors when they become a column of a dataframe.

Arrays Multi-dimensional arrays of data of the same type.

Lists An collection of an arbitrary number of elements each of which is one of the data storage types listed here (including lists).

Variables and Assignment

* Data structures will generally be stored as a variable.
* Variables need to be given a name.
* To assign a data structure to a variable we use the assignment operator <-
* Data creation functions include `c`, `matrix`, `data.frame`, `list`.
* Each element of a list can be given a name.

Data Frames

* One of the most useful data structures is the `data.frame`.
* This is a rectangular data structure similar to a matrix.
* Unlike a matrix, however, the entries in a data frame need not all be of the same type.
* A column can only contain one type of data (numeric, factor, character, ...) but different columns can have different data types.
* A data frame is internally implemented a list of columns with the constraint that all columns must be of the same length.
* Columns can be given names through which they can be accessed.

Importing Data Frames

* Dataframes are used to hold datasets with each column corresponding to a variable and each row to an observation.

* We often receive datasets as external files and need to import them to R in order to analyze them.

* Often data comes as a plain text file with each observation on a single line and each column separated by white space.

* The function `read.table` will read such a file and construct a dataframe from it.

* Alternatively you can read in a comma-separated (`.csv`) file using the function `read.csv`.

* Most spreadsheet packages allow you to export a spreadsheet as a comma-separated file.

Indexing

* A variable name refers to the entire object.

* We can refer to one or more parts of an object using indexing.

* An index is specified directly after the variable name in `[i]`.

* For vectors there are three types of indices

  Positive Integers Select the entry (entries) specified by the index.

  Negative Integers Select the entire vector except those specified in the index.

  Logical Vector Select those elements for which the index evaluates to `TRUE`.

Indexing

* Matrices are indexed with two indices in the `[i,j]` separated by a comma. The first refers to the rows and the second to the columns. The comma **must** be included but if one of the indices is not given it defaults to the entire range of that index.

* An element of a list can be extracted using a single positive integer in `[[i]]`.

* If the elements of the list have names then they can be accessed using `$` followed by the name of the element required.

* Dataframes can be indexed like matrices or columns can be accessed like the elements of a list.

* Indexed variables can be used on the left of the assignment operator to assign values to the indexed parts of the variable.

Getting Help

* R has an extensive online help facility.

* You can get help on any function using `help(`*function*`)` or `?`*function*.

* Help files give details of arguments and the returned value as well as examples of use.

* The function `help.search` takes a keyword as a string and tries to find functions related to that keyword.

Operators

* Mathematical operators work on numeric data and return numeric results.

`^`	exponentiation
`-`	negation (unary minus)
`*, /`	multiplication, division
`+, -`	addition, subtraction
`%/%, %%`	Integer division, modulo (remainder)

* Comparison operators return logical results and have lower precedence to mathematical operators.

`==` `<` `>` `>=` `<=` `!=`

* The logical operators `&` (*AND*) and `|` (*OR*) work on logical data.

* All operators will work on single data points or element-wise on vectors.

Mathematical and Statistical Functions

* Most standard mathematical functions are available in R

  `sqrt`, `log`, `log10`, `exp`, `abs`, `sin`, `cos`, `tan`, ...

* All such functions will work element-wise on a vector argument.

* Functions such as `max`, `min`, `sum`, `prod`, `length` take a vector and return a single numeric response.

* Many functions also exist for summary statistics

  `mean`, `var`, `sd`, `median`, `iqr`, `summary`, ...

Statistical Modelling

* R has many methods to fit statistical models.

* The function `lm` fits standard linear models using least squares estimation.

* The main argument to `lm` (and many other modelling functions) is a formula.

* A formula is of the form `y~x1+x2+x3` where `y` is the name of the response variable and `x1`, `x2` and `x3` are the covariates.

* Interactions can be included using `*` or `:`. Note that `x1*x2` is the equivalent of `x1+x2+x1:x2`.

* If one or more covariates is a factor then R produces appropriate dummy variables with the first level being the reference level by default.

Statistical Modelling

* The returned value from a call to `lm` is an object with class `lm`. This is like a list and contains a number of components.

* What is printed on the screen is a short description of the fit.

* Functions like `coefficients`, `fitted` and `residuals` can be used to access parts of the object.

* The object can also be indexed similar to a named list.

* The function `summary` is particularly useful. It prints out a more detailed summary of the fit.

* `summary` also produces another object with more information than in the original model object.

Statistical Modelling

* Generalized linear models can be fitted using `glm`.
* This also takes a formula but you should also specify a distribution and link function.
* `loess` is a scatter-plot smoother using local regression.
* Other models such as mixed models, generalized additive models, classification and regression trees etc. are available in add-on packages.

R Functions

* All computations in R are carried out by functions.
* As we have seen many of these are available in R by default.
* Typing a function name will cause the definition of that function to be displayed.
* A function definition is just a list of R commands that are carried out when the function is executed with a set of arguments.
* One of the best features of R is the ability to create our own functions with their own definitions.
* This allows us to either change how an in-built function works (generally not advised) or create new functions to do things the in-built functions do not do.

R Functions

* A function must have a name like a variable and is stored in the workspace.

* A function takes a number of arguments. In the definition we need to give the arguments names by which they will be referred to in the function.

* The body of the function is simply a list of R commands enclosed in braces.

* The returned value is the value from the final line of the function.

Syntax of Function Definition

```
function.name <- function(argument1, argument2, ...) {
    R command 1
    R command 2
          .
          .
          .
    Return Value
}
```

Writing R Functions

* The commands in the function body can be any valid R commands (including other function definitions).

