A Quick Course in R

1 Introduction

What is R?

- R is free statistical software.
- R is high level programing language.
  - It is an open source implementation of the S programming language.
  - Official description is: R is free software environment for statistical computing and graphics.
  - A lot of extensions. Users can write functions and easily add software libraries, a.k.a. *packages*, to R.
  - More than ten thousand packages are available form the official R depositary. Even more at GitHub.

- R is *interactive*: you type what you want and get out results. Much easier to get started as you can immediately address syntax and other errors.
- *RStudio* is free, open-source IDE (integrated development environment) for R.
  - Commonly used as it makes life easier. *syntax highlighting, interactive autocomplete, easier data/object/graph viewing, etc.*
- RStudio and R can be automatically installed via Purdue Fort Wayne software deployment system on *classroom and lab* computers.
  - Double click at the *Purdue FW Software Center* icon at your Desktop and search for *RStudio*. Both R and RStudio will install automatically.

- Installation:
  - Download from https://cloud.r-project.org/ (or just google "download R").
  - Save the executable file and remember the location.
  - Double click the file and follow the instructions.
  - Once the installation is complete, double click the icon on your Desktop to start an R session.
  - Test R by computing `2+2` (or something else :).
  - To quit R, type `q()`.
- To quickly run something in R (without actually installing R) you can use an online emulator in a web browser, e.g.: https://www.tutorialspoint.com/execute_r_online.php

2 Help

There are many ways to get help for R:

- On the Internet at www.r-project.org
- If you know the command you want help for, from the command line type:
  `help(command,` e.g., `help(lm)`
  `?command,` e.g., `?lm`
- If you only know the topic you want help for, from the command line type:
  `??topic,` e.g., `??logarithm`

Many online resources. We recommend Quick-R for a quick external reference:
https://www.statmethods.net/about/sitemap.html

3 Data Structures

R operates on **data structures**. A data structure is an object, some sort of "container" that holds certain kinds of information.

Common R data structures:

- Vector (a sequence of *numerical*, *character*, *factor/categorical*, or *logical* elements of the *same type*)
- Lists (collection of other objects, e.g., vectors of *any type* and *any length*)
- Matrices/Arrays (multi-dimensional collection of vectors of the *same type* and *same length*)
- Data Frame (list of vectors of *equal length* but possibly *different data types*)

A **vector** is a sequence of values of the same data type.

The `c` function (concatenate) can be used to join data from end to end to create vectors.

The calls below will create numeric, character, and Boolean vectors.

```
c(1, 2, 5.3, 6, -2, 4)
c("one", "two", "three")
c(TRUE, FALSE, TRUE)
```


The `seq` function (from sequence) can be used to create an equidistant series of values.

- A sequence of numbers from 1 to 10 in increments of 1.
  ```
  seq(1, 10)
  1:10
  ```
- A sequence of numbers from 1 to 20 in increments of 2.
  ```
  seq(1, 20, by = 2)
  ```
- A sequence of numbers from 10 to 20 of length 100
  ```
  seq(10, 20, len = 100)
  ```

The `rep` function (from replicate) can be used to create a vector by replicating values.

- Repeat the sequence 1, 2, 3 three times in a row.
  ```
  rep(1:3, times = 3)
  ```
- Repeat "trt1" once, "trt2" twice, and "trt3" three times.
  ```
  rep(c("trt1", "trt2", "trt3"), times = 1:3)
  ```
- Repeat each element of the sequence 1, 2, 3 four times
  ```
  rep(1:3, each = 4)
  ```
- To access the last output in R, type
  ```
  .Last.value # last output
  ```
- Anything on a line after a # character is *comment* (R will ignore the comment).

- To store a data structure in the computer's memory we must assign it to an object (name). Names are case sensitive.
- Data structures can be stored using the assignment operator "<-" or ("=")
  - E.g., store the sequence from 1 through 5 in an object named `v1`.
    ```
    v1 <- 1:5
    ```
- To access the data stored in an object, we simply type the variable name into R and hit enter.
  ```
  v1
  ```
- Vectors can be combined and stored in a single vector using the c function and the assignment operator.
  ```
  v2 <- c(1, 10, 11)
  new <- c(v1, v2; new
  ```


A **matrix** can be created with the `matrix()` function. For example:

```
A<-matrix(data=1:6, nrow=3,ncol=2); A
B<-matrix(data=1:6, nrow=5, ncol=10, byrow=T); B
```

Notice that for B above the matrix function automatically repeats the "data" `1,2,3,4,5,6` as needed in order to fill-in all the entries. In recent versions of R it will give a warning about this.


```
  dim(B) #gives both dimensions of the
matrix B
```

For all ways to create a matrix, type ?matrix.

