ACAI R training material

# Preface

This material aims to provide a manual that will be used by the ACAI NARS partners to train R users. The questions we asked ourselves before coming up with this content are:

1. What would be the best way to reach a high number of users with the training content?
2. What would be the best way to make sure the lessons learnt at the training remain active?

The training team came up with 2 solutions:

* Stepping the training down to researchers who are running field experiments this year and who will be facing problems to be solved that are part of the training package
* The participants of the ToT implement the lessons learnt to a hypothetical problem as part 2 of the training.

## Previous trainings

May – July 2020: three virtual trainings “Introduction to R statistical programming language” --- Meklit Chernet (IITA) and Turry Ouma (IITA)

November 2020: In-person training with focus on intermediate and advance levels --- Ibnou Dieng (IITA), Meklit Chernet (IITA), Turry Ouma (IITA) and Sam Ofodile (IITA)

Who would benefit from this training?

Those who have attended the previous trainings and have a clear grasp of the basics of R. This is a linear training in the sense that it solves some of the most common issues that arise from data management and visualization of field trial data. This training uses data collected using ODK and hosted on ona.io.

## Feedback

There will be pre and post course online questionnaires hosted on ona.io to enable us get feedback to improve future training sessions.

# Session 1: Getting started with R and Rstudio

## Is R and Rstudio same?

***R*** is a programming language used for statistical computing and graphics. ***Rstudio*** is an integrated development environment (IDE) that provides an interface which allows users to write and edit programs in ***R*** by offering statistical packages, many convenient features and tools. You can use ***R*** on its own to write a program and run the code. It is however not the case with ***Rstudio*** since it can only be used together with ***R***.

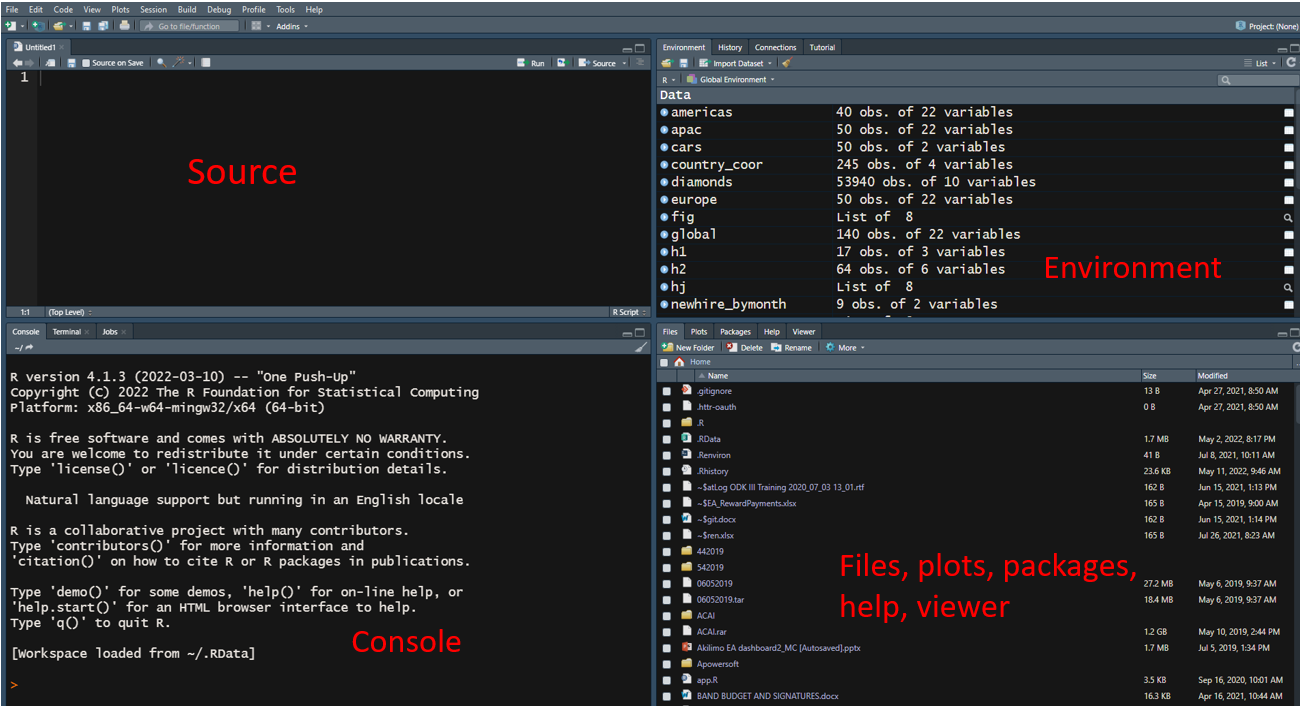
In order to use ***Rstudio*** your ***R*** version should be 3.0.1 or higher. The most recent version of ***R*** can be downloaded from [CRAN](https://cran.r-project.org/), the Comprehensive R Archive Network.

If you already have ***R*** and/or ***RStudio*** installed, let’s get started!

## R pane layout:

Below is the ***R*** pane layout with four components: Upper left corner: Source pane, Bottom left corner: Console pane, Upper right corner: Environment, Bottom right corner: files, plots, packages, help.

If you wish to customize this layout just go to “Tools” –> “Global options” –> “Pane layout”.



### Before we start…

## Set the Working directory

1. Create a folder called ACAI\_R\_training

2. Navigate to the file path and copy the path

Now let’s direct R to where we want our inputs and outputs to be. To do this we will use the function setwd().

A function is a set of modules of code that are put together to accomplish a specific task. R has a many in-built functions that you can use but you can also create functions of your own to avoid repetitive work. Functions take in a data structure processes it in the code, and then returns a result.

1. Use setwd() to set the directory: Paste the path in between the parentheses:setwd("D:\Training Materials\ACAI\_R\_training")

In R, a single backslash is an escape character, and using it for directory paths will always produce an error.

1. Specify directory paths correctly in R, using a forward slash: setwd("D:/Training Materials/ACAI\_R\_training")

If you need to determine which directory your R session is using as its current working directory, type getwd().

This is an example of a function that doesn’t take arguments.

## Install relevant packages

R packages are free libraries of code, documentation and sample data developed by R’s active user community. For every problem you encounter, chances are someone already wrote code to solve it and you can download the package for free to apply to your problem.

Three of the most popular repositories for Rpackages are:

• CRAN: the official repository, a network of serversmaintained by the R community around the world.

• Bioconductor: a topic specific repository, open source forbioinformatics.

