ALTUNA AKALIN

R FOR GENOMICS

Contents

What can you do with R?

R is not only a powerful statistical programming language but also go-to data analysis tool for many computational genomics experts. Genomics usually produces high dimensional data sets that are suitable to be analyzed with core R packages and functions. On top of that, Bioconductor and CRAN have an array of specialized tools for doing genomics specific analysis.

Here is a list of computational genomics tasks that can be completed using R.

General data analysis and exploration

Most genomics data sets are suitable for application of general data analysis tools. In some cases, you may need to preprocess the data to get it to a state that is suitable for application such tools.

- unsupervised data analysis: clustering (k-means, hierarchical), matrix factorization (PCA, ICA etc)
- supervised data analysis: generalized linear models, support vector machines, randomForests

Visualization

Visualization is an important part of all data analysis techniques including computational genomics. Again, you can use core visualization technniques in R and also genomics specific ones with the help of specific packages.

- Basic plots: Histograms, scatter plots, bar plots, box plots
- ideograms and circus plots for genomics
- heatmaps
- meta-profiles of genomic features, read enrichment over all promoters
- genomic track visualization for given locus

Dealing with genomic intervals

Most of the genomics data come in a tabular format that contains the location in the genome and some other relevant values, such as scores for those genomic features and/or names. R/Bioconductor has dedicated methods to deal with such data. Here are a couple of example tasks that you can achieve using R.

- Overlapping CpG islands with transcription start sites, and filtering based on overlaps.
- Aligning reads.
- Overlapping aligned reads with exons and counting aligned reads per gene.

Application of other bioinformatics specific algorithms

In addition to genomic interval centered methods, R/Bioconductor gives you access to multitude of other bioinformatics specific algorithms. Here are some of the things you can do.

- sequence analysis: TF binding motifs, GC content and CpG counts of a given DNA sequence
- Differential expression (or arrays and sequencing based measurements)
- Gene set/Pathway analysis: What kind of genes are enriched in my gene set.

Introduction to R

Here are some basic R operations and data structures that will be good to know if you do not have prior experience with R.

Computations in R

R can be used as an ordinary calculator. Here are a few examples:

```r
2 + 3 * 5  # Note the order of operations.

## [1] 17

log(10)  # Natural logarithm with base e=2.718282

## [1] 2.303

4^2  # 4 raised to the second power

## [1] 16

3/2  # Division

## [1] 1.5

sqrt(16)  # Square root

## [1] 4

abs(3 - 7)  # Absolute value of 3-7

## [1] 4

pi  # The mysterious number

## [1] 3.142

exp(2)  # exponential function

## [1] 7.389

# This is a comment line
```

Vectors

R handles vectors easily and intuitively.

```
x <- c(1, 3, 2, 10, 5)  #create a vector x with 5 components
x

## [1]  1  3  2 10  5

y <- 1:5  #create a vector of consecutive integers
y

## [1] 1 2 3 4 5

y + 2  #scalar addition

## [1] 3 4 5 6 7

2 * y  #scalar multiplication

## [1]  2  4  6  8 10

y^2  #raise each component to the second power

## [1]  1  4  9 16 25

2^y  #raise 2 to the first through fifth power

## [1]  2  4  8 16 32

y  #y itself has not been unchanged

## [1] 1 2 3 4 5

y <- y * 2
y  #it is now changed

## [1]  2  4  6  8 10
```

Matrices

A matrix refers to a numeric array of rows and columns. One of the easiest ways to create a matrix is to combine vectors of equal length using cbind(), meaning "column bind":

```
x <- c(1, 2, 3, 4)
y <- c(4, 5, 6, 7)
m1 <- cbind(x, y)
```

```
m1

##      x y
## [1,] 1 4
## [2,] 2 5
## [3,] 3 6
## [4,] 4 7

t(m1)  # transpose of m1

##   [,1] [,2] [,3] [,4]
## x    1    2    3    4
## y    4    5    6    7

dim(m1)  # 2 by 5 matrix

## [1] 4 2
```

You can also directly list the elements and specify the matrix:

```
m2 <- matrix(c(1, 3, 2, 5, -1, 2, 2, 3, 9),
    nrow = 3)
m2

##      [,1] [,2] [,3]
## [1,]    1    5    2
## [2,]    3   -1    3
## [3,]    2    2    9
```

