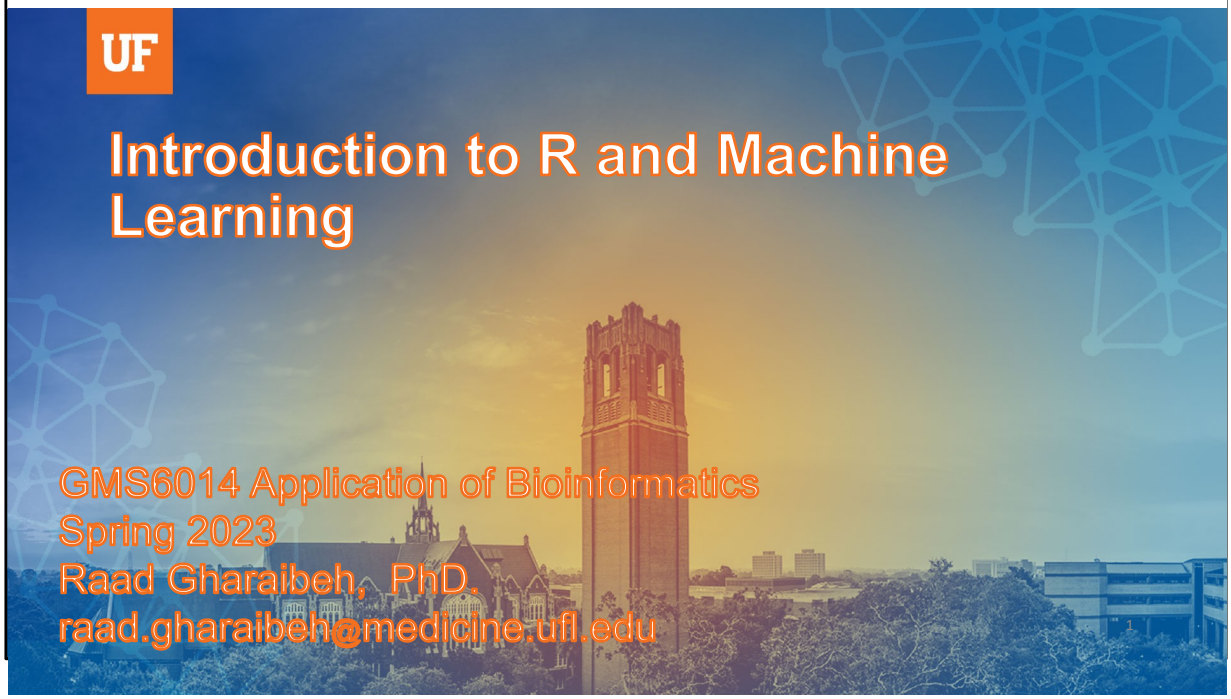

UF
Introduction to R and Machine Learning
GMS6014 Application of Bioinformatics
Spring 2023
Raad Gharaibeh, PhD.
raad.gharaibeh@medicine.ufl.edu

Outline

I. Introduction to R:

a. Getting started with R, Rstudio and installing packages.
b. R Data types and structures.
c. Reading and writing data in R.
d. R Programming concepts.
e. Tidyverse practice.

II. Introduction to Machine Learning:

a) What is Machine Learning: prediction vs. classification.
b) The dataset: training and testing datasets.
c) Machine learning evaluation metrics with examples.

2

Getting started with R and Rstudio

3

References :

R for Beginners: https://cran.r-project.org/doc/contrib/Paradis-rdebuts_en.pdf
Cookbook for R: http://www.cookbook-r.com/
Hands-On Programming with R: https://rstudio-education.github.io/hopr/index.html
Advanced R: http://adv-r.had.co.nz/

Getting started with R and Rstudio

Why R?

1. Free and open source.
2. Cross-platform software: it runs on Windows, Mac OS and UNIX/Linux.
3. Scripts and data objects can be shared across platforms.
4. There is a large and active community of R users (i.e. a lot of support available).
5. Easy to share implementations of new methodologies through R package system (your code can reach wider audience, your paper receives more citations).

4

Getting started with R and Rstudio

1. Download and install R: http://cran.r-project.org
2. Download and install Rstudio: https://www.rstudio.com/products/rstudio/download/#download

Rstudio is an Integrated Development Environment (IDE) for R. You don't have to use Rstudio if you don't want to, but it makes your life easier with R, especially if you are a new R user.