• Github: although this is not R specific, Github is probably the most popular repository for open source projects

Let’s use ‘tidyverse’, a collection of the most commonly used package, to provide an example of how to install packages:

1. Install it from CRAN with: install.packages("tidyverse") or from GitHub: devtools::install\_github("tidyverse/tidyverse").OR in the Files pane of RStudio: Click on the “Packages” tab, Click on “Install”, Type the name of the package under “Packages (separate multiple with space or comma):” Click “Install”

For GitHub you may need to install the{devtools} package. You can do this from CRAN with install.packages("devtools") then load the devtools package using library(devtools).

In most cases just use install\_github(“author/package\_name”) as shown in the example above.

1. Load it for use in R with: [library("tidyverse")](https://rdrr.io/r/base/library.html).
2. Get help on it with: package? tidyverse and [help(package = " tidyverse")](https://rdrr.io/pkg/x/man).

#### Packages we will use in this exercise:

tidyverse (comes with our packages of interest: readr, lubridate, tidyr, dplyr, ggplot2)

RColorBrewer

# Session 2: Building our analysis dataset

## Prepare data

### Read the data in R

We will use {readr} which is a part of the core {tidyverse} and supports seven file formats with seven read\_ functions. For our case these two apply: read\_csv() and write\_csv().

read\_csv("data/dataVAL\_FR\_PO.csv")

An alternative is to use read.csv which is inbuilt in base R and reads the data as a dataframe.

read.csv ("data/dataVAL\_FR\_PO.csv", header=T, na.strings=c("","NA"))

Now let’s assign this data frame to a variable so that it is stored in R’s memory:

dataVAL\_FR\_PO <- read\_csv("data/dataVAL\_FR\_PO.csv")

### Let’s EXPLORE our data:

The function str() is one of the functions you will need to apply before you work on any new dataset. It gives you more insight into the structure of your dataset. Type str(dataVAL\_FR\_PO) in the console and see the result. You will get the total number of variables, observations, data type of each variable, the first observations etc.

The function head() allows us to see the first 6 rows by default. You can also specify the number of rows: head(dataVAL\_FR\_PO, 10)

The function summary() shows the data type of each variable and a few other attributes that are useful for numeric attributes. It also displays min, 1st quartile, median, mean, 3rd quartile and max values: summary(dataVAL\_FR\_PO)

R’s **basic data types** are:

* Decimal values are called **numerics** in R
* Whole numbers like are called **integers**. Integers are also numerics.
* Boolean values e.g TRUE or FALSE are called **logical**.
* Text/string values are called **characters**.

## Selection of data frame elements

Since we have specific parameters of interest for this exercise (Identifier information, location data, selected agronomic events and dates & yield data), let’s proceed to subset our data. e.g.

Now if we knew the exact column number it would be easier to use square brackets in base R, as in example 1 below:

### Example 1: Selecting Columns by Index

See below how we used indexes to extract columns and rows:

dataVAL\_FR\_PO [ ,1:8] we use the sequence to select all the metadata columns of our dataset

dataVAL\_FR\_PO [2,2] to select the value at the second row and second column

dataVAL\_FR\_PO [1:2,6:10] to select rows 1, 2 and columns 6,7,8,9,10

OR using dplyr:

dataVAL\_FR\_PO %>%

select(c(2, 5, 6))

### Example 2: Selecting Specific Columns by their Names

But often times and as you have seen when exploring this dataset, we have numerous columns so it is hard to select by row/column. In our case, it is better to select the columns by name.

We use names(dataVAL\_FR\_PO) to identify the names of the columns, then put all the column names in a vector:

#subset/select

dataVAL\_FR\_PO [, c("ENID", "HHID", "geopoint-Latitude","geopoint-Longitude",

"plantingDetails/plantingDate", "plantingDetails/variety", "harvest/intHarvestDate\_CON", "harvest/effHarvestDate\_CON", "harvest/tuberizedMarketableRootsFW\_CON", harvest/intHarvestDate\_SSR", "harvest/effHarvestDate\_SSR", harvest/tuberizedMarketableRootsFW\_SSR")]

#OR use dplyr

yield\_datFR <- dataVAL\_FR\_PO %>%

select(c("ENID", "HHID", "geopoint-Latitude","geopoint-Longitude",

"plantingDetails/plantingDate", "plantingDetails/variety", "harvest/intHarvestDate\_CON","harvest/effHarvestDate\_CON", "harvest/tuberizedMarketableRootsFW\_CON", "harvest/intHarvestDate\_SSR", "harvest/effHarvestDate\_SSR", "harvest/tuberizedMarketableRootsFW\_SSR"))

## Assigning new field names

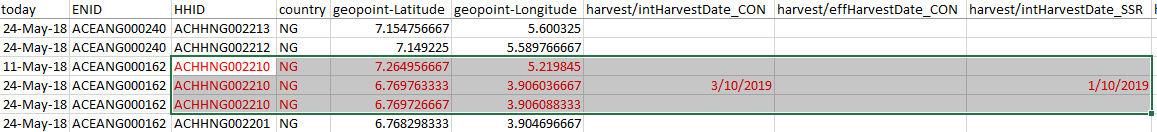
As you may have noticed, the variable names are quite lengthy and may be hard to remember. So, we are going to change that by assigning new names using the function colnames()

colnames(yield\_datFR) = c("ENID", "HHID", "Latitude", "Longitude", "plantingDate", "variety", "intHarvestDate\_CON", "HarvestDate\_CON", "yield\_CON","intHarvestDate\_SSR", "HarvestDate\_SSR","yield\_SSR")

## Data cleaning

### Repetitive coordinates

When you explore this data in order to get unique identifiers, you will realize that in some cases there were several records from the same HH because the gps coordinates were captured multiple times, probably due to poor visibility:



In order to clean this data, we round off the gps coordinates to drop repetitive ones:

yield\_datFR$Latitude <- round(yield\_datFR$Latitude,digits=3)#drop some repetitive gps coordinates

yield\_datFR$Longitude <- round(yield\_datFR$Longitude,digits=3)

### No yield data

Since our main concern is with data that has yield data, we now proceed to drop any records that have no yield.

Here we apply the ifelse statement:

yield\_datFR$yieldInfo <- ifelse( is.na(yield\_datFR$plantingDate) & is.na(yield\_datFR$variety) & is.na(yield\_datFR$HarvestDate\_SSR) & is.na(yield\_datFR$HarvestDate\_CON) & is.na(yield\_datFR$yield\_CON) & is.na(yield\_datFR$yield\_SSR), "NO YIELD", "YIELD") #drop rows without any yield information

yield\_datFR2 <- droplevels(yield\_datFR[!yield\_datFR$ yieldInfo == "NO YIELD", ])

# Session 3: Data Wrangling

### Working with dates

Explore the structure of dfFR, specifically the ‘harvest date’ variable for SSR and Control treatments. This is given as a character type when it takes in data in date format. In order to correct this, we are going to combine two lines of functions as below:

We will use the **lubridate package** to change the type and then get the difference between harvest dates and planting dates by creating a new variable using **mutate**.