Data Frames

A data frame is more general than a matrix, in that different columns can have different modes (numeric, character, factor, etc.). Small to moderate size data frame can be constructed by data.frame() function. For example, we illustrate how to construct a data frame from genomic intervals or coordinates.

```
chr <- c("chr1", "chr1", "chr2", "chr2")
strand <- c("-", "-", "+", "+")
start <- c(200, 4000, 100, 400)
end <- c(250, 410, 200, 450)
mydata <- data.frame(chr, start, end, strand)
# change column names
names(mydata) <- c("chr", "start", "end",
    "strand")
```

```
mydata

##    chr start end strand
## 1 chr1   200 250      -
## 2 chr1  4000 410      -
## 3 chr2   100 200      +
## 4 chr2   400 450      +

# OR this will work too
mydata <- data.frame(chr = chr, start = start,
    end = end, strand = strand)
mydata

##    chr start end strand
## 1 chr1   200 250      -
## 2 chr1  4000 410      -
## 3 chr2   100 200      +
## 4 chr2   400 450      +
```

There are a variety of ways to extract the elements of a data frame .

```
mydata[, 2:4]  # columns 2,3,4 of data frame

##   start end strand
## 1   200 250      -
## 2  4000 410      -
## 3   100 200      +
## 4   400 450      +

mydata[, c("chr", "start")]  # columns chr and Age from data frame

##    chr start
## 1 chr1   200
## 2 chr1  4000
## 3 chr2   100
## 4 chr2   400

mydata$start  # variable start in the data frame

## [1]  200 4000  100  400

mydata[c(1, 3), ]  # get 1st and 3rd rows

##    chr start end strand
## 1 chr1   200 250      -
## 3 chr2   100 200      +
```

Lists

An ordered collection of objects (components). A list allows you to gather a variety of (possibly unrelated) objects under one name.

```
# example of a list with 4 components - a
# string, a numeric vector, a matrix, and
# a scaler
w <- list(name = "Fred", mynumbers = c(1,
    2, 3), mymatrix = matrix(1:4, ncol = 2),
    age = 5.3)
w

## $name
## [1] "Fred"
##
## $mynumbers
## [1] 1 2 3
##
## $mymatrix
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
##
## $age
## [1] 5.3
```

You can extract elements of a list using the [[]] convention.

```
w[[3]]  # 3rd component of the list

##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4

w[["mynumbers"]]  # component named mynumbers in list

## [1] 1 2 3
```

Plotting

R has great support for plotting and customizing plots. We will show only a few below. Let us sample 50 values from normal distribution and plot them as a histogram. See the output at Figure 1

```r
# sample 50 values from normal distribution
# and store them in vector x
x<-rnorm(50)
hist(x) # plot the histogram of those values
```

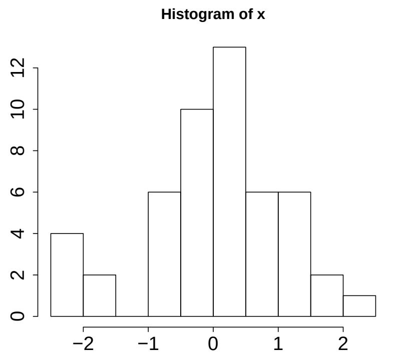


Figure 1: Histogram of 50 random values.

We can modify all the plots by providing certain arguments to the plotting function. Now let's give a title to the plot using `'main'` argument. We can also change the color of the bars using `'col'` argument. You can simply provide the name of the color. Below, we are using "red" for the color. See Figure 2 for the result this chunk.

```r
hist(x, main = "Hello histogram!!!", col = "red")
```

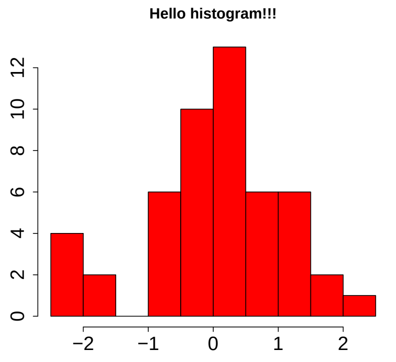
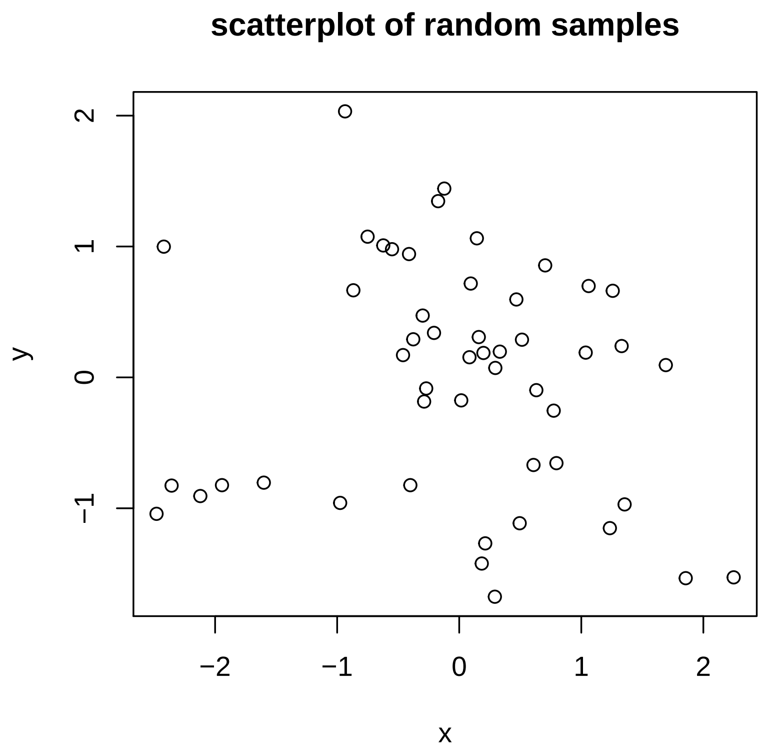


Figure 2: Histogram with a title

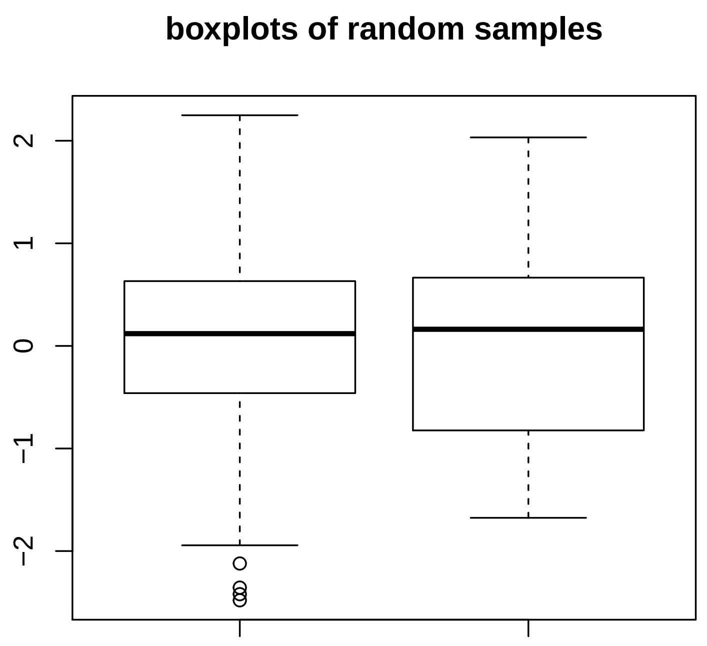
Next, we will plot a scatter plot. We sampled another set of 50 values and plotted those against the ones we sampled earlier.

```r
# randomly sample 50 points from normal distribution
y<-rnorm(50)
#plot a scatter plot
plot(x,y,main="scatterplot of random samples")
```



We can also plot box plots for x and y vectors.

```r
boxplot(x,y,main="boxplots of random samples")
```



Saving plots

If you want to save your plots to an image file there are couple of ways of doing that. Normally, you will have to do the following:

1. Open a graphics device
2. Create the plot
3. Close the graphics device

```r
pdf("mygraphs/myplot.pdf")
plot(x, y)
dev.off()
```

Alternatively, you can first create the plot then copy the plot to a graphical device.

```
plot(x, y)
dev.copy(pdf, "mygraphs/myplot.pdf")
dev.off()
```

Getting help on R functions/commands

Most R functions have great documentation on how to use them. Try `?` and `??`. `?` will pull the documentation on the functions and `??` will find help pages on a vague topic. Try on R terminal:
`?hist`
`??histogram`

Short introduction for Genomics and R

R and Bioconductor has many packages that will help analyze genomics data. Some of the most popular ones are GenomicRanges, IRanges, Rsamtools and BSgenome. Next subsections will show how to use R and bioconductor to read-in and manipulate genomic intervals. Knowing more about R and Bioconductor packages will be useful if you want to customize or enhance your data analysis.

Reading the genomics data

Most of the genomics data are in the form of genomic intervals associated with a score. That means mostly the data will be in table format with columns denoting chromosome, start positions, end positions, strand and score. One of the popular formats is BED format used primarily by UCSC genome browser but most other genome browsers and tools will support BED format. We have all the annotation data in BED format. In R, you can easily read tabular format data with read.table() function.

```
# read enhancer marker BED file
enh.df=read.table("data/subset.enhancers.bed",header=FALSE)
# read CpG island BED file
cpgi.df=read.table("data/subset.cpgi.hg18.bed",header=FALSE)
# check first lrows to see how the data looks like
head(enh.df)

##      V1     V2     V3 V4   V5 V6    V7    V8 V9
## 1 chr20 266275 267925  . 1000  .  9.11 13.17 -1
## 2 chr20 287400 294500  . 1000  . 10.53 13.02 -1
## 3 chr20 300500 302500  . 1000  .  9.10 13.39 -1
## 4 chr20 330400 331800  . 1000  .  6.39 13.51 -1
## 5 chr20 341425 343400  . 1000  .  6.20 12.99 -1
## 6 chr20 437975 439900  . 1000  .  6.31 13.52 -1

head(cpgi.df)

##      V1     V2     V3       V4
## 1 chr20 195575 195851  CpG:_28
## 2 chr20 207789 208148  CpG:_32
## 3 chr20 219055 219437  CpG:_33
## 4 chr20 225831 227155 CpG:_135
## 5 chr20 252826 256323 CpG:_286
## 6 chr20 275376 276977 CpG:_116

# get CpG islands on chr21
head( cpgi.df[cpgi.df$V1=="chr21",] )

##        V1       V2       V3       V4
## 800 chr21  9906603  9906958  CpG:_46
## 801 chr21  9917382  9917652  CpG:_30
## 802 chr21 10011784 10013284 CpG:_152
## 803 chr21 10128589 10129003  CpG:_38
## 804 chr21 13331283 13332372  CpG:_73
## 805 chr21 13957814 13958107  CpG:_24
```

Using GenomicRanges package for operations on genomic intervals

One of the most useful operations when working with genomic intervals is the overlap operation. For example, we may want to know how many of enhancers overlap with CpG islands or how many of the binding sites overlap with promoters, etc. Unfortunately, basic R functions are not designed to deal with such problems, however bioconductor packages: IRanges and GenomicRanges provide efficient ways to handle genomic interval data and provide many functions for operating on genomic intervals. Below, we will show how to convert your data to GenomicRanges objects/data structures and do overlap between enhancers and CpG islands.

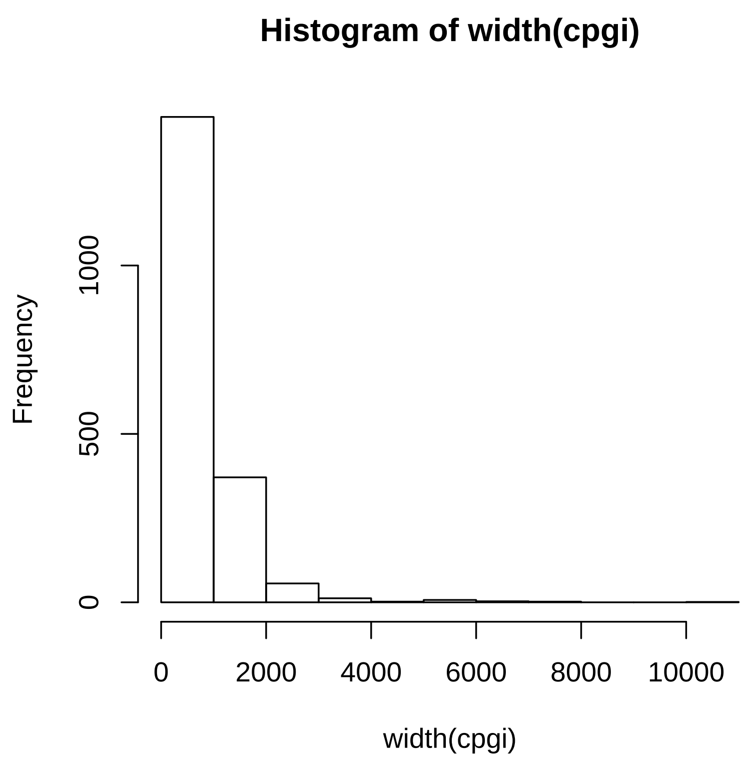
```
# covert enhancer data frame to GenomicRanges object
library(GenomicRanges) # load the package
enh <- GRanges(seqnames=enh.df$V1,
               ranges=IRanges(start=enh.df$V2,end=enh.df$V3)
               )
cpgi = GRanges(seqnames=cpgi.df$V1,
               ranges=IRanges(start=cpgi.df$V2,end=cpgi.df$V3),
               ids=cpgi.df$V4
               )
# find enhancers overlapping with CpG islands
cpg.enh=subsetByOverlaps(enh, cpgi)
# number of enhancers overlapping with CpG islands
length(cpg.enh)

## [1] 1062

# number of all enhancers in the set
length(enh)

## [1] 50416

# plot histogram of lenths of CpG islands
hist(width(cpgi) )
```


Histogram of width(cpgi)
Frequency
width(cpgi)