Biostatistics 615 - Statistical Computing

Lecture 8 Advanced R – Foundations

Jian Kang

Oct 8, 2015

Final Project Principles

- Team project (3 person). Two-person team needs permission from the instructor, and the expected amount of work is the same.
- The project should be chosen from a given list of problems or proposed by the group (recommended). Proposal of new project related to your research interest is encouraged.
- The project requires C++, R, or C++/R combined programming. Other programming languages need permission from the instructor.
- The project is graded based on novelty (15%), difficulty (15%), degree of completion (20%), presentation (25%) and writing (25%).

Final Project Timeline

- Form a team and propose a project by **November 3rd**, and submit your proposal at http://goo.gl/forms/A6tlCAmdzy
- Discuss your project (regarding novelty, difficulty, amount of work, etc) with the instructor as necessary.
- Present your project (10 minutes) in class (during the last three classes in December). It will be graded both by the instructor and by other students.
- Write up a final report and submit it together with your code via email by **December 18th**. The report should include an introduction of your problem, the method/algorithm that you used, results on simulated/real data experiments, and conclusion/discussion. The report should be no more than six pages (11pt font) including tables, figures and references. It will be graded based on the clarity of writing and the thoughtfulness of your method and experiments.

Sample Journals

Possible journals of interest are (you are not limited to theses):

- Journal of Computational and Graphical Statistics: (http://amstat.tandfonline.com/loi/jcgs).
- Journal of Statistical Software: (http://www.jstatsoft.org/).
- Annals of Applied Statistics: (http://imstat.org/aoas/).
- Journal of the American Statistical Association: (http://www.tandfonline.com/loi/jasa).
- Journal of the Royal Statistical Society: Series B: (http://onlinelibrary.wiley.com/journal/10.1111/%28ISSN%291467-9868).
- Journal of the Royal Statistical Society: Series C: (http://onlinelibrary.wiley.com/journal/10.1111/%28ISSN%291467-9876).

Sample Papers

Possible papers of interest are (you are not limited to theses):

- Bayesian classifier for multi-type spatial point patterns: (http://wagerlab.colorado.edu/files/papers/KangNichols.pdf).
- Hidden Markov model for haplotype reconstruction: (http://stephenslab.uchicago.edu/MSpapers/Scheet2006.pdf).
- Coordinate descent algorithm for generalized linear models: (http://www.jstatsoft.org/v33/i01/paper).
- Dynamic programming algorithm for fused lasso: (http://www.tandfonline.com/doi/abs/10.1080/10618600.2012.681238).
- Sequential importance sampling algorithm for density estimation: (http://www.tandfonline.com/doi/abs/10.1080/01621459.2013.813389).

Possible Projects

2. Re-implement an existing R package/function(s), and compare it to the existing one.

- Find an R package/function of interest. Possible R packages/functions of interest are (you are not limited to theses):
  - Generalized linear models (glm)
  - Elastic-net (glmnet)
  - Support vector machine (svm)
  - Any clustering algorithm
- find the reference paper regarding the method/algorithm used in the R package/function. Read the paper carefully and fully understand its method/algorithm.
- Re-implement the method/algorithm presented in the paper as an R package. Alternatively, you can also develop your own method/algorithm solving the same problem.
- Compare your R package/function to the existing one, using simulated and/or real datasets.

Possible Projects

3. Suggest your own topic

- Propose the topic within your own research interest.
- Review with instructor the computational/statistical requirements to be implemented and set the goal for the class project.
- Implement your method/algorithm and test it with simulated and/or real datasets.

Advanced R – Foundations

- Data structures
- Subsetting
- Vocabulary
- Style guide
- Functions
- OO field guide
- Environments
- Debugging, condition handling and defensive programming

Data structures

Dimension	Homogeneous	Heterogeneous
1d	Atomic vector	List
2d	Matrix	Data frame
nd	Array	

Vectors

- Two main structures
  - atomic vectors
  - lists
- Three common properties
  - Type, `typeof()`, what it is
  - Length, `length()`, how many elements it contains
  - Attributes, `attributes()`, additional arbitrary metadata.

Atomic vectors

- Atomic vectors are usually created with `c()` short for combine:

```
num.vec = c(1, 2.5, 4.5)
# The L suffix indicates integer rather than double
 int.vec = c(1L, 6L, 8L)
# Use TRUE and FALSE (or T and F) to create logical vectors
 logic.vec = c(TRUE, FALSE, T, F)
 chr.vec = c("Hello", "World")
```

- Atomic vectors are always flat, even if you nest `c()`'s:

```
> c(1, c(2, c(3, 4)))
[1] 1 2 3 4
# the same as
> c(1, 2, 3, 4)
[1] 1 2 3 4
```

- Missing values are specified with `NA`, which is a logical vector of length one. Specific types: `NA_real_`, `NA_integer_` and `NA_character_`

```
> c(1L, NA_integer_)
# This is an integer vector
> c(1L, NA) # NA will always be coerced to the correct type
# This is an integer vector
> c(1, NA)
# This is a numeric (double) vector
```