Lubridate package makes it easier to work with dates and times.  Notice the ‘dmy’ in the code below. This stands for date, month and year. You need to identify the order in which these appear in your data and arrange them in the same order. Lubridate uses this order (dmy or ymd or mdy) as the name of the function it uses to parse your dates.

The order of our dates is this: 10-Jun-19 so dmy suffices.

dfFR <- dfFR %>% mutate(

diffeffH = lubridate::dmy(dfFR$HarvestDate\_SSR) - lubridate::dmy(dfFR$plantingDate),

diffeffC = lubridate::dmy(dfFR$HarvestDate\_CON) - lubridate::dmy(dfFR$plantingDate)

)

### Merging data

For the next step in getting graphics out of this data, we need to know from which region the data came from. Since this was not in our original questionnaire, we are lucky that we have a separate registration dataset that has this information linked to the HHID and ENID. We use the merge function to get this information:

dfFR2 <- droplevels(unique(merge(dsENHH, dfFR, by=c("HHID", "ENID"))))

### Dates AGAIN! ... and FOR loops

In order to generate bar graphs of yield by month we need to convert the dates into month abbreviations. We achieve this by splitting the dates to get the month number then we use the function month.abb to convert this number into the month abbreviation:

#For loops

For Loop is a type of control statement used to iterate over items of a sequence.

**For loop Syntax:**

for (value in sequence)

{

statement

}

#work on dates to get m, d, y

onpldate <- NULL

for(hids in unique(dfFR2$HHID)){

hdata <- dfFR2[dfFR2$HHID == hids, ]

hdata$pmnth <- lubridate::month(dmy(hdata$plantingDate))

hdata$pday <- lubridate::day(dmy(hdata$plantingDate))

hdata$pyear <- lubridate::year(dmy(hdata$plantingDate))

hdata$rtdmy <- ifelse(is.na(hdata$pmnth), NA, paste(hdata$pyear, hdata$pmnth, hdata$pday, sep = "/"))

hdata$rtdmy <- as.Date(hdata$rtdmy)

hdata <- hdata[order(hdata$rtdmy, decreasing = TRUE), ][1,]

onpldate <- rbind(onpldate, hdata)

}

head(onpldate)

str(onpldate)

onpldate$f0m <- month.abb[onpldate$pmnth]

### Reshaping your data frame

Often times you will want to reshape your data frame from wide to ‘longer’layouts or vice versa. In our case for plotting we need to have a column that gives us the yield (by treatment type) alongside the value in a separate column. Therefore, since our data is is in wide format, we use the tidyr function “gather” to reshape our data into long format, favorable for plotting.

#change the layout of data

#gather

pldate <- onpldate %>% tidyr::gather(harvest, yield, yield\_SSR, yield\_CON)

pldate$harvest <- as.character(pldate$harvest)

pldate$harvest[pldate$harvest == "yield\_SSR"] <- "SSR"

pldate$harvest[pldate$harvest == "yield\_CON"] <- "Control"

hvstpldate <- pldate %>% tidyr::gather(diff, days, diffeffH, diffeffC)

head(hvstpldate)

hvstpldate2 <- hvstpldate %>% tidyr::gather(hvstype, daterc, HarvestDate\_SSR, HarvestDate\_CON)

head(hvstpldate2)

### Factors & levels

If you asses the regions/states where the data was collected, you will see that these are abbreviated. In order to plot by region, we need to provide correct full names. When you check str(rti) you see that region\_state is listed as character. In order to rename the levels, we have to convert this to factor and then recode the factor levels:

rti$region3 <- as.factor(rti$region\_state)

levels(rti$region\_state) #asses the levels

levels(rti$region\_state) <- c("Kwara", "Ogun", "Ondo", "Oyoo") #rename the levels

# Session 4: Data visualisation

## Graphics

To create a plot, specify the data in the ggplot() function and add the required layers: variables, aesthetic elements and the type of plot:

ggplot(data) +

aes(x = var\_x, y = var\_y) +

geom\_x()

* data in ggplot() is the name of the data frame which contains the variables var\_x and var\_y.
* The + symbol used to indicate the different layers
* The layer aes() indicates the variables to be used in the plot and
* more generally, the aesthetic elements of the plot
* x in geom\_x() represents the type of plot: geom\_point(),
* geom\_line(), etc.

### Scatter plot

A scatter plot is used to visualize the relation between two quantitative variables.

Often used to visualize a potential correlation between the two variables

We create a scatter plot using geom\_point()

ggplot(rti) + # data

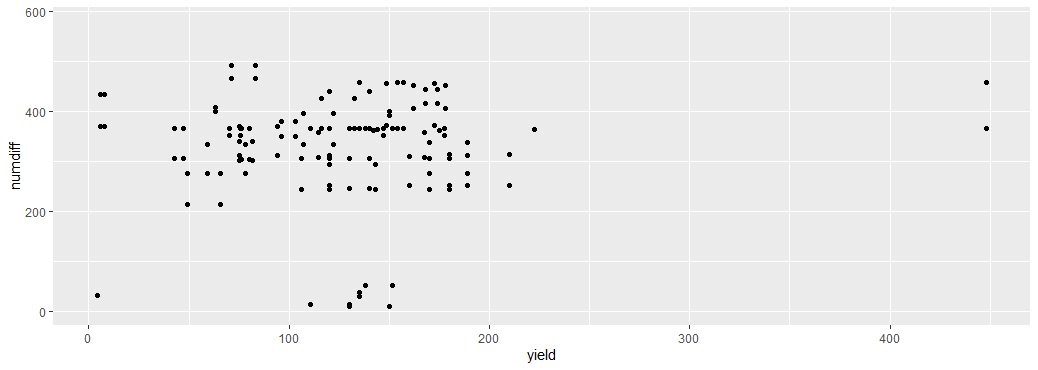
aes( x=yield, y=numdiff) + # variables

geom\_point() # type of plot

Look at the output. It is OK but we can do better: personalize the plot to

make it more informative. We can add a title, subtitle, caption and edit axis labels

with the labs() function:



It is possible to use mathematical equations instead of text strings. The quote() can be used for that. Read about the available options in ?plotmath

Save the “main” plot in an object, and add more layers. You can edit the alignment, the size and the shape of the title and subtitle via the theme() layer and the element\_text() function.

