20.109 Intro to R & Clustering

Amanda Kedaigle, Ernest Fraenkel

1/6/2019

Part 1 - Introduction to R

We'll be analyzing data from RNA-seq experiments on DNA repair performed by your instructors next time. Today, let's get familiar with the tools we'll use to do it.

We'll perform the analysis using the programming language called R[1]. R is free to use and is quite popular for scientific computing. There is a collection of tools for R written for bioinformatics in particular, including RNA-seq data analysis, called Bioconductor. We'll be using R through a interface called RStudio.cloud, which will let you access and analyze the data through an online portal.

1. **Starting R** – You should have recieved an invitation to the 20.109 Computational Lab Rstudio.cloud space in your email (if not, please talk to your instructors). This will let you get an account and open links to the 20.109 space. Click the "Exercise #1" assignment to begin interacting with this lab. Your account will get its own copy of the exercise, which you can edit and run as you want. You can access this copy of the exercise from any computer. As we progress through the course, more assignments will become available at this link.

If you are interested in getting R, and the GUI program for interacting with R called RStudio, for your own computers, you can download them from their websites, www.r-project.org and www.rstudio.org.

2. **RStudio Layout** – Once you click "Lab #1" you will see the RStudio interface in your web browser. The RStudio interface consists of several windows.
    - Bottom left: console window (also called command window). Here you can type simple commands after the ">" prompt and R will then execute your command. This is the most important window, because this is where R actually does stuff. You can try it out by using R as a calculator. Try typing a simple command in this window, i.e. "2+2". Hit enter and R should calculate the answer for you.
    - Top left: editor window (also called script window). Collections of commands (scripts) can be edited and saved. When you don't get this window, you can open it with File → New → R script. Just typing a command in the editor window is not enough, it has to get into the command window before R executes the command. If you want to run a line from the script window (or the whole script), you can click Run or press CTRL+ENTER while your cursor is in that line to send it to the command window. In addition to scripts, you can read "R Markdown" files in this window. This pdf file was created with an R Markdown file.
    - Top right: workspace / history window. In the workspace window you can see which data and values R has in its memory. You can view and edit the values by clicking on them. The history window shows what has been typed before. There's nothing there yet, but there will be.
    - Bottom right: files / plots / packages / help window. Here you can open files, view plots (also previous plots), install and load packages or use the help function. To follow along with this tutorial, open the "Intro_to_R_Clustering.Rmd" R Markdown file, which will open in your editor window.
3. **Working Directory** - Your working directory is the folder on your computer in which you are currently working. When you ask R to open a certain file, it will look in the working directory for this file, and when you tell R to save a data file or figure, it will save it in the working directory. In RStudio.cloud, your working directory is automatically set to the project directory, so we can leave it alone. But if you

[1] These instructions for learning R are adapted from "A (very) short introduction to R" by Paul Torfs and Claudia Bauer.

were working on your own local computer, you would want to set your working directory to where your project files were stored. The command to change the working directory might look like this:

```
setwd("~/Desktop/RNA-seq_data_analysis/")
```

4. **Libraries** – R can do many statistical and data analyses. They are organized in so-called packages or libraries. With the standard installation, most common packages are installed. We'll need to load some packages from Bioconductor. Go to the packages tab in the bottom right window of RStudio. Packages listed in this window are installed, and if the checkbox next to them is ticked, the package is loaded (activated) and can be used. Load the package pheatmap, which we'll later use to analyze our data. You can either click Install and then check the box next to pheatmap, or run the following code.

```
install.packages("pheatmap")
library("pheatmap")
```

5. **Workspace** – In addition to using R like a simple calculator, you can also give numbers a name. By doing so, they become "variables" which can be used later. For example, you can type in the command window:

```
a=4
```

You can see that a appears in the workspace window, which means that R now remembers what a is[2]. You can also ask R what a is (just type a ENTER in the command window):

```
a
```

```
## [1] 4
```

or do calculations with a:

```
a*5
```

```
## [1] 20
```

If you specify a again, it will forget what value it had before. You can also assign a new value to a using the old one.

```
a = a + 10
a
```

```
## [1] 14
```

If you want to remove the variable a, you can type rm(a). To remove all variables from R's memory, click "clear all" in the workspace window.