R will automatically perform an operation to all entries of a matrix, a vector, a data frame variable, or even a data frame:

```
(1:4)^2 # will return 1,4,9,16
log(B)  # will take natural log on each
entry of the matrix B
```

Categorical data should be stored as a *factor* in R. The factor function takes vectors of any data type and converts them to factors.

- Examples:

```
f1 <- factor(rep(1:6, times = 3))
f1
f2 <- factor(c("a", 7, "blue", "blue"))
f2
is.factor(f2)
```

- Many model fitting R functions will automatically handle categorical variables when properly coded as a factor.

4 Helpful Functions

General Functions

```
str(x)    # compact overview of object x
length(x)# length of x
sum(x)    # sum elements in x
mean(x)   # mean of elements in x
var(x)    # sample variance of elements in x
sd(x)  # standard deviation of elements in x
range(x) # range of elements in x
log(x)    # ln of elements in x
summary(x)# 5-number summary of x
```


Functions related to statistical distributions

Suppose that a random variable X has the "dist" distribution

`p[dist](q, ...)` – returns the cdf of X evaluated at q, i.e., $p = \Pr(X \leq q)$.

`q[dist](p, ...)` – returns the inverse cdf (or quantile function) of X evaluated at `p`, i.e., $q = \inf\{x: \Pr(X \leq x) \geq p\}$.

`d[dist](x, ...)` – returns the mass or density of X evaluated at `x` (depending on whether it's discrete or continuous).

`r[dist](n, ...)` – returns an i.i.d. random sample of size `n` having the same distribution as X.

... indicates that additional arguments describing the shape of the distribution (but default values are assumed).

Examples:

```
pnorm(1.96, mean = 0, sd = 1)
```

returns the probability that a normal random variable with mean 0 and standard deviation 1 is less than or equal to 1.96. That is, for Z~N(0,1), the function returns P(Z≤ 1.96).

- Note that mean=0, sd=1 are the default values so you can skip them, `pnorm(1.96)` will give the same result.

- If you want upper tail, P(Z> 1.96), we can get it with
  `pnorm(1.96, lower.tail=FALSE)`
  or
  `1-pnorm(1.96)`

```
qunif(0.6, min =0, max = 1)
```

returns the value x such that $P(X \leq x) = 0.6$ for a **uniform** random variable on the interval [0, 1].

- Here `min =0, max = 1` are defaults bounds for the uniform so same result if you type `qunif(0.6)`.

```
dbinom(2, size = 20, prob = .2)
```

returns the probability that $\Pr(X = 2)$ for $X \sim \text{Binomial}(n = 20, p = 0.2)$ random variable

- `dexp(1, rate = 4)` returns the density of an exponential random variable with mean = ¼ evaluated at 1
- `rchisq(100, df = 5)` returns a random sample of 100 observations from a chi-squared random variable with 5 df.

- For list of all distributions built in R check:

  https://cran.r-project.org/web/views/Distributions.html

5 Basic Plotting

Good graphics are essential in data analysis.

- They help us avoid mistakes.
- They help us decide on a model.
- They help communicate the results of our analysis.
- Graphics can be more convincing than text many times.

  A picture is worth a thousand words.

- The plotting capabilities of R are one of its most powerful and attractive features.
- It is relatively simple to construct histograms, (parallel) boxplots, scatterplots, etc.
- A histogram is created using the `hist` function.
- A boxplot is created using the `boxplot` function.
- A scatterplot is created using the `plot` function.

  Many R packages exist for "fancier" graphics. For instance, `ggplot2` package is often recommended.

