### Исследование методов прогнозирования с помощью среды R. Часть 2. Основы языка R.

##### Факторные данные

Факторы представляют собой структуру данных для хранения векторов категориальных данных (классов), т.е. величин, которые могут принимать значения из конечного и неупорядоченного множества.

Факторы создаются с помощью функции *factor(x = character(), levels, labels= levels).*

Предположим, что у нас имеется 3 класса *Yes, No, Perhaps*. И некоторая выборка из 6 объектов, каждый из которых принадлежит одному из этих классов.

|  |
| --- |
| > v = c("Yes", "No", "Yes", "Perhaps", "No", "Perhaps")  > f = factor(v)  > f  [1] Yes No Yes Perhaps No Perhaps  Levels : No Perhaps Yes |

Для получения и изменения текстового вектора, содержащего имена уровней фактора, служит функция *levels(x).*

##### Структуры данных

Структуры данных (data frames) один из самых важных типов данных в R, позволяющий объединять данные разных типов вместе. Структура является специальной версией списка, где все элементы имеют одинаковую длину. Можно считать, что структура данных это двумерная таблица, в которой (в отличие от числовых матриц), разные столбцы могут содержать данные разных типов (но все данные в одном столбце имеют один тип).

Создать структуру данных можно с помощью функции *data.frame(...),* аргументами которой являются произвольное количество элементов (столбцов) фрейма. В качестве элементов структуры данных могут выступать векторы, факторы, матрицы, списки или другие структуры. При этом все векторы должны иметь одинаковую длину, а матрицы и фреймы одинаковое число строк. Функция *data.frame(...)* просто собирает все данные вместе. Символьные векторы конвертируются в факторы. Остальные данные собираются во фрейм такими, какие они есть, поэтому для правильной работы некоторых алгоритмов приходится конвертировать их самостоятельно.

|  |
| --- |
| > a = matrix(1:8 , nrow = 4, ncol = 2)  > b = c("a", "b", "c", "a")  > d = (1:4 %% 2 == 0)  > e = factor(c("soft", "hard", "soft", "medium"))  > f = data.frame(a , b , d , e ) |

С помощью функции *colnames(x)* можно изменить имена столбцов фрейма, с помощью *rownames(x)* – имена строк. Для загрузки наборов данных из файла в структуру или таблицу может быть использована функция *read.table(file, header = FALSE, sep = "", ...),* в которой на вход необходимо задать путь к текстовому файлу с данными. Параметр *header* позволяет указать, следует ли интерпретировать первую строку файла, как имена столбцов таблицы.

##### Импортирование данных в R

Следует отметить, что возможности системы R по вводу и редактированию данных умышленно ограничены ее создателями, которые предполагали, что для этого будут использоваться другие средства (например, программа Microsoft Excel или базы данных). Поэтому подлежащие анализу объемные таблицы данных обычно подготавливаются при помощи сторонних приложений, и только потом загружаются в рабочую среду R из внешних файлов.

Правила подготовки загружаемых файлов:

В импортируемой таблице с данными не должно быть пустых ячеек. Если некоторые значения по тем или иным причинам отсутствуют, вместо них следует ввести NA.

Импортируемую таблицу с данными рекомендуется преобразовать в простой текстовый файл с одним из допустимых расширений. На практике обычно используются файлы с расширением .txt, в которых значения переменных разделены знаками табуляции (tab-delimited files), а также файлы с расширением .csv (comma separated values), в которых значения переменных разделены запятыми или другим разделяющим символом.

В качестве первой строки в импортируемой таблице рекомендуется ввести заголовки столбцов-переменных. Такая строка – удобный, но не обязательный элемент загружаемого файла. Если она отсутствует, то об этом необходимо сообщить в описании команды, которая будет управлять загрузкой файла (например, read.table() – см. ниже). Все последующие строки файла в качестве первого элемента могут содержать заголовки строк (если таковые предусмотрены), после которых следуют значения каждой из имеющихся в таблице переменных.

Подлежащий импортированию файл рекомендуется поместить в рабочую папку программы, т.е. папку, в которой R по умолчанию будет "пытаться найти" этот файл.