Types and tests

- `typeof()` is used to determine the specific type.

```
>typeof(c(1L, NA_integer_))
[1] "integer"
```

- Check the specific type: `is.character()`, `is.double()`, `is.integer()`, `is.logical()` or more generally, `is.atomic()`

```
>is.integer(c(1L, NA_integer_))
[1] TRUE
>is.double(c(1, NA))
[1] TRUE
>is.character(c(1,NA_character_))
[1] TRUE
>is.integer(c(1,NA_integer_))
[1] FALSE
>is.atomic(c(1,NA_integer_))
[1] FALSE
> a = c(1, 2)
> is.vector(a)
[1] TRUE
> attributes(a) = list(names=c("X","Y"))
> a
X Y
1 2
> is.vector(a)
```

Types and tests

- `is.vector()` does not test if the object is a vector or not; It returns TRUE only if the object has no attributes apart from names.

```
> a = c(1, 2)
> is.vector(a)
[1] TRUE
> attributes(a) = list(names = c("X", "Y"))
> a
X Y
1 2
> is.vector(a)
[1] TRUE
> attributes(a) = list(width = 2)
> a
[1] 1 2
attr(,"width")
[1] 2
> is.vector(a)
[1] FALSE
```

- `is.numeric()` tests the "numberliness" of vector and return TRUE for both integer and numeric vectors

```
> c(is.numeric(int.vec), is.numeric(num.vec), is.numeric(logic.vec))
[1]  TRUE  TRUE FALSE
```

Coercion in an atomic vector

- All elements of an atomic vector must be the same type.
- When different types are combined they will be coerced to the most flexible type. (Character $>$ numeric $>$ integer $>$ logical)

```
#str() compactly display the structure of an arbitrary R object
> str(c("a",1))
 chr [1:2] "a" "1"
> str(c("a",TRUE))
 chr [1:2] "a" "TRUE"
> str(c("a",1L))
 chr [1:2] "a" "1"
#any type combining with character will be coerced to character
> str(c(1,1L))
 num [1:2] 1 1
> str(c(1,FALSE,TRUE))
 num [1:3] 1 0 1
 #integer and logical type combining with numeric will be coerced to numeric
> str(c(1L,FALSE,TRUE))
 int [1:3] 1 0 1
# logical type combining with integer will be coerced to integer
```

- Explicitly coerce `as.character()`, `as.double()`, `as.integer()` and `as.logical()`

Lists

- Lists are different from atomic vectors because their elements can be of any type, including lists.

```
> a.list = list(1:3,"a",c(TRUE,FALSE,TRUE),function(x,y) x+y)
> str(a.list)
List of 4
 $ : int [1:3] 1 2 3
 $ : chr "a"
 $ : logi [1:3] TRUE FALSE TRUE
 $ :function (x, y)

> x = list(list(list()))
> str(x)
List of 1
 $ :List of 1
  ..$ : list()
> is.recursive(x)
[1] TRUE
```

Lists

- `c()` will coerce the vectors to list before combining them

```
> x = list(list(1,2),c(3,4))
> y = c(list(1,2),c(3,4))
> str(x)
List of 2
 $ :List of 2
  ..$ : num 1
  ..$ : num 2
 $ : num [1:2] 3 4
> str(y)
List of 4
 $ : num 1
 $ : num 2
 $ : num 3
 $ : num 4
```

Lists

- `typeof()` a list is list. You can test for a list with `is.list()` and coerce to a list with `as.list()`. You can turn a list into an atomic vector with `unlist()` . If the elements of a list have different types, `unlist()` uses the same coercion rules as `c()`.

```
> typeof(y)
[1] "list"
> is.list(y)
[1] TRUE
> str(as.list(c(1,2,3,4)))
List of 4
 $ : num 1
 $ : num 2
 $ : num 3
 $ : num 4
> unlist(list(a=2L,b=TRUE,c=0.5))
  a   b   c
2.0 1.0 0.5
> unlist(list(a=2L,b=TRUE,c=c(0.5,1,2,4)))
  a   b  c1  c2  c3  c4
2.0 1.0 0.5 1.0 2.0 4.0
```

Attributes

- All objects can have arbitrary additional attributes, used to store meta-data about the object.
- Attributes can be thought of as a named list (with unique names).
- Attributes can be accessed individually with `attr()` or all at once (as a list) with `attributes()`.

```
> y = 1:10
> attr(y,"information") = "This is an atomic vector"
> attr(y,"information")
[1] "This is an atomic vector"
> attributes(y)
$information
[1] "This is an atomic vector"
```

- `structure()` function returns a new object with modified attributes

```
 > structure(1:10, type=1)
 [1]  1  2  3  4  5  6  7  8  9 10
attr(,"type")
[1] 1
```

Attributes

By default, most attributes are lost when modifying a vector

```
y = 1:10
attr(y,"information") = "This is an atomic vector"
attr(y,"information")
attributes(y[1])
attributes(sum(y))
```

The only attributes not lost are the three most important:

- Names: a character vector giving each element a name
- Dimensions: used to turn vectors into matrices and arrays
- Class: used to implement the `S3` object system
- Each of these attributes has a specific accessor function to set and get values: `names()`, `dim()`, and `class()`

```
> y = 1:10
> names(y) = paste("V",1:10,sep="")
> names(y)
 [1] "V1"  "V2"  "V3"  "V4"  "V5"  "V6"  "V7"  "V8"  "V9"  "V10"
> attr(y,"names")
 [1] "V1"  "V2"  "V3"  "V4"  "V5"  "V6"  "V7"  "V8"  "V9"  "V10"
```

Names

- You can name a vector in three ways
  - When creating it:

```
x = c(abc = 1, def = 2, ghi = 3)
```

  - By modifying an existing vector in place:

```
x = 1:3; names(x) = c("abc","def","ghi")
```

  - By creating a modified copy of a vector:

```
> x = setNames(1:3, c("abc","def","ghi"))
> x
abc def ghi
  1   2   3
```

- Remarks:
  - Names does not need to be unique, but it is most useful when names are unique
  - Not all elements of a vector need to have a name.
    - If some names are missing, `names()` will return an empty string `" "` or `<NA>` for those elements.
    - If all names are missing, `names()` will return `NULL`

Factors

- A factor is a vector that can contain only predefined values and is used to store categorical data.
- Factors are built on top of integer vectors using two attributes: `class()` and `levels()`

```
> x = factor(c("abc","def","def","abc"))
> x
[1] abc def def abc
Levels: abc def
> class(x)
[1] "factor"
> levels(x)
[1] "abc" "def"
> attr(x,"class")
[1] "factor"
> attr(x,"levels")
[1] "abc" "def"

# you can not use values that are not in the levels
> x[1] = "ghi"
Warning message:
In `[<-.factor`(`*tmp*`, 1, value = "ghi") :
  invalid factor level, NA generated
```

Factors

- Factors are useful when you know the possible values a variable may take, even if you do not see all values in a given dataset. Using a factor instead of a character vector makes it obvious when some groups contain no observations

```
> gender.chr = c("female","female","female")
> gender.fct = factor(gender.chr, levels=c("male","female"))
> table(gender.chr)
gender.chr
female
     3
> table(gender.fct)
gender.fct
  male female
     0      3
```

Factors

- R can create a data frame reading from a TXT or CSV file, but it might produce a factor vector for a column that should be a numeric vector.
- This is caused by a non-numeric value in the column, often a missing value encoded in a special way like "N/A" or "-".
- Solution: coerce the vector from a `factor` to a `character` vector, then from a `character` to a `double` vector.

```
> z = read.csv(text = "value\n12\n1\nN/A\n9")
> z$value
[1] 12  1   N/A 9
Levels: 1 12 9 N/A
> class(z$value)
[1] "factor"
> typeof(z$value)
[1] "integer"
> as.double(z$value)
[1] 2 1 4 3
```

Factors

- Another solution: using `na.strings` argument in `read.csv()`

```
> z = read.csv(text = "value\n12\n1\nN/A\n9", na.strings="N/A")
> z$value
[1] 12  1 NA  9
> typeof(z$value)
[1] "integer"
> as.double(z$value)
[1] 12  1 NA  9
```

- Most data loading functions in `R` automatically convert character vectors to factors.
- Using `stringsAsFactors` = `FALSE` to suppress this behavior, and then manually convert character vectors to factors using your knowledge of the data.

```
> z = read.csv(text="value\nabc\ndef\nghi\njkl")
> z$value
[1] abc def ghi jkl
Levels: abc def ghi jkl
> z = read.csv(text="value\nabc\ndef\nghi\njkl",stringsAsFactors = FALSE)
> z$value
[1] "abc" "def" "ghi" "jkl"
```

Matrices and Arrays

- Adding a `dim()` attribute to an atomic vector allows it to behave like a multi-dimensional `array`
- A special case of the array is the `matrix`, which has two dimensions.

```
> A = 1:6
> attr(A,"dim") = c(2,3)
> A
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> dim(A) = c(3,2)
> A
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
> dim(A) = c(1,5)
Error in dim(A) = c(1, 5) :
  dims [product 5] do not match the length of object [6]
> is.matrix(A)
[1] TRUE
> is.array(A)
[1] TRUE
```

Matrices and Attributes

- `length()` and `names()` have high-dimensional generalizations:
- `length()` generalizes to `nrow()` and `ncol()` for matrices, and `dim()` for arrays
- `names()` generalizes to `rownames()` and `colnames()` for matrices, and `dimnames()`, a list of character `vectors` for arrays

```
> length(A)
[1] 6
> nrow(A)
[1] 2
> ncol(A)
[1] 3
> dim(A)
[1] 2 3
> rownames(A) = c("abc","def")
> colnames(A) = c("ghi","jkl","mno")
> A
    ghi jkl mno
abc   1   3   5
def   2   4   6
> dimnames(A) = list(c("xyz","uvw"),c("lmn","opq","rst"))
```

Matrices and Arrays

- `c()` generalizes to `cbind()` and `rbind()` for matrices, and to `abind()` (package `abind`) for arrays.
- You can transpose a matrix with `t()`; the generalized equivalent for arrays is `aperm()`.