Histograms

Histogram with a custom x-axis label and title

```
x <- rnorm(100, mean = 100, sd = 10)
hist(x, xlab = "x-values",
   main = "Histogram of 100 observations from
Normal(100, 10^2)distribution")
```


Boxplots

Single Boxplot

```
y <- rnorm(100, mean = 80, sd = 3)
boxplot(y)
```

Parallel Boxplot

```
grp <- factor(rep(c("Grp 1", "Grp 2"), each
= 100)) #make groups for x and y
dat <- c(x, y)
boxplot(dat ~ grp, xlab = "Group")
```


Scatterplots

Construct a scatterplot with x on the x-axis and y on the y-axis:

```
#generate vectors
x <- runif(20)
y <- 2 + 3 * x + rnorm(20)
plot(x, y)
```

Scatterplot with custom labels and title and a line for the deterministic part of the relationship:

```
plot(x, y, xlab="1st variable", ylab="2nd
variable", main="Title of the plot")
abline(2,3,col="blue")
```


Line plot of standard normal density

```
x <- seq(-4, 4, len = 1000)
y <- dnorm(x, mean = 0, sd = 1)
plot(x, y, xlab="x", ylab="density", type =
"l")
title("Density of Standard Normal") # same
as adding a main="Density of Standard
Normal" in the plot function call
```

"Histogram" scatterplot of probability mass function

```
#plot of Binomial(n = 20, p = .3) pmf
x <- 0:20
y <- dbinom(x, size = 20, prob = .3)
plot(x, y, xlab="# Successes", ylab="Prob",
type = "h")
title("pmf of Binomial(n = 20, p = .3)")
```


6 Data Frames

- Date frames are created by passing vectors into the `data.frame` function.
  - The names of the columns in the data frame are the names of the vectors you give the `data.frame` function.
- Example:

```
d <- c(1, 2, 3, 4)
e <- c("red", "white", "blue", NA)
f <- c(TRUE, TRUE, TRUE, FALSE)
mydataframe <- data.frame(d,e,f)
mydataframe
```


- The columns of a data frame can be renamed using the `names` function on the data frame.

```
names(df) <- c("ID", "Color", "Passed")
df
```

- The columns of a data frame can also be named when you are first creating the data frame by using "name =" for each vector of data.

```
df2 <- data.frame(ID=d, Color=e,
Passed=f); df2
```


- The vectors in a data frame may be accessed using "$" and specifying the name of the desired vector.
- Access the Color vector in `df`:

```
df$Color
```

- The vectors of a data frame may also be accessed by specifying the desired row(s) or column(s) in square brackets.
- Access first row of `df`

```
df[1,]
```

- Access third column of `df`

```
df[,3]
```

- Access the ID column of `df2` and assign it to `newID`

```
newID <- daf2$ID
```

7 Importing Data

- The `read.table` function imports data into R as a data frame.
- Usage: `read.table(file, header = TRUE, sep = ",")`
- `file` is the filepath and name of the file you want to import into R
- To change the directory use setwd, e.g., in Windows
```
setwd("PFW Teaching//QuickRCourse")
getwd() # prints the current directory
```
- If you don't know the file path, type `file = file.choose()`. This will bring up a dialog box asking you to locate the file you want to import.

- `header` specifies whether the data file has a header (labels for each column of data in the first row of the data file).
  - If you don't specify this option in R or use `header=FALSE`, then R will assume the file doesn't have any headings.
  - `header=TRUE` tells R to read in the data as a data frame with column names taken from the first row of the data file.

- `sep` specifies the delimiter separating elements in the file.
  - If each column of data in the file is separated by a space, then use `sep = " "`
  - If each column of data in the file is separated by a comma, then use `sep = ","`
  - If each column of data in the file is separated by a tab, then use `sep = "\t"`.

- In a toy example, we will read 4 rows of data with header and tab separators.

```
data <- read.table("example.txt",
     header = TRUE, sep = "\t")
str(data) # 3 numeric variables
mean(data$Third) #mean of the variable Third
```


8 Accessing Elements of a Data Structure

- Subsets of the elements of a vector may be selected by appending to the name of the vector an index vector in square brackets.
```
a <- seq(2, 16, by = 2); a
```
- Access the 2nd, 4th, and 6th elements of `a`.
```
a[c(2, 4, 6)]
```
- Access all elements in `a` EXCEPT the 2nd, 4th, and 6th.
```
a[-c(2, 4, 6)]
```
- Access all elements in `a` except elements 3 through 6.
```
a[-(3:6)]
```
- The following will produce an error (indices start from 1)
```
a[-3:6]
```

```
B <- matrix(data=1:12, nrow=3, ncol=4,
byrow=T); B
```

- To access the last entry in the last row of the matrix:
  `B[3,4]`
- To grab the second row only.
  `B[2,]`
- What is the median of the forth column?
  `median(B[,4])`

- Sometimes we need to know if the elements of an object satisfy certain conditions. This can be determined using the logical operators `<`, `<=`, `>`, `>=`, `==`, `!=`
  `"=="` means "equal to" and `"!="` means not equal to.