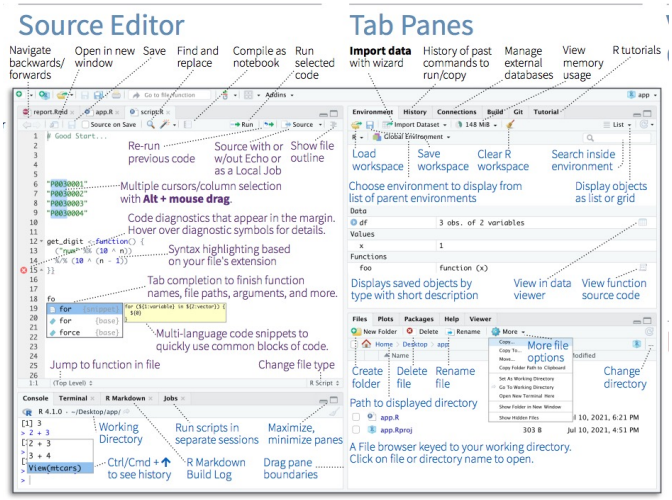
5

Integrated Development Environment (IDE) is a software application that provides comprehensive facilities to computer programmers for software development.
If you need help, watch those tutorial movies:

1) Install R: https://learnr-examples.shinyapps.io/ex-setup-r/#section-install-r

2) Install Rstudio: https://learnr-examples.shinyapps.io/ex-setup-r/#section-install-rstudio

Getting started with R and Rstudio



https://rstudio.org/links/ide_cheat_sheet

6

RStudio includes an editor with many R specific features, a console to execute your code, and other useful panes.
RStudio provides a useful cheat sheet with the most widely used commands. You can get it from RStudio: https://rstudio.org/links/ide_cheat_sheet

R style: http://adv-r.had.co.nz/Style.html

Installing packages

See GMS6014S23_Lec12_r_code.R

Installing packages

- Installing packages depends on the source of the package you are want to install:

1. CRAN packages can be installed using R command: installed.packages() from R console:

```
install.packages("caret", dependencies = T)
install.packages("tidyverse", dependencies = T)
```

2. Bioconductor (PMID: 15461798) packages. To install core packages:

```
if (!requireNamespace("BiocManager", quietly = T))
install.packages("BiocManager")
BiocManager::install()
```

To install specific packages:

```
BiocManager::install(c("GenomicFeatures", "AnnotationDbi"))
```

8

CRAN: The Comprehensive R Archive Network
Bioconductor: https://www.bioconductor.org/
Install packages tutorial: https://learnr-examples.shinyapps.io/ex-setup-r/#section-install-packages

Installing packages

3. github packages: installed using install_github() function from devtools package:

```
install.packages("devtools")
```

To install specific packages:

```
install_github("benjjneb/dada2")
```

Working with packages

Getting help on packages and functions:

```
library(readr) #load package you installed
```

- List all functions in the package **readr**:

```
ls("package:readr")
```

- Get help page for **read_table** function:

```
?read_table or help("read_table")
```

- Search the word "**read**" in help pages:

```
help.search("read") or ??read
```

- Getting help on operators:

```
?"+"
?">"
```

10

R Data types

R Data types

There are several data types in R:

1. Numeric: 1,2,3,4,5, ….
2. Logical : TRUE, FALSE or T, F
3. Character: `'a'`, `'b'`, `'hello'`, `'qual'`
4. Integer: 1L, 2L, 3L
5. Dates: “01/05/2000", "03/07/2001", "08/15/2002", "10/30/2003", "12/22/2004"

12

R Data types

The `class()` function in R helps us determine the type of data we have:

```
> class(1)
[1] "numeric"
> class('a')
[1] "character"
> class(T)
[1] "logical"
> class('T')
[1] "character"
> class(1.4)
[1] "numeric"
> class(1L)
[1] "integer"
```

```
> myDates <- as.Date(c("01/05/2000",
"03/07/2001", "08/15/2002",
"10/30/2003", "12/22/2004"),
"%m/%d/%y")

> class(myDates)
[1] "Date"
```

R Data structures

R Data structures

There are four common data structures in R:

1. **Vectors**: the core R data structure, everything else is just a collection of vectors. A vector is a one-dimensional group of elements of the same type (numeric, character or logical). We create a vector using the `c()` function:

```
myVec<-c(1, 2, 3, 4, 5)
myVec<-c('a', 'b', 'c', 'd')
myVec<-letters[1:4]
```

15

R Data structures

2. **Matrices**: A two-dimensional structure (rows and columns) that stores entries of the same type. A Matrix can't store more than one data type and if you try to create a matrix of numeric and character values, R will automatically convert everything to character. We create a matrix using the `matrix()` function:

```
myMat<-matrix(data, num.rows, num.columns)
myMat<-matrix(c(1,2,3,11,12,13), 2, 3)
```