6. **Scalars, vectors, and Matrices** – Like in many other programs, R organizes numbers in scalars (a single number), vectors (a row of numbers, also called arrays or lists) and matrices (like a table). The a you defined before was a scalar. To define a vector with the numbers 3, 4 and 5, you need the function c, which is short for concatenate (paste together).

```
b = c(3,4,5)
```

7. **Functions** – If you would like to compute the mean of all the elements in the vector b from the example above, you could type

```
(3+4+5)/3
```

```
## [1] 4
```

But when the vector is very long, this is very boring and time-consuming work. This is why things you do often are automated in so-called functions. Some functions are standard in R or in one of the

[2]Some people prefer to use <- instead of = (they do the same thing). <- consists of two characters, < and -, and represents an arrow pointing at the object receiving the value of the expression.

packages. You can also program your own functions. When you use a function to compute a mean, you'll type:

```
mean(x=b)
```

```
## [1] 4
```

Within the brackets you specify the arguments. Arguments give extra information to the function. In this case, the argument x says of which set of numbers (vector) the mean should be computed. Sometimes, the name of the argument is not necessary: mean(b) works as well.

The function rnorm, as another example, is a standard R function which creates random samples from a normal distribution. It takes an argument specifying how many random numbers you want.

```
rnorm(10)
```

```
##  [1] -1.2097791  1.4173686  0.3917976  0.9059368 -0.5227675 -0.8784516
##  [7]  1.2964015 -0.7044103 -0.7079504  1.4626632
```

Entering the same command again produces 10 new random numbers. Instead of typing the same text again, you can also press the upward arrow key (↑) to access previous commands. If you want 10 random numbers out of normal distribution with mean 1.2 and standard deviation 3.4 you can type

```
rnorm(10, mean=1.2, sd=3.4)
```

```
##  [1]  2.6827494 -1.8423179 -3.7477572  0.3287504  3.4259838  3.7721244
##  [7] -3.8601789  0.1141922 -0.5803698  2.3148872
```

showing that the same function (rnorm) may have different interfaces and that R has "named arguments" (in this case mean and sd). By the way, the spaces around the "," and "=" do not matter. Comparing this example to the previous one also shows that for the function rnorm only the first argument (the number 10) is compulsory. R gives default values to the other so-called optional arguments. RStudio has a nice feature: when you type "rnorm(" in the command window and press TAB, RStudio will show the possible arguments. You can also get more information about any function and its arguments by typing "help(rnorm)" or "?rnorm" and see examples of its use by typing "example(rnorm)".

8. **Scripts** – R is an interpreter that uses a command line based environment. This means that you have to type commands, rather than use the mouse and menus. You can store your commands in files, the so-called scripts. These scripts typically have file names with the extension .R. You can open an editor window to edit these files by clicking File and New or Open file... You can run (send to the console window) part of the code by selecting lines and pressing CTRL+ENTER or click Run in the editor window. If you do not select anything, R will run the line your cursor is on. You can always run the whole script with the console command source. You can insert comments into your scripts by starting lines with the # sign. These lines will not be run in the console. You can use them to make your scripts more readable by other users, and to leave notes for yourself in the middle of your code. #Here's a comment that won't be included in the pdf!

9. **Data Structures** – Here are some basic ways to save your data in structures.

- *Vectors* were already introduced, but they can do more. Run each of the following lines to see what they do:

```
vec1 = c(1,4,6,8,10)
vec1
```

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

creates a vector with the given data points.

```
vec1[5]
```

```
## [1] 10
```

This line prints the 5th element in the vector called vec1.

```
vec1[3] = 12
vec1
```

```
## [1]  1  4 12  8 10
```

This line changed the 3rd element of the vec1 vector by accessing it with the vectorname[i] format.

```
vec2 = seq(from=0, to=1, by=0.25)
vec2
```

```
## [1] 0.00 0.25 0.50 0.75 1.00
```

The seq function is another useful way to construct vectors.

```
sum(vec1)
```

```
## [1] 35
```

```
vec1+vec2
```

```
## [1]  1.00  4.25 12.50  8.75 11.00
```

These lines show some useful vector calculations.

- *Matrices* are nothing more than 2-dimensional vectors. To define a matrix, use the function matrix:

```
mat=matrix(data=c(9,2,3,4,5,6),ncol=3)
mat
```

```
##      [,1] [,2] [,3]
## [1,]    9    3    5
## [2,]    2    4    6
```

The argument data specifies which numbers should be in the matrix. Use either ncol to specify the number of columns or nrow to specify the number of rows. Matrix-operations are similar to vector operations:

```
mat[1,2]
```

```
## [1] 3
```

You can access matrix elements with matrixname[row,column].

```
mat[2,]
```

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

When you want to select a whole row, you leave the spot for the column number empty (the other way around for columns of course).

```
mean(mat)
```

```
## [1] 4.833333
```

- *Data frames* are matrices with names above the columns. This is nice, because you can call and use one of the columns without knowing in which position it is.

```
t = data.frame(x = c(11,12,14), y = c(19,20,21), z = c(10,9,7))
t
```

```
##    x  y  z
## 1 11 19 10
## 2 12 20  9
## 3 14 21  7
```

```
mean(t$z)
```

```
## [1] 8.666667
```

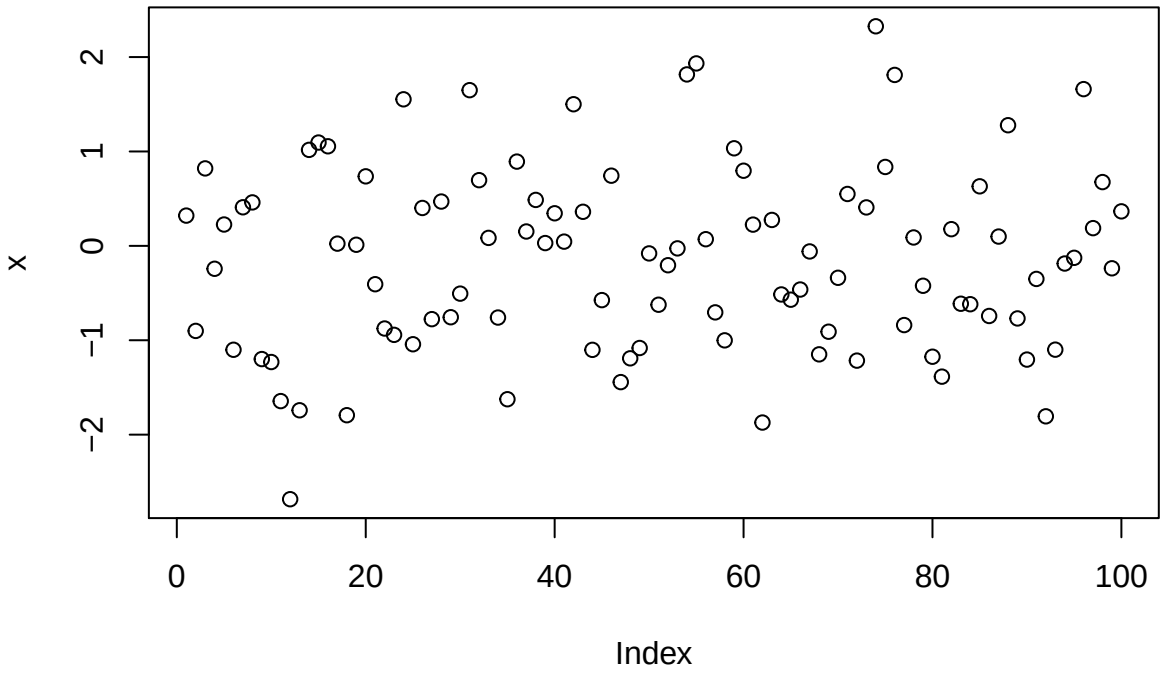
```
mean(t[["z"]])
```

```
## [1] 8.666667
```

These lines show two ways of how you can select the column called z from the data frame called t.

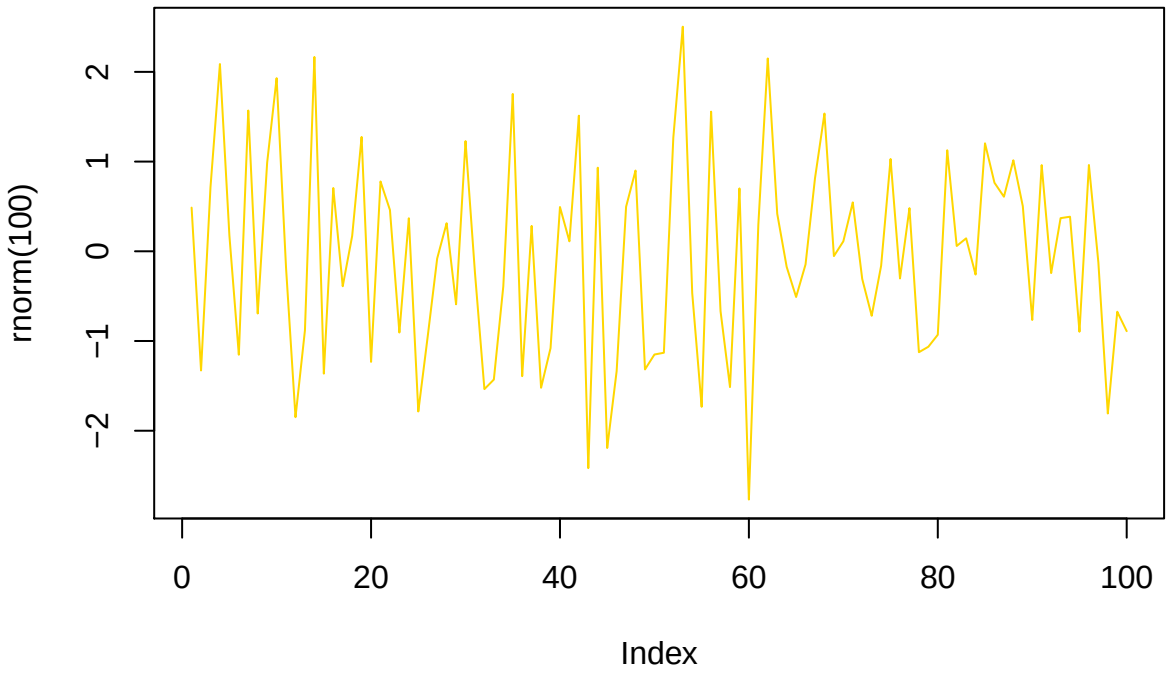
10. **Plotting** - R can make graphs. Here is a very simple example, plotting a vector of 100 random numbers.

```
x = rnorm(100)
plot(x)
```



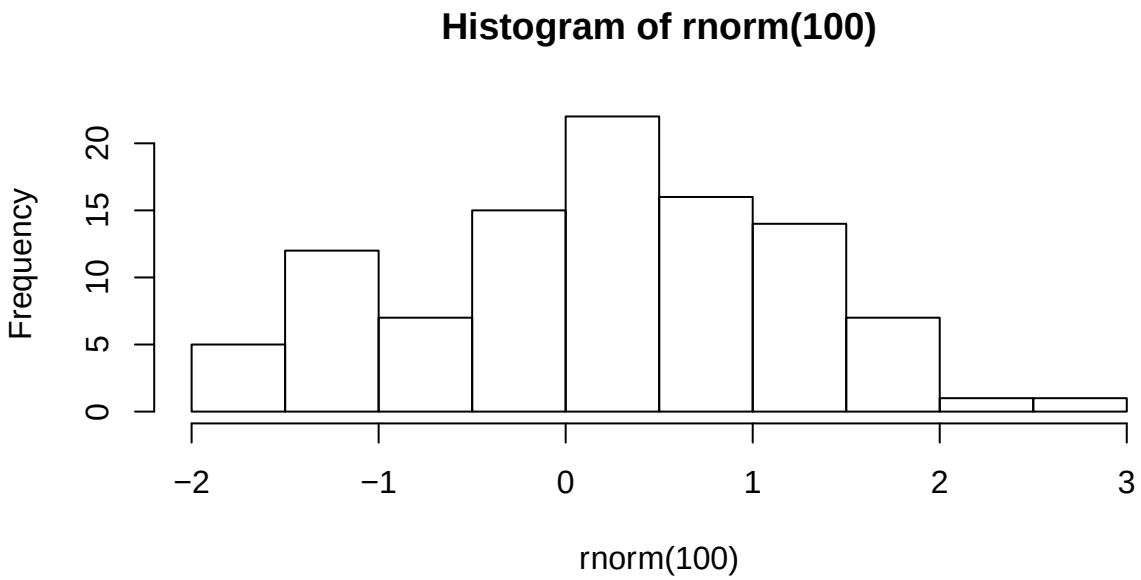
You can also play around with how the graph looks using arguments to the plot function. Here, we change the color and using a line instead of circles (the symbol between quotes after the type=, is the letter l, not the number 1) :

```
plot(rnorm(100), type="l", col="gold")
```



Another very simple example is the classical statistical histogram plot, generated by the simple command

```
hist(rnorm(100))
```



To learn more about formatting plots, search for par in the R help. Google "R color chart" for a pdf file with a wealth of color options. To copy your plot to a document, go to the plots window, click the "Export" button, choose the nicest width and height and click Copy or Save.

11. **Reading and Writing Data Files** – There are many ways to write data from within the R environment to files, and to read data from files. We will illustrate one way here. The following lines illustrate the essential:

```
d = data.frame(a = c(3,4,5), b = c(12,43,54))
d
```

```
##   a  b
```

```
## 1 3 12
## 2 4 43
## 3 5 54
```

```
write.table(d, file="tst0.txt",row.names=FALSE)
d2 = read.table(file="tst0.txt", header=TRUE)
d2
```

```
##   a  b
## 1 3 12
## 2 4 43
## 3 5 54
```

12. **Missing Data** – When you work with real data, you will encounter missing values because instrumentation failed or because you didn't want to measure in the weekend. When a data point is not available, you write NA instead of a number.

    ```
    j=c(1,2,NA)
    ```

    Computing statistics of incomplete data sets is strictly speaking not possible. Maybe the largest value occurred during the weekend when you didn't measure. Therefore, R will say that it doesn't know what the largest value of j is:

    ```
    max(j)
    ```

    ```
    ## [1] NA
    ```

    If you don't mind about the missing data and want to compute the statistics anyway, you can add the argument na.rm=TRUE (Should I remove the NAs? Yes!).

    ```
    max(j, na.rm=TRUE)
    ```

    ```
    ## [1] 2
    ```

Part 2 - Clustering

An important technique in many bioinformatic analyses is clustering, a way to assign similar samples to groups. We'll explore two types of clustering by hand before returning to R.

1. **Hierarchical Clustering** builds a hierarchy of clusters. In the agglomerative, or bottom-up, method of hierarchical clustering, the two most similar samples or clusters are joined into one cluster repetitively. By hand, work out the hierarchical clustering of the following 2D data points, using Euclidean distance (the length of a straight line between two points) and complete linkage (once a cluster contains more than one point, calculate the distance between two clusters as the longest distance between any two points in the respective clusters). Show each step, and when you're done, draw a dendrogram to show the results.

   Point 1: (0,0)

   Point 2: (3,0)

   Point 3: (0,6)

   Point 4: (21,2)

   Point 5: (23,2)

   Now, open the script "intro_clustering.R" on RStudio.cloud and run the code for Problem 1 to see if you get the same dendrogram as R. Save your heatmap to a jpeg image file by clicking Export in the Plots window.

2. **K-means Clustering** is another clustering technique. In this algorithm, you decide ahead of time on a value for k, which is the number of clusters you'll get. You randomly choose k points to be the center or "centroid" of each cluster, and then iteratively assign nearby points to those clusters. Once a cluster has more than one point, the "centroid" is the average of the points in that cluster. Cluster the above points by k-means clustering, with k=3 by hand. Show your work.

- Use points 3, 4, and 5 as your initial centroids.
- Use points 1, 3, and 4 as your initial centroids.
- Finally, try out the code in intro_clustering.R and save your colored plot as a jpeg image.

3. **Principal Components Analysis** is another technique, in which the data points (which often have more dimentions than our 2D points) are projected onto the 2D plane such that they spread out in the two directions that explain most of the difference between them. The x-axis (PC1) is the direction that separates the data points the most. The y-axis (PC2) is a direction (it must be orthogonal to the first direction) that separates the data the second most. The percent of the total variance that is contained in the direction is printed in the axis label. Here we'll use R to cluster a larger high-dimensional dataset. Use the code in intro_clustering.R to generate a random dataset for 15 samples, with information on 100 genes each. Create a heatmap and run principal components analysis on these data.

- After hierarchical clustering, save your heatmap as a jpg file. What are the two most similar samples to sample 7?
- Now run principle components analysis on this data. Save the individuals factors graph to a jpeg image. Are the same samples the most similar to sample 7 with this technique? Notice how much of the variation is explained in this 2-D chart. Do you think this is a good representation of the data?