- values of `a` greater than 10
  `a > 10`
- values of `a` less than or equal to 4
  `a <= 4`
- values of v `a` equal to 10
  `a == 10`
- values of `a` not equal to 10
  `a != 10`

- More complicated logical arguments can be made using & and |.
  - & means "and"
  - | means "or"
- Elements of `a` greater than 6 and less than or equal to 10
  `(a > 6)&(a <= 10)`
- Elements of `a` less than or equal to 4 or greater than or equal to 12.
  - `(a <= 4)|(a >= 12)`

- Logical statements can be used to return parts of an object satisfying the appropriate criteria.
- Return elements of `a` less than 6.
  `a[a < 6]`
- Return elements of `a` equal to 10.
  `a[a == 10]`
- Return elements of `a` less than 6 or equal to 10.
  `a[(a < 6)|(a == 10)]`

9 Functions

- A function is essentially a sequence of commands executed based on certain arguments supplied to the function.

In R, a function is defined using the general format:

```
myfunction <- function(arg1, arg2, arg3)
{
   code to execute
}
```

- The name of the function is "myfunction", and to use this function, we need to supply 3 arguments.

- A function may or may not return something back that you can store for later use.

- To call the function, you simply type

```
myfunction(10, 3.17, "CU")
```

Default values might be provided by the function so arguments can be omitted when making the function call.

```
myfunction <- function(arg1, arg2=3,
arg3="Purdue" )…
```


Example of a function that returns the standard deviation of a vector x

The sole argument is `x`, the vector of values for which we want to determine the standard deviation

```
stdev <- function(x)
{
   s <- sqrt(sum((x - mean(x))^2)/(length(x) - 1))
   s
}
```

```
z <- rnorm(20) # 20 random values N(0,1)
z
stdev(z)
sd(z) # the sd() built-in R function gives
the same result
```


Example of a function that returns the density of a normal random variable with mean mu and standard deviation sigma for a vector x.

The arguments are:

- x, the vector of values at which I want to determine the density
- mean, the mean of the normal distribution
- sigma, the standard deviation of the normal distribution

```
normal.density <- function(x, mu = 0, sigma = 1)
{
return(exp(-(x-mu)^2/(2*sigma^2))/sqrt(2*pi*sigma^2))
}
```

Example: Create function that returns the mean and standard deviation of a vector x.

The sole argument is:

- x, the vector of values for which I want to determine the mean and standard deviation

```
ms <- function(x)
{
   m <- mean(x)
   s <- sd(x)
   return(list(m = m, s = s))
}
y <- 1:10
rslt <- ms(y); rslt
rslt$m-mean(y) #should give 0
```


10 Installing Packages

- Many useful R functions come in packages, free libraries written by R's community.
- The CRAN online package repository features more than 12,000 available packages. More packages are available outside of CRAN.
- To install an R package, type:

`install.packages ("package_name")`, e.g.:

```
install.packages ("faraway")
```

- Or use the menu `Packages->Install package(s).`
- You need to install a package only once.

- R will download the package from CRAN, so you must be connected to the internet.
- To access the functions and/or the data in the package in your current R session:

`library("package_name")`, e.g.:

```
library(faraway)
```

To check the names of all *functions* and *datasets* in the package, type:

```
ls("package:faraway")
```


- Prominently used packages are well supported and documented.
- Some less prominently used packages might be slow and less refined.
- A good starting point to consider for R packages is RStudio's *Quick list of useful R packages available*:

https://support.rstudio.com/hc/en-us/articles/201057987-Quick-list-of-useful-R-packages

11 Data and Plotting Example

It is critical to perform initial data analysis of real data.

Calculate numerical summaries:

- means
- standard deviations (SDs)
- maximum and minimum, correlations
- anything else that may be appropriate.