If the title or subtitle is long, divide it into multiple lines, use \n

**Let’s now see it in terms of the treatments applied:**

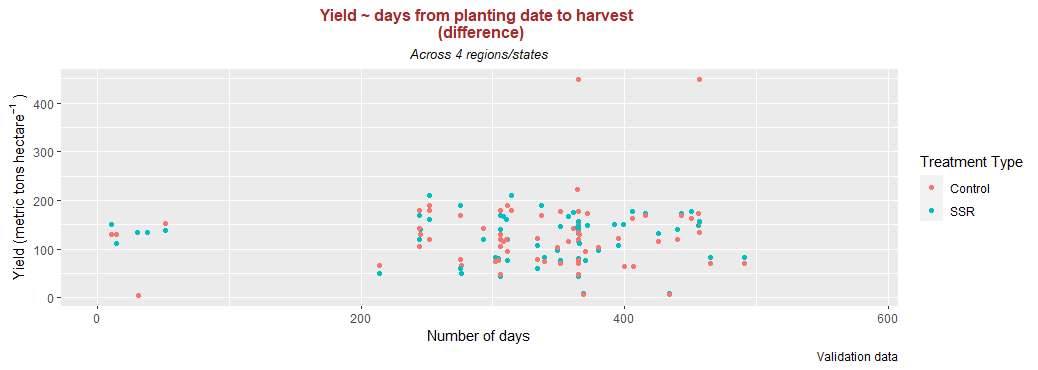
p <- ggplot(rti) + # data

aes( x=numdiff, y=yield, color=harvest) + # variables

geom\_point() # type of plot

**We can do better by changing the title of the legend: “Treatment Type ” instead of “harvest”**

**p <- p + scale\_color\_discrete (name ="Treatment Type")**



Let’s change the default color:

The {RColorBrewer} package makes it easy to quickly load sensible color palettes

install.packages("RColorBrewer")

library(RColorBrewer)

There are three types of palettes: sequential, diverging and qualitative.

Sequential palettes are suited to ordered data that progress from low to high.

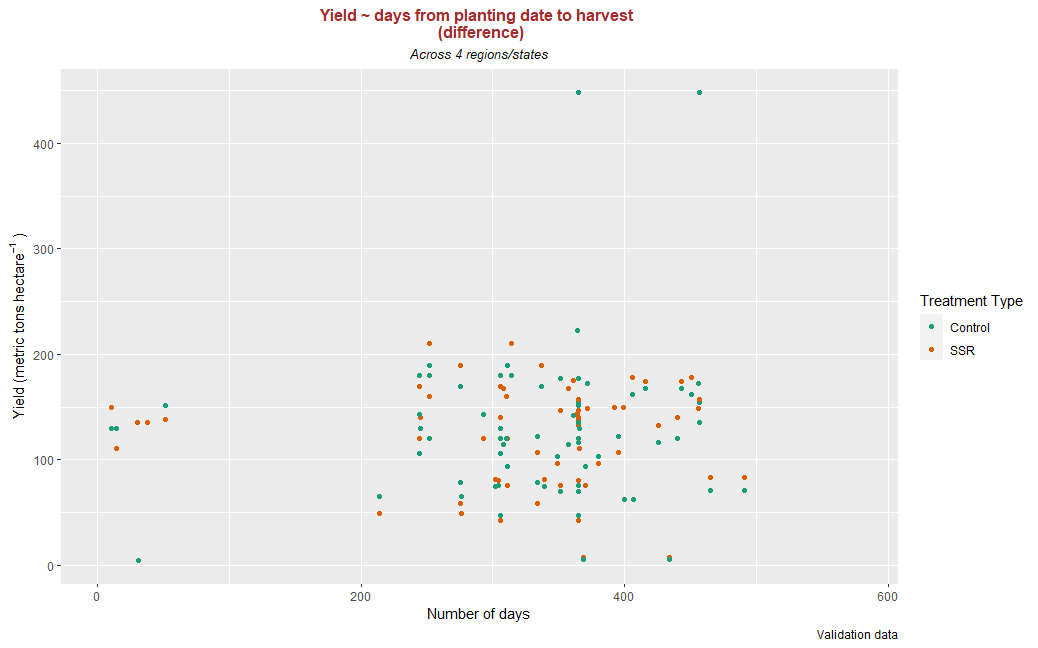
Diverging palettes are suited to centered data with extremes in either direction.

Qualitative palettes are suited to nominal or categorical data.

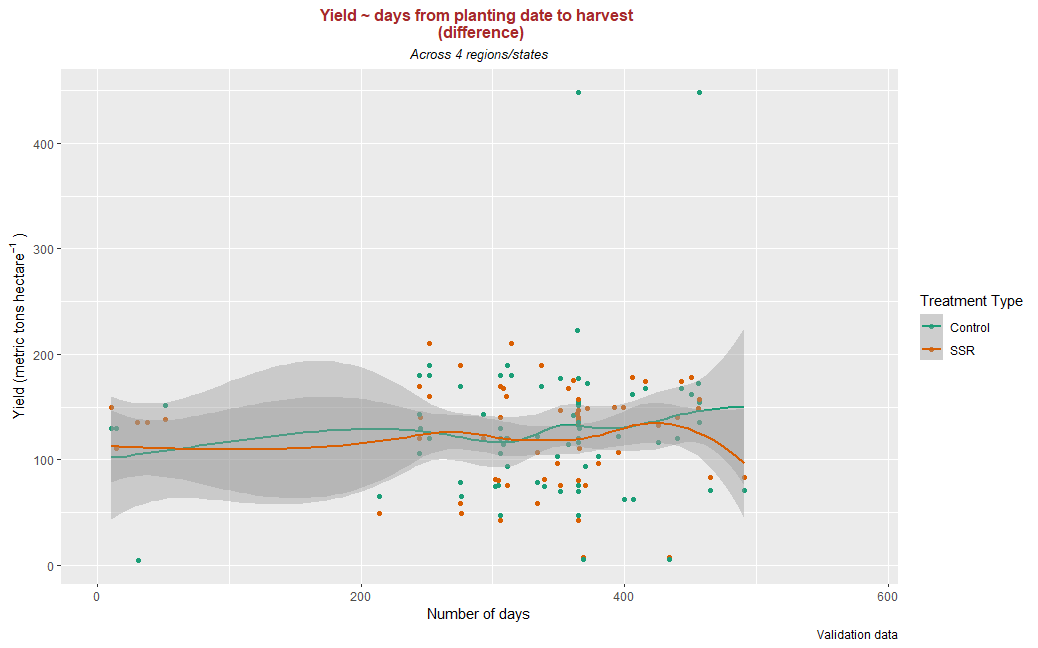
The display.brewer.all() will plot the available palettes