Основной функцией для импортирования данных в рабочую среду R является read.table().

|  |  |
| --- | --- |
| Аргумент | Назначение |
| file | Служит для указания пути к импортируемому файлу. Путь приводят либо в абсолютном виде (например, file = "C:/Temp/MyData.dat"), либо указывают только имя импортируемого файла (например, file = "MyData.txt"), но при условии, что последний хранится в рабочей папке программы В качестве имени можно также указывать полную URL-ссылку на файл, который предполагается загрузить из Сети (например: file = "http://somesite.net/YourData.csv"). Начиная с версии R 2.10, появилась возможность импортировать архивированные файлы в zip-формате. |
| header | Служит для сообщения программе о наличии в загружаемом файле строки с заголовками столбцов. По умолчанию принимает значение FALSE. Если строка с заголовками столбцов имеется, этому аргументу следует присвоить значение TRUE. |
| row.names | Служит для указания номера столбца, в котором содержатся имена строк (например, в рассмотренном выше примере это был первый столбец, поэтому row.names = 1). Важно помнить, что все имена строк должны быть уникальными, т.е. одинаковые имена для двух или более строк не допускаются. |
| sep | Служит для указания разделителя значений переменных, используемого в файле (separator – разделитель). По умолчанию предполагается, что значения переменных разделены "пустым пространством", например, в виде пробела или знака табуляции (sep = ""). В файлах формата csv значения переменных разделены запятыми, и поэтому для них sep = ",". |
| dec | Служит для указания знака, используемого в файле для отделения целой части числа от дроби. По умолчанию dec = "." |
| nrows | Выражается целым числом, указывающим количество строк, которое должно быть считано из загружаемой таблицы. Отрицательные и иные значения игнорируются. Пример: nrows = 100. |
| skip | Выражается целым числом, указывающим количество строк в файле, которое должно быть пропущено перед началом импортирования. Пример: skip = 5 |

В качестве примера предположим, что нам необходимо загрузить файл Dreissena.txt, который хранится в рабочей папке R.

Функции read.table() в этом случае может быть применена следующим образом:

Моллюски <- read.table("Dreissena.txt", header=TRUE, sep="\t", na.strings="NA", strip.white=TRUE)

##### Организация вычислений: функции, ветвления, циклы

Абсолютное большинство процедур обработки данных в R реализуется с помощью функций. Функции представляют собой поименованный программный код, состоящий из некоторого набора переменных, констант, операторов и других функций, и предназначенный для выполнения конкретных операций и задач. Как правило (но не всегда), функции возвращают результат своего выполнения в виде объекта языка R – переменной определенного класса: вектора, списка, таблицы и т.д.

По своему назначению функции можно разделить на характерные группы: арифметические, символьные, статистические и прочие. Функции могут быть встроенными (т.е. представленными в базовых или подгружаемых пакетах) и собственными (т.е. написанными непосредственно самими пользователями).

Некоторые наиболее употребительные встроенные функции представлены ниже:

|  |  |
| --- | --- |
| Вызов функции и описание | Пример и результат |
| Арифметические функции | |
| abs(х) – модуль величины х | abs(-1)  1 |
| ceiling(х) – округление до целого в большую сторону | ceiling(9.435)  10 |
| floor(х) – округление до целого в меньшую сторону | floor(2.975)  2 |
| round(x, digits=n) – округление до указанного числа digits знаков после десятичной точки | round(5.475, 2)  5.48 |
| signif(x, digits=n) – округление до указанного числа digits значащих цифр | signif(3.475, 2)  3.5 |
| trunc(x) – округление до целого числа | trunc(4.99)  4 |
| ехр(х) – ех | ехр(2.87)  17.637 |
| log(x) – логарифм натуральный х | log(3.12)  1.137 |
| log10(x) – логарифм десятичный х | log10(3.12)  0.494 |
| sqrt(x) – корень квадратный х | sqrt(2.12)  1.456 |
| cos(x) sin(x) tan(x) acos(x) cosh(x) acosh(x) – тригонометрические функции от х | cos(1.27\*pi)  -0.661 |
| Функции для работы с символьными типами данных | |
| grep(pattern,x,ignore.case=FALSE,fixed=FALSE) – возврат индекса первого найденного элемента pattern в х | grep("A",c("x","y","A", "z"),fixed=TRUE)  3 |
| substr(x, start=nl, stop=n2) – выбор или замена символов в строках символьного вектора х | substr("язык R", 2, 4)  "зык" |
| paste(..., sep="") – объединение символов или строк через значение разделителя sep | paste("x",l:3,sep="")  "xl" "x2" "хЗ" |
| strsplit(x, split) – разделяет элементы вектора по разделителям split | strsplit("абв","")  "а" "б" "в" |
| toupper(x) и tolower(x) – преобразуют буквы текстового вектора х в прописные и обратно | toupper("Мал")  "МАЛ" |

Написание собственных функций

Группа команд ниже выполняет расчет среднего и стандартного отклонения натурального ряда чисел от 1 до 10 и возвращает вектор из этих значений:

{aver <- mean(1:10); stdev <- sd(1:10); c(MEAN=aver, SD=stdev)}

MEAN SD

5.50000 3.02765

Однако если этот расчет необходимо выполнить неоднократно для различных наборов исходных данных, то его стоит оформить в виде функции. Общий синтаксис оформления собственной функции пользователя таков:

имя\_функции <- function(argl, arg2,...) {

группа\_выражений

return(object) }

где имя\_функции – имя создаваемой функции, argl, arg2,... – формальные аргументы функции. Оператор return() нужен в случаях, когда группа выражений не возвращает целевого результата.

Перед своим первым выполнением функция должна быть определена в текущем скрипте, либо загружена с помощью команды source() из скриптового файла, где она была предварительно подготовлена. Тогда вызов функции может быть осуществлен как

имя\_функции (argl, arg2,...)

где argl, arg2,... – фактические аргументы, связанные с формальными параметрами функции по порядку их следования, либо по наименованиям.

Для представленного выше примера можно оформить функцию:

stat\_param <- function(x){

aver <- mean(x); stdev <- sd(x); c(MEAN=aver, SD=stdev)}

и включить ее в коллекцию собственных функций, расположенных в файле my\_func.R. Тогда необходимый нам результат, приведенный выше, можно получить, выполнив

source("my\_func.R")

stat\_param (1:10)

Компоненты списка аргументов в заголовке функций могут быть обязательными или принимать опциональные значения. Например, следующая функция возводит числовой объект x в степень n, но если степень не указана, то автоматически происходит возведение в куб:

power <- function(x,n = 3){ x^n }

Аргументами функций могут быть объекты самого разного типа, например, названия других функций. Так, следующая функция выполняет произвольные преобразования случайных равномерно распределенных величин:

my\_exampl <- function(n, func\_trans)

{ x <- runif(n) ; abs(func\_trans(x)) }

Тогда сгенерировать 5 прологарифмированных значений можно, если записать:

my\_exampl(5, log)

[1] 0.6345459 0.6522557 2.4180118 0.1007311 1.6983938

Условия и циклы

Условный оператор имеет следующую структуру:

if( логическое\_выражение )

{ группа\_выражений\_1 если логическое\_выражение равно TRUE }

else { группа\_выражений\_2 в противном случае }

Например, следующая функция сравнивает размеры двух векторов:

compare <- function(x, y){ nl <- length(x) ; n2 <- length(у)

if(nl!= n2){

if(nl > n2){ z=(nl - n2)

cat("Первый вектор имеет на ",z," элементов 6ольше \n") }

else{ z=(n2 - nl)

cat("Второй вектор имеет на ",z," элементов 6ольше \n")}}

else{cat("Количество элементов одинаково ",nl,"\n") }

}

х <- c(1:4)

у <- c(1:9)

compare(х, у)

Первый вектор имеет на 5 элементов 6ольше

Имеется также сокращенная форма реализации ветвлений:

ifelse(логическое\_выражение, группа\_выражений\_1, группа\_выражений\_2)

Повторение в цикле одних и тех же вычислительных операций осуществляется с использованием конструкций for(), while() или repeat(), которые имеют следующий синтаксис:

for (index in for\_object) { группа\_выражений }

while(логическое\_выражение) { группа\_выражений }

repeat { группа\_выражений ; break }

Здесь объект for\_object может быть вектором, массивом, таблицей, или списком, а группа\_выражений выполняется каждый раз для каждого элемента index этого объекта.

Рассмотрим в качестве примера функцию оценки доверительного интервала среднего значения для выборки размером n с использованием непараметрического бутстрепа. Будем генерировать из исходной выборки множество псевдовыборок того же размера, состоящих из случайных комбинаций исходного набора элементов. При этом используем алгоритм "случайного выбора с возвратом" (random sampling with replacement), т.е. извлеченный элемент возвращается в исходную совокупность и имеет шанс быть выбранным снова. В результате некоторые члены в каждой отдельной псевдовыборке могут повторяться два или более раз, тогда как другие – отсутствовать.

Этот алгоритм в R реализован в функции sample(data,replace=T). Для каждой псевдовыборки мы рассчитаем значение среднего, а в качестве границ 95%-ного доверительного интервала (bootstrap percentile interval) примем 2.5% и 97.5% квантили бутстреп-распределения:

boot\_np <- function(data , Nboot=5000) {

boots <- numeric(Nboot) # Пустой вектор для хранения результатов

for (i in 1:Nboot) { boots[i] <- mean(sample(data,replace=T)) }

CI <- quantile(boots, prob=c(0.025,0.975))

return (c(m = mean(data),CI))

}

x <-c(5, 5, 8, 10, 10, 10, 19, 20, 20, 20, 30, 40, 42, 50, 50)

boot\_np(x)

m 2.5% 97.5%

22.6 15.0 30.8

Здесь конструкция for() осуществляет формирование Nboot = 5000 значений средних для генерируемых псевдовыборок.