Construct the appropriate plots.

- For one variable, consider boxplots, histograms, density plots, etc.

- For two variables, scatterplots.
- For three or more variables, you can construct interactive and dynamic graphics.

Look for outliers, data-entry errors, skewness, unusual distributions, structure.

Make sure to clean the data of any errors.

- This may be very time consuming.

Kidney Example

The National Institute of Diabetes and Digestive and Kidney Diseases conducted a study on 768 adult female Pima Indians living near Phoenix. The following variables were recorded: number of times pregnant, plasma glucose concentration at 2 hours in an oral glucose tolerance test, diastolic blood pressure (mmHg), triceps skin fold thickness (mm), 2-hour serum insulin (mu U/ml), body mass index (weight in kg/(height in m2)), diabetes pedigree function, age (years) and a test whether the patient showed signs of diabetes (coded zero if negative, one if positive). The data may be obtained from the UCI Machine Learning Repository [http://archive.ics.uci.edu/ml].

Note that the dataset is also available in the R package *faraway* by Julian Faraway.

In principle, we should find out the purpose of the study and more about how the data were collected. But let's get to analysis.

```
>require(faraway) #load package faraway with
the dataset only if is not already loaded
>data(pima) # load data

>head(pima) # first six rows of pima
  pregnant glucose diastolic triceps insulin
1        6     148        72      35       0
2        1      85        66      29       0
3        8     183        64       0       0
4        1      89        66      23      94
5        0     137        40      35     168
6        5     116        74       0       0
```

Numerical summaries of each variable:

```
> summary(pima)
     pregnant         glucose
 Min.   : 0.00   Min.   :  0
 1st Qu.: 1.00   1st Qu.: 99
 Median : 3.00   Median :117
 Mean   : 3.85   Mean   :121
 3rd Qu.: 6.00   3rd Qu.:140
 Max.   :17.00   Max.   :199
```


The `summary` command is a quick way to get the five-number univariate summary information for each quantitative variable

```
> summary(pima)
```

- Look for anything unusual or unexpected, perhaps indicating a data-entry error.
- The minimum `diastolic` blood pressure is zero! (That's often an indication of a problem).

```
> sort(pima$diastolic)
 [1]   0   0   0   0   0   0   0   0   0   0
[11]   0   0   0   0   0   0   0   0   0   0
[21]   0   0   0   0   0   0   0   0   0   0
[31]   0   0   0   0   0  24  30  30  38  40
```


The first 35 values are zero—there's a problem.

- It seems that 0 was used in place of a *missing value*.
- This is very bad since 0 is a number, this problem may be overlooked, which can lead to faulty analysis!
- This is why we must check our data carefully for potential problems and things that don't make sense.

The value for missing data in R is `NA`.

Several variables share this problem. Let's set the 0s that should be missing values to `NA`.

```
pima$diastolic[pima$diastolic == 0]  <- NA
pima$glucose[pima$glucose == 0] <- NA
pima$triceps[pima$triceps == 0]  <- NA
pima$insulin[pima$insulin == 0] <- NA
pima$bmi[pima$bmi == 0] <- NA
```

The variable `test` is a categorical variable, not numerical.

- R thinks it is numerical.
- In R, a categorical variable is a `factor`.

- We need to convert this variable to a factor.

```
> pima$test <- factor(pima$test)
> summary(pima$test)
  0   1
500 268
```

500 of the cases were negative and 268 were positive. We can provide *more descriptive* labels using the `levels` function.

```
levels(pima$test) <-c("negative","positive")
```

Now the data is cleaned up.

Let's create some plots.

A histogram of diastolic blood pressure.

```
> hist(pima$dias, xlab="Diastolic")
```

We had to use the `$` symbol to access the `dias` variable in the `pima` dataframe.

- This isn't a big deal for a few variables, but if we need to access multiple variables in a dataframe, this can be laborious.

An alternative way of doing this is to use the `with` function.

- The first argument to the with function is the dataframe of reference.

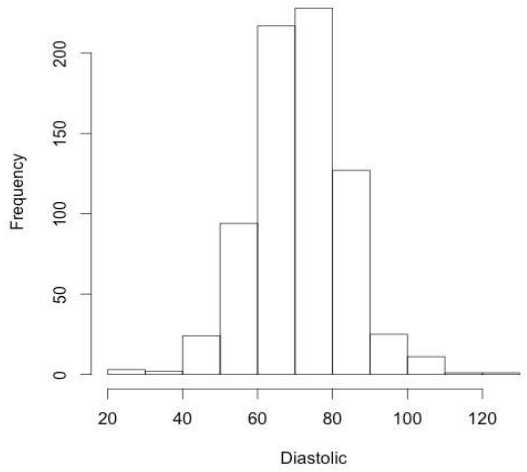
- The second argument is the command you want to execute, referencing the variables in the dataframe directly instead of with the $ symbol.

```
> with(pima, hist(diastolic)) #alternative 1
```

Yet another way to do this is to use the `attach()` function, selecting the dataframe for all the calls until you `detach()`.

```
> attach(pima)
> hist(diastolic)
> detach(pima)
```

`attach()` is useful if you need to do play with the dataset but one often forgets to detach and some names can overlap too.



The histogram is approximately bell-shaped and centered around 70.

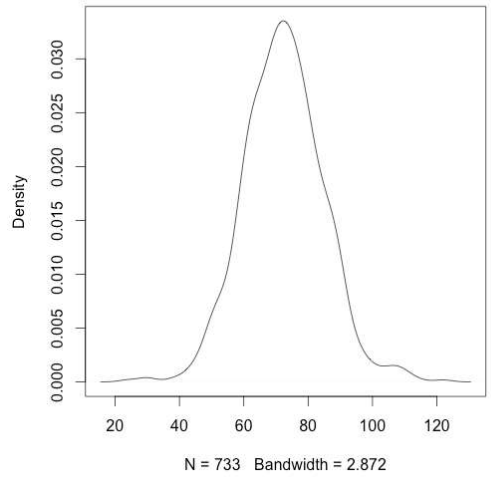
- A histogram can look very different depending on certain choices such as the number of bins and their spacing.

Many people prefer the density plot over the histogram since the histogram is so sensitive to its options.

- A density plot is essentially a smoothed version of a histogram.
- A "kernel" smoother is used to construct a weighted average of data points and create a smooth surface.

```
> plot(density(pima$diastolic,na.rm=TRUE),
main="")
```

The density plot isn't as blocky (though you might see weird things happen at the boundaries).

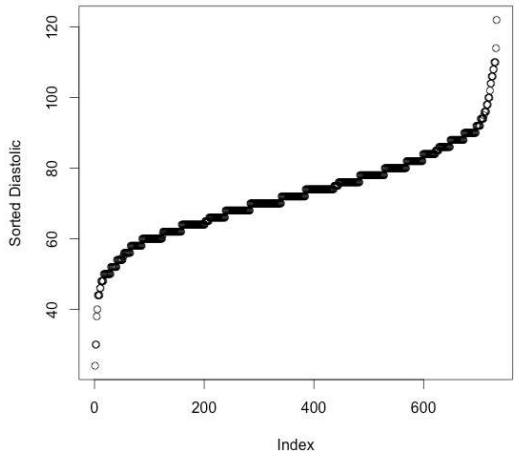


We could simply plot the sorted data against its index.

```
> plot(sort(pima$diastolic),ylab="Sorted
Diastolic")
```

In this plot, we get to see

- the individual values
- the data distribution
- possible outliers
- the "discreteness" of the data.

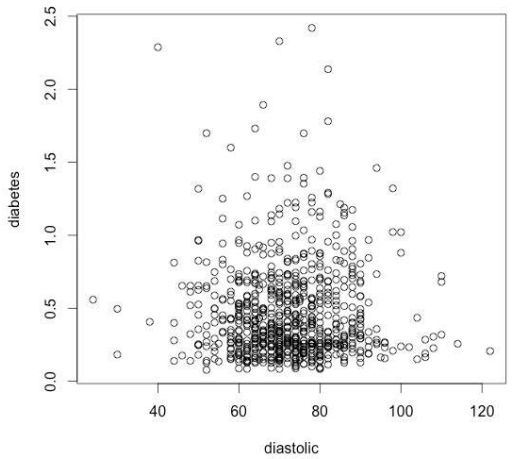


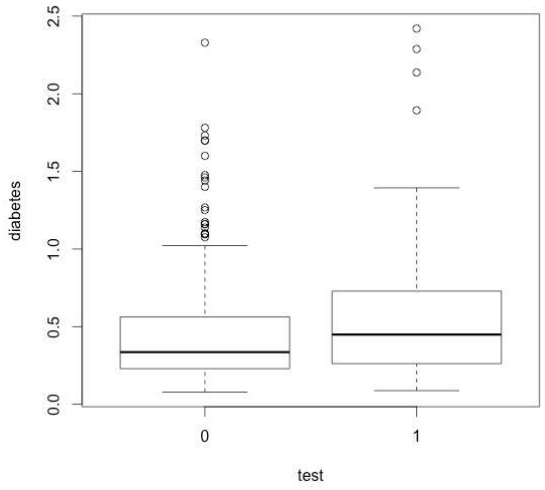
We can create useful bivariate plots using the plot function.

The first plot is a standard scatterplot of diabetes vs diastolic blood pressure.

The second is a parallel boxplot of diabetes vs test result.

```
> plot(diabetes ~ diastolic, data = pima)
# this is the 4th way to specify pima data
> plot(diabetes ~ test, data = pima)
```



The plots we have just created are using the built-in base graphics systems in R.

- These are very fast, simple, and yet professional.

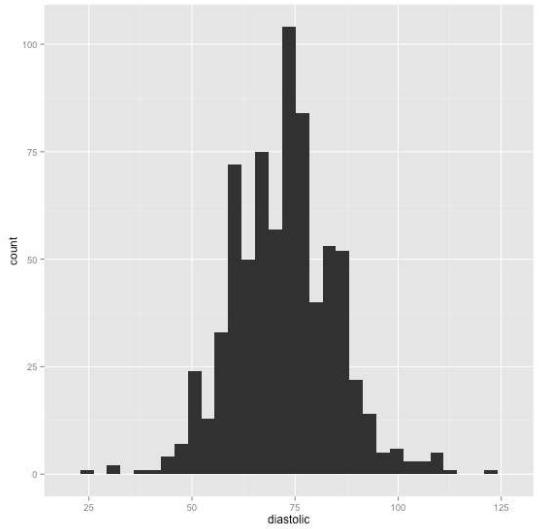
A fancier alternative is to construct plots using the `ggplot2` package.

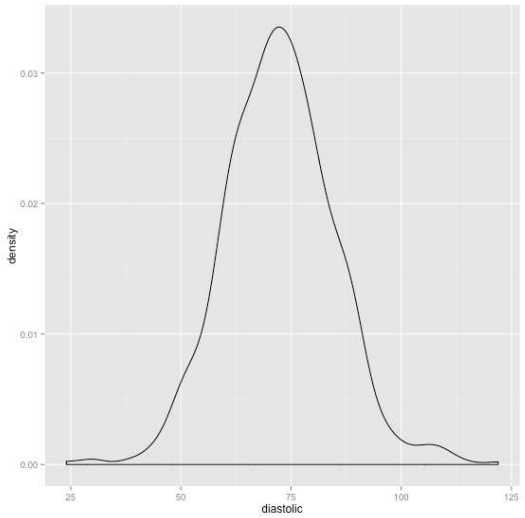
- The plots look perhaps more elegant with a bit more effort.
- First, we need to load the package:

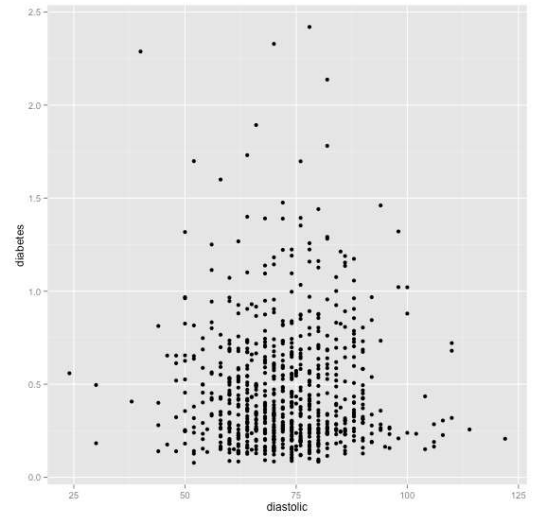
```
if(!require("ggplot2")install.packages("ggplot2")
library(ggplot2)
```

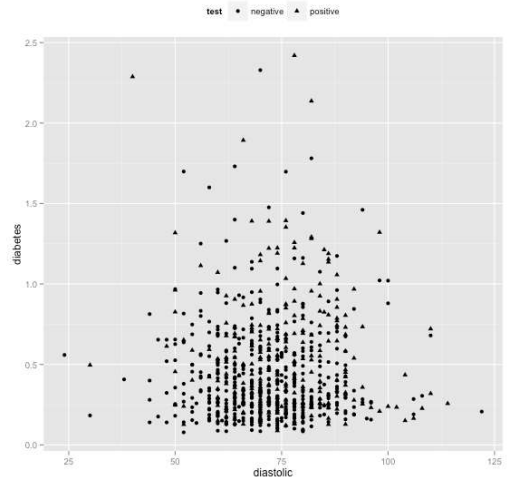
The same plots can now be constructed using `ggplot2`.

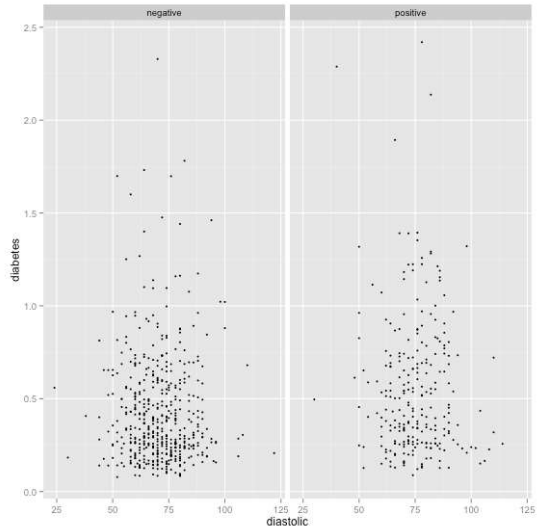
```
ggpimax  = ggplot(pima, aes(x=diastolic))
ggpimaxy = ggplot(pima, aes(x=diastolic, y=diabetes))
ggpimax + geom_histogram()
ggpimax + geom_density()
ggpimaxy + geom_point()
ggplot(pima,aes(x=diastolic,y=diabetes,shape=test)) +
  geom_point() +
  theme(legend.position = "top", legend.direction =
"horizontal")
ggpimaxy + geom_point(size=1) + facet_grid(~ test)
```









For `ggplot2`:

- You first need to create a ggplot object using the `ggplot` function that specifies where the data comes from (the data frame is `pima`) and an **aesthetic** using `aes`.
- The aesthetic specifies what you see such as position in the *x* or *y* direction or aspects such as shape or color.
- The second part of the command (after the +) in each case is specifying the particular geometry for the plot (how you want to map the aesthetics).
- The advantage of `ggplot2` is more apparent in producing more complex plots involving more than just two variables.
- A **theme** specifies options for the appearance of the plot.
  - We specified where the legend should appear in one plot and to use more than one panel (facets) in another.

Other resources about R that may be helpful:

- A rather thorough "**Short Reference Card**": https://cran.r-project.org/doc/contrib/Baggott-refcard-v2.pdf
- Free "official" R notes from R-Project.org:

- R for Beginners, by E. Paradis
https://cran.r-project.org/doc/manuals/R-intro.pdf

- Notes on R: A Programming Environment for Data Analysis and Graphics Version 3.4.3 (2017-11-30), by W. N. Venables, D. M. Smith and the R Core Team
https://cran.r-project.org/doc/contrib/Paradis-rdebuts_en.pdf

Thank you to Dr. Josh French from University of Colorado Denver for sharing his Crash Course in R notes.

Affordable books:

- *The Art of R Programming,* by Norman Matloff (Very good reviews)
- *Linear Models with R, Second Edition,* by Julian J. Faraway (a lot of R code)
- *R Cookbook*, by Paul Teetor

Many free and paid tutorials, courses, and videos are available online (including on YouTube); for instance:
http://cyclismo.org/tutorial/R/
https://www.datacamp.com/courses/free-introduction-to-r

To practice some of the skills covered in this tutorial, please download a short R practicum from:
https://tinyurl.com/R-Ttrl