Let’s use the scale\_color\_brewer() to apply the relevant palette:

p <- p + scale\_color\_brewer(name ="Treatment Type" ,palette=“Dark2")



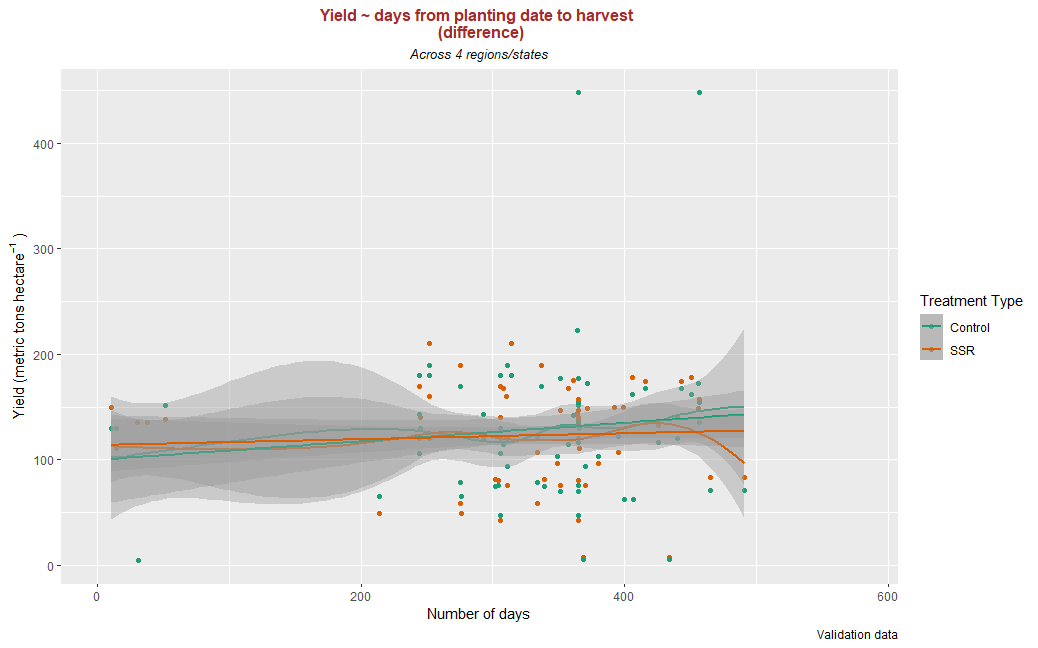
It is also possible to add a smooth line fitted to the data:



In case of simple linear regression, it is possible to display the regression line on the plot.

This can be done by adding method = lm in geom\_smooth(), as below:

p <- p + geom\_smooth(method = lm)



Several functions are available in the **{ggplot2}** package to change the theme of the plot.

The default theme is: **theme\_gray()**

Other common themes are:

* the black and white: **theme\_bw()**
* minimal: **theme\_minimal()**
* classic: **theme\_classic()**

#### Facetting

Instead of plotting all the *regions* in the same plot and using color to differentiate them, we can use facet\_grid() to divide the same graphic into several panels according to the values of one (*region*) or even two qualitative variables.

p <- ggplot(data = rti) + # data

aes( x=numdiff, y=yield, color=harvest) + # variables

geom\_point() + # type of plot

labs(title= "Yield ~ days from planting date to harvest \n (difference)", #Add labels

subtitle = "Across 4 regions/states",

caption = "Validation data",

x="Number of days",

y = quote("Yield (metric tons" ~ "hectare"^{-1} ~ ")"))+

scale\_color\_discrete(name="Treatment Type")+#Add mathematical equation

facet\_grid(. ~ region3)+ #facet by region

theme(

plot.title = element\_text(hjust = 0.5, size = 12, color = "brown",face = "bold"),

plot.subtitle = element\_text(hjust = 0.5, size = 10,color = "black",face = "italic"))

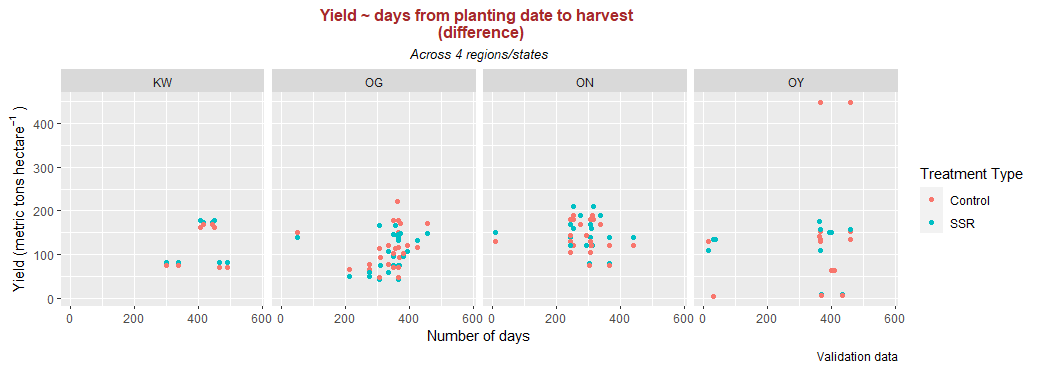
p

#facet by region

f <- p + facet\_grid(. ~ region3)

f + xlab("days between planting & harvest") +

ylab(quote("Yield (metric tons" ~ "hectare"^{-1} ~ ")"))



Line plots, particularly useful in time series, can be created by using geom\_line()

b <- ggplot(rti) + # data

aes( x=numdiff, y=yield, color=harvest) + # variables

geom\_point()+# type of plot

geom\_line()

b = b + labs(title= "Yield ~ days from planting date to harvest \n (difference)", #Add labels

subtitle = "Across 4 regions/states",

caption = "Validation data",

x="Number of days",

b = quote("Yield (metric tons" ~ "hectare"^{-1} ~ ")")) #Add mathematical equation

b <- b + theme(

plot.title = element\_text(hjust = 0.5, size = 12, color = "brown",face = "bold"),

plot.subtitle = element\_text(hjust = 0.5, size = 10,color = "black",face = "italic")

)

### Boxplot

A boxplot can be plotted using geom\_boxplot()

• A boxplot graphically represents the distribution of a quantitative variable by visually displaying five common location summary (minimum, median, first/third quartiles and

maximum) and any observation that was classified as a suspected outlier using the interquartile range (IQR) criterion.