##### Векторизованные вычисления в R с использованием apply-функций

Использование оператора циклов for(), что характерно для языков низкого уровня типа С/С++, не считается хорошим стилем программирования на языке R, ориентированном на векторизацию вычислений. Вместо того, чтобы выполнять последовательно скалярные операции над каждым из элементов массива, гораздо эффективнее выполнять параллельные вычисления, при которых программа обрабатывает одновременно весь массив (вектор) целиком или по несколько элементов вектора в каждый момент времени. Очевидно, что такой подход потенциально может привести к значительному ускорению однотипных вычислений над большими массивами данных:

x <- 1:10

sqrt(x)

[1] 1.000 1.414 1.732 2.000 2.236 2.449 2.645 2.828 3.000 3.162

Принцип векторизованных вычислений применим не только к векторам как таковым, но и к более сложным объектам R – матрицам, спискам и таблицам данных (для R разницы между последними двумя типами объектов не существует: фактически таблица данных является списком из нескольких компонентов – векторов одинакового размера).

Функция apply() – используется в случаях, когда необходимо применить какую-либо функцию ко всем строкам или столбцам матрицы (или массивам большей размерности):

apply(x, MARGIN, FUN, ...)

где x – это преобразуемый объект, MARGIN – индекс, обозначающий направление процесса вычислений (по столбцам или строкам), FUN – применяемая для вычислений функция, ... – это любые другие параметры применяемой функции. Для матрицы или таблицы данных MARGIN = 1 обозначает строки, а MARGIN = 2 – столбцы.

Функция **lapply()** – используется в случаях, когда необходимо применить какую-либо функцию к каждому компоненту списка и получить результат также в виде списка (буква "l" в названии lapply() означает list – "список").

# Создадим список с тремя компонентами-векторами:

x <- list(a = 1, b = 1:3, c = 10:100)

# Выясним размер каждого компонента списка х

# (функция length()):

lapply(x, FUN = length)

$a

[1] 1

$b

[1] 3

$c

[1] 91

# Выполним суммирование элементов в каждом компоненте списка х:

lapply(x, FUN = sum)

$a

[1] 1

$b

[1] 6

$c

[1] 5005

Функция **sapply()** – используется в случаях, когда необходимо применить какую-либо функцию к каждому компоненту списка, но результат вывести в виде вектора (буква "s" в названии sapply() означает simplify – "упростить").

# Список из трех компонентов:

x <- list(a = 1, b = 1:3, c = 10:100)

# Выясним размер каждого компонента списка х:

sapply(x, FUN = length)

a b c

1 3 91 # результат возвращен в виде вектора

# Суммирование всех элементов в каждом компоненте списка х:

sapply(x, FUN = sum)

a b c

1 6 5005 # результат возвращен в виде вектора

Функция **replicate ()** является своего рода "оберткой" для функции sapply() и позволяет провести серию вычислений с целью генерации набора чисел по заданному алгоритму. Синтаксис функции имеет вид:

replicate(n, expr, simplify=TRUE)

где n – число повторов, expr – функция или группа выражений, которые надо повторить n раз , simplify = TRUE – необязательный параметр, который пробует упростить результат и представить его в виде вектора или матрицы значений.

Рассмотрим пример использования функции replicate() для проверки статистической гипотезы о равенстве медиан двух выборок бутстреп-методом. Созданная нами функция boot\_med() из каждой исходной выборки x и y извлекает по N псевдовыборок, используя алгоритм "случайного выбора с возвратом", и находит разность их медиан. Здесь функция sample.int() формирует целочисленные наборы случайных индексов indx и indy для каждого из сравниваемых векторов:

boot\_med <- function(x, y, N=100) {

replicate(N, {

indx <- sample.int(length(x), length(x), replace=T)

indy <- sample.int(length(y), length(y), replace=T)

median(x[indx]) - median(y[indy])

})

}

Для тестирования функции проверим однородность медиан для двух случайных выборок из нормального распределения со средним mean=10 и стандартным отклонением sd=4:

Y <- rnorm(100, sd=4, mean=10)

X <- rnorm(100, sd=4, mean=10)

quantile(boot\_med(Y,X,10000), probs = c(0.025, 0.975))

2.5% 97.5%

-0.3834803 2.5177006

Поскольку доверительный интервал разности медиан включает 0, то нулевую гипотезу отклонять не следует.

Функция **vapply()** – схожа с sapply(), но работает несколько быстрее за счет того, что пользователь однозначно указывает тип возвращаемых значений (буква "v" в названии vapply() означает velocity – "скорость").

При вызове vapply() пользователь должен привести пример ожидаемого типа возвращаемых значений. Для этого служит аргумент FUN.VALUE:

# Аргументу FUN.VALUE присвоено логическое значение FALSE.

# Этим задается тип возвращаемых функцией значений,

# который ожидает пользователь

a <- vapply(NULL, is.factor, FUN.VALUE = FALSE)

# Функция sapply() применена к тому же NULL-объекту:

b <- sapply(NULL, is.factor)

# Проверка типа переменных:

is.logical(a)

[1] TRUE

is.logical(b)

[1] FALSE

Функция **mapply()** – используется в случаях, когда необходимо поэлементно применить какую-либо функцию одновременно к нескольким объектам (например, получить сумму первых элементов векторов, затем сумму вторых элементов векторов, и т.д.). Результат возвращается в виде вектора или массива другой размерности. Буква "m" в названии mapply() означает multivariate – "многомерный" (имеется в виду одновременное выполнение вычислений над элементами нескольких объектов).

mapply(sum, 1:5, 1:5, 1:5)

[1] 3 6 9 12 15

Функция **rapply()** – используется в случаях, когда необходимо применить какую-либо функцию к компонентам вложенного списка (буква "r" в названии rapply() означает recursively – "рекурсивно").

# Пользовательская функция, которая добавляет ! к элементу

# объекта x если этот элемент является текстовым выражением

# или добавляет 1 если этот элемент является числом:

myFun <- function(x){

if (is.character(x)){

return(paste(x,"!",sep=""))

}

else{

return(x + 1)

}

}

# Пример вложенного списка:

l <- list(a = list(a1 = "Boo", b1 = 2, c1 = "Eeek"),

b = 3, c = "Yikes",

d = list(a2 = 1, b2 = list(a3 = "Hey", b3 = 5)))

l

$a

$a$a1

[1] "Boo"

$a$b1

[1] 2

$a$c1\_\_

[1] "Eeek"

$b

[1] 3

$c

[1] "Yikes"

$d

$d$a2

[1] 1

$d$b2

$d$b2$a3

[1] "Hey"

$d$b2$b3

[1] 5

# Рекурсивное применение функции myFun к списку l:

rapply(l,myFun)

a.a1 a.b1 a.c1 b c d.a2 d.b2.a3 d.b2.b3

"Boo!" "3" "Eeek!" "4" "Yikes!" "2" "Hey!" "6"

# Если необходимо вернуть результат в виде вложенного

# списка, можно воспользоваться аргументом how = "replace".

# В этом случае исходные значения в списке l будут

# заменены на новые:

rapply(l, myFun, how = "replace")

$a

$a$a1

[1] "Boo!"

$a$b1

[1] 3

$a$c1

[1] "Eeek!"

$b

[1] 4

$c

[1] "Yikes!"

$d

$d$a2

[1] 2

$d$b2

$d$b2$a3

[1] "Hey!"

$d$b2$b3

[1] 6

Функция **tapply()** – используется в случаях, когда необходимо применить какую-либо функцию fun к отдельным группам элементов вектора x, заданным в соответствии с уровнями какого-либо фактора group:

tapply(x, group, fun, ...)

Например, в следующем фрагменте кода функция sample() используется для создания двух случайных выборок: из 50 значений целых чисел от 1 до 4 и связанных с ними меток четырех групп A-D. Функция tapply() подсчитывает суммы х для каждого из значений фактора:

х <- sample(1:4, size=50, replace=T)

gr <- as.factor(sample(c("A","B","C","D"),size=50, replace=T))

tapply(x, gr,sum)

А В С D

29 25 30 37

Функция by() является своего рода аналогом функции tapply(), с той разницей, что она применяется для таблиц. Таблица data разделяется в соответствии с заданным столбцом-фактором group на подмножество подтаблиц и для обработки каждой такой части определяется функция fun:

by(data, group, fun, ...)

Рассмотрим пример с таблицей Моллюски (Dreissena.txt) и рассчитаем средние значения длины раковины ZMlength и численности инфузорий CAnumber для каждого из трех обследованных озер Lake (Баторино, Мястро и Нарочь):

Моллюски <- read.table("Dreissena.txt", header=TRUE, sep="\t", na.strings="NA", strip.white=TRUE)

by(Моллюски[,5:6], Моллюски$Lake, colMeans)

Моллюски$Lake: Batorino

ZMlength CAnumber

18.61339 772.80315

-------------------------

Моллюски$Lake: Myastro

ZMlength CAnumber

16.36944 803.08889

-------------------------

Моллюски$Lake: Naroch

ZMlength CAnumber

14.4929 340.9941

Функция **outer ()** позволяет выполнить комбинаторную операцию fun над элементами двух массивов или векторов x и у, не прибегая к явному использованию "двойного" цикла:

outer(x, у, fun="\*", ...)

По умолчанию осуществляется операция попарного перемножения

х <- 1:5 ; у <- 1:5

outer(х,у)

[,1] [,2] [,3] [,4] [,5]

[1,] 1 2 3 4 5

[2,] 2 4 6 8 10

[3,] 3 6 9 12 15

[4,] 4 8 12 16 20

[5,] 5 10 15 20 25

Используя, например, функцию outer() вместе с функцией paste(), можно сгенерировать все возможные попарные комбинации "связок" элементов символьного и целочисленного векторов:

х <- с("А", "В", "С", "D")

у <- 1:10

outer(x, у, paste, sep="")

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

[1,] "А1" "А2" "A3" "А4" "А5" "А6" "А7" "А8" "А9" "А10"

[2,] "В1" "В2" "ВЗ" "В4" "В5" "В6" "В7" "В8" "В9" "В10"

[3,] "С1" "С2" "СЗ" "С4" "С5" "С6" "С7" "С8" "С9" "С10"

[4,] "Dl" "D2" "D3" "D4" "D5" "D6" "D7" "D8" "D9" "D10"

##### Задание

1. Выполнить приведенные выше примеры.

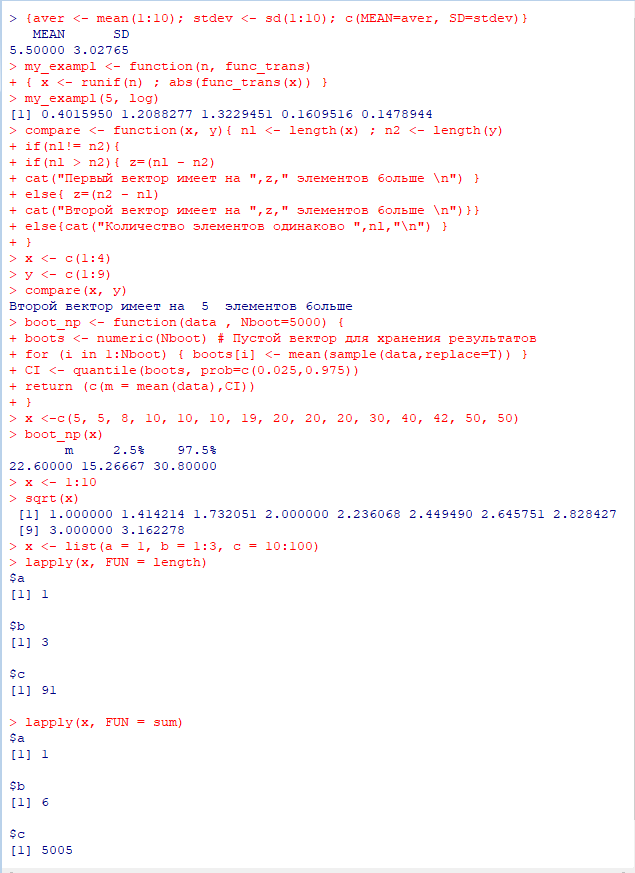


Рисунок 1 – Выполнение команд.

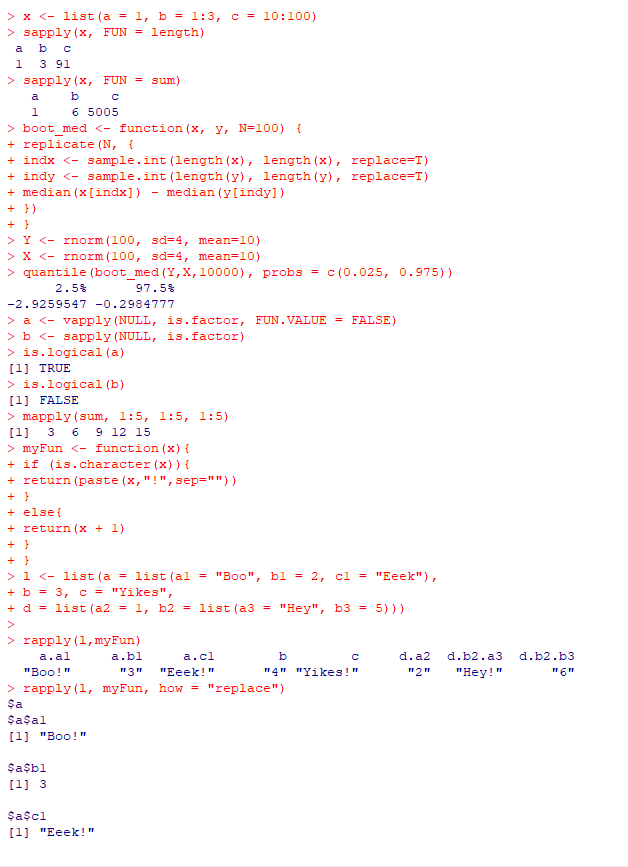


Рисунок 2 – Выполнение команд.

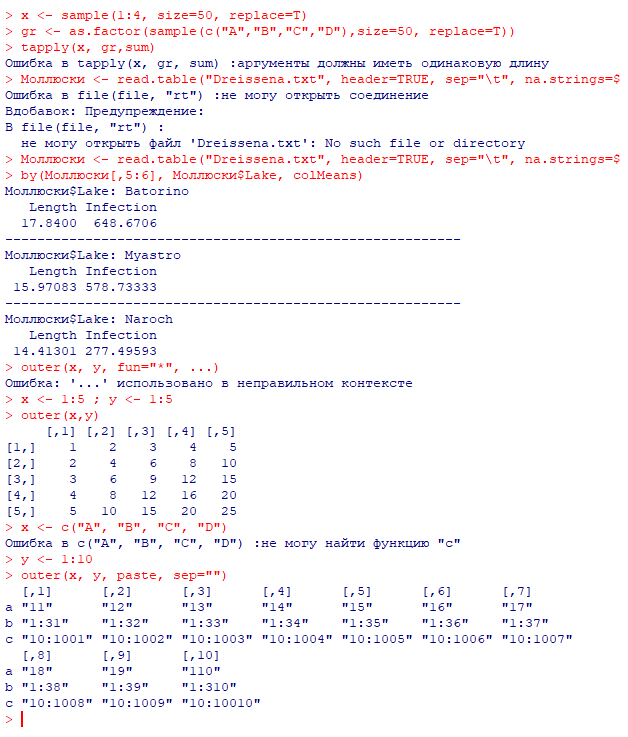


Рисунок 3 – Выполнение команд.

2) В качестве примера организации ветвлений, циклов и функций выполнить прилагаемые скрипты Example1.R – фрактальное дерево и Example2.R – фрактал Мандельброт. Меняя различные параметры, попытаться добиться изменения, например, вида фрактального дерева. Рассмотреть отдельные фрагменты фрактала Мандельброт.

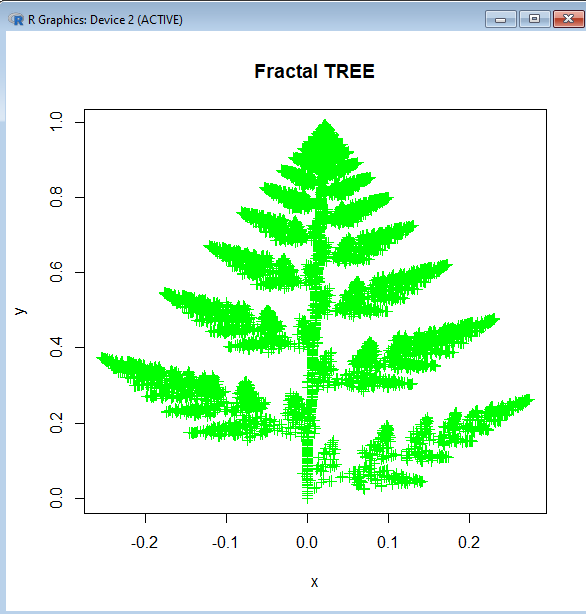


Рисунок 4 – Выполнение скрипта Example1.R.

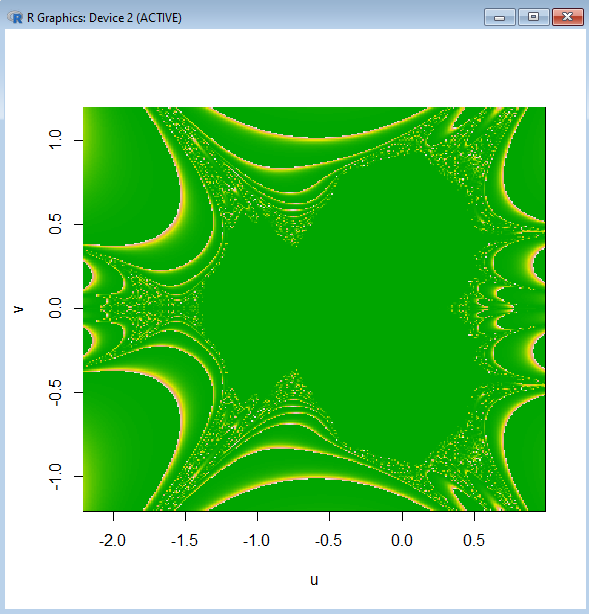


Рисунок 5 – Выполнение скрипта Example2.R.

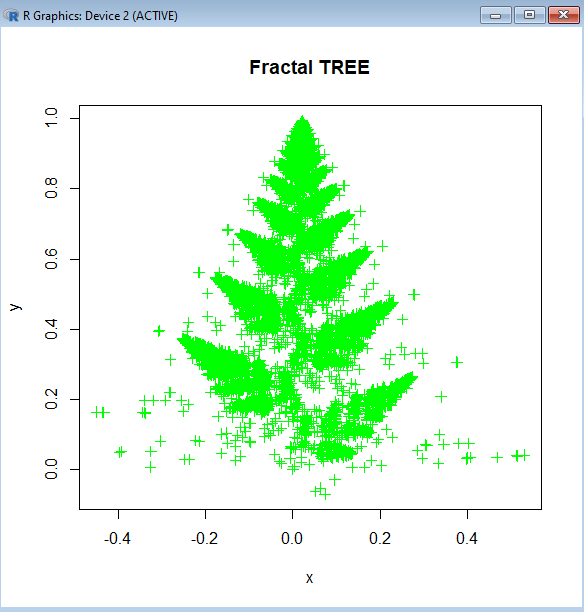


Рисунок 6 – Выполнение скрипта Example1.R с измененными данными.

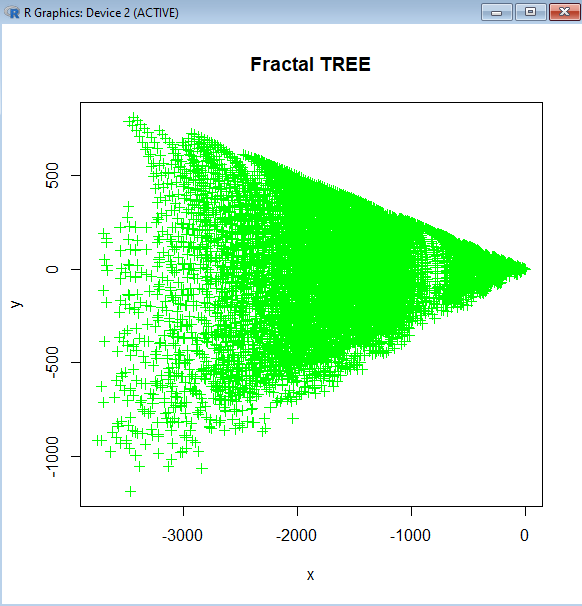


Рисунок 7 – Выполнение скрипта Example1.R с измененными данными.

3) Загрузить свои данные, используя функцию read.table().

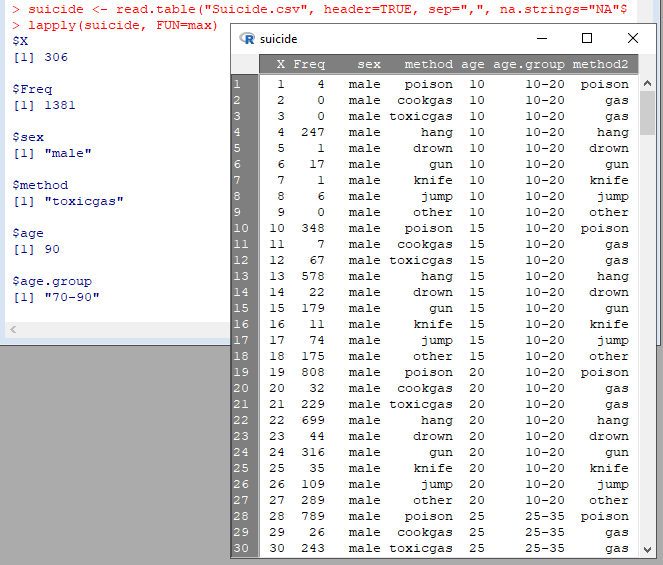


Рисунок 8 – Загрузка данных по самоубийствам в Германии.

4) Используя свои данные, выполнить несколько векторных функций (apply), например, вычисление средних, минимальных, сумм по строкам/столбцам, умножение на коэффициент. Выбор конкретных функций зависит от набора данных.

Больше совершило самоубийство: мужчины.  
Частая причина смерти: токсичный газ, яд.

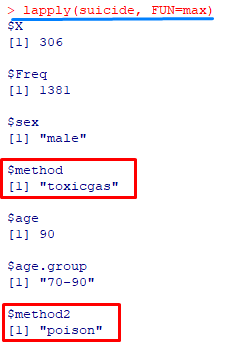


Рисунок 9 – Выполнение lapply(FUN=max).

Меньше совершило самоубийство: женщины.  
Редкая причина смерти: бытовой газ, утопление.

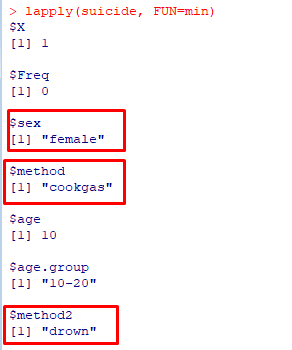


Рисунок 10 – Выполнение lapply(FUN=min).