* Variables (or functions) created within a function are only visible within that function and are not part of the global R workspace.

* Once the function execution is finished these values no longer exist.

* In general, you should not use variables from the global R workspace within a function since that can result in different values being returned for the same input arguments.

* Instead pass the values of the variables you wish to access through the arguments.

Function Arguments

* The function argument list gives the names of variables that will be used within the function.

* When the function is called, the user specifies values for these arguments.

* Default values for function arguments can be specified in the function definition using `argument.name=`<*default value*>.

* The arguments in a function call are mapped to the arguments in the definition. Named input arguments are mapped first and then any unnamed arguments are mapped to the other arguments in the order they appear.

* Note that only the values are assigned to the function arguments so changes to the arguments within a function do not affect the variables in the global workspace.

Conditional Evaluations

* The simplest way is using
  `if (`<*condition*>`)` <*statement1*> `else` <*statement2*>

* *condition* should return a single logical value.

* The `else` clause is optional.

* Multiple statements can be evaluated for either (or both) parts by enclosing them in braces.

* `if ... else` statements can be nested when there are multiple conditions.

* Note that only one of the two statements (code chunks) is ever evaluated in this construct.

Conditional Evaluations

* There is also a function `ifelse`
  `ifelse(`<*condition*>`,` <*true*>`,` <*false*>`)`

* In this case <*condition*> should be a logical vector.

* <*true*> and <*false*> should be vectors of the same length as <*condition*>.

* All three vectors are fully evaluated.

* For every `TRUE` in <*condition*>, the corresponding element of <*true*> is returned otherwise the corresponding element of <*false*> is returned.

Loops

* A fundamental concept in computer programming is the loop which allows a chunk of code to be run multiple times.

* R provides a number of types of loops.

* It is important to note, however, that very often in R a loop is not required since most functions and operators are designed to operate element-wise on vectors.

* Nevertheless there are some cases in which looping is required.

* One such case is for iterative algorithms which are common in advanced statistical and mathematical applications.

The `for` Loop

`for` (<*index*> `in` <*index vector*>) {<*Loop Body*>}

* In a `for` loop the body of the loop is run as many times as there are elements in the index vector.

* The first time the loop body is evaluated the index variable is set to the first element in the index vector.

* For the second iteration the index variable is set to the second element of the index vector and so on.

* The index vector is evaluated once before the first iteration and cannot be changed in the loop body.

* Most commonly the index vector is something like `1:n` which evaluates as the vector $c(1, 2, ..., n)$.

The `while` Loop

`while` (<*condition*>) {<*Loop Body*>}

* The condition should evaluate to a logical (`TRUE` or `FALSE`).
* If the condition is `TRUE` then the loop body is evaluated.
* Before each iteration condition is evaluated again and the body executed as long as the condition stays `TRUE`.
* Once the condition is `FALSE` the loop is terminated and the next command after the loop is executed.
* Generally the condition should be `TRUE` initially and get updated in the loop body each time so that it eventually becomes `FALSE`.

`apply` and related functions

* Loops are often used to do the same thing to every column (or row) of a matrix.

* R provides a function `apply` to do this without an explicit loop.

* `apply` takes a matrix, a dimension (1=rows, 2=columns) to loop over and a function to be applied.

* `apply` returns a vector or matrix of the results from applying the function provided they are all the same length.

* `lapply` and `sapply` evaluate a function on each element of a list.

* `lapply` always returns a list, `sapply` tries to simplify the returned value into a vector or matrix if possible.

Ragged arrays and `tapply`

* A ragged array consists of a vector along with a number of factor indices.

* The combination of the indices partitions the vector into a number of exhaustive and non-overlapping groups.

* The function `tapply` can be used to evaluate the same function on each of these groups.

* By default the method will attempt to simplify the results into a vector or matrix.

Debugging

* You will make mistakes in your code!!!!!
* It is therefore important to check your code and output.
* Sometimes you get odd results and don't understand why. Trying to figure out and fix this is a very important part of programming.
* The `browser()` command is very useful to see what is going on within a loop or function.
* This returns command to the user but at the point at which the browser command was found and so we can look at values of local variables within a function.
* This can often help to track down where there is a problem and how to fix it.

Writing Programs

* Although you can simply run commands directly in the R console this is not generally advised.

* Instead you should write your command in a plain text file. R has an inbuilt editor allowing you to do this easily.

* The free RStudio software may make this easier for some users but is not required.

* The # symbol is used to put comments into the code. Anything to the right of this symbol is ignored by R.

* This method of interacting with R allows you to keep track of your work, correct mistakes and print out your code. All of which will be required for this course.

Histograms

* `hist` takes a quantitative argument and constructs a histogram.

* The `breaks` argument allows us to change the number or location of the default bins for the histogram.

* By default, for equal length bins, the height is proportional to the count in a bin.

* Specifying `prob=TRUE` rescales the histogram so that the entire area is one. This is also the default if the bins are not all equal.

* The returned value has the information used to construct the plot such as the bin boundaries and midpoints and the frequencies and estimated density values for each bin.

Scatterplots

* The basic command in R is `plot` which produces scatterplots.
* In its basic form it takes a vector of x coordinates and a vector of y coordinates.
* Special plots are produced for many R objects including the objects returned from `lm`.
* The plotting symbol can be changed with the `pch=` argument.
* `points` and `lines` can be used to add to a plot.
* `abline` adds a line with given intercept and slope.
* R can be told to directly send a plot to a file. This file can be one of many possible graphic types (pdf, postscript, jpeg, bmp, png, ...)