• All observations above

o above q0.75 + 1.5⋅IQR

o below q0.25−1.5⋅IQR

• where q0.25 and q0.75 correspond to first and third quartile respectively are considered as potential outliers by R.

• The minimum and maximum in the boxplot are represented without these suspected outliers

Use this code to get the boxplot below:

p <- ggplot(rti) + # data

aes( x=numdiff, y=yield, color=harvest) + # variables

geom\_boxplot() # type of plot

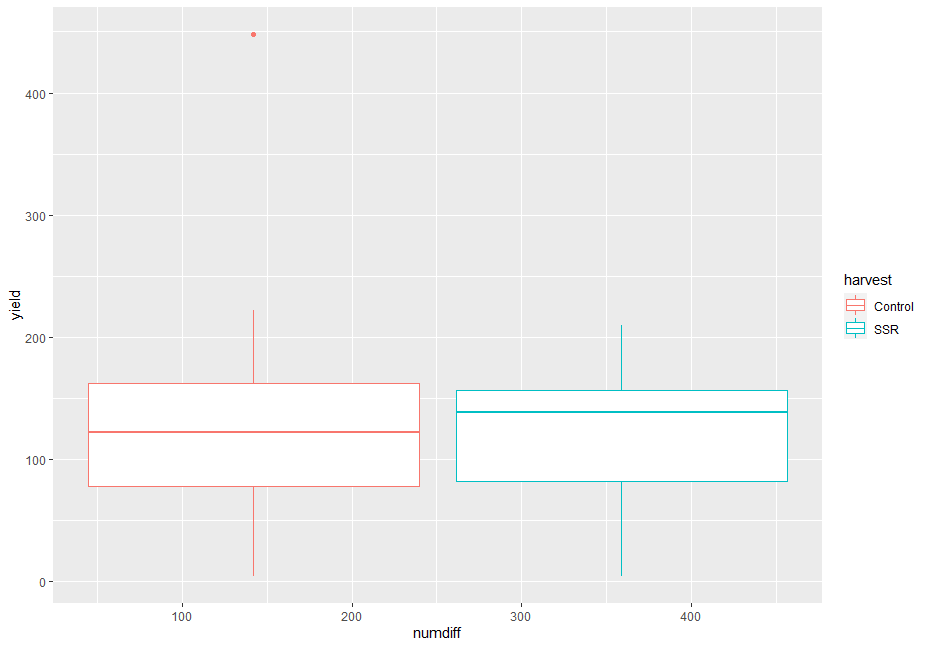
p <- p + labs(x = "",

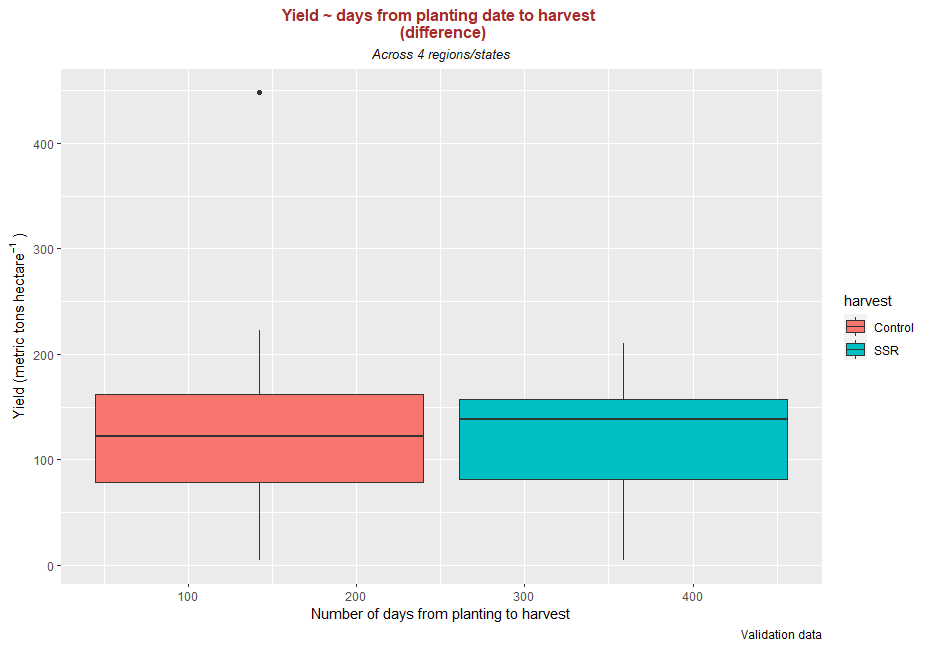
y = "Yield (metric tons/hectare)")

p + theme\_classic()

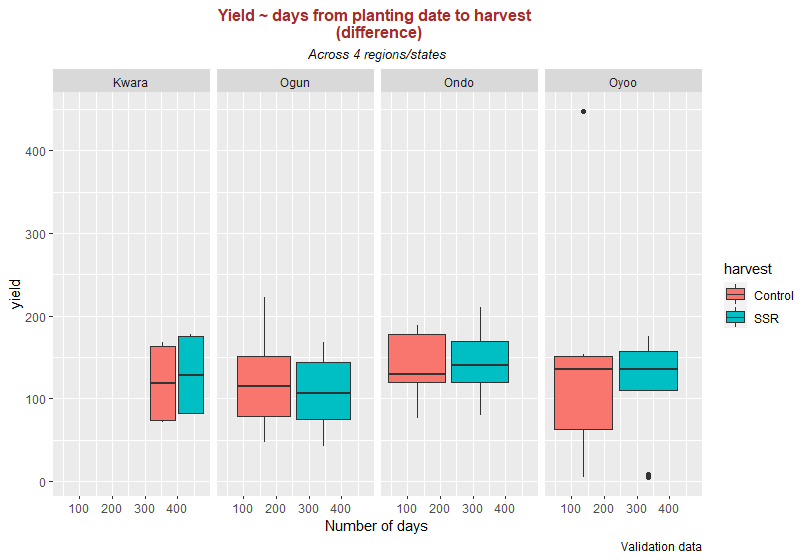
p

Boxplot by factor using colors:





Boxplot by factor: divided into several panels



### Barplot

A barplot can be plotted using geom\_bar(). A barplot is a tool to visualize the distribution of a qualitative variable.