16

R Data structures

```
myMat<-matrix(c(1,2,3,11,12,13), 2, 3)
```

```
     [,1] [,2] [,3]
[1,]    1    3   12
[2,]    2   11   13
```

The matrix is filled by columns by default, but you can fill it by rows if you set `byrow` option to `T`

R Data structures

3. **Data frames**: A two-dimensional structure (rows and columns) that stores entries of multiple types. We create a data frame using the `data.frame()` function:

```
myDF<-data.frame(chr=c(1,2,3),
strand=c('-', '+', '+'),
start=c(100,2000,10000), end= c(550,3000,
11230))
```

18

R Data structures

```
myDF<-data.frame(chr=c(1,2,3), strand=c('-', '+',
'+'), start=c(100,2000,10000), end= c(550,3000,
11230))
```

```
chr  strand  start   end
1       -    100     550
2       +    2000    3000
3       +    10000   11230
```

R Data structures

4. **Lists**: is an ordered collection of structures that stores a variety of objects under one name. We create a list using the `list()` function:

```
myList<-list(vec=myVec, mat=myMat,
df=myDF, age=25, correct=F)
```

R Data structures

```
myList<-list(vec=myVec, mat=myMat, df=myDF, age=25,
correct=F)
```

```
$vec
[1] "a" "b" "c" "d"
$mat
     [,1] [,2] [,3]
[1,]    1    3   12
[2,]    2   11   13
$df
  chr strand start   end
1   1      -   100   550
2   2      +  2000  3000
3   3      + 10000 11230
$age
[1] 25
$correct
[1] FALSE
```

Reading and writing data in R

Reading and writing data in R

`read.table` and `write.table` will read and write ASCII text files:

```
read.table('myFile.txt', sep='\t', header=T)
read.delim('myFile.txt')
read.csv('myFile.csv')

write.table(myDF, file ='myDF.txt', sep='\t',
row.names=T, col.names=T)

write.csv(myDF, file ='myDF.csv')
```

23

ASCII: American Standard Code for Information Interchange,

Reading and writing data in R

Binary data in R are read and written using **save**, **load** or **saveRDS**, **readRDS** :

```
save(myDF, file='myDF.dat')  #can save one or many objects
load('myDF.dat')

saveRDS(myDF, file='myDF.rds')  #saves single object
myData<-readRDS('myDF.rds')
```

24

Programming concepts: conditional expressions, loops and functions

Programming concepts: conditional expressions

- `if` and `else` statements:

Let's say we want to print the square root of the variable `x` except when `x` equals 1:

```
x <- 1
if(x!=1){
 print(sqrt(x))
} else {
 print("I dont want to calculate the square root of 1")
}
```

26

Programming concepts: loops

- **Loops** are used when we need to repeat a certain task or a function multiple times. A loop will execute the task until a certain condition is met. The most famous example is the **for** loop:

Let's say we have the vector **x**, which has 10 numbers, and we want to find the sum of all those numbers:

```
x <- c(1,2,3,4,5,6,7,8,9,10)
x_sum <- 0
for(i in 1:length(x)){
 x_sum <- x_sum + x[i]
}
print(x_sum) # [1] 55
```

27

Programming concepts: functions

- An R function is a chunk of code used to perform the same task with different input values. A function takes in different arguments and returns an output (*R is a collection of functions*).
- There are two types of functions in R: user defined (you write them) and pre-defined (supplied by R or a package you install).

Let's revisit the `for` loop example, we can rewrite x sum as a function:

```
x_sum <- function(y){
my_sum <- 0
for(i in 1:length(y)){
 my_sum <- my_sum + y[i]
}
return(my_sum)
}

x <- c(1,2,3,4,5,6,7,8,9,10)
x_sum(x)
[1] 55
```

28

Programming concepts: functions

- **sum** is a pre-defined function in R provided by "base" package:

```
x <- c(1,2,3,4,5,6,7,8,9,10)
sum(x)
[1] 55
```

29

https://stat.ethz.ch/R-manual/R-devel/library/base/html/sum.html

Tidyverse practice

30

See GMS6014S23_Lec12_tidyverse_code.R

Reference:
https://www.tidyverse.org/learn/

The tidyverse

- Most of the time we will be dealing with data frame (the preferred type for data storage for many).
- Data frames can be organized in a specific data format called “tidy” and operations performed on these tidy data frames (reordering, subsetting, adding, etc ..) is streamlined using tidyverse.
- tidyverse (https://www.tidyverse.org/) is a collection of packages and can be installed with:

```
install.packages("tidyverse",
dependencies = T)
```

The tidyverse

- tidyverse has many packages, the mostly used ones are:

1. The `readr` package for reading and writing data.
2. The `dplyr` package for manipulating data frames.
3. The `purrr` package for working with functions.
4. The `ggplot2` package for plotting and data visualization.

- All of these work on "tidy" data: a data table with each row represents one observation and columns represent different variables associated with each of these observations.

The tidyverse

- To demonstrate the utility of tidyverse, we will be using a locally processed RNAseq dataset for cancer and normal cell lines.
- Let's read the data directly from dropbox:

```
gene_exp01 <-
read.csv("https://www.dropbox.com/s/qrynlw7wzncesl9/diff_gene_exp.csv?dl=1")


dim(gene_exp01)
[1] 33117    14 # 33,117 rows and 14 columns


colnames(gene_exp01)
 [1] "test_id"    "gene_id"     "gene"        "locus"       "sample_1"
 [6] "sample_2"    "status"      "value_1"     "value_2"     "log2FC"
[11] "test_stat"   "p_value"     "q_value"     "significant"
```

33

The tidyverse

- Let's say we don't want all 14 columns. We just want 6 of them:
  gene, locus, status, log2FC, p_value, q_value
- We can use `select` function to select the columns we want by name:

```
gene_exp02 <- gene_exp01 %>% select(gene, locus, status,
log2FC, p_value, q_value)

dim(gene_exp02)
[1] 33117     6
```

The tidyverse

- Let's suppose we want to filter out genes that did not get "OK" in their test status (or keep only genes with "OK" status):

```
gene_exp03 <- gene_exp02 %>% filter(status =="OK")
```

- Let's clean the log2FC column to remove non-numeric values:

```
gene_exp04 <- gene_exp03 %>%
filter(!is.na(as.numeric(as.character(log2FC))))  %>%
mutate(log2FC=as.numeric(as.character(log2FC))) %>%
filter(log2FC != "Inf")
```

35

The tidyverse

- Let's say we want to annotate `gene_exp04` with a new column. The new column name is "Significant" and it will have one of three values:

a) **If log2FC > 0 and q_value < 0.05 then value is "Significant and Up in Normal".**

b) **If log2FC < 0 and q_value < 0.05 then value is "Significant and Up in Cancer"**

c) **If q_value >= 0.05 then value is: "Not Significant"**

We create a function for that (x: log2FC and y: q_value):

```
sigFunction <- function(x, y){
if(x > 0 & y < 0.05){return(c("Significant and Up in Normal"))}
if(x < 0 & y < 0.05){return(c("Significant and Up in Cancer"))}
if(y >= 0.05){return(c("Not Significant"))}
}
```

36

The tidyverse

- We use `mutate` to add the new column "Significant":

```
gene_exp04 <- gene_exp04 %>%
mutate(Significant=mapply(function(x, y)
sigFunction(x,y), log2FC,  q_value))
```

- We can do all the above in just one step if you don't mind the long code:

```
gene_exp04a <-
read.csv("https://www.dropbox.com/s/qrynlw7wzncesl9/diff_gene_exp.c
sv?dl=1") %>% select(gene, locus, status, log2FC, p_value, q_value)
%>% filter(status =="OK")  %>%
filter(!is.na(as.numeric(as.character(log2FC)))) %>%
mutate(log2FC=as.numeric(as.character(log2FC))) %>% filter(log2FC
!= "Inf") %>%  mutate(Significant=mapply(function(x, y)
sigFunction(x,y), log2FC,  q_value))
```

37

mapply:
https://www.rdocumentation.org/packages/base/versions/3.6.2/topics/mapply
mapply is a functional. Functionals are functions that help us apply the same function to each entry of a vector, matrix, data frame or list.
R has an array of functionals, the most famous are: apply and its siblings (sapply, lapply, mapply, tapply, …).

The tidyverse

- We can generate log2FC summary statistics, using **summarize**:

```
gene_exp04 %>% summarize(average = mean(log2FC, na.rm
=T), standard_deviation = sd(log2FC, na.rm = T))
```

```
 average          standard_deviation
0.2262424           2.417315
```

The tidyverse

- We can sort data:

```
gene_exp04 %>% arrange(log2FC) %>% head()
gene_exp04 %>% arrange(desc(log2FC)) %>% head()
```

- We can do nested sort :

```
gene_exp04 %>% arrange(q_value, log2FC) %>% head()
```

39

The tidyverse

- The **ggplot2** package provides powerful plotting and data visualization functions.
- Examine the code under: #######Plotting Volcano plot with ggplot2#

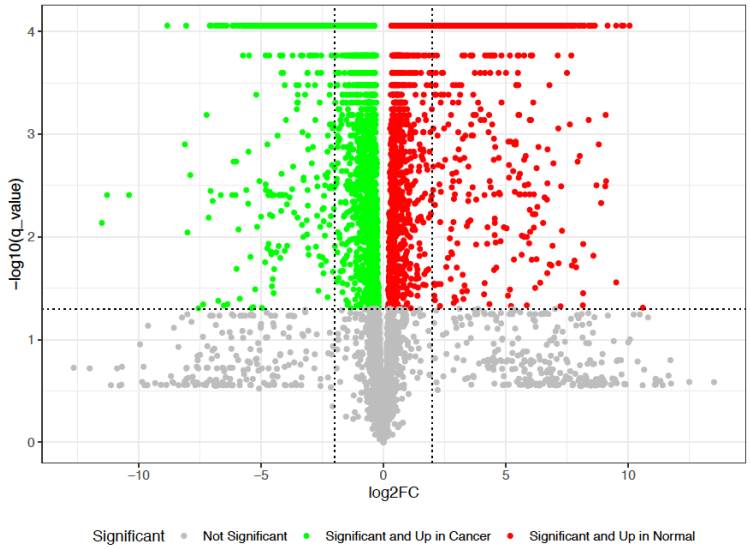
```
v01 <- gene_exp04 %>% ggplot(aes(x = log2FC, y = -log10(q_value))) +
geom_point(aes(color = Significant)) +
scale_color_manual(values = c("grey", "green", "red")) +
theme_bw(base_size = 12) + theme(legend.position = "bottom") +
geom_hline(yintercept = -log10(0.05), linetype = "dotted") +
geom_vline(xintercept = c(-2, 2), linetype = "dotted")
```

- We can save our plot to PDF file with `ggsave`:

```
ggsave(filename="vol_fig1.pdf", plot = v01, height=6, width=8)
```

40

The tidyverse



The tidyverse

- We can customize on the fly, so to generate a volcano plot using genes with log2FC more than -1 and less than 1, we use:

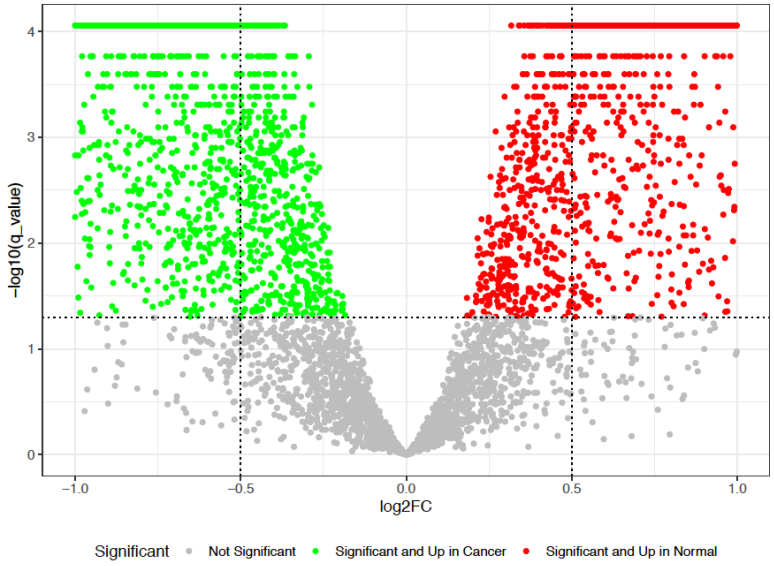
```
v02 <- gene_exp04 %>% filter(log2FC > -1 & log2FC < 1) %>%
ggplot(aes(x = log2FC, y = -log10(q_value)))+
geom_point(aes(color = Significant)) +
scale_color_manual(values = c("grey", "green", "red")) +
theme_bw(base_size = 12) + theme(legend.position = "bottom") +
geom_hline(yintercept = -log10(0.05), linetype = "dotted") +
geom_vline(xintercept = c(-0.5, 0.5), linetype = "dotted")
```

- **We can save our plot to PDF file with `ggsave`:**

```
ggsave(filename="vol_fig2.pdf", plot = v02, height=6,
width=8)
```

42

The tidyverse



43

Machine Learning (ML)

44

See GMS6014S23_Lec12_ml_code.R

References:
Data Mining: Practical Machine Learning Tools and Techniques. Mark A. Hall, Ian H. Witten, Eibe Frank, Christopher Pal
Introduction to machine learning. Ethem Alpaydin
Rafael Irizarry's edX
Google Machine Learning Crash Course Courses.
https://developers.google.com/machine-learning/crash-course

What is Machine Learning

- **Machine learning** is the science (and art) of programming computers so they can **learn** from, and make predictions on **data**. (Aurélien Géron)

- Machine learning has been successfully applied to many areas:

1. Image processing.
2. Speech recognition.
3. Movie recommendation.
4. Spam and malware detectors.

45

These days Artificial Intelligence (AI) and ML are often used interchangeably, but there is a fundamental distinction between the two:
AI implements decision making based on programmable rules derived from **theory** or **first principles**. ML decisions are based on algorithms built with **data**.

What is Machine Learning

- In machine learning, we have two main parts:

1. The **outcome** (what we want to predict).
2. The **features** (what we will use to predict the outcome).

- The mission is to build a system (algorithm) that takes in ***feature*** as input and returns a ***prediction*** for the ***outcome***.

- The approach is to **train** an algorithm using a set of features **for which we know the outcome** and then apply it to another dataset that we **don't know** its outcome.

46

Prediction and Classification

Outcome	Feature 1	Feature 2	Feature 3
y1	x1,1	x1,2	x1,3
y2	x2,1	x2,2	x2,3
y3	x3,1	x3,2	x3,3

- When the outcome is continuous we refer to the machine learning task as **prediction**.
- When the outcome is categorical, we refer to the machine learning task as **classification**.

47

The dataset: Expression level and sample type

- The **outcome**: cancer, normal
- The **features**: *geneX* expression levels.

- The mission is to build an algorithm that takes in *geneX* expression level as input and returns a classification for the sample type {cancer, normal}.

- The machine learning approach is to train the algorithm using this feature (which we know the outcome for) and then apply it to another set of features that we don't know the outcome for.

Table of 1050 x 2
(rows are samples)

Outcome (*y*)	Feature (*x*)
cancer	750
normal	700
cancer	680
normal	740

48

```
dataF<-
read.csv("https://dl.dropbox.com/s/5qe56ysmmigh398/ml_lecture_data.csv?dl=1",
stringsAsFactors=T)
```

Training and Test Datasets

- A machine learning algorithm is evaluated on how it performs in real world with completely new dataset(s).
- When developing an algorithm, we work on a dataset with known outcomes (training dataset).
- In the *geneX* expression level table, the outcome for each sample in the dataset is known.

49

Training and Test Datasets

- The dataset is split into two parts and we pretend as if we don't know the outcome for one of these parts:

1. **Training set** (60-75% of the dataset): we know the outcome and use it to develop and optimize the algorithm. Once we are done developing the algorithm, we don't use this dataset.
2. **Test set** (40-25% of the dataset): we pretend we don't know the outcome and we use it to test the performance of our algorithm.

50

Training and Test Datasets

Outcome (*y*)	Feature (*x*)
cancer	750
normal	700
cancer	680
normal	740

Training set

Outcome (*y*)	Feature (*x*)
cancer	750
normal	700

Test set

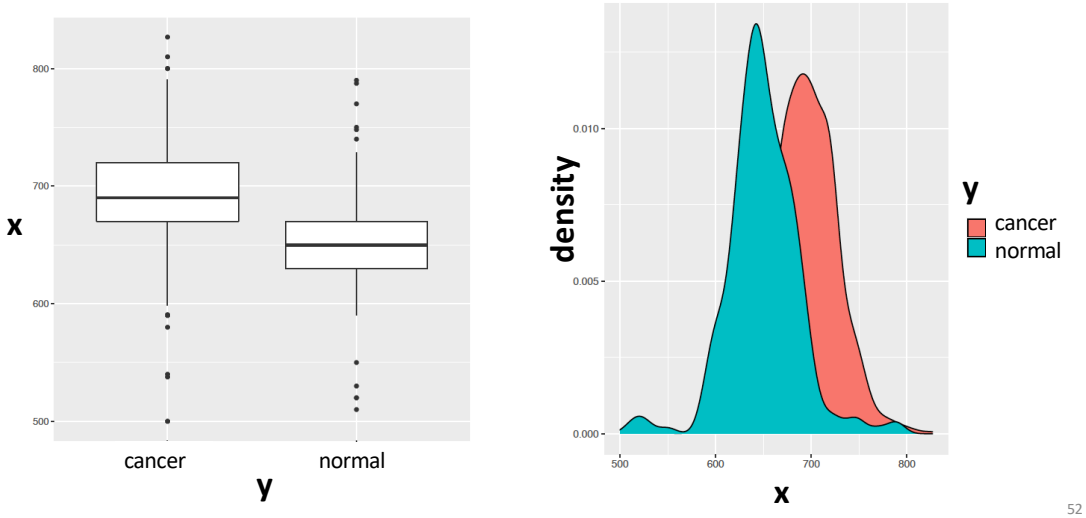
Outcome (*y*)	Feature (*x*)
?	680
?	740

We will stop pretending when we are done constructing our algorithm and start evaluating it.

51

```
train_rows <- createDataPartition(dataF$y, list=F)
train_set <- dataF[train_rows, ]
test_set <- dataF[-train_rows, ]
```

Machine Learning is NOT Magic



52

dataF %>% ggplot(aes(y, x)) + geom_boxplot()
dataF %>% ggplot(aes(x, fill = y)) + geom_density()

Machine Learning Evaluation Metrics

- Let us assume we are done developing our algorithm using the ***training set***. Now, we switch to evaluating it using the ***test set***.
- The simplest way to evaluate the algorithm in our example is by using **Overall Accuracy**: the proportion of cases that were correctly predicted in the ***test set***.
- We will start by developing the simplest possible algorithm: **randomly guessing** the outcome (we are ignoring x and just guessing y).

53

Machine Learning Evaluation Metrics

- Since we are ignoring x and just guessing y, this approach becomes more like flipping a coin.
- As expected, the **Overall Accuracy** for randomly guessing the outcome is 0.4895 (~50%).
- **Can we do better than 50%?**

54

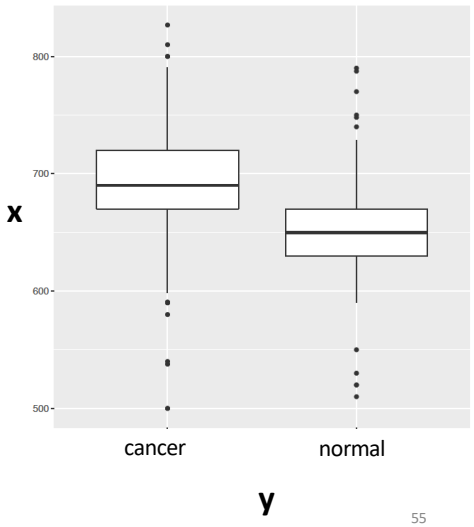
```
#random guessing
y_predict <- sample(c("cancer", "normal"), nrow(test_set), replace = TRUE) %>%
factor(levels = levels(test_set$y))
#overall Accuracy
cat("1) random guessing overall accuracy: ", mean(y_predict == test_set$y), "\n\n")
```

Machine Learning Evaluation Metrics

- Let us explore our training dataset:

y	mean(*x*)	sd(x)
cancer	694	38.7
normal	652	38.4

Algorithm: classify as **cancer** if *geneX* expression level is within two standard deviations from the average *geneX* expression level for **cancer samples**.



55

train_set %>% group_by(y) %>% summarize(mean(x), sd(x))

Machine Learning Evaluation Metrics

- Algorithm: classify as **cancer** if *geneX* expression level is within two standard deviations from the average *geneX* expression level for **cancer** samples**:**

694-(2*39) = 616

- if *geneX* expression level is more than 616 the sample type is **cancer**, else sample type is **normal.**

y	mean(*x*)	sd(x)
cancer	694	38.7
normal	652	38.4

Overall Accuracy is ~ 0.773

Can we do better than 0.773?

56

```
y_predict <- ifelse(train_set$x > (694-(2*39)), "cancer", "normal") %>%  factor(levels = levels(train_set$y))
cat("2) mean/sd overall accuracy: ", mean(y_predict == train_set$y), "\n\n")
```

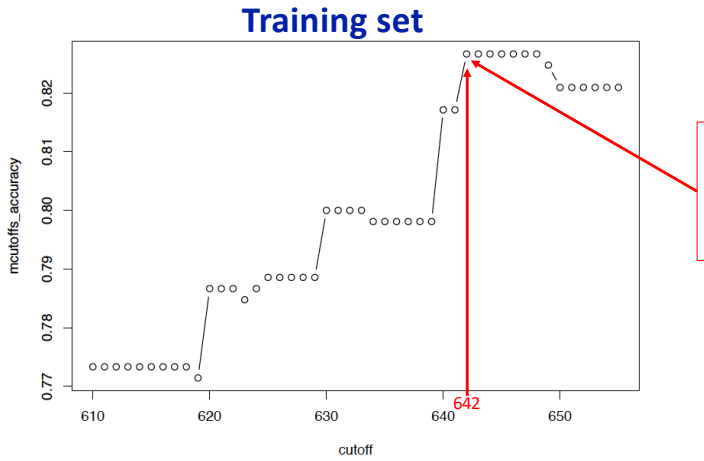
Machine Learning Evaluation Metrics

- Instead of using a single cutoff (616), let us consider multiple cutoffs (from 610 to 655) and then pick the one that provides the best result.
- It is important that we optimize the cutoff using only the **training set** (the test set is only for evaluation).
- Evaluating an algorithm using the training set can lead to **overfitting** (over-optimistic assessments).

57

```
cutoff <- seq(610, 655)
```

Machine Learning Evaluation Metrics



Max **Overall Accuracy** is ~0.83 with *geneX* expression level = 642

58

```
mcutoffs_map_fun<-function(x){
  y_predict <- ifelse(train_set$x > x, "cancer", "normal") %>% factor(levels =
levels(train_set$y))
  mean(y_predict == train_set$y)
}
#####
mcutoffs_accuracy <- sapply(cutoff, mcutoffs_map_fun)
plot(cutoff, mcutoffs_accuracy, type="b")
#####
cat("3a) multiple cutoff highest accuracy (training dataset): ",
max(mcutoffs_accuracy), "\n")
cat("3b) cutoff that resulted in the highest accuracy (training dataset): ",
cutoff[which.max(mcutoffs_accuracy)], "\n")
```

Machine Learning Evaluation Metrics

- Lets take this algorithm that uses 642 as a cutoff and apply it to **the test set**.
- This will result in an **Overall Accuracy** of ~ 0.84

59

```
best_cutoff <- cutoff[which.max(mcutoffs_accuracy)]
y_predict <- ifelse(test_set$x > best_cutoff, "cancer", "normal") %>% factor(levels = levels(test_set$y))
cat("3c) best cutoff overall accuracy on test dataset: ", mean(y_predict == test_set$y), "\n\n")
```

Machine Learning Evaluation Metrics

- Overall accuracy can be a deceptive measure, especially when we have an imbalanced dataset (**prevalence** of one outcome over the other). In the previous example ~77% of the outcome is cancer.
- If we calculate the accuracy for each sample type alone, we will find that our accuracy for cancer= 0.94 but our accuracy for normal= 0.48
- This means although we can classify cancer with high accuracy, we are classifying almost half of normal as cancer!!!
- Clearly, we need better evaluation metrics.

60

```
cat("4) accuracy for each sample type alone:\n")
test_set %>% mutate(y_predict = y_predict) %>% group_by(y) %>%
summarize(accuracy = mean(y_predict == y)) %>% print()
cat("\n\n")
```

Machine Learning Evaluation Metrics

- Sensitivity and specificity to the rescue!
- **Sensitivity**: the ability of an algorithm to predict/classify a *positive outcome* when the *actual outcome is positive*: classified= cancer when actual= cancer.
- But sensitivity on its own is not enough to evaluate an algorithm!
- **Specificity**: the ability of an algorithm to **not classify** a sample as cancer when the **actual** outcome is normal.

61

Machine Learning Evaluation Metrics

Confusion Matrix:

	Actually positive	Actually negative
Classified positive	True positives (TP)	False positives (FP)
Classified negative	False negatives (FN)	True negatives (TN)

Test set Confusion Matrix:

	Actually cancer	Actually normal
Classified cancer	382	62
Classified normal	24	57

62

```
#generate confusion matrix
cm <- confusionMatrix(data = y_predict, reference = test_set$y)
cat("5) best cutoff evaluation metrics:\n")
cat("confusion matrix:\n")
print(cm$table)
```

Machine Learning Evaluation Metrics

Confusion Matrix:

	Actually cancer Actually Positive	**Actually normal** Actually Negative
Classified cancer Classified positive	True positives (TP)	False positives (FP)
Classified normal Classified negative	False negatives (FN)	True negatives (TN)

- **Sensitivity** (on target)= TP/(TP+FN)
- Also called **recall**

63

Machine Learning Evaluation Metrics

Confusion Matrix:

	Actually cancer Actually Positive	Actually normal Actually Negative
Classified cancer Classified positive	True positives (TP)	False positives (FP)
Classified normal Classified negative	False negatives (FN)	True negatives (TN)

- **Specificity** (off target)=TN/(TN+FP)
- Also called **precision**.

Machine Learning Evaluation Metrics

Measure of:	AKA:	Definition:
Sensitivity	Recall	TP/(TP+FN)
Specificity	Precision	TN/(TN+FP)

65

Machine Learning Evaluation Metrics

Confusion Matrix:

	Actually cancer	Actually normal
Classified cancer	382	62
Classified normal	24	57

- **Sensitivity** (on target)= TP/(TP+FN)

= 382 /(382 +24)= 0.94

- **Specificity** (off target)=TN/(TN+FP)

=57/(57+62)=0.48

66

```
cat("\n\n")
print(cm$overall["Accuracy"])
cat("\n")
print(cm$byClass[c("Sensitivity","Specificity", "Prevalence")])
cat("\n\n")
```

Questions?

Notes:

- Those topics are covered in more details in GMS6232
- Evaluation on GatorEvals