So, when you asses our data, what is(are) the qualitative(s) variable(s) we can use for barplot?

p <- ggplot(rti) +

aes(x = variety) +

geom\_bar()

p <- p + labs(x = "Variety", y = "Count")

p + theme\_classic()

The levels of variety are: "other\_improved" "other\_local" "TME419" "TMS30572"

"TMS98\_0581". We can recode these using mutate() and case\_when()

rti <- rti %>% mutate(

variety\_new = case\_when(

variety %in% "other\_local" ~ "Other local",

variety %in% "other\_improved" ~ "Other improved",

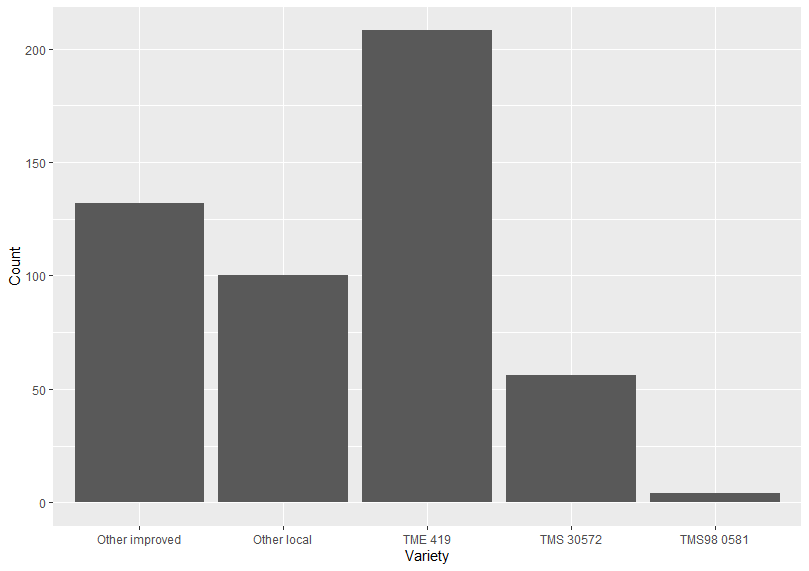
variety %in% "TME419" ~ "TME 419",

variety %in% "TMS30572" ~ "TMS 30572",

variety %in% "TMS98\_0581" ~ "TMS98 0581"

)

)



The output is rather dull. Let’s asses the geom\_bar() syntax to make it more presentable:

geom\_bar(stat, fill, color, width)

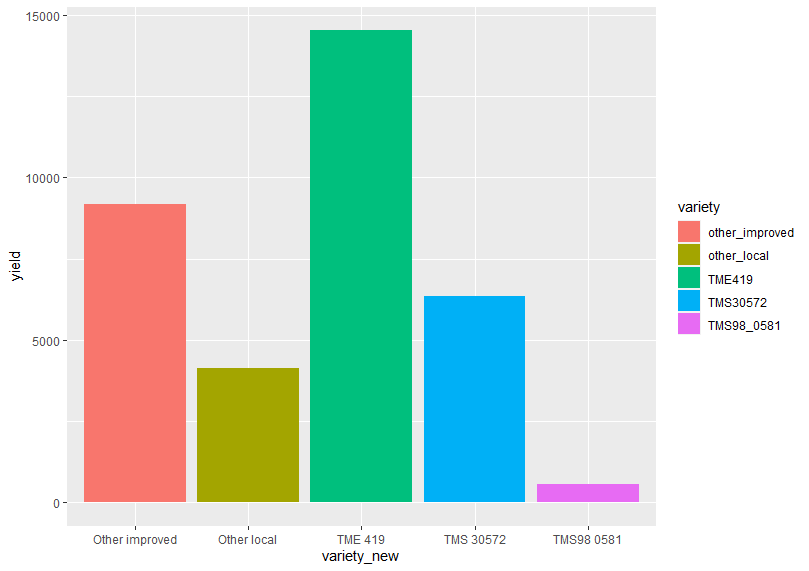
Parameters :

* stat : Set the stat parameter to identify the mode.
* fill : Represents color inside the bars.
* color : Represents color of outlines of the bars.
* width : Represents width of the bars.

f <-ggplot(data=rti, aes(x=variety\_new, y=yield,fill=variety))+

geom\_bar(stat="identity")

f



Now let’s see a barplot with two qualitative variables

p <- ggplot(rti) +

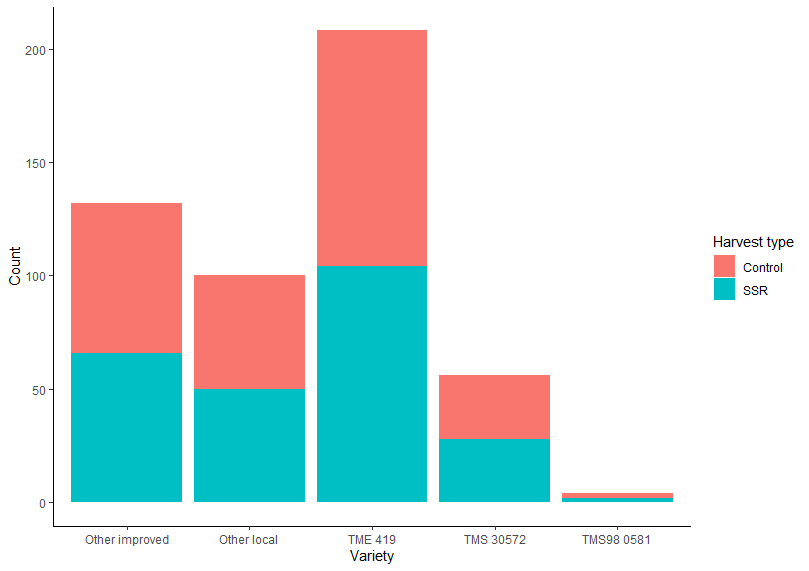
aes(x = variety\_new, fill=factor(harvest)) +

geom\_bar()

p <- p + labs(x = "Variety", y = "Count")

p <- p + scale\_fill\_discrete(name="Year")

p + theme\_classic()



Let’s plot by region/state:

#### Example 1

p <- ggplot(rti) +

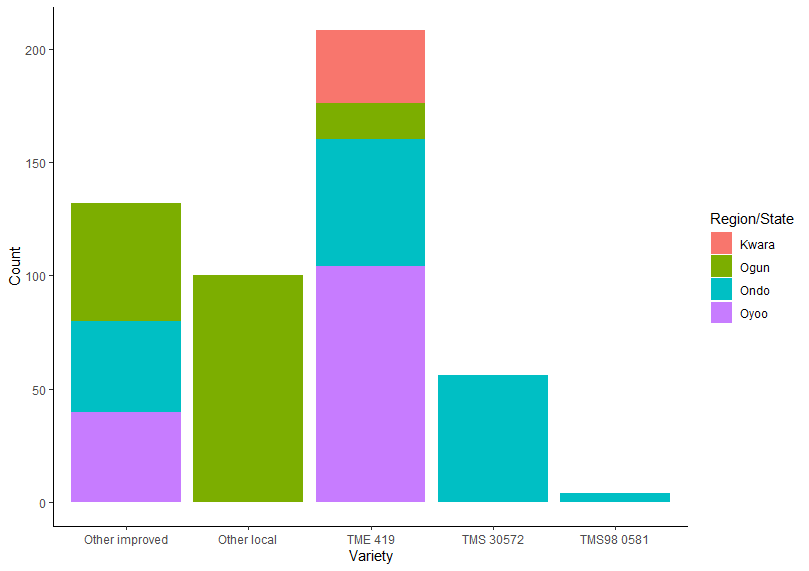
aes(x = variety\_new, fill=factor(region3)) +

geom\_bar()

p <- p + labs(x = "Variety", y = "Count")

p <- p + scale\_fill\_discrete(name="Region/State")

p + theme\_classic()



#### Example 2

p <- ggplot(rti) +

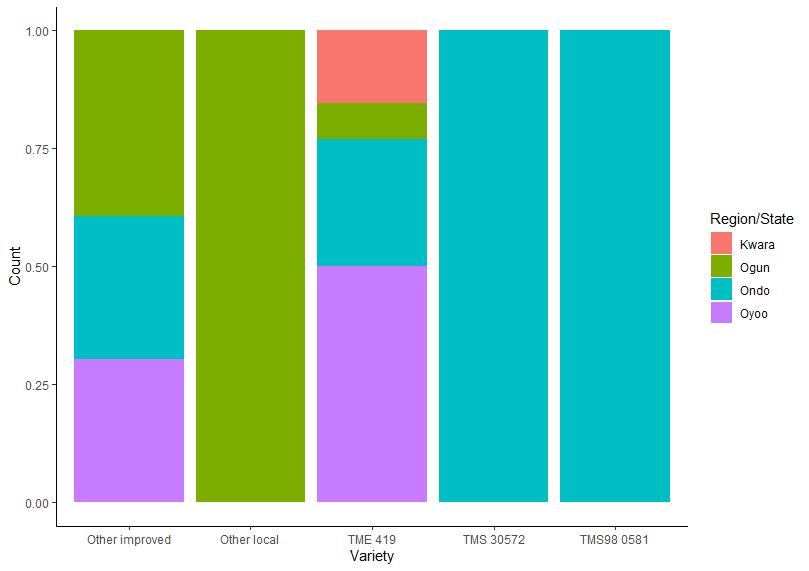
aes(x = variety\_new, fill=factor(region3 )) +

geom\_bar(position="fill")

p <- p + labs(x = "Variety", y = "Count")

p <- p + scale\_fill\_discrete(name="Region/State")

p + theme\_classic()



Here’s a barplotwith two qualitative variables:

#position dodge

p <- ggplot(rti) +

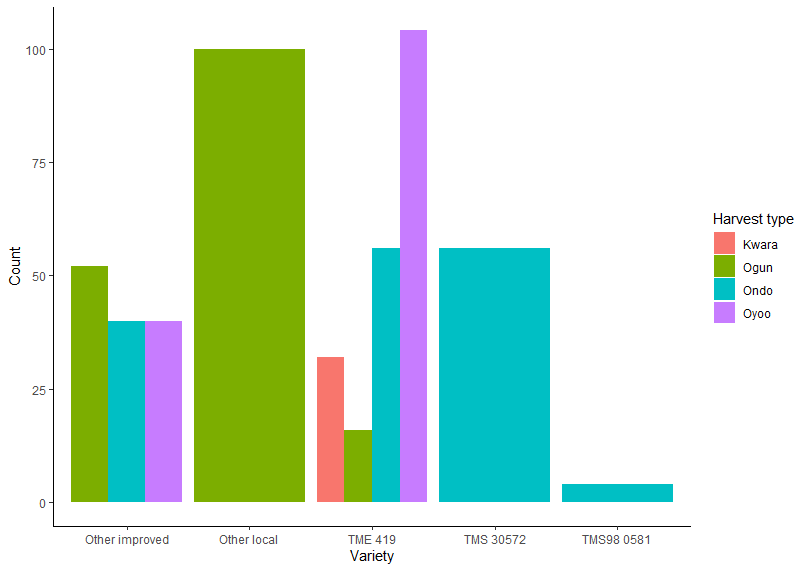
aes(x = variety\_new, fill=factor(harvest )) +

geom\_bar(position="dodge")

p <- p + labs(x = "Variety", y = "Count")

p <- p + scale\_fill\_discrete(name="Harvest type")

p + theme\_classic()



Apply manual colors:

p <- ggplot(rti) +

aes(x = variety\_new, fill=factor(harvest )) +

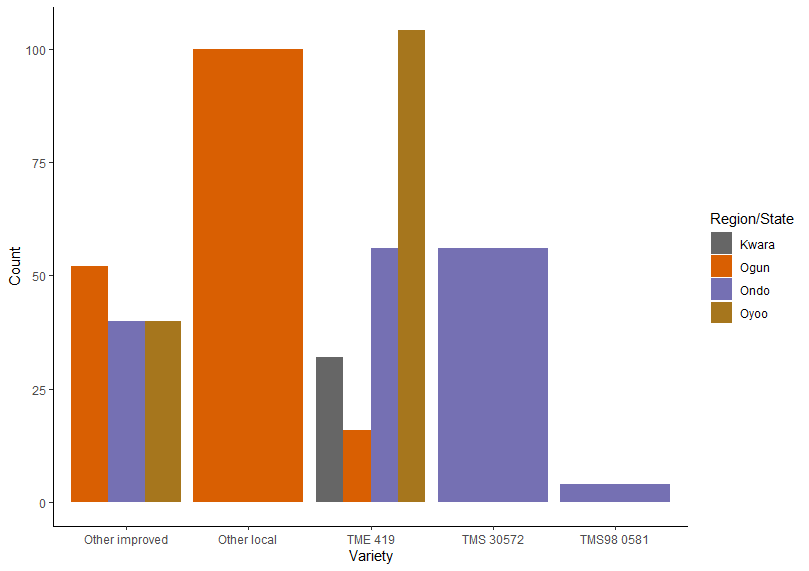
geom\_bar(position="dodge")

p <- p + labs(x = "Variety", y = "Count")

p <- p + scale\_fill\_manual(values=c("#666666", "#D95F02", "#7570B3", "#A6761D"),

name="Region/State")

p + theme\_classic()



### Saving your graphic output:

ggsave("varHarvest.pdf", p, width = 10)

TASK

See example below and try to write a code that would give this output:

