Experiment 1

Aim:

Installation of R and R Studio

Theory:

R is a programming language and free software environment for statistical computing and graphics supported by the R Foundation for Statistical Computing.The R language is widely used among statisticians and data miners for developing statistical software and data analysis. Polls, data mining surveys, and studies of scholarly literature databases show substantial increases in popularity; as of February 2020, R ranks 13th in the TIOBE index, a measure of popularity of programming languages.

A GNU package, the official R software environment is written primarily in C, Fortran, and R itself (thus, it is partially self-hosting) and is freely available under the GNU General Public License. Pre-compiled executables are provided for various operating systems. Although R has a command line interface, there are several third-party graphical user interfaces, such as RStudio, an integrated development environment, and Jupyter, a notebook interface.

Procedure:

The process of installing R on different operating systems is different, we shall be seeing it for Ubuntu and WIndows in our study.

- **Windows**
  - Installing R-base
    1. Navigate to https://cran.r-project.org/bin/windows/
    2. Click on base, and select the link to download the base package.
    3. Double click and install by following the instructions on the Screen.
  - Installing R-studio
    a. Go to www.rstudio.com and click on the "Download RStudio" button.
    b. Click on "Download RStudio Desktop."
    c. Click on the version recommended for your system, or the latest Windows version, and save the executable file. Run the .exe file and follow the

installation instructions.

- **Linux**
  - Installing R-base

i. Open a terminal by ALT+CTRL+T (or anything Identical)

ii. Type in `sudo apt update`

iii. Then so as to install R, `sudo apt install r-base`

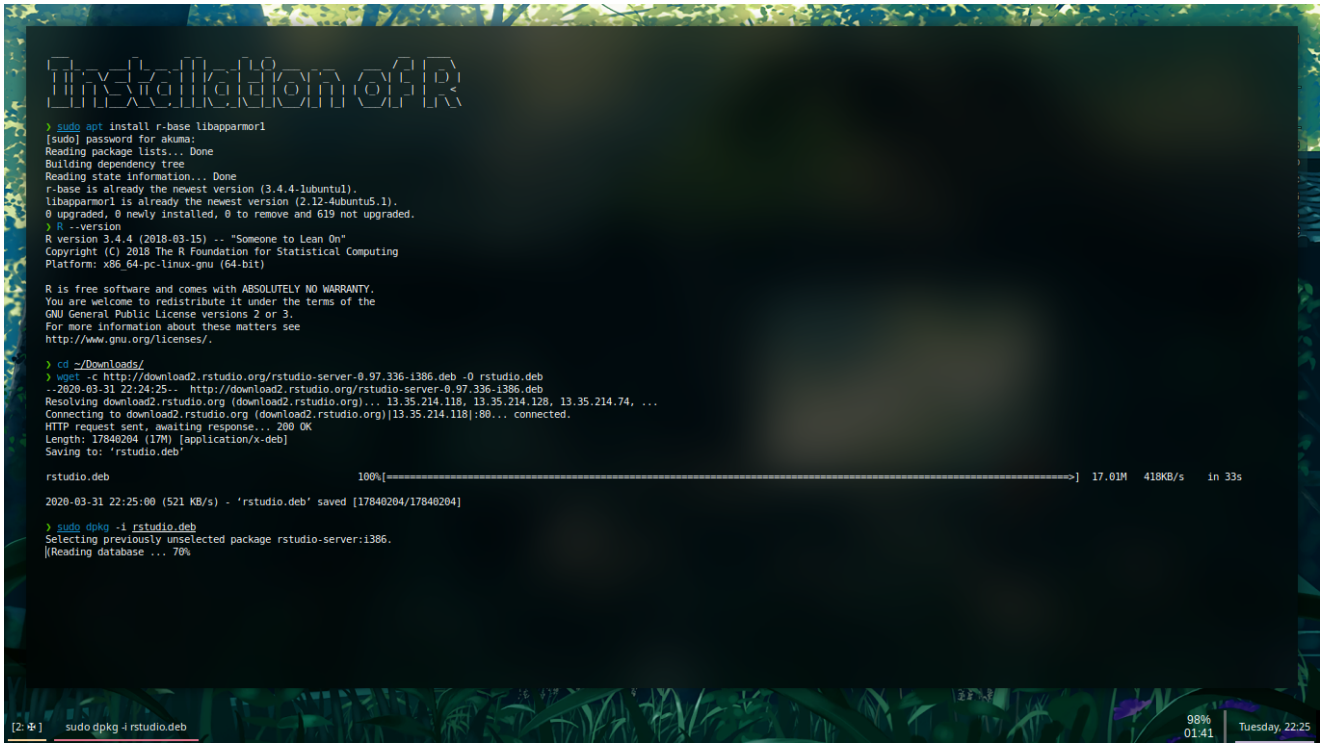
  - Installing R-studio

a. Go to terminal and type in `wget https://download2.rstudio.org/server/trusty/amd64/rstudio-server-1.2.5033-amd64.deb`

b. Then `sudo dpkg -i rstudio-server-1.2.5033.amd64.deb`

c. Wait for installation to finish.

Screenshots :



Conclusion:

Henceforth, we have successfully installed r-base and rstudio.

Experiment 2

Aim:

Basic functionality of R - Variables, basic arithmetics and help commands.

Theory:

Every programming language has a syntax, definitely needs one as long as its not esoteric programming language like brainfuck, pikachu. Variables are basically the placeholders for user inputs and data, in R they have a very significant role as it allows us to keep an entire dataset (and refer to it) using variables.

Basic arithmetics and help commands enable us to get acquainted with the language itself.

Arithmetics and Basics:

The console is where you enter commands for **R** to execute *interactively*, meaning that the command is executed and the result is displayed as soon as you hit the `Enter` key. For example, at the command prompt `>`, type in `2+2` and hit `Enter`; you will see

```
2+2
```

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

The results from calculations can be stored in (*assigned to*) variables. For example:

```
a <- 2+2
```

R automatically creates the variable `a` and stores the result (4) in it, but by default doesn't print anything. To ask **R** to print the value, just type the variable name by itself

```
a
```

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

`print(a)` also works to print the value of a variable. By default, a variable created this way is a *vector* (an ordered list), and it is *numeric* because we gave **R** a number rather than (e.g.) a character string like `"pxqr"`; in this case `a` is a numeric vector of length 1;

You could also type

```
a <- 2+2; a
```

using a semicolon to put two or more commands on a single line. Conversely, you can break lines *anywhere that* **R** *can tell you haven't finished your command* and **R** will give you a "continuation" prompt (`+`) to let you know that it doesn't think you're finished yet: try typing

```
a <- 3*(4+5)
```

```
x <- 5
y <- 2
z1 <- x*yz2 <- x/yz3 <- x^yz1; z2; z3
```

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

```
## [1] 2.5
```

```
## [1] 25
```

Help system in R

R has a help system, although it is generally better for reminding you about syntax or details, and for giving cross-references, than for answering basic "how do I …?" questions.

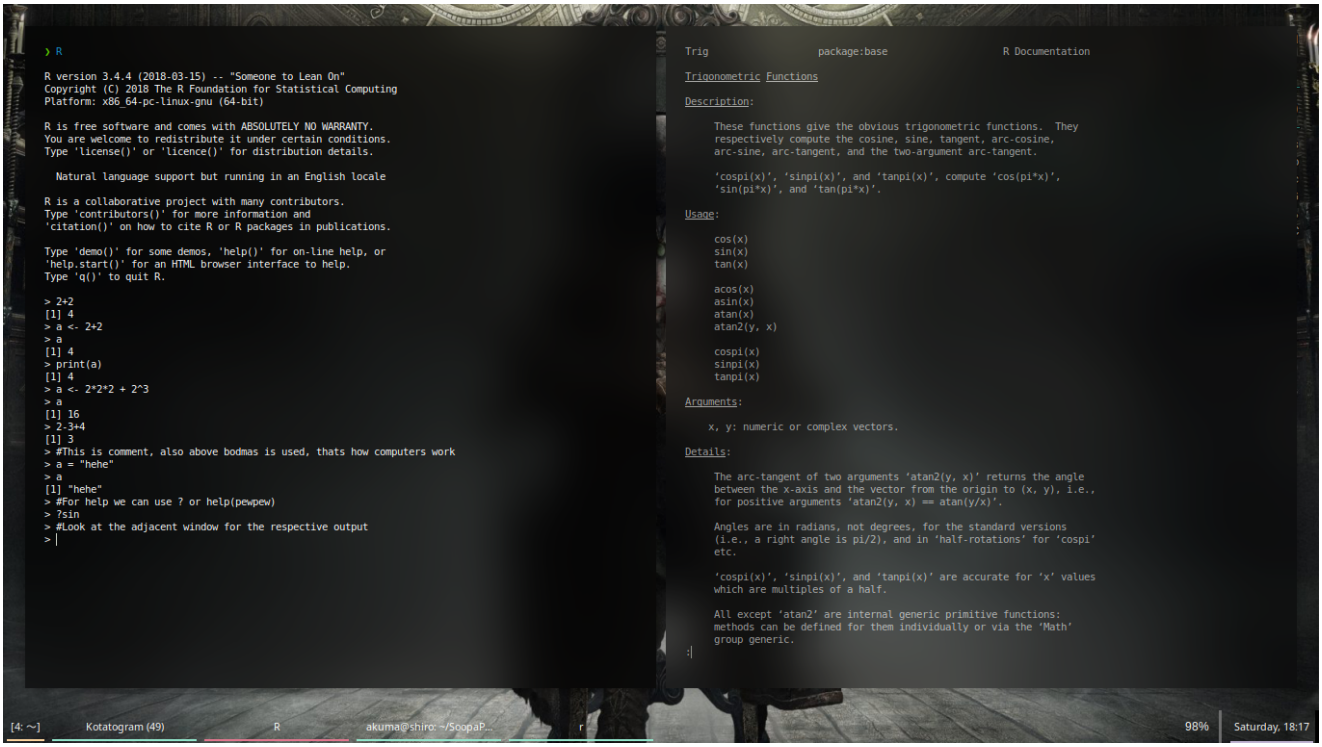
- You can get help on an **R** function named `foo` by entering

```
?foo
```

or

```
help(foo)
```

Screenshots:



Conclusion:

Henceforth, we have successfully studied the variables,assignments and help command in R.

Experiment 3

Aim:

Basic datatypes and data structures in R - vectors, metrics, list and dataframes.

Theory:

Variables can contain values of specific types within R. The six **data types** that R uses include:

- `"numeric"` for any numerical value
- `"character"` for text values, denoted by using quotes (“”) around value
- `"integer"` for integer numbers (e.g., `2L`, the `L` indicates to R that it’s an integer)
- `"logical"` for `TRUE` and `FALSE` (the Boolean data type)
- `"complex"` to represent complex numbers with real and imaginary parts (e.g., `1+4i`) and that’s all we’re going to say about them
- `"raw"` that we won’t discuss further

The table below provides examples of each of the commonly used data types:

Data Type	Examples
Numeric:	1, 1.5, 20, pi
Character:	“anytext”, “5”, “TRUE”
Integer:	2L, 500L, -17L
Logical:	TRUE, FALSE, T, F

Data Structures

We know that variables are like buckets, and so far we have seen that bucket filled with a single value. Even when `number` was created, the result of the mathematical operation was a single value. **Variables can store more than just a single value, they can store a multitude of different data structures.** These include, but are not limited to, vectors (`c`), factors (`factor`), matrices (`matrix`), data frames (`data.frame`) and lists (`list`).

Vectors

A vector is the most common and basic data structure in R, and is pretty much the workhorse of R. It’s basically just a collection of values, mainly either numbers, characters or logical values.

Note: All values in a vector must be of the same data type. If you try to create a vector with more than a single data type, R will try to coerce it into a single data type.

Each element of this vector contains a single numeric value, and three values will be combined together into a vector using `c()` (the combine function). All of the values are put within the parentheses and separated with a comma.

```
glengths <- c(4.6, 3000, 50000)
glengths
```

Similarly,

```
species <- c("ecoli","human","corn")
```

Matrix

A `matrix` in R is a collection of vectors of **same length and identical datatype**. Vectors can be combined as columns in the matrix or by row, to create a 2-dimensional structure.

Matrices are used commonly as part of the mathematical machinery of statistics. They are usually of numeric datatype and used in computational algorithms to serve as a checkpoint. For example, if input data is not of identical data type (numeric, character, etc.), the `matrix()` function will throw an error and stop any downstream code execution.

Data Frame

A `data.frame` is the *de facto* data structure for most tabular data and what we use for statistics and plotting. A `data.frame` is similar to a matrix in that it's a collection of vectors of the **same length** and each vector represents a column. However, in a dataframe **each vector can be of a different data type** (e.g., characters, integers, factors).

We can create a dataframe by bringing **vectors** together to **form the columns**. We do this using the `data.frame()` function, and giving the function the different vectors we would like to bind together. *This function will only work for vectors of the same length.*

```
df <- data.frame(species, glengths)
```

Beware of `data.frame()`'s default behaviour which turns **character vectors into factors**. Print your data frame to the console:

```
df
```

Lists

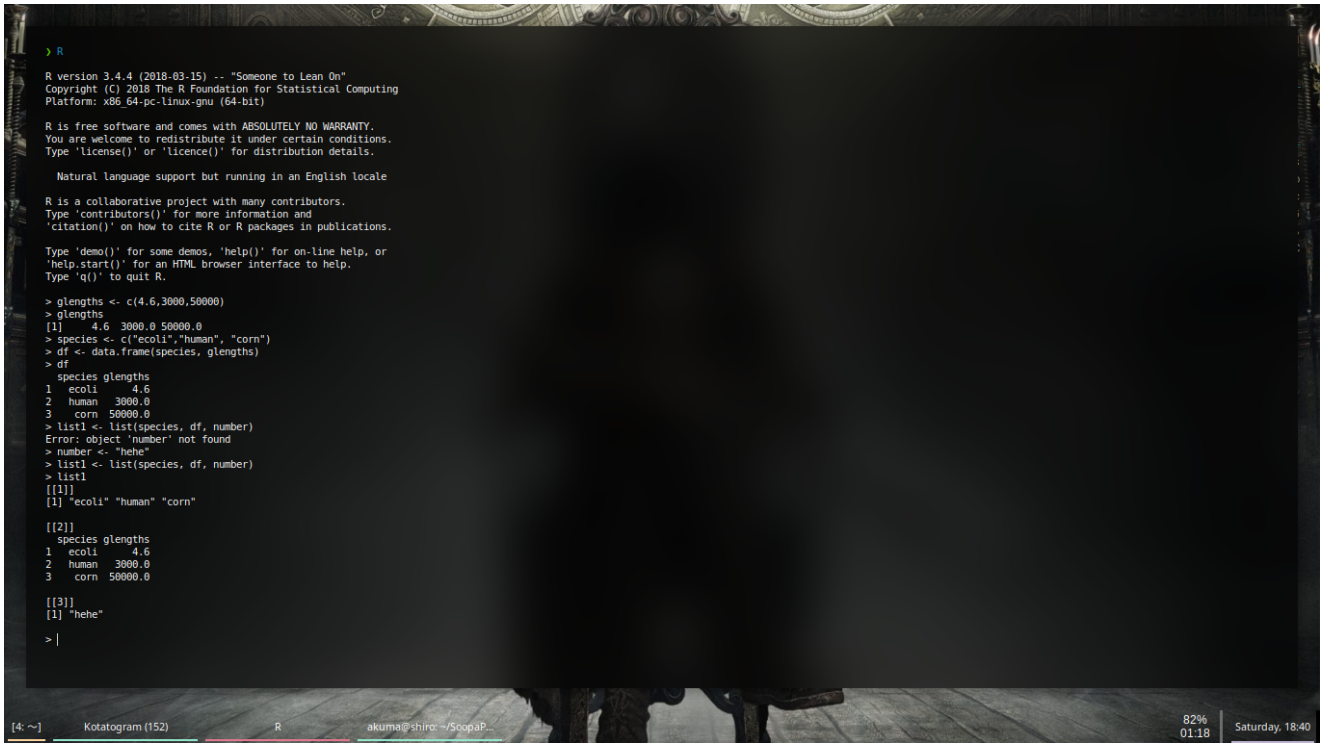
Lists are a data structure in R that can be perhaps a bit daunting at first, but soon become amazingly useful. A list is a data structure that can hold any number of any types of other data structures.

```
list1 <- list(species, df, number)
```

Print out the list to screen to take a look at the components:

```
list1

[[1]]
[1] "ecoli" "human" "corn"

[[2]]
  species glengths
1   ecoli      4.6
2   human   3000.0
3    corn  50000.0

[[3]]
[1] 5
```

Screenshot:



Conclusion:

Henceforth, we have successfully understood R datatypes and datastructures.

Experiment 4

Aim

To study programming constructs in R - loops and conditional executions.

Theory

In R programming, while loops are used to loop until a specific condition is met.

Syntax of while loop

```
while (test_expression){statement}
```

Here, `test_expression` is evaluated and the body of the loop is entered if the result is `TRUE`.

The statements inside the loop are executed and the flow returns to evaluate the `test_expression` again.

This is repeated each time until `test_expression` evaluates to `FALSE`, in which case, the loop exits.

A for loop is used to iterate over a vector in R programming.

Syntax of for loop

```
for (val in sequence){statement}
```

Here, `sequence` is a vector and `val` takes on each of its value during the loop. In each iteration, `statement` is evaluated.

Syntax of ifelse() function

```
ifelse(test_expression, x, y)
```

Here, `test_expression` must be a logical vector (or an object that can be coerced to logical). The return value is a vector with the same length as `test_expression`.

This returned vector has element from `x` if the corresponding value of `test_expression` is `TRUE` or from `y` if the corresponding value of `test_expression` is `FALSE`.

This is to say, the `i-th` element of result will be `x[i]` if `test_expression[i]` is `TRUE` else it will take the value of `y[i]`.

The vectors `x` and `y` are recycled whenever necessary.

```
> a = c(5,7,2,9)> ifelse(a %% 2 == 0,"even","odd")[1] "odd"  "odd"  "even"
"odd"
```

Decision making is an important part of programming. This can be achieved in R programming using the conditional `if...else` statement.

R if statement

The syntax of if statement is:

```
if (test_expression) {statement}
```

If the `test_expression` is `TRUE`, the statement gets executed. But if it's `FALSE`, nothing happens.

Here, `test_expression` can be a logical or numeric vector, but only the first element is taken into consideration.

In the case of numeric vector, zero is taken as `FALSE`, rest as `TRUE`.

if…else statement

The syntax of if…else statement is:

```
if (test_expression) {statement1} else {statement2}
```

The else part is optional and is only evaluated if `test_expression` is `FALSE`.

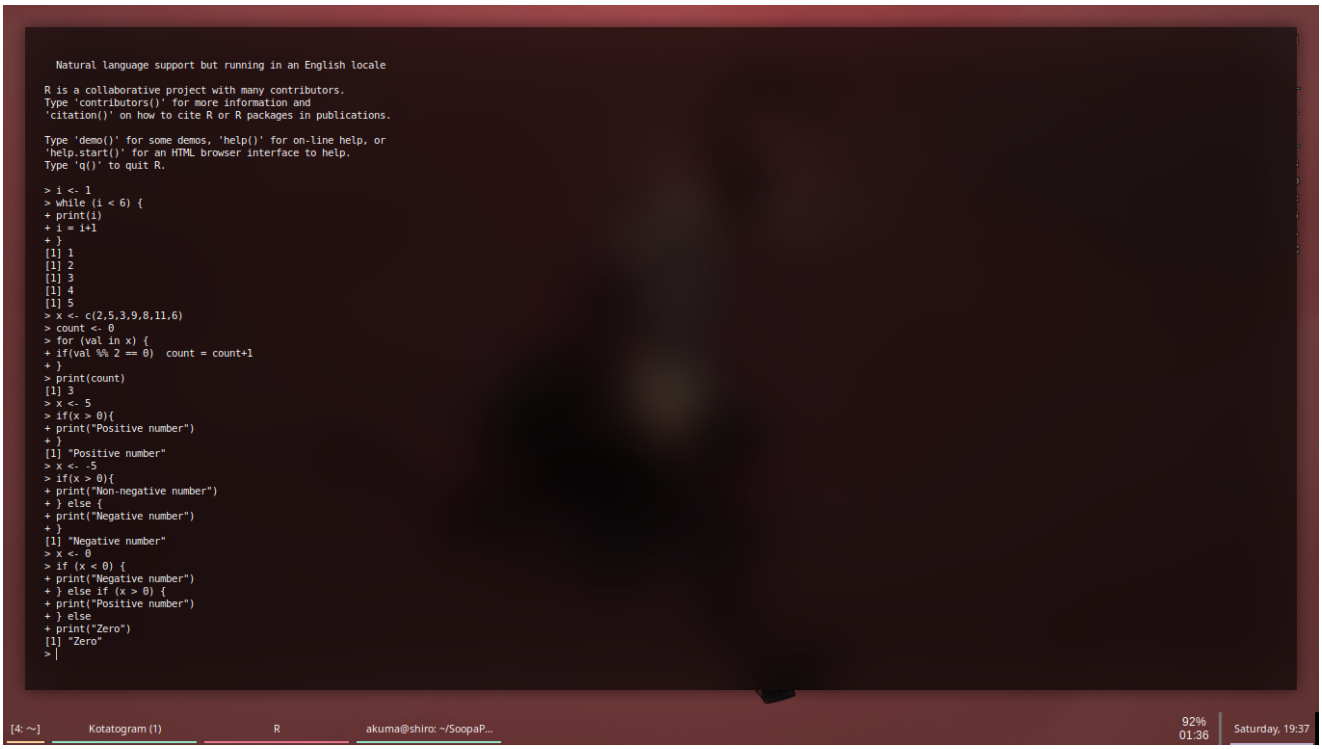
if…else Ladder

The if…else ladder (if…else…if) statement allows you execute a block of code among more than 2 alternatives

The syntax of if…else statement is:

```
if ( test_expression1) {statement1} else if ( test_expression2) {statement2}
else if ( test_expression3) {statement3} else {statement4}
```

Screenshot:



Conclusion

Henceforth, we have successfully studied conditionals and looping in R.

Experiment 5

Aim:

Tables with labels in R - multiple response variables excel function translations

Theory:

`expss` computes and displays tables with support for ‘SPSS’-style labels, multiple / nested banners, weights, multiple-response variables and significance testing. There are facilities for nice output of tables in ‘knitr’, R notebooks, ‘Shiny’ and ‘Jupyter’ notebooks. Proper methods for labelled variables add value labels support to base R functions and to some functions from other packages. Additionally, the package offers useful functions for data processing in marketing research / social surveys - popular data transformation functions from ‘SPSS’ Statistics (‘RECODE’, ‘COUNT’, ‘COMPUTE’, ‘DO IF’, etc.) and ‘Excel’ (‘COUNTIF’, ‘VLOOKUP’, etc.). Package is intended to help people to move data processing from ‘Excel’/‘SPSS’ to R. See examples below. You can get help about any function by typing `?function_name` in the R console.

Installation

`expss` is on CRAN, so for installation you can print in the console `install.packages("expss")`.

```
library(expss)data(mtcars)mtcars = apply_labels(mtcars,
mpg = "Miles/(US) gallon",                      cyl = "Number of cylinders",
disp = "Displacement (cu.in.)",                      hp = "Gross
horsepower",                       drat = "Rear axle ratio",
wt = "Weight (1000 lbs)",                     qsec = "1/4 mile time",
vs = "Engine",                       vs = c("V-engine" = 0,
"Straight engine" = 1),                          am = "Transmission",
am = c("Automatic" = 0,                                "Manual"=1),
gear = "Number of forward gears",                      carb = "Number of
carburetors"
)
```

For quick cross-tabulation there are `fre` and `cro` family of function. For simplicity we demonstrate here only `cro_cpct` which caluclates column percent. Documentation for other functions, such as `cro_cases` for counts, `cro_rpct` for row percent, `cro_tpct` for table percent and `cro_fun` for custom summary functions can be seen by typing `?cro` and `?cro_fun` in the console.

```
# 'cro' examples
# just simple crosstabulation, similar to base R 'table' function
cro(mtcars$am, mtcars$vs)
```

```
# Table column % with multiple banners
cro_cpct(mtcars$cyl, list(total(), mtcars$am, mtcars$vs))
```

	#Total		Transmission		Engine
			Automatic	Manual	
---	---	---	---	---	---
Number of cylinders					
4	34.4		15.8	61.5	
6	21.9		21.1	23.1	
8	43.8		63.2	15.4	
#Total cases	32		19	13	

```
# or, the same result with another notation
mtcars %>% calc_cro_cpct(cyl, list(total(), am, vs))
```

	#Total		Transmission		Engine
			Automatic	Manual	
---	---	---	---	---	---
Number of cylinders					
4	34.4		15.8	61.5	
6	21.9		21.1	23.1	
8	43.8		63.2	15.4	
#Total cases	32		19	13	

```
# Table with nested banners (column %).
mtcars %>% calc_cro_cpct(cyl, list(total(), am %nest% vs))
```

We have more sophisticated interface for table construction with `magrittr` piping. Table construction consists of at least of three functions chained with pipe operator: `%>%`. At first

we need to specify variables for which statistics will be computed with `tab_cells`. Secondary, we calculate statistics with one of the `tab_stat_*` functions. And last, we finalize table creation with `tab_pivot`, e. g.: `dataset %>% tab_cells(variable) %>% tab_stat_cases() %>% tab_pivot()`. After that we can optionally sort table with `tab_sort_asc`, drop empty rows/columns with `drop_rc` and transpose with `tab_transpose`. Resulting table is just a `data.frame` so we can use usual R operations on it. Detailed documentation for table creation can be seen via `?tables`. For significance testing see `?significance`. Generally, tables automatically translated to HTML for output in knitr or Jupyter notebooks. However, if we want HTML output in the R notebooks or in the RStudio viewer we need to set options for that: `expss_output_rnotebook()` or `expss_output_viewer()`.

Excel functions translation guide

Let us consider Excel toy table:

	A	B	C
1	2	15	50
2	1	70	80
3	3	30	40
4	2	30	40

Code for creating the same table in R:

```
library(expss)w = text_to_columns("        a  b  c        2 15 50        1
70 80        3 30 40        2 30 40")
```

`w` is the name of our table.

IF

Excel: `IF(B1>60, 1, 0)`

R: Here we create new column with name `d` with results. `ifelse` function is from base R not from 'expss' package but included here for completeness.

```
w$d = ifelse(w$b>60, 1, 0)
```

If we need to use multiple transformations it is often convenient to use `compute` function. Inside `compute` we can put arbitrary number of the statements:

```
w = compute(w, {    d = ifelse(b>60, 1, 0)    e = 42
    abc_sum = sum_row(a, b, c)    abc_mean = mean_row(a, b, c)})
```

COUNTIF

Count 1's in the entire dataset.

Excel: `COUNTIF(A1:C4, 1)`

R:

```
count_if(1, w)
```

or

```
calculate(w, count_if(1, a, b, c))
```

Count values greater than 1 in each row of the dataset.

Excel: `COUNTIF(A1:C1, ">1")`

R:

```
w$d = count_row_if(gt(1), w)
```

or

```
w = compute(w, {    d = count_row_if(gt(1), a, b, c) })
```

Count values less than or equal to 1 in column A of the dataset.

Excel: `COUNTIF(A1:A4, "<=1")`

R:

```
count_col_if(le(1), w$a)
```

Table of criteria:

Excel	R
<1	lt(1)
<=1	le(1)
<>1	ne(1)
=1	eq(1)
>=1	ge(1)
>1	gt(1)

SUM/AVERAGE

Sum all values in the dataset.

Excel: `SUM(A1:C4)`

R:

```
sum(w, na.rm = TRUE)
```

Calculate average of each row of the dataset.

Excel: `AVERAGE(A1:C1)`

R:

```
w$d = mean_row(w)
```

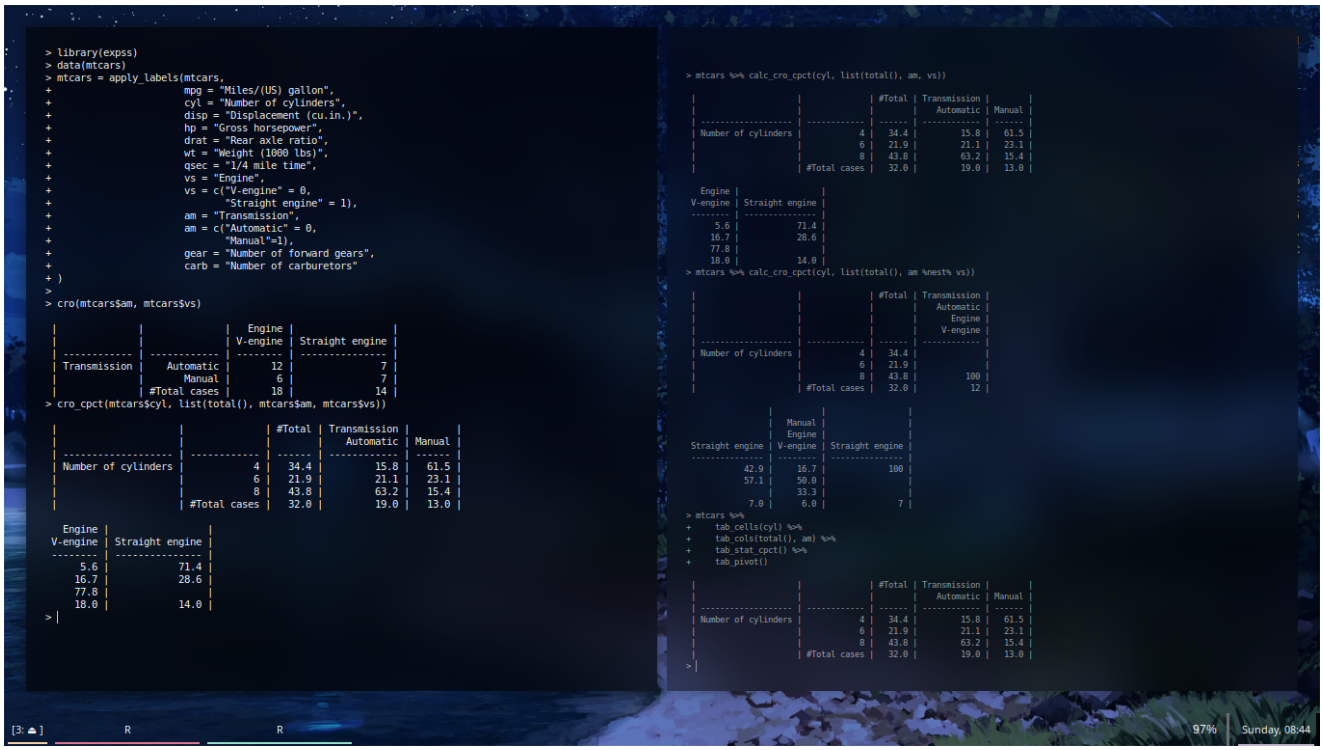
or

```
w = compute(w, {    d = mean_row(a, b, c) })
```

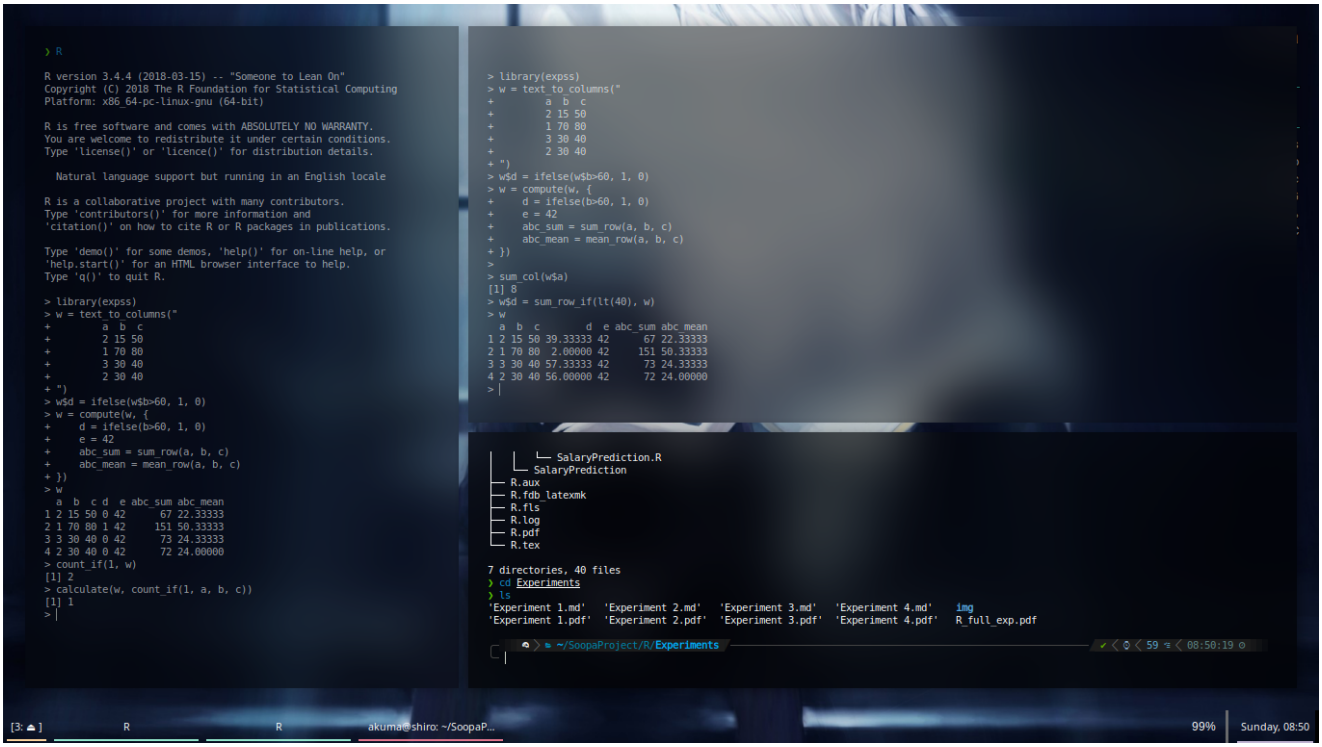
Screenshots:

Installation of `expss`

Cross-tabulation Example:



Excel functions Example:



Conclusion:

Henceforth, we have successfully seen cross-tabulation and excel functions.

Experiment 6

Aim:

Working with large datasets with dplyr and data.table

Theory:

dplyr is the next iteration of plyr, focussed on tools for working with data frames (hence the `d` in the name). It has three main goals:

- Identify the most important data manipulation tools needed for data analysis and make them easy to use from R.
- Provide blazing fast performance for in-memory data by writing key pieces in C++.
- Use the same interface to work with data no matter where it's stored, whether in a data frame, a data table or database.

One can install:

- the latest released version from CRAN with

```
install.packages("dplyr")
```

- the latest development version from github with

```
if (packageVersion("devtools") < 1.6) {
install.packages("devtools")}devtools::install_github("hadley/lazyeval")
devtools::install_github("hadley/dplyr")
```

`tbls`

The key object in dplyr is a *tbl*, a representation of a tabular data structure. Currently `dplyr` supports:

- data frames
- [data tables]
- SQLite
- PostgreSQL
- MySQL/MariaDB

- Bigquery
- MonetDB
- data cubes with arrays (partial implementation)

```
library(dplyr) # for functions
library(hflights) # for data
head(hflights)

# Coerce to data table
hflights_dt <- tbl_dt(hflights)

# Caches data in local SQLite db
hflights_db1 <- tbl(hflights_sqlite(), "hflights")

# Caches data in local postgres db
hflights_db2 <- tbl(hflights_postgres(), "hflights")
Each tbl also comes in a grouped variant which allows one to easily perform
operations "by group":

carriers_df  <- group_by(hflights, UniqueCarrier)
carriers_dt  <- group_by(hflights_dt, UniqueCarrier)
carriers_db1 <- group_by(hflights_db1, UniqueCarrier)
carriers_db2 <- group_by(hflights_db2, UniqueCarrier)
```

`dplyr` can be effectively used to

- Pick observations by their values (`filter()`).
- Reorder the rows (`arrange()`).
- Pick variables by their names (`select()`).
- Create new variables with functions of existing variables (`mutate()`).
- Collapse many values down to a single summary (`summarise()`).

These can all be used in conjunction with `group_by()` which changes the scope of each function from operating on the entire dataset to operating on it group-by-group. These six functions provide the verbs for a language of data manipulation.

All verbs work similarly:

1. The first argument is a data frame.
2. The subsequent arguments describe what to do with the data frame, using the variable names (without quotes).
3. The result is a new data frame.

Together these properties make it easy to chain together multiple simple steps to achieve a complex result. Let’s dive in and see how these verbs work.

Filter rows with `filter()`

`filter()` allows one to subset observations based on their values. The first argument is the name of the data frame. The second and subsequent arguments are the expressions that filter the data frame. For example, we can select all flights on January 1st with:

```
filter(flights, month == 1, day == 1)
#> # A tibble: 842 x 19
#>    year month   day dep_time sched_dep_time dep_delay arr_time
sched_arr_time
#>   <int> <int> <int>    <int>          <int>     <dbl>    <int>
<int>
#> 1  2013     1     1      517            515         2      830
819
#> 2  2013     1     1      533            529         4      850
830
#> 3  2013     1     1      542            540         2      923
850
#> 4  2013     1     1      544            545        -1     1004
1022
#> 5  2013     1     1      554            600        -6      812
837
#> 6  2013     1     1      554            558        -4      740
728
#> # … with 836 more rows, and 11 more variables: arr_delay <dbl>, carrier
<chr>,
#> #   flight <int>, tailnum <chr>, origin <chr>, dest <chr>, air_time
<dbl>,
#> #   distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

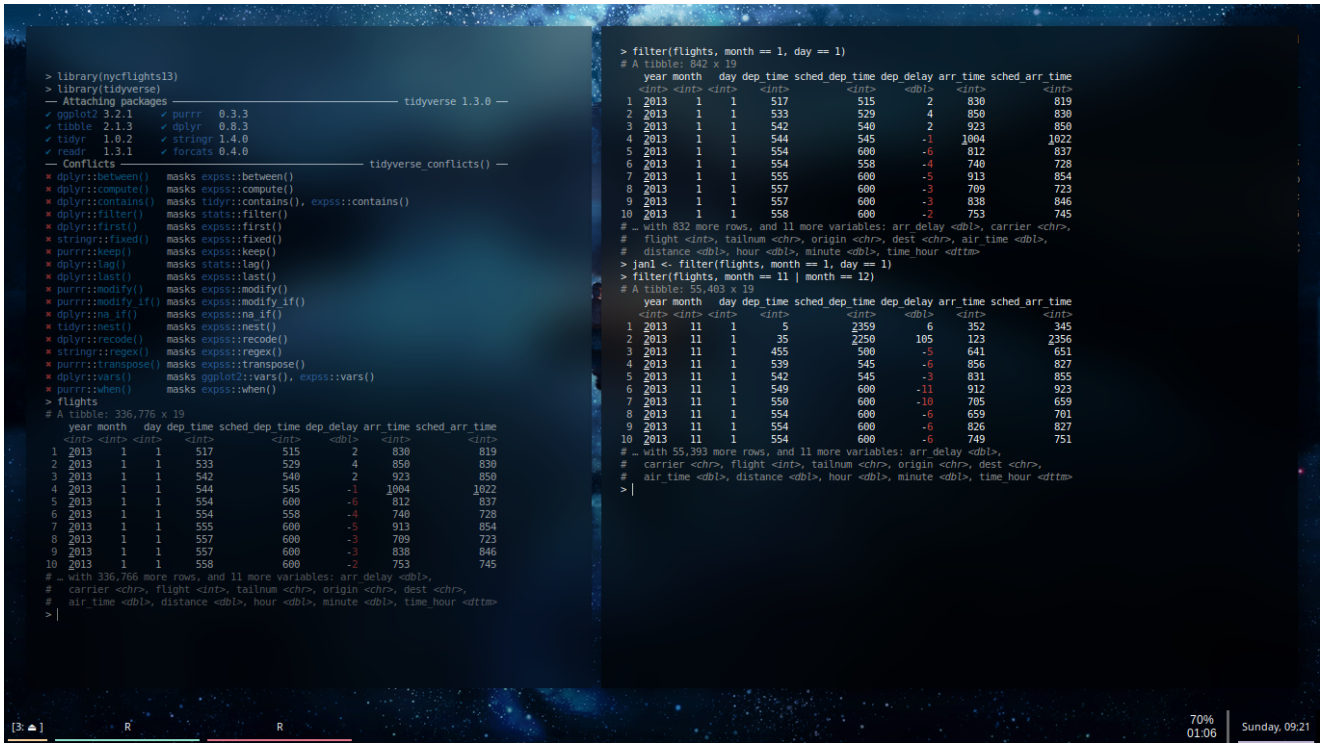
Comparisons

To use filtering effectively, one have to know how to select the observations that one want using the comparison operators. R provides the standard suite: `>`, `>=`, `<`, `<=`, `!=` (not equal), and `==` (equal).

Logical operators

Multiple arguments to `filter()` are combined with “and”: every expression must be true in order for a row to be included in the output. For other types of combinations, one’ll need to use Boolean operators themself: `&` is “and”, `|` is “or”, and `!` is “not”

```
filter(flights, month == 11 | month == 12)
```



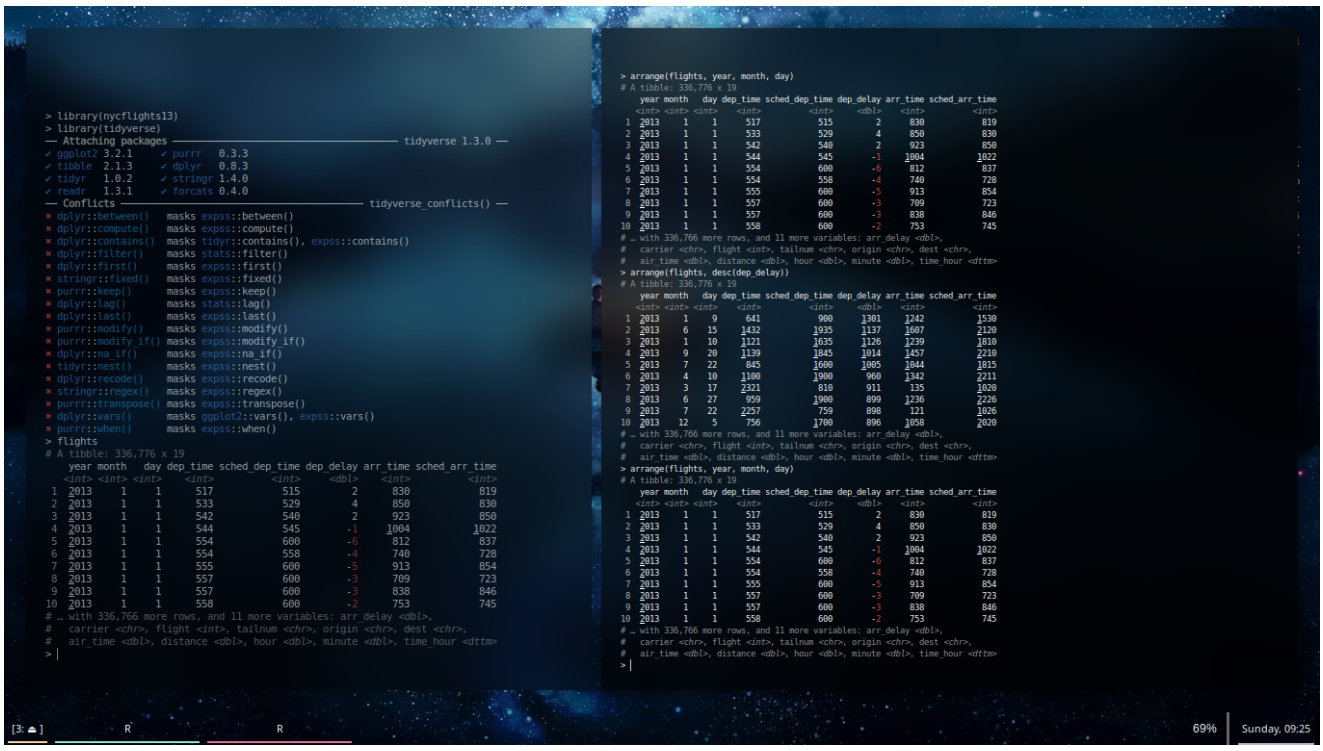
Arrange rows with `arrange()`

`arrange()` works similarly to `filter()` except that instead of selecting rows, it changes their order. It takes a data frame and a set of column names (or more complicated expressions) to order by. If one provide more than one column name, each additional column will be used to break ties in the values of preceding columns:

```
arrange(flights, year, month, day)
#> # A tibble: 336,776 x 19
#>    year month   day dep_time sched_dep_time dep_delay arr_time
sched_arr_time
#>   <int> <int> <int>    <int>          <int>     <dbl>    <int>
<int>
#> 1  2013     1     1      517            515         2      830
819
#> 2  2013     1     1      533            529         4      850
830
#> 3  2013     1     1      542            540         2      923
850
#> 4  2013     1     1      544            545        -1     1004
1022
#> 5  2013     1     1      554            600        -6      812
837
#> 6  2013     1     1      554            558        -4      740
728
#> # … with 3.368e+05 more rows, and 11 more variables: arr_delay <dbl>,
#> #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
#> #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour
<dttm>
```

Use `desc()` to re-order by a column in descending order:

```
arrange(flights, desc(dep_delay))
#> # A tibble: 336,776 x 19
#>    year month   day dep_time sched_dep_time dep_delay arr_time
sched_arr_time
#>   <int> <int> <int>    <int>          <int>     <dbl>    <int>
<int>
#> 1  2013     1     9      641            900      1301     1242
1530
#> 2  2013     6    15     1432           1935      1137     1607
2120
#> 3  2013     1    10     1121           1635      1126     1239
1810
#> 4  2013     9    20     1139           1845      1014     1457
2210
#> 5  2013     7    22      845           1600      1005     1044
1815
#> 6  2013     4    10     1100           1900       960     1342
2211
#> # … with 3.368e+05 more rows, and 11 more variables: arr_delay <dbl>,
#> #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
#> #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour
<dttm>
```



Select columns with `select()`

It’s not uncommon to get datasets with hundreds or even thousands of variables. In this case, the first challenge is often narrowing in on the variables one’re actually interested in.

`select()` allows one to rapidly zoom in on a useful subset using operations based on the names of the variables.

`select()` is not terribly useful with the flights data because we only have 19 variables, but one can still get the general idea:

```
# Select columns by name
select(flights, year, month, day)
#> # A tibble: 336,776 x 3
#>    year month   day
#>   <int> <int> <int>
#> 1  2013     1     1
#> 2  2013     1     1
#> 3  2013     1     1
#> 4  2013     1     1
#> 5  2013     1     1
#> 6  2013     1     1
#> # … with 3.368e+05 more rows
# Select all columns between year and day (inclusive)
select(flights, year:day)
#> # A tibble: 336,776 x 3
#>    year month   day
#>   <int> <int> <int>
#> 1  2013     1     1
#> 2  2013     1     1
#> 3  2013     1     1
#> 4  2013     1     1
#> 5  2013     1     1
#> 6  2013     1     1
#> # … with 3.368e+05 more rows
# Select all columns except those from year to day (inclusive)
select(flights, -(year:day))
#> # A tibble: 336,776 x 16
#>   dep_time sched_dep_time dep_delay arr_time sched_arr_time arr_delay
carrier
#>      <int>          <int>     <dbl>    <int>          <int>     <dbl>
<chr>
#> 1      517            515         2      830            819        11 UA
#> 2      533            529         4      850            830        20 UA
#> 3      542            540         2      923            850        33 AA
#> 4      544            545        -1     1004           1022       -18 B6
#> 5      554            600        -6      812            837       -25 DL
#> 6      554            558        -4      740            728        12 UA
#> # … with 3.368e+05 more rows, and 9 more variables: flight <int>,
#> #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance
<dbl>,
#> #   hour <dbl>, minute <dbl>, time_hour <dttm>
```

Add new variables with `mutate()`

Besides selecting sets of existing columns, it's often useful to add new columns that are functions of existing columns. That's the job of `mutate()`.

`mutate()` always adds new columns at the end of one's dataset so we'll start by creating a narrower dataset so we can see the new variables. Remember that when one is in RStudio, the easiest way to see all the columns is `View()`.

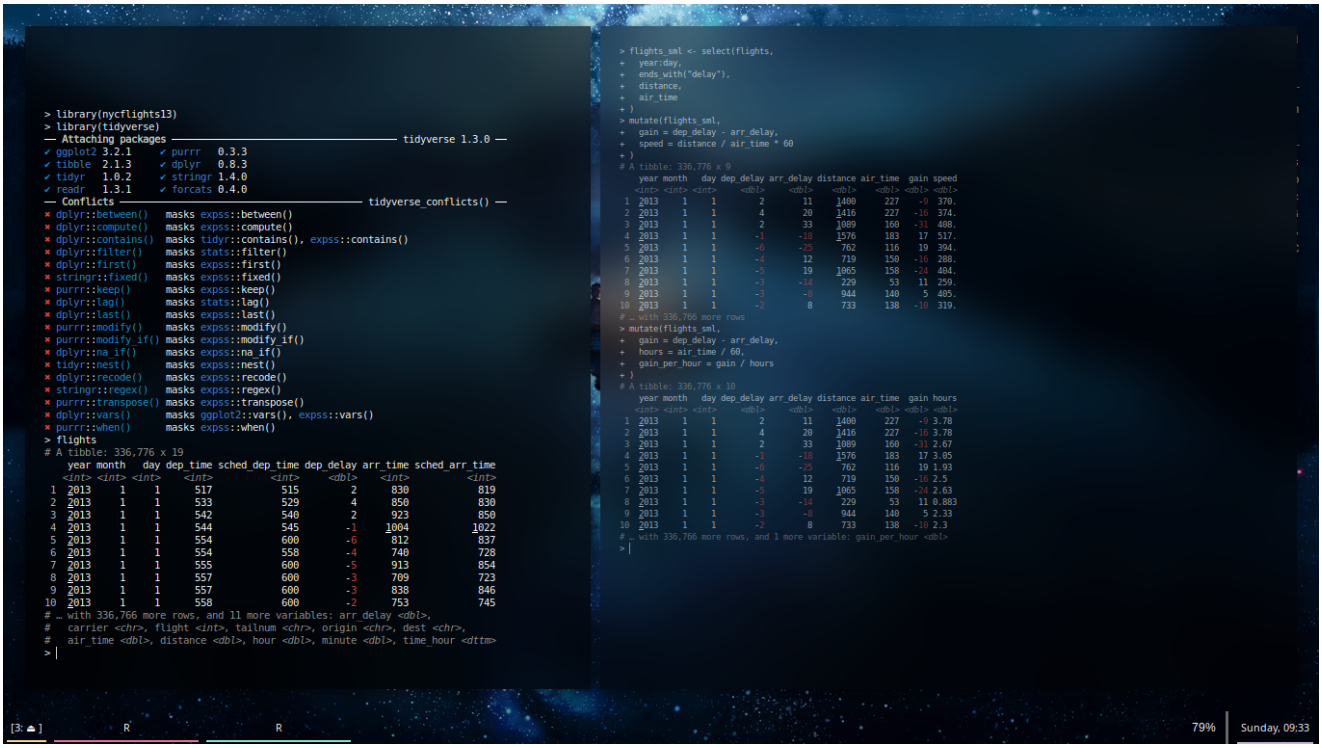
```
flights_sml <- select(flights,
  year:day,
  ends_with("delay"),
  distance,
  air_time
)
mutate(flights_sml,
  gain = dep_delay - arr_delay,
  speed = distance / air_time * 60
)
```

Note that you can refer to columns that one has just created:

```
mutate(flights_sml,
  gain = dep_delay - arr_delay,
  hours = air_time / 60,
  gain_per_hour = gain / hours
)
```

If you only want to keep the new variables, use `transmute()`:

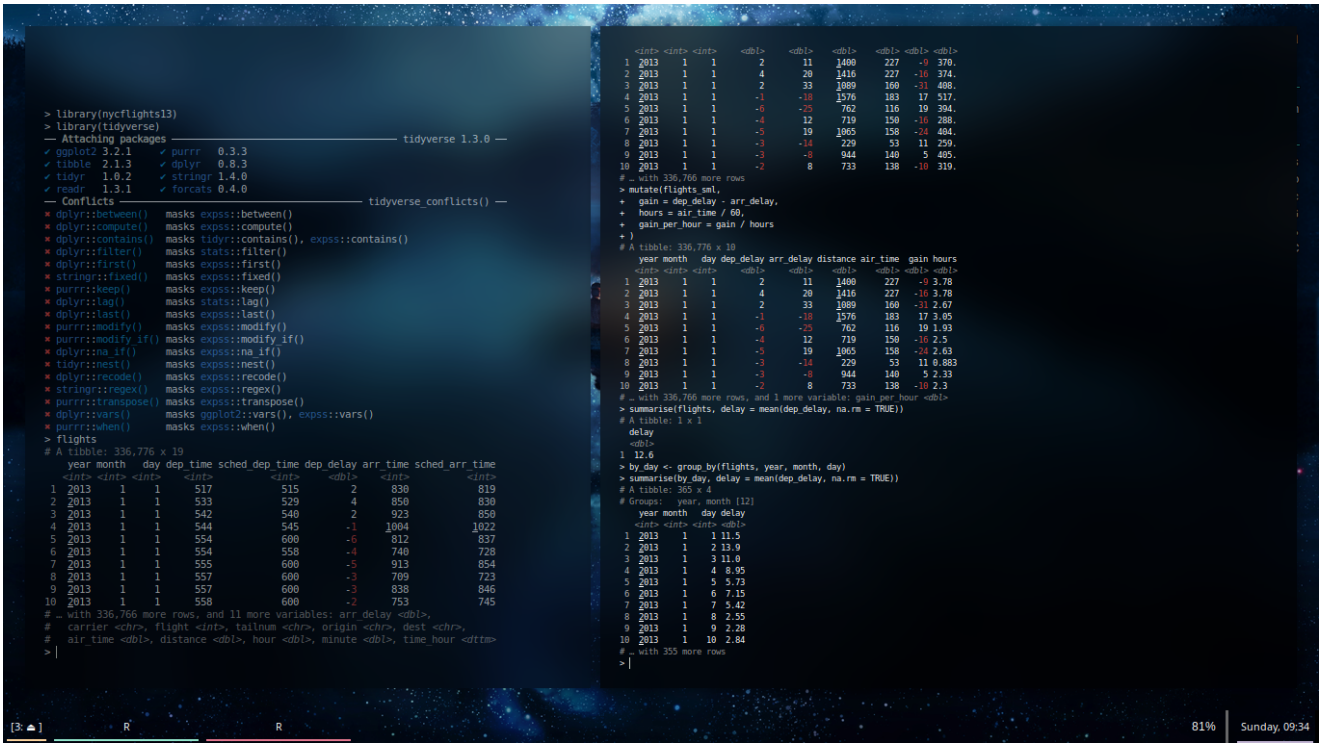
```
transmute(flights,
  gain = dep_delay - arr_delay,
  hours = air_time / 60,
  gain_per_hour = gain / hours
)
```



Grouped summaries with `summarise()`

The last key verb is `summarise()`. It collapses a data frame to a single row:

```
summarise(flights, delay = mean(dep_delay, na.rm = TRUE))
#> # A tibble: 1 x 1
#>   delay
#>   <dbl>
#> 1  12.6
```



Conclusion

Henceforth, we have Successfully studied dplyr and its supportive functions.

Experiment 7

Aim:

EDA with R - Range, Summary, Mean, Variance, Median, Standard Deviation, Histogram, Boxplot, Scatter Graph

Theory:

EDA is an iterative cycle. wherein One:

1. Generate questions about one's data.
2. Search for answers by visualising, transforming, and modelling one's data.
3. Use what one learn to refine oner questions and/or generate new questions.

EDA is not a formal process with a strict set of rules. More than anything, EDA is a state of mind. During the initial phases of EDA one should feel free to investigate every idea that occurs to one. Some of these ideas will pan out, and some will be dead ends. As one's exploration continues, one will home in on a few particularly productive areas that one'll eventually write up and communicate to others.

EDA is an important part of any data analysis, even if the questions are handed to one on a platter, because one always need to investigate the quality of oner data. Data cleaning is just one application of EDA: one ask questions about whether oner data meets oner expectations or not. To do data cleaning, one'll need to deploy all the tools of EDA: visualisation, transformation, and modelling.

Definitive Terms:

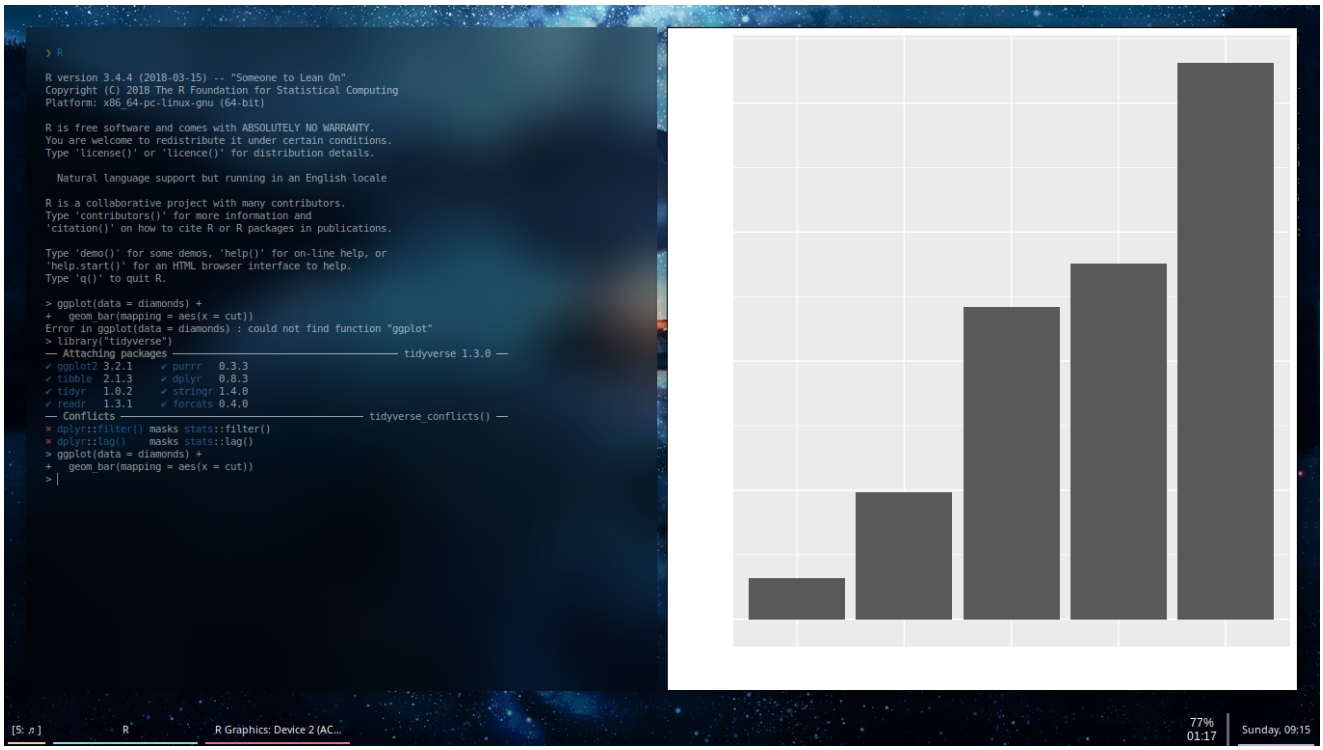
- A **variable** is a quantity, quality, or property that one can measure.
- A **value** is the state of a variable when one measure it. The value of a variable may change from measurement to measurement.
- An **observation** is a set of measurements made under similar conditions (one usually make all of the measurements in an observation at the same time and on the same object). An observation will contain several values, each associated with a different variable. I'll sometimes refer to an observation as a data point.
- **Tabular data** is a set of values, each associated with a variable and an observation. Tabular data is *tidy* if each value is placed in its own "cell", each variable in its own column, and each observation in its own row.

Variation

Variation is the tendency of the values of a variable to change from measurement to measurement. one can see variation easily in real life; if one measure any continuous variable twice, one will get two different results. This is true even if one measure quantities that are constant, like the speed of light. Each of one's measurements will include a small amount of error that varies from measurement to measurement. Categorical variables can also vary if one measure across different subjects (e.g. the eye colors of different people), or different times (e.g. the energy levels of an electron at different moments). Every variable has its own pattern of variation, which can reveal interesting information. The best way to understand that pattern is to visualise the distribution of the variable's values.

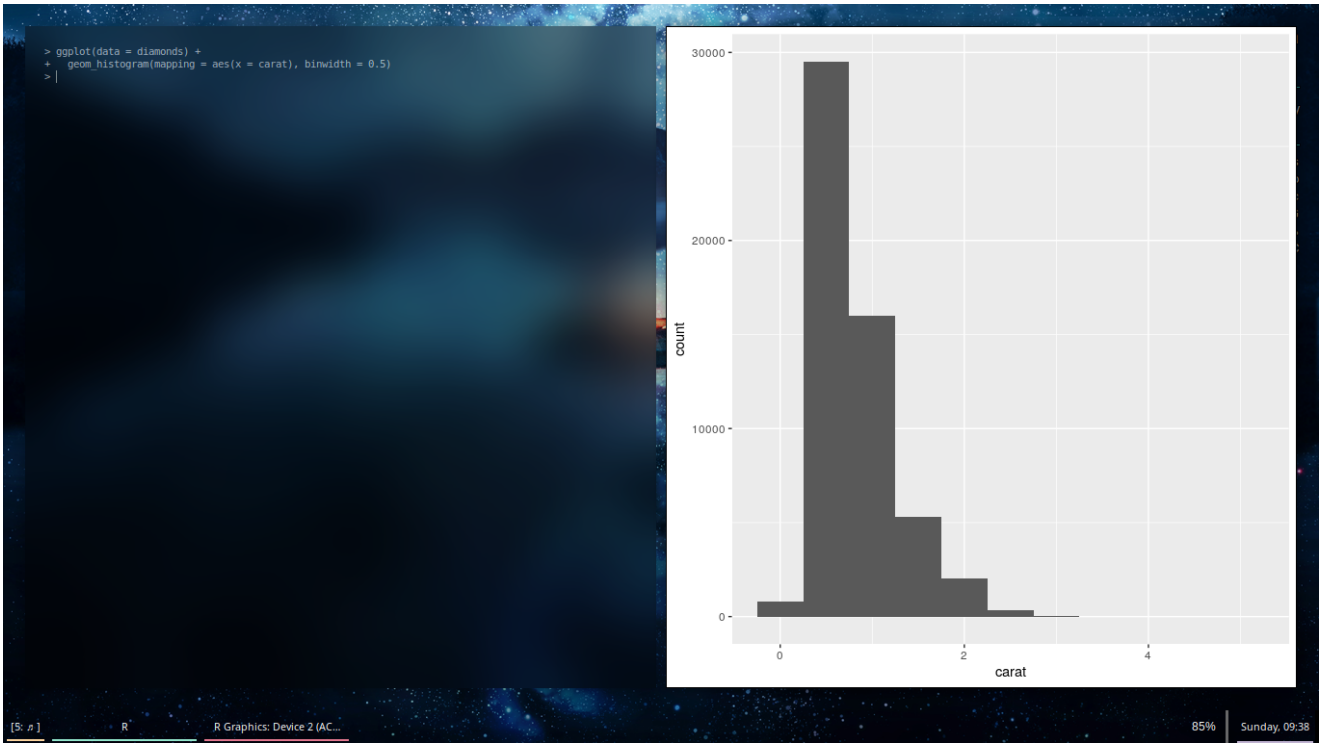
A variable is **categorical** if it can only take one of a small set of values. In R, categorical variables are usually saved as factors or character vectors. To examine the distribution of a categorical variable, use a bar chart:

```
ggplot(data = diamonds) +
  geom_bar(mapping = aes(x = cut))
```



A variable is **continuous** if it can take any of an infinite set of ordered values. Numbers and date-times are two examples of continuous variables. To examine the distribution of a continuous variable, use a histogram:

```
ggplot(data = diamonds) +
  geom_histogram(mapping = aes(x = carat), binwidth = 0.5)
```



Unusual values

Outliers are observations that are unusual; data points that don't seem to fit the pattern. Sometimes outliers are data entry errors; other times outliers suggest important new science. When one have a lot of data, outliers are sometimes difficult to see in a histogram. For example, take the distribution of the `y` variable from the diamonds dataset. The only evidence of outliers is the unusually wide limits on the x-axis.

```
ggplot(diamonds) +
  geom_histogram(mapping = aes(x = y), binwidth = 0.5)
```

Missing values

If one has encountered unusual values in one's dataset, and simply want to move on to the rest of one's analysis, one have two options.

1. Drop the entire row with the strange values:

   ```
   diamonds2 <- diamonds %>%
     filter(between(y, 3, 20))
   ```

   I don't recommend this option because just because one measurement is invalid, doesn't mean all the measurements are. Additionally, if one have low quality data, by time that one've applied this approach to every variable one might find that one don't have any data left!

2. Instead, I recommend replacing the unusual values with missing values. The easiest way to do this is to use `mutate()` to replace the variable with a modified copy. one can use the `ifelse()` function to replace unusual values with `NA`:

```
diamonds2 <- diamonds %>%
  mutate(y = ifelse(y < 3 | y > 20, NA, y))
```

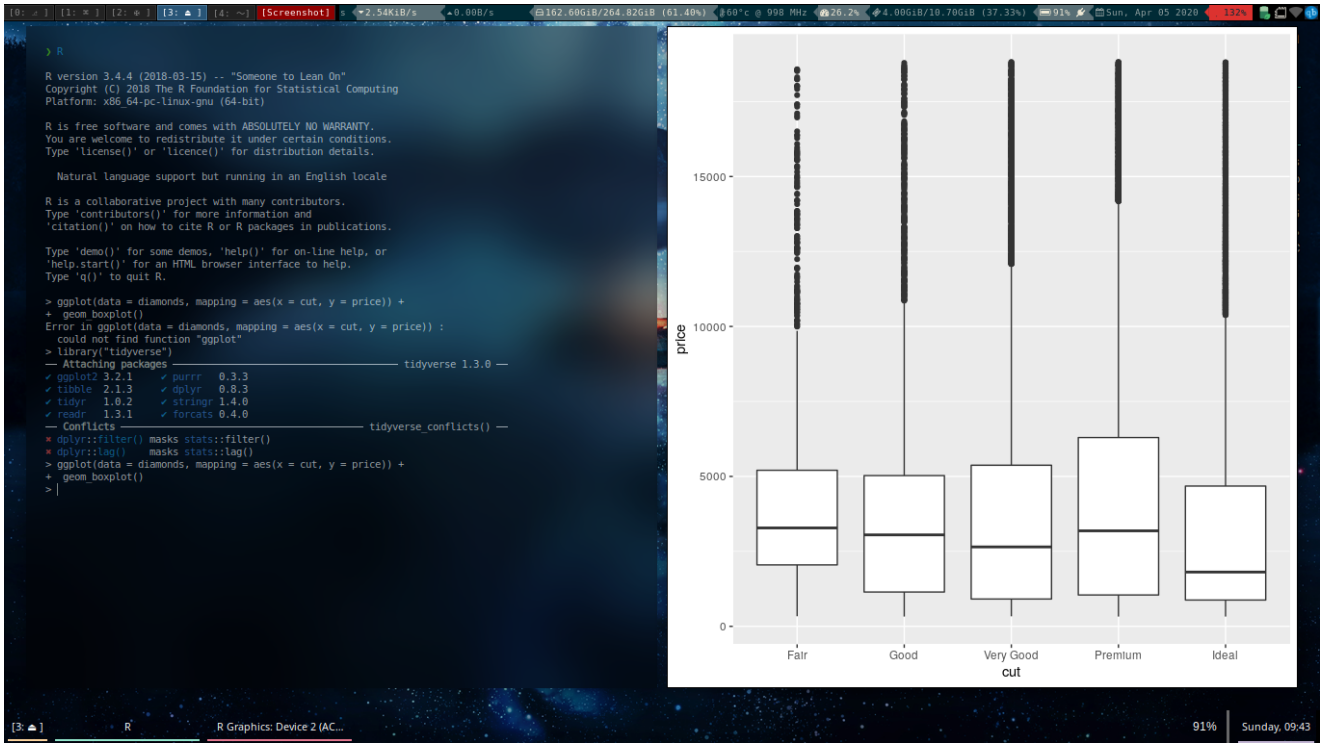
Covariation

If variation describes the behavior *within* a variable, covariation describes the behavior *between* variables. **Covariation** is the tendency for the values of two or more variables to vary together in a related way. The best way to spot covariation is to visualise the relationship between two or more variables. How you do that should again depend on the type of variables involved.

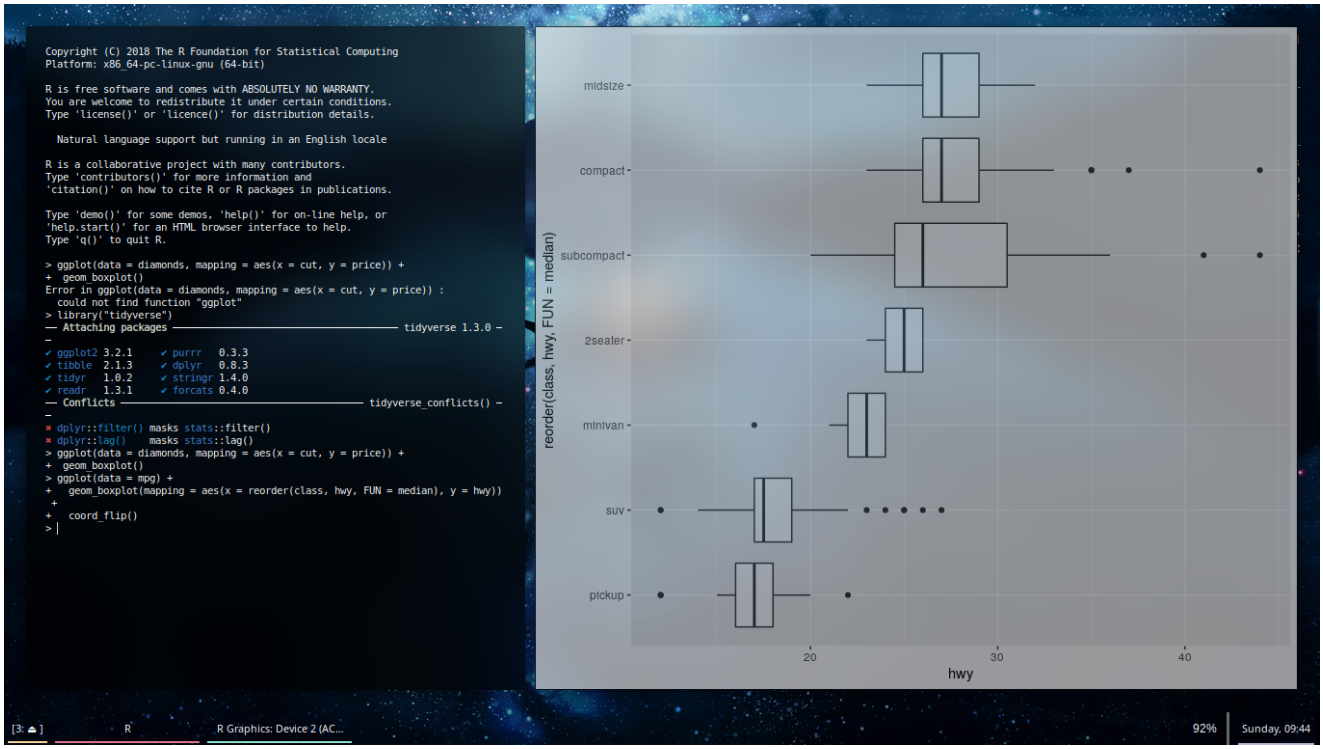
A **boxplot** is a type of visual shorthand for a distribution of values that is popular among statisticians. Each boxplot consists of:

- A box that stretches from the 25th percentile of the distribution to the 75th percentile, a distance known as the interquartile range (IQR). In the middle of the box is a line that displays the median, i.e. 50th percentile, of the distribution. These three lines give you a sense of the spread of the distribution and whether or not the distribution is symmetric about the median or skewed to one side.
- Visual points that display observations that fall more than 1.5 times the IQR from either edge of the box. These outlying points are unusual so are plotted individually.
- A line (or whisker) that extends from each end of the box and goes to the farthest non-outlier point in the distribution.

```
ggplot(data = diamonds, mapping = aes(x = cut, y = price)) +
  geom_boxplot()
```




```
ggplot(data = mpg) +
  geom_boxplot(mapping = aes(x = reorder(class, hwy, FUN = median), y = hwy)) +
  coord_flip()
```



Central Tendency

Mean: Calculate sum of all the values and divide it with the total number of values in the data set.

```
x <- c(1,2,3,4,5,1,2,3,1,2,4,5,2,3,1,1,2,3,5,6) # our data set

mean.result = mean(x) # calculate mean

print (mean.result)
```

Median: The middle value of the data set.

```
x <- c(1,2,3,4,5,1,2,3,1,2,4,5,2,3,1,1,2,3,5,6)
median.result <- median(x)
print(median.result)
```

Mode: The most occurring number in the data set. For calculating mode, there is no default function in R. So, we have to create our own custom function.

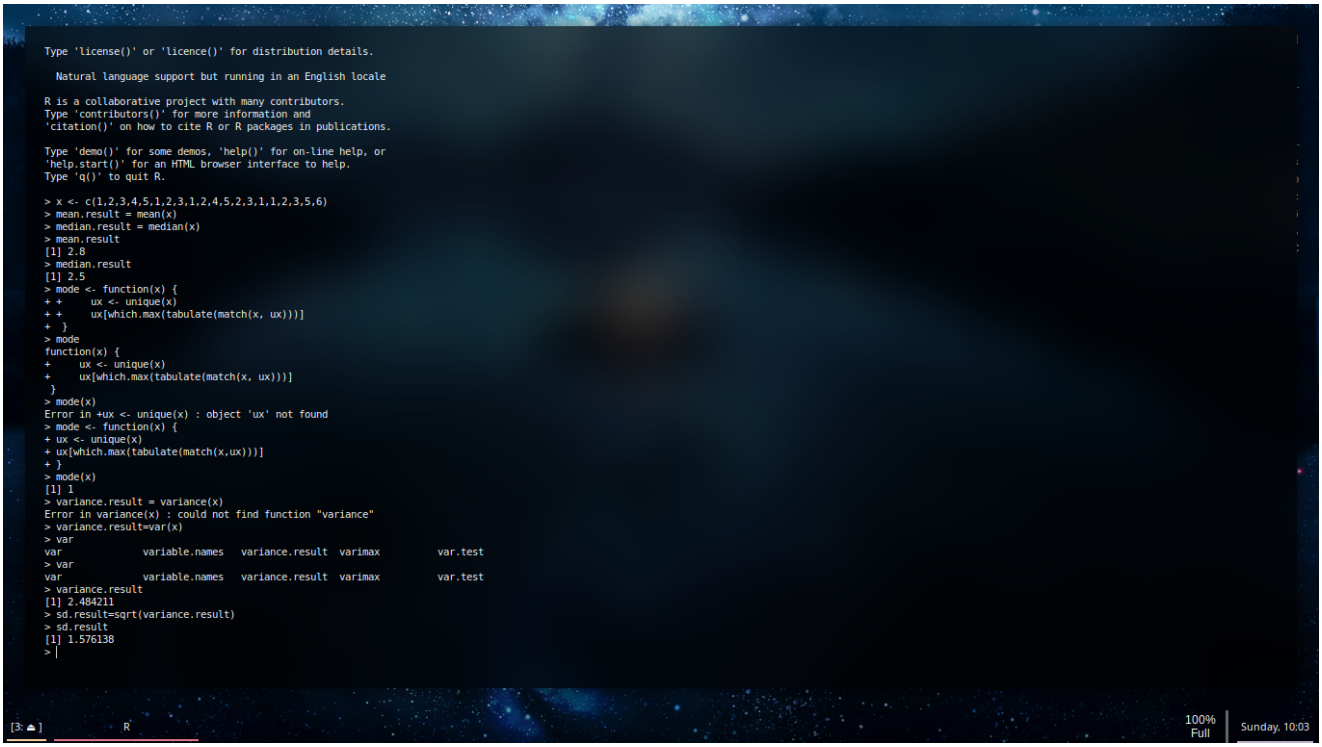
```
mode <- function(x) {
+     ux <- unique(x)
+     ux[which.max(tabulate(match(x, ux)))]
+ }
 x <- c(1,2,3,4,5,1,2,3,1,2,4,5,2,3,1,1,2,3,5,6)
 mode.result = mode(x)
 print (mode.result)
```

Variance: How far a set of data values are spread out from their mean.

```
variance.result = var(x)
print (variance.result)
```

Standard Deviation: A measure that is used to quantify the amount of variation or dispersion of a set of data values.

```
sd.result = sqrt(var(x))
print (sd.result)
```



Conclusion:

Henceforth, we have successfully performed EDA, and studied central tendencies.

Experiment 8

Aim:

Basic and Advanced Graphics in R

Theory:

Graphics are important for conveying important features of the data.**R** includes at least three graphical systems, the standard **graphics** package, the **lattice** package for Trellis graphs. R has good graphical capabilities but there are some alternatives like gnuplot.

R provides the usual range of standard statistical plots, including scatterplots, boxplots, histograms, barplots, piecharts, and basic 3D plots.

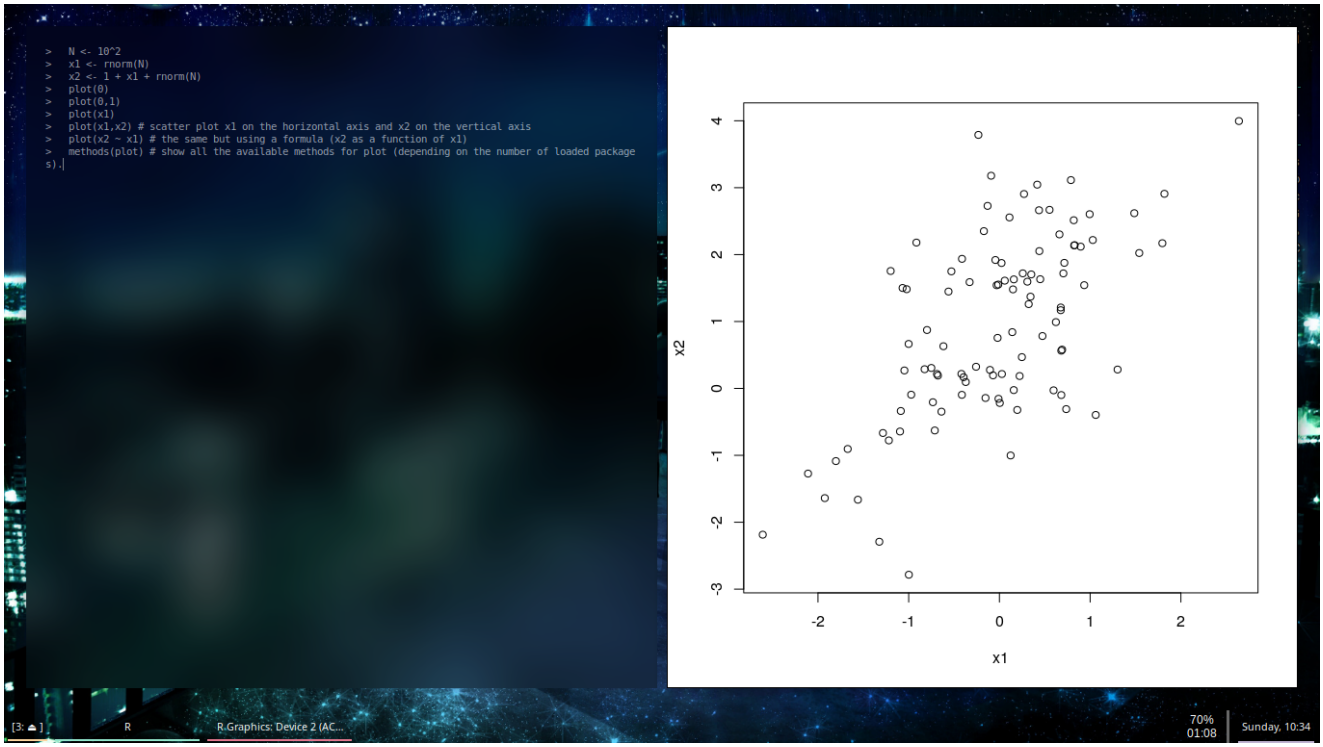
They can be used to examine • Marginal distributions • Relationships between variables •Summary of very large data

Important complement to many statistical and computational techniques.

Graphics in R

- `plot()` is the main function for graphics. The arguments can be a single point such as 0 or c(.3,.7), a single vector, a pair of vectors or many other R objects.
- `par()` is another important function which defines the default settings for plots.
- There are many other plot functions which are specific to some tasks such as `hist()`, `boxplot()`, etc. Most of them take the same arguments as the `plot()` function.

```
  N <- 10^2
  x1 <- rnorm(N)
  x2 <- 1 + x1 + rnorm(N)
  plot(0)
  plot(0,1)
  plot(x1)
  plot(x1,x2) # scatter plot x1 on the horizontal axis and x2 on the
vertical axis
  plot(x2 ~ x1) # the same but using a formula (x2 as a function of x1)
```



R Histograms

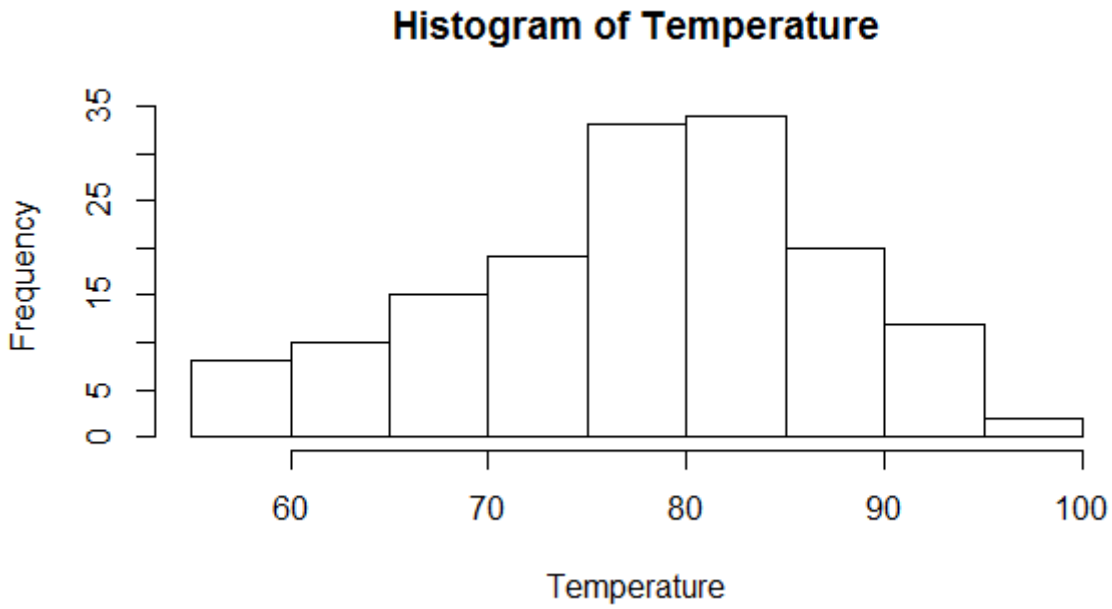
In this article, you'll learn to use hist() function to create histograms in R programming with the help of numerous examples.

Histogram can be created using the `hist()` function in R programming language. This function takes in a vector of values for which the histogram is plotted.

Let us use the built-in dataset `airquality` which hasDaily air quality measurements in New York, May to September 1973.-R documentation.

```
 str(airquality)
 'data.frame':153 obs. of  6 variables:$ Ozone   : int  41 36 12 18 NA 28 23
19 8 NA ...$
Solar.R: int  190 118 149 313 NA NA 299 99 19 194 ...$
Wind   : num  7.4 8 12.6 11.5 14.3 14.9 8.6 13.8 20.1 8.6 ...$
Temp   : int  67 72 74 62 56 66 65 59 61 69 ...$
Month  : int  5 5 5 5 5 5 5 5 5 5 ...$
Day    : int  1 2 3 4 5 6 7 8 9 10 ...
```

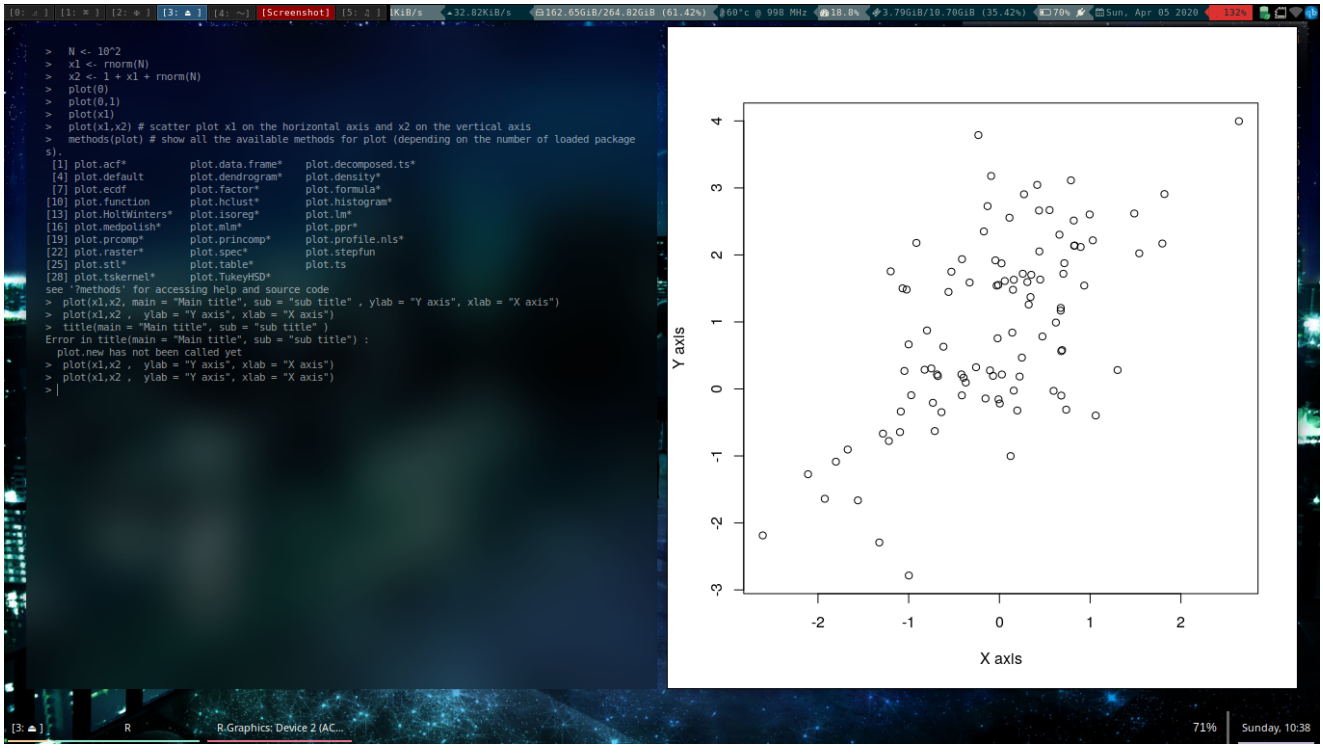
```
Temperature <- airquality$Temphist(Temperature)
```



Titles, legends and annotations

Titles main gives the main title, sub the subtitle. They can be passed as argument of the plot() function or using the title() function. xlab the name of the x axis and ylab the name of the y axis.

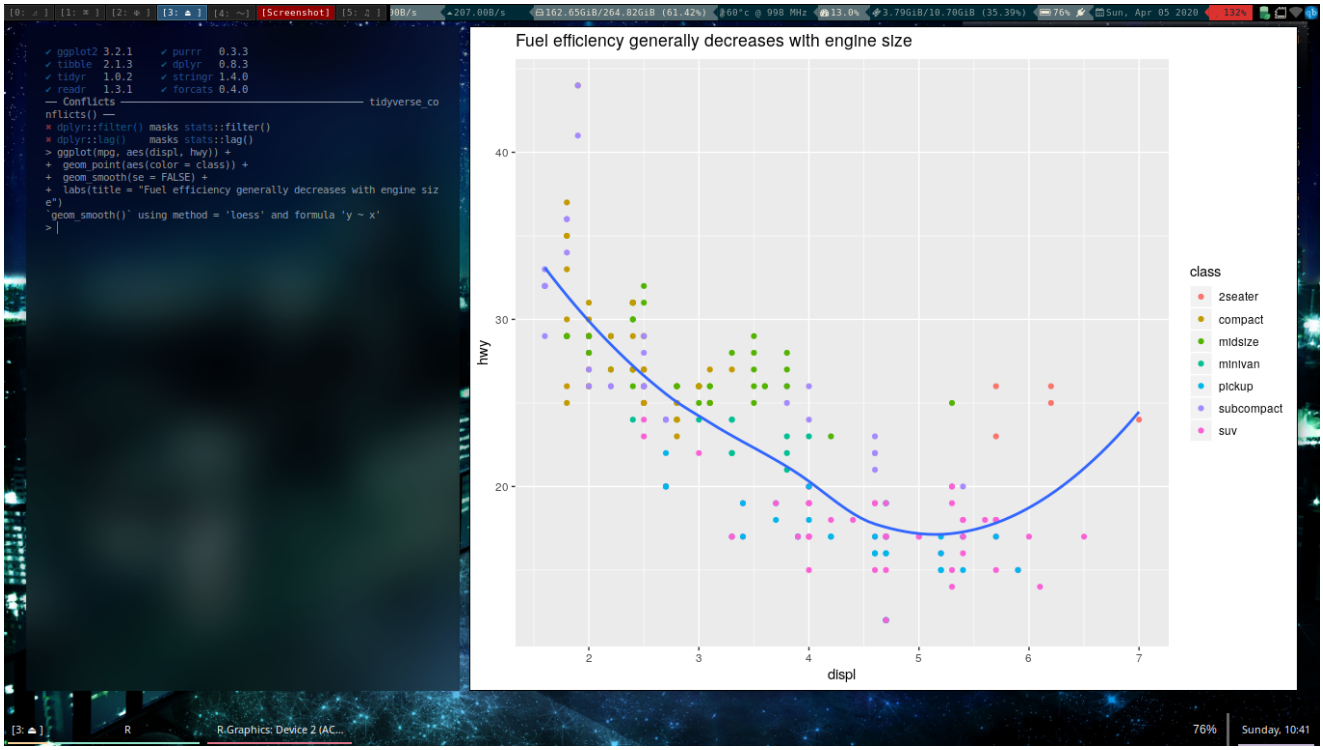
```
  plot(x1,x2, main = "Main title", sub = "sub title" , ylab = "Y axis", xlab
= "X axis")
  plot(x1,x2 ,  ylab = "Y axis", xlab = "X axis")
  title(main = "Main title", sub = "sub title" )
```



Label

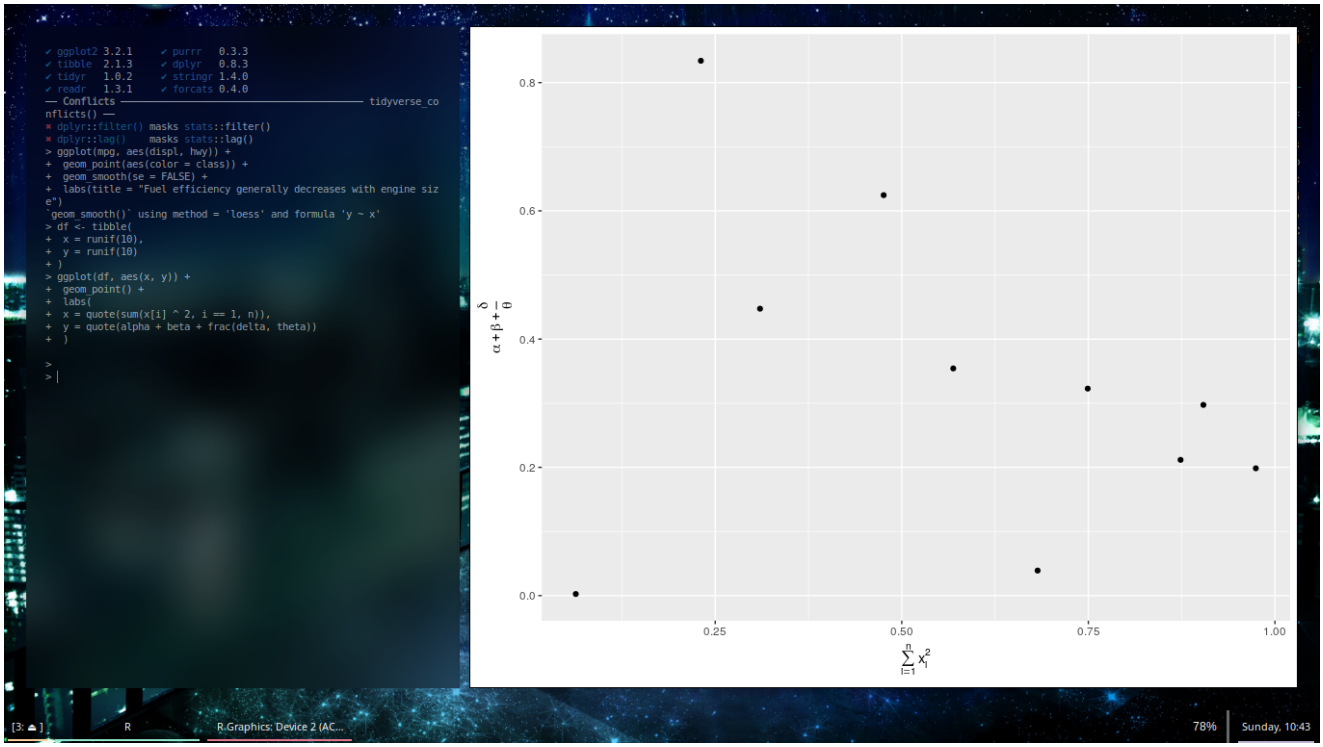
The easiest place to start when turning an exploratory graphic into an expository graphic is with good labels. You add labels with the `labs()` function. This example adds a plot title:

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(color = class)) +
  geom_smooth(se = FALSE) +
  labs(title = "Fuel efficiency generally decreases with engine size")
```



It's possible to use mathematical equations instead of text strings. Just switch `""` out for `quote()` and read about the available options in `?plotmath`:

```
df <- tibble(
  x = runif(10),
  y = runif(10)
)
ggplot(df, aes(x, y)) +
  geom_point() +
  labs(
    x = quote(sum(x[i] ^ 2, i == 1, n)),
    y = quote(alpha + beta + frac(delta, theta))
  )
```

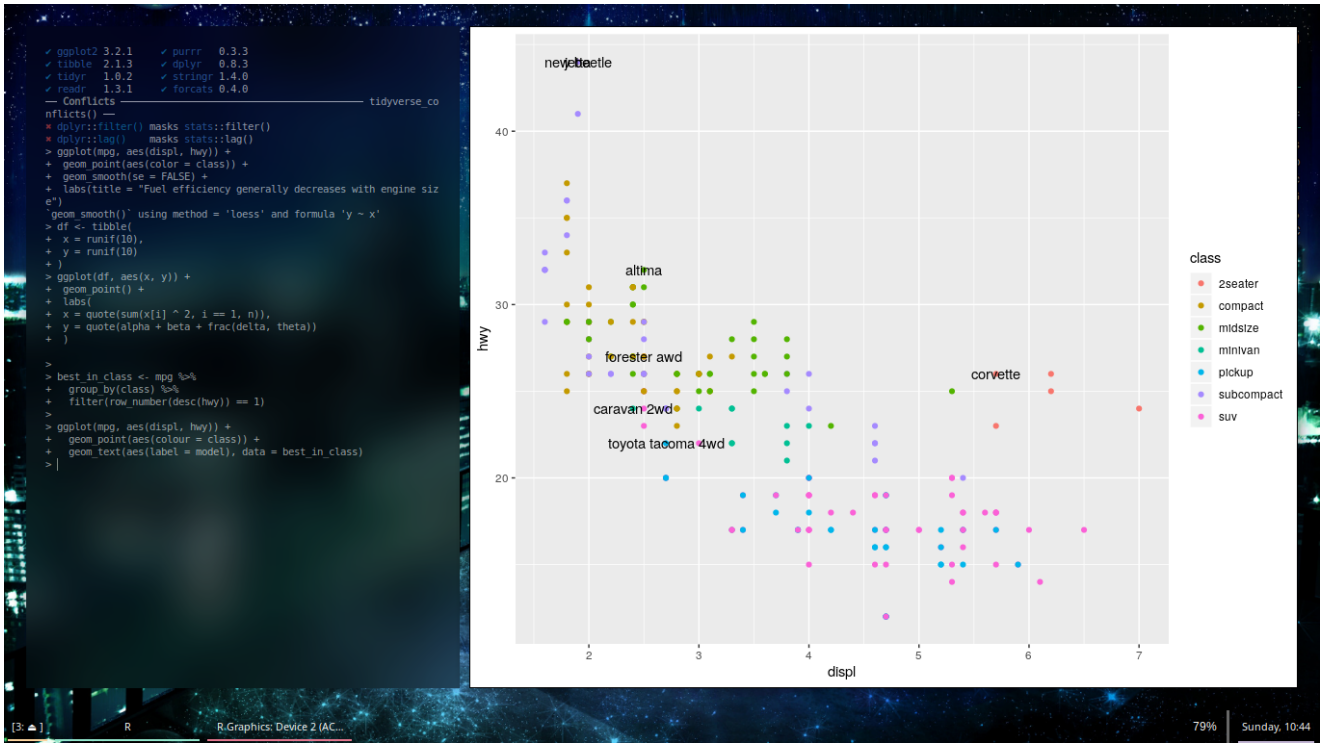


Annotations

In addition to labelling major components of your plot, it's often useful to label individual observations or groups of observations. The first tool you have at your disposal is `geom_text()`. `geom_text()` is similar to `geom_point()`, but it has an additional aesthetic: `label`. This makes it possible to add textual labels to your plots.

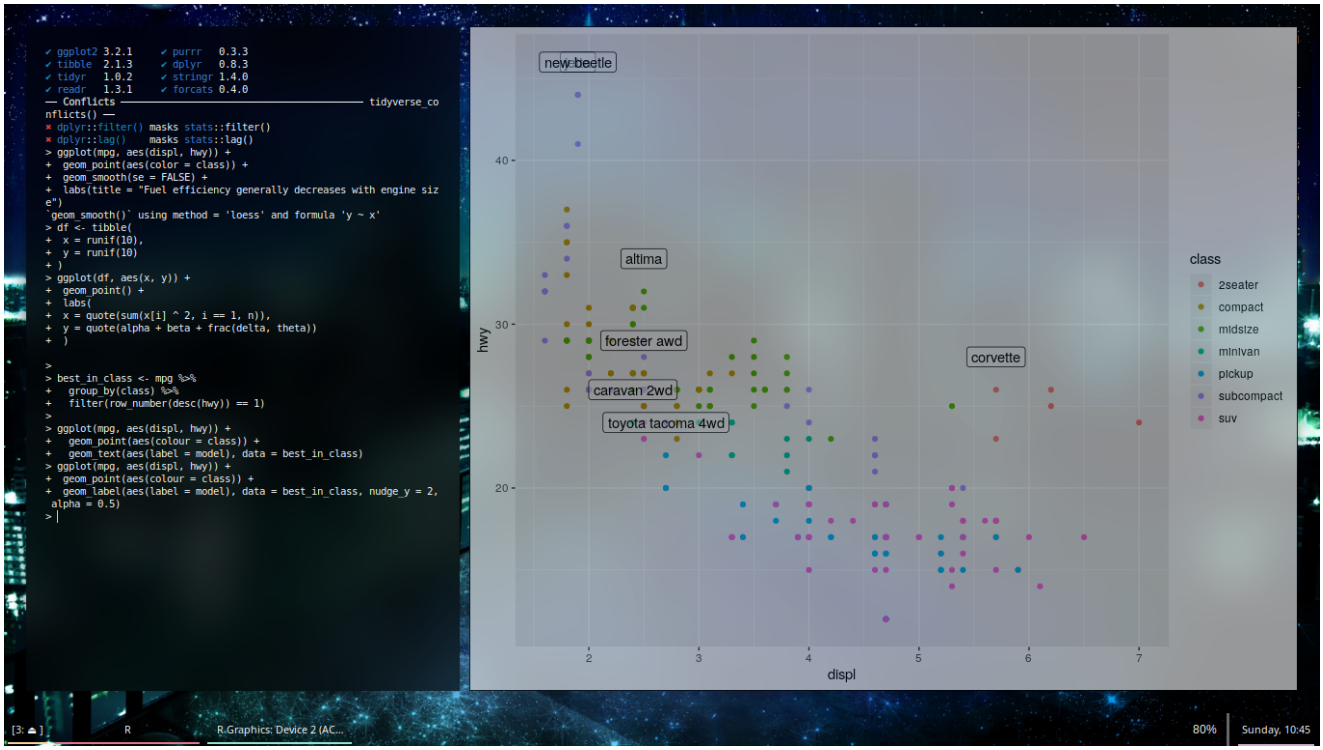
There are two possible sources of labels. First, you might have a tibble that provides labels. The plot below isn't terribly useful, but it illustrates a useful approach: pull out the most efficient car in each class with dplyr, and then label it on the plot:

```
best_in_class <- mpg %>%
  group_by(class) %>%
  filter(row_number(desc(hwy)) == 1)

ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(colour = class)) +
  geom_text(aes(label = model), data = best_in_class)
```



This is hard to read because the labels overlap with each other, and with the points. We can make things a little better by switching to `geom_label()` which draws a rectangle behind the text. We also use the `nudge_y` parameter to move the labels slightly above the corresponding points:

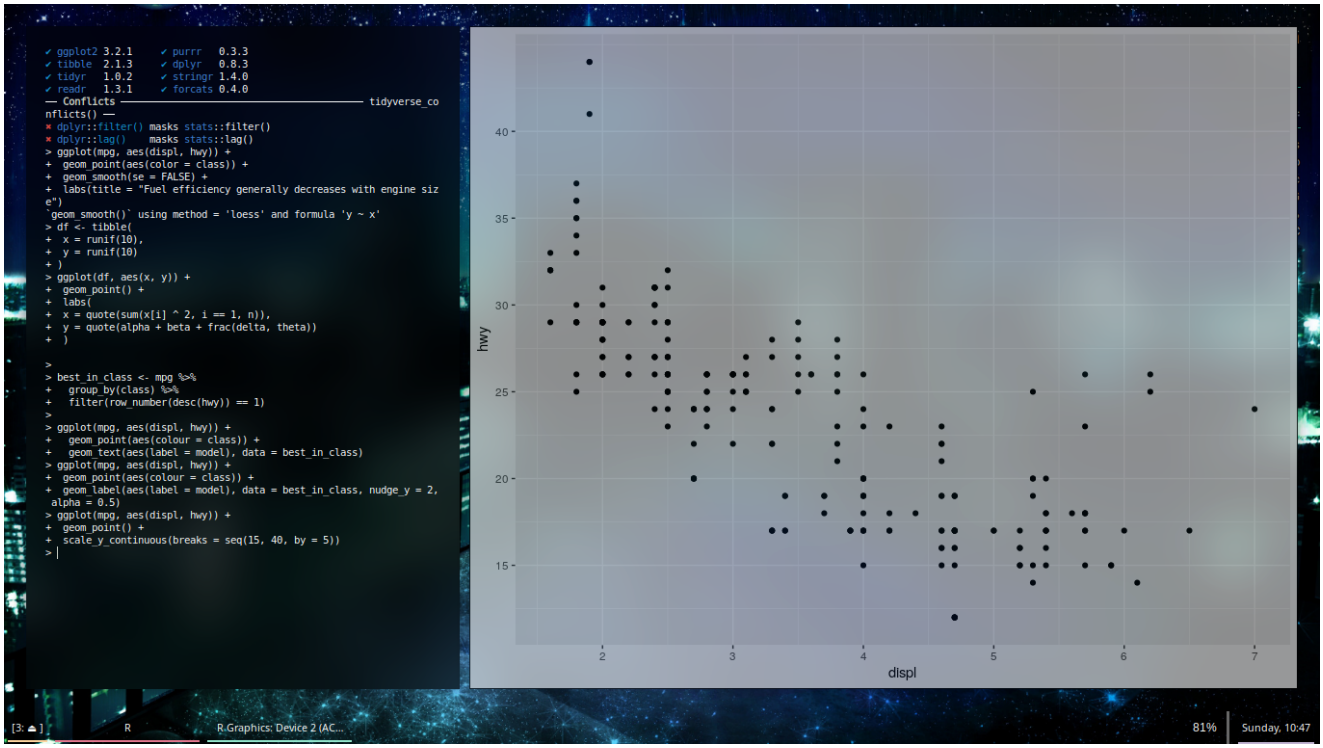
```
ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(colour = class)) +
  geom_label(aes(label = model), data = best_in_class, nudge_y = 2, alpha =
0.5)
```



Axis ticks and legend keys

There are two primary arguments that affect the appearance of the ticks on the axes and the keys on the legend: `breaks` and `labels`. Breaks controls the position of the ticks, or the values associated with the keys. Labels controls the text label associated with each tick/key. The most common use of `breaks` is to override the default choice:

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point() +
  scale_y_continuous(breaks = seq(15, 40, by = 5))
```

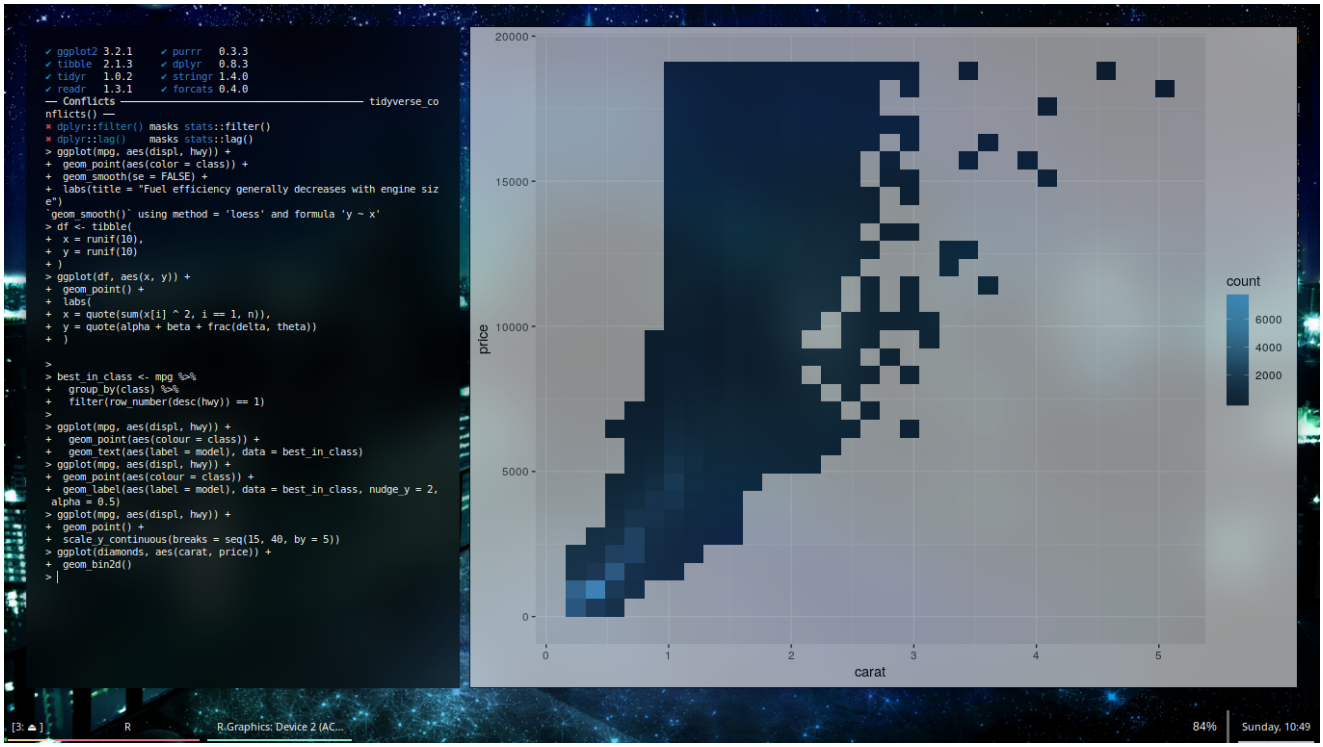


Replacing a scale

Instead of just tweaking the details a little, you can instead replace the scale altogether. There are two types of scales you're mostly likely to want to switch out: continuous position scales and colour scales. Fortunately, the same principles apply to all the other aesthetics, so once you've mastered position and colour, you'll be able to quickly pick up other scale replacements.

It's very useful to plot transformations of your variable. For example, as we've seen in diamond prices it's easier to see the precise relationship between `carat` and `price` if we log transform them:

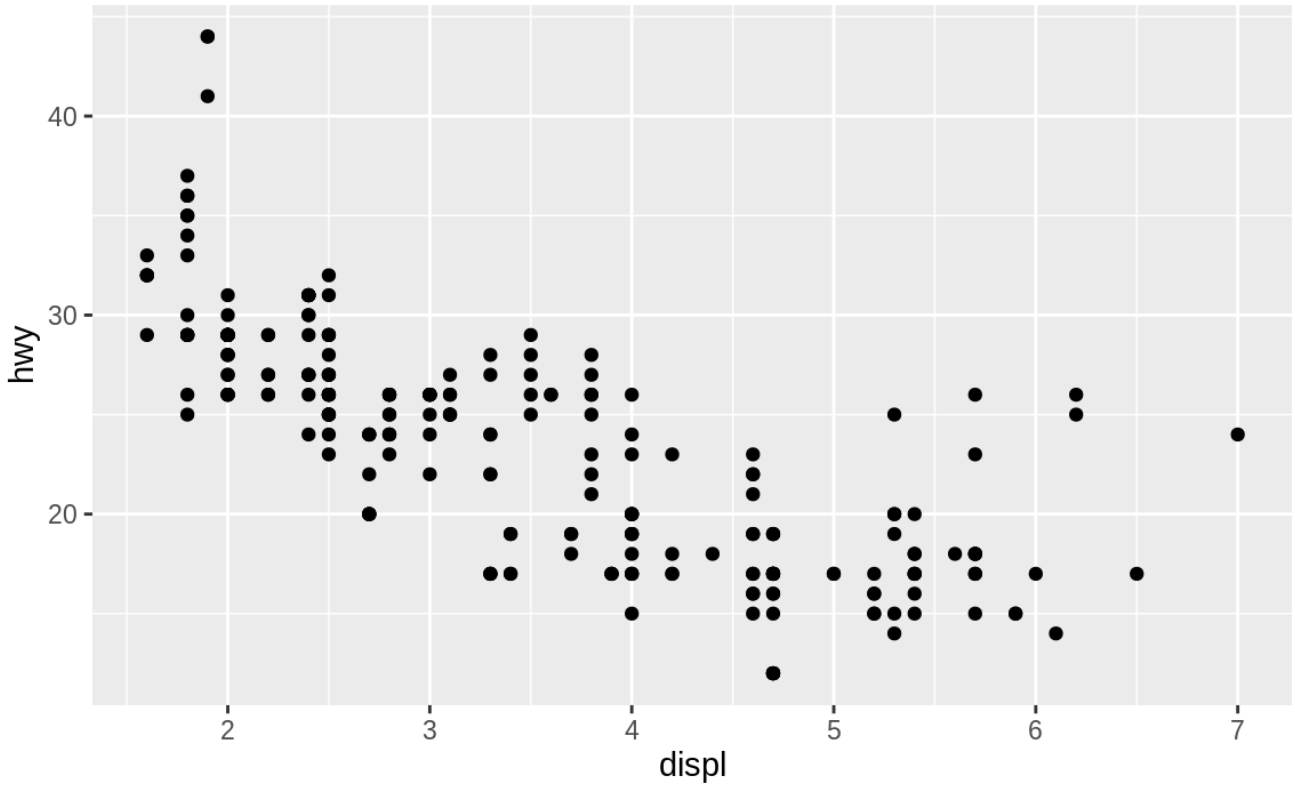
```
ggplot(diamonds, aes(carat, price)) +
  geom_bin2d()
```



Saving your plots

There are two main ways to get your plots out of R and into your final write-up: `ggsave()` and knitr. `ggsave()` will save the most recent plot to disk:

```
ggplot(mpg, aes(displ, hwy)) + geom_point()
```



```
ggsave("my-plot.pdf")
#> Saving 6 x 3.7 in image
```

Figure sizing

The biggest challenge of graphics in R Markdown is getting your figures the right size and shape. There are five main options that control figure sizing: `fig.width`, `fig.height`, `fig.asp`, `out.width` and `out.height`. Image sizing is challenging because there are two sizes (the size of the figure created by R and the size at which it is inserted in the output document), and multiple ways of specifying the size (i.e., height, width, and aspect ratio: pick two of three).

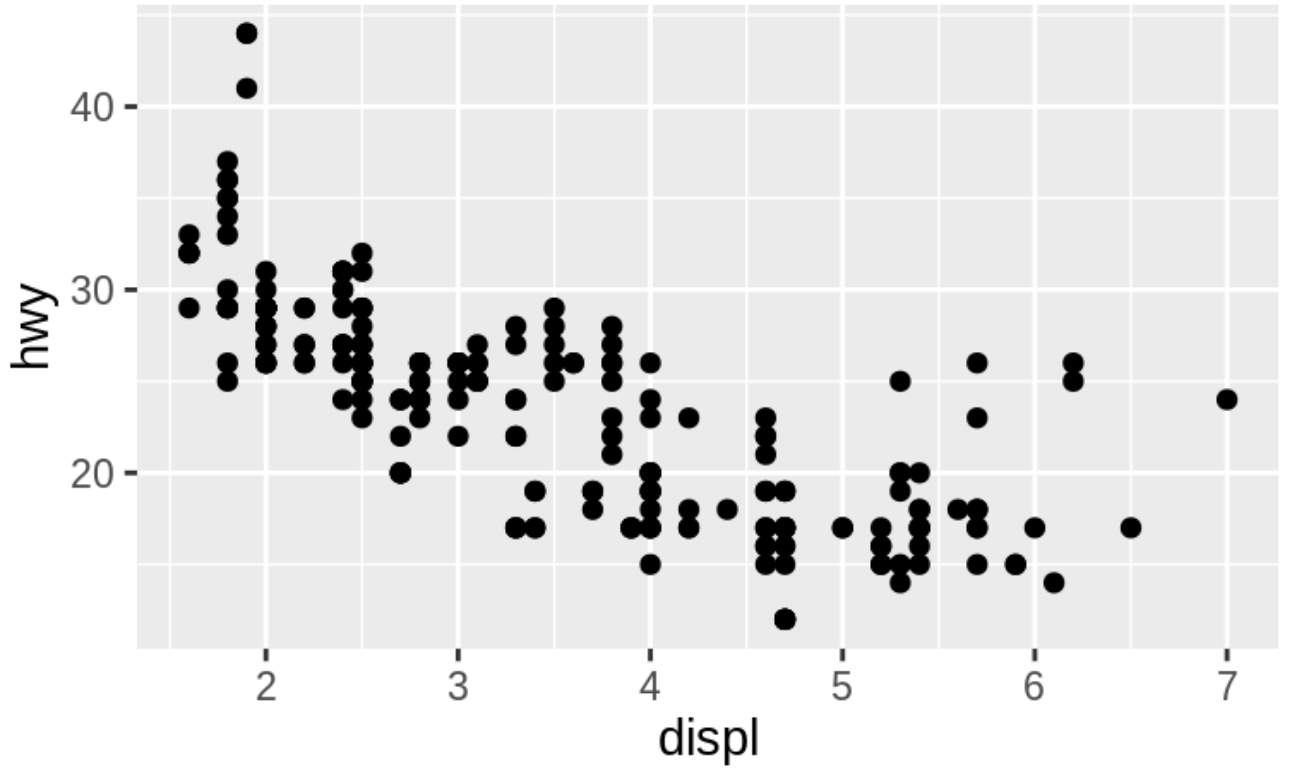
I only ever use three of the five options:

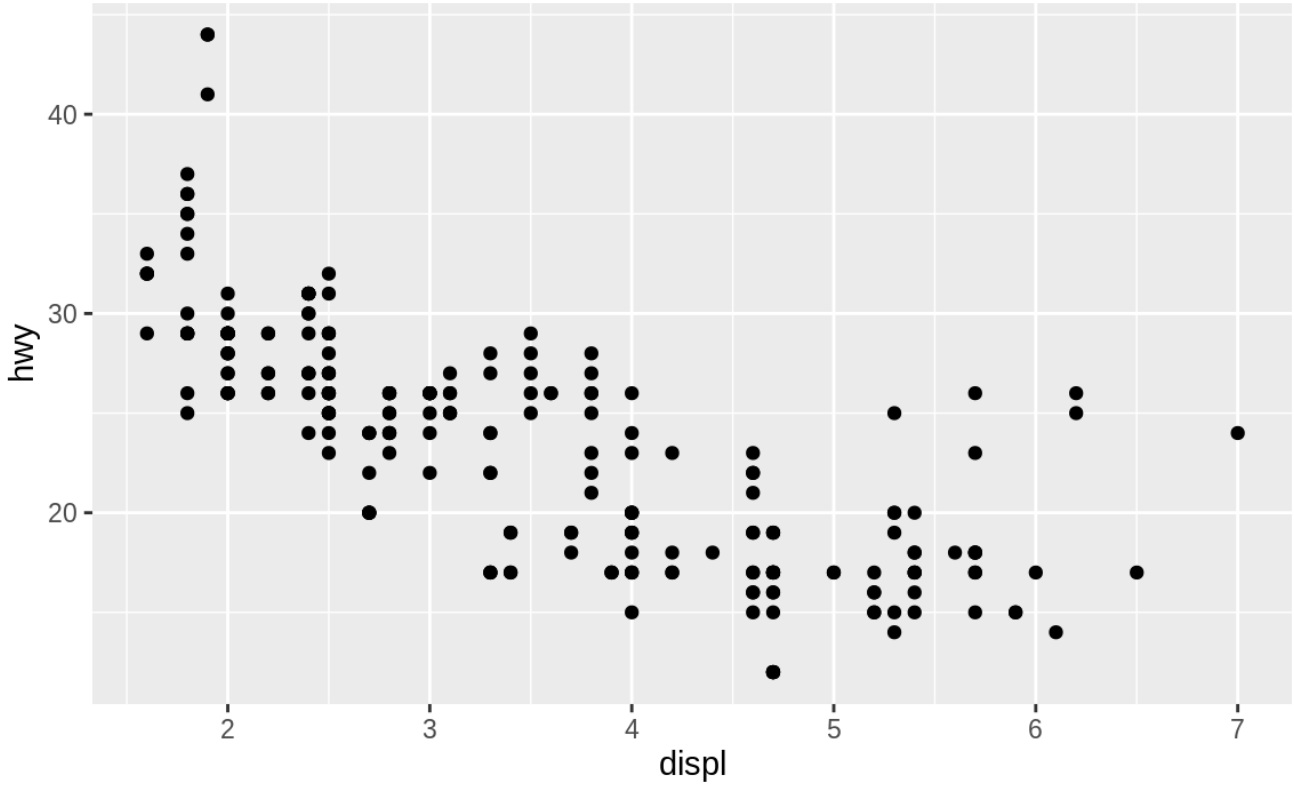
- I find it most aesthetically pleasing for plots to have a consistent width. To enforce this, I set `fig.width = 6` (6") and `fig.asp = 0.618` (the golden ratio) in the defaults. Then in individual chunks, I only adjust `fig.asp`.

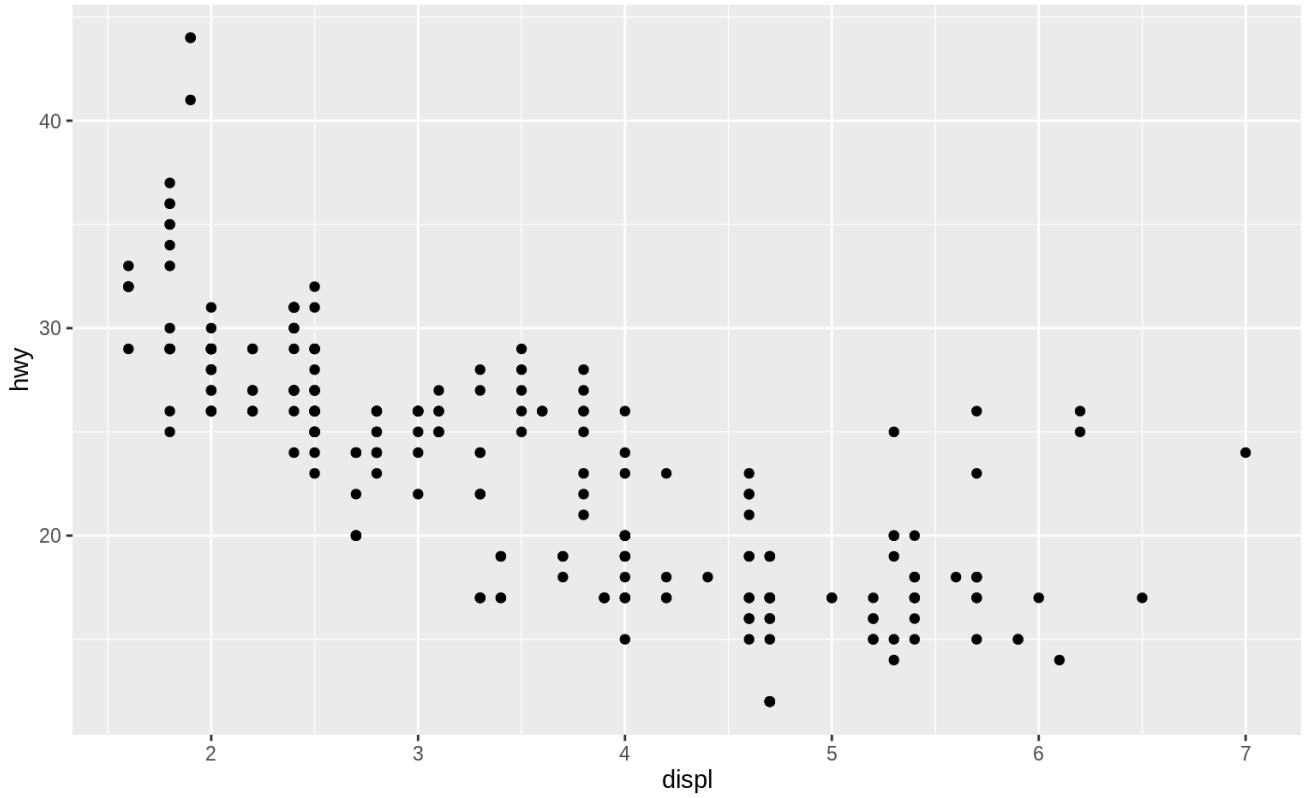
- I control the output size with `out.width` and set it to a percentage of the line width. I default to `out.width = "70%"` and `fig.align = "center"`. That give plots room to breathe, without taking up too much space.

- To put multiple plots in a single row I set the `out.width` to `50%` for two plots, `33%` for 3 plots, or `25%` to 4 plots, and set `fig.align = "default"`. Depending on what I'm trying to illustrate (e.g. show data or show plot variations), I'll also tweak `fig.width`, as discussed below.

If you find that you're having to squint to read the text in your plot, you need to tweak `fig.width`. If `fig.width` is larger than the size the figure is rendered in the final doc, the text will be too small; if `fig.width` is smaller, the text will be too big. You'll often need to do a little experimentation to figure out the right ratio between the `fig.width` and the eventual width in your document. To illustrate the principle, the following three plots have `fig.width` of 4, 6, and 8 respectively:


40
30
20
hwy
2
3
4
5
6
7
displ


40
30
20
hwy
2
3
4
5
6
7
displ



Conclusion:

Henceforth, we have successfully studied and explored plotting and graphs in R.

Experiment 9

Aim:

Basic of correlation, simple and multiple regression in R

Theory:

Correlation and linear regression each explore the relationship between two quantitative variables. Both are very common analyses.

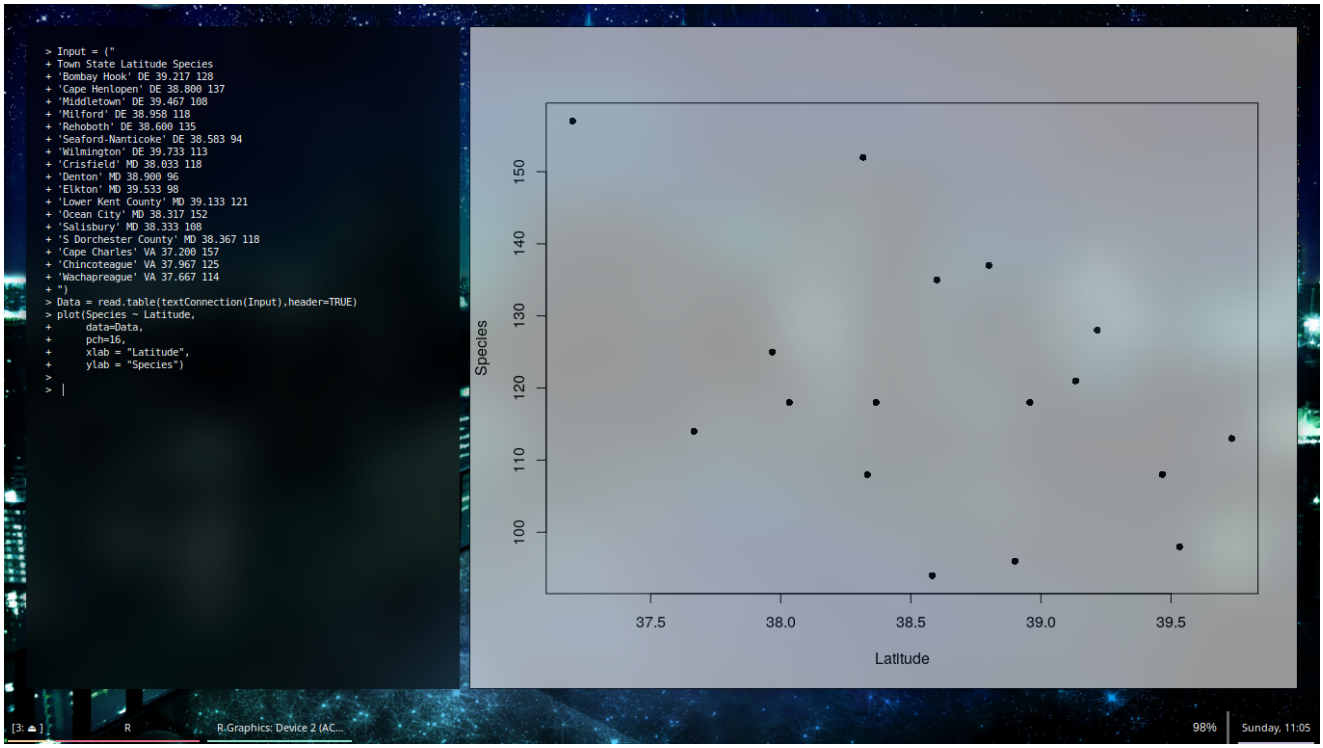
Correlation determines if one variable varies systematically as another variable changes. It does not specify that one variable is the dependent variable and the other is the independent variable. Often, it is useful to look at which variables are correlated to others in a data set, and it is especially useful to see which variables correlate to a particular variable of interest.

For this experiment we will be using the custom data and performing the experiments.

```
### ---------------------------------------------------------------
### Correlation and linear regression, species diversity example
### pp. 207-208
### ---------------------------------------------------------------

Input = ("
Town                   State  Latitude  Species
'Bombay Hook'           DE     39.217    128
'Cape Henlopen'         DE     38.800    137
'Middletown'            DE     39.467    108
'Milford'               DE     38.958    118
'Rehoboth'              DE     38.600    135
'Seaford-Nanticoke'     DE     38.583     94
'Wilmington'            DE     39.733    113
'Crisfield'             MD     38.033    118
'Denton'                MD     38.900     96
'Elkton'                MD     39.533     98
'Lower Kent County'     MD     39.133    121
'Ocean City'            MD     38.317    152
'Salisbury'             MD     38.333    108
'S Dorchester County'   MD     38.367    118
'Cape Charles'          VA     37.200    157
'Chincoteague'          VA     37.967    125
'Wachapreague'          VA     37.667    114
")

Data = read.table(textConnection(Input),header=TRUE)
```

So as to generate the simple plot for the data,

```
plot(Species ~ Latitude,
     data=Data,
     pch=16,
     xlab = "Latitude",
     ylab = "Species")
```



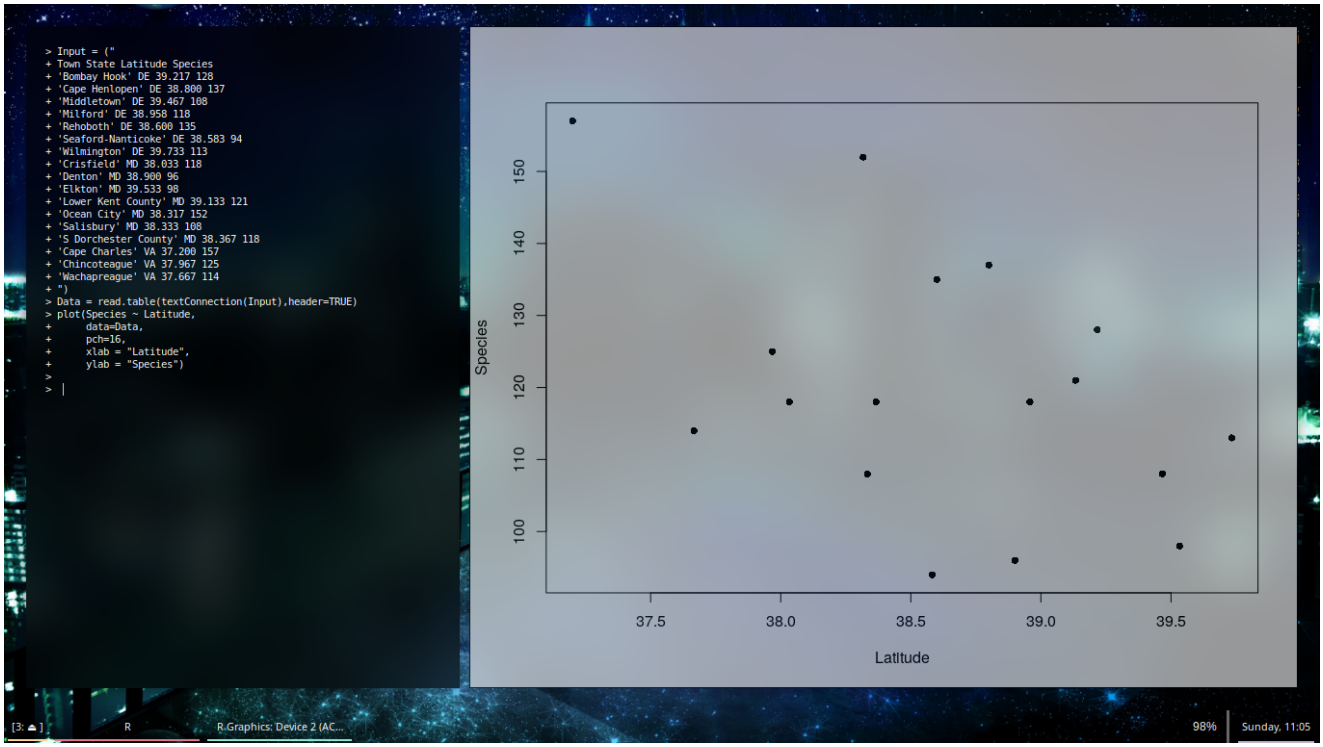
Correlation

Correlation can be performed with the *cor.test* function in the native *stats* package. It can perform Pearson, Kendall, and Spearman correlation procedures. Methods for multiple correlation of several variables simultaneously are discussed in the *Multiple regression* chapter.

Pearson correlation

Pearson correlation is the most common form of correlation. It is a parametric test, and assumes that the data are linearly related and that the residuals are normally distributed.

```
cor.test( ~ Species + Latitude,
          data=Data,
          method = "pearson",
          conf.level = 0.95)
#Pearson's product-moment correlation
t = -2.0225, df = 15, p-value = 0.06134
       cor
-0.4628844
```



Kendall correlation

Kendall rank correlation is a non-parametric test that does not assume a distribution of the data or that the data are linearly related. It ranks the data to determine the degree of correlation.

```
cor.test( ~ Species + Latitude,
          data=Data,
          method = "kendall",
          continuity = FALSE,
          conf.level = 0.95)
Kendall's rank correlation tau
z = -1.3234, p-value = 0.1857
        tau
-0.2388326
```

Spearman correlation

Spearman rank correlation is a non-parametric test that does not assume a distribution of the data or that the data are linearly related. It ranks the data to determine the degree of correlation, and is appropriate for ordinal measurements.

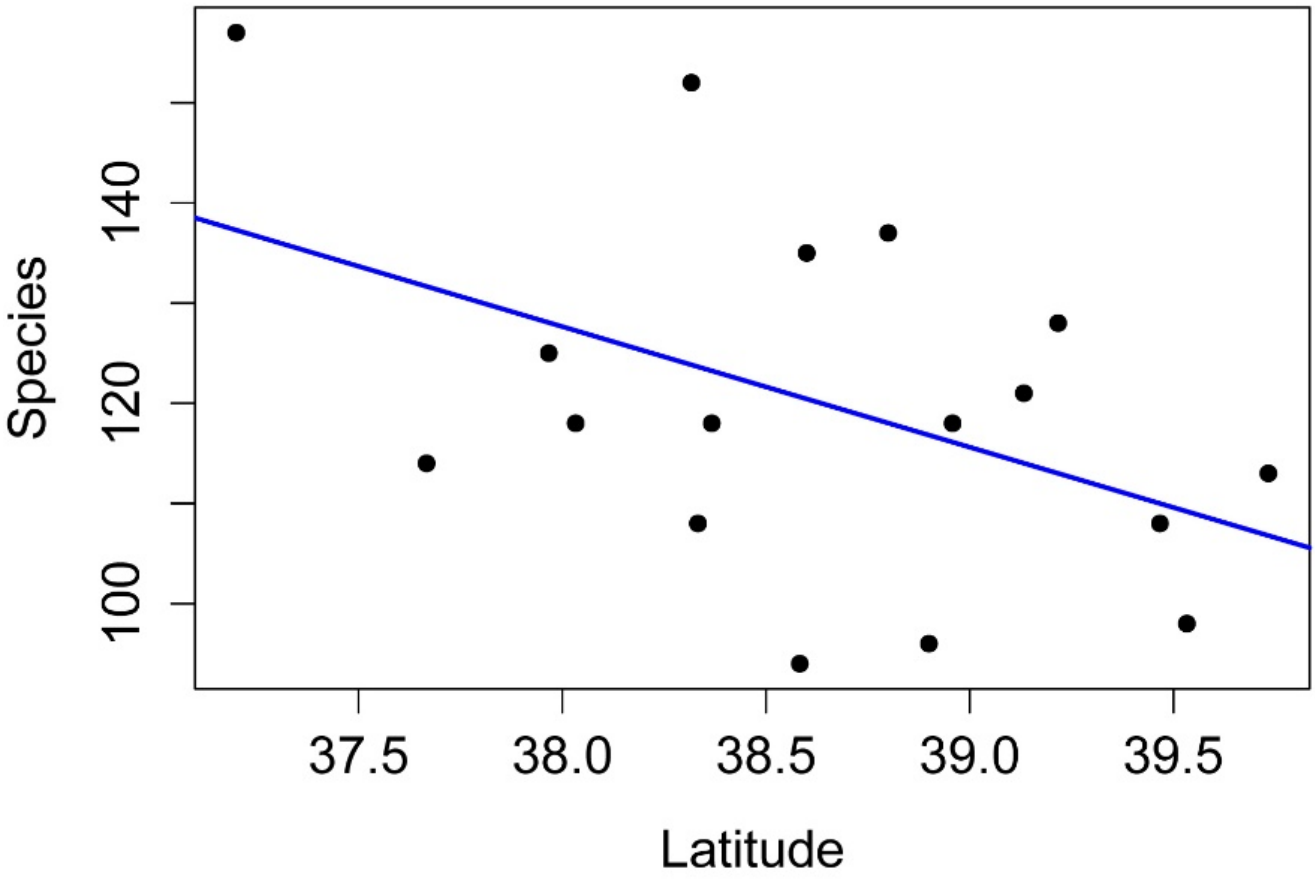
```
cor.test( ~ Species + Latitude,
          data=Data,
          method = "spearman",
          continuity = FALSE,
          conf.level = 0.95)



Spearman's rank correlation rho
S = 1111.908, p-value = 0.1526
       rho
-0.3626323
```

Linear regression

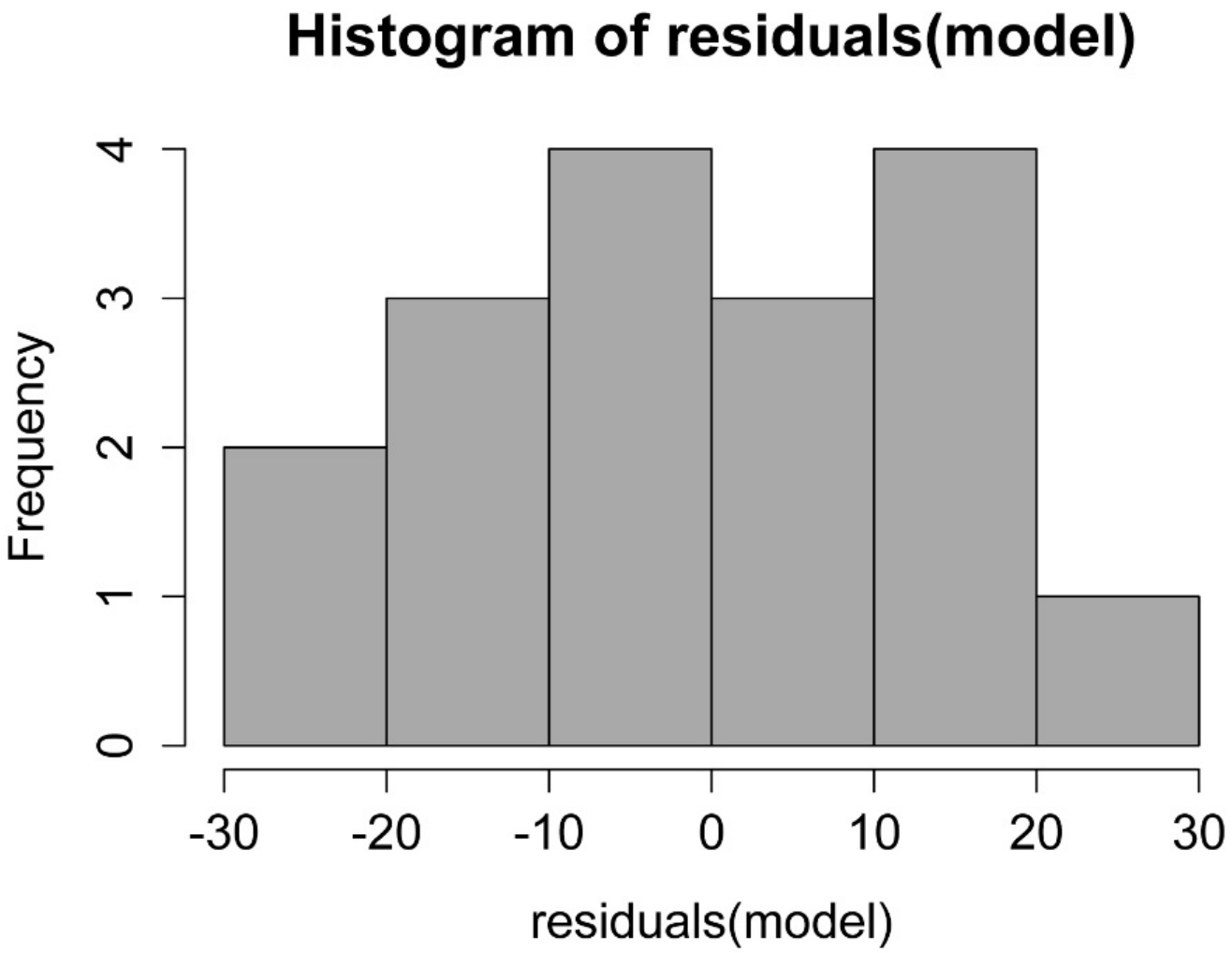
Linear regression can be performed with the *lm* function in the native *stats* package. A robust regression can be performed with the *lmrob* function in the *robustbase* package.

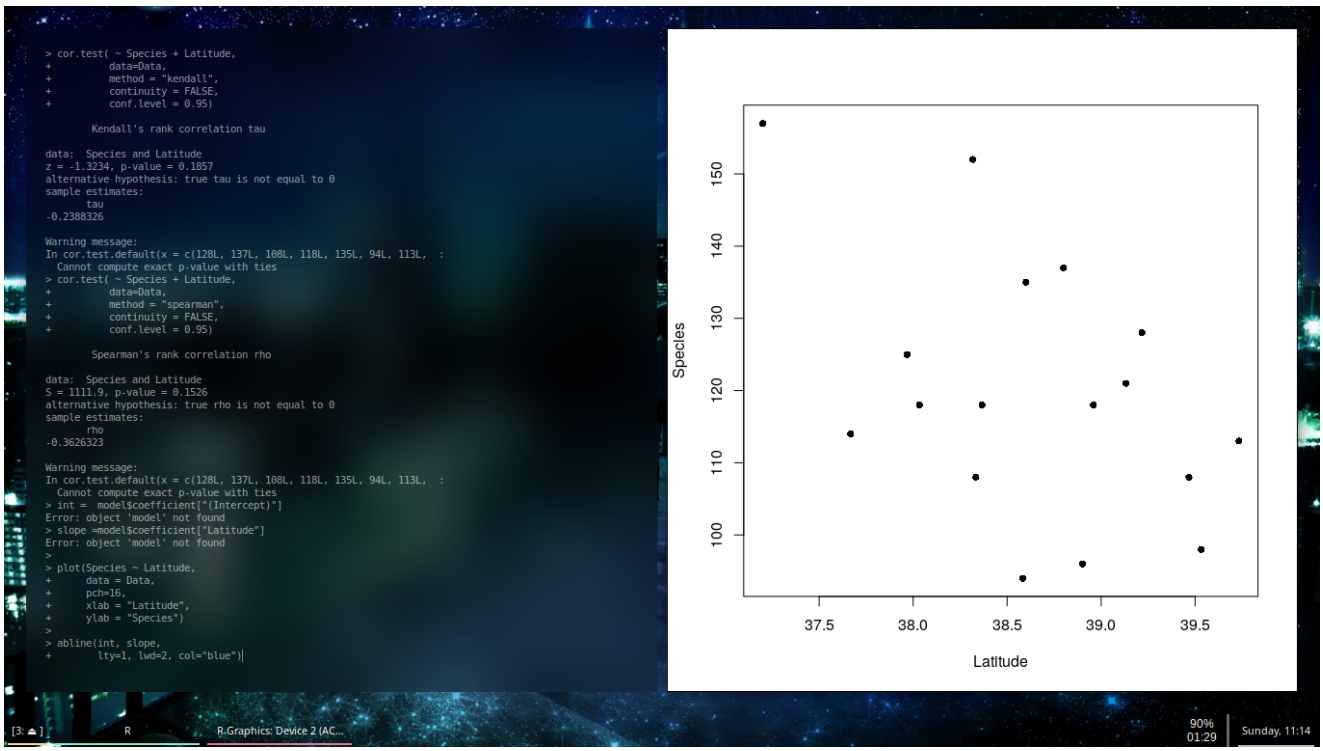
```
model = lm(Species ~ Latitude,
           data = Data)

summary(model)                     # shows parameter estimates,
                                   # p-value for model, r-square
             Estimate Std. Error t value Pr(>|t|)
(Intercept)  585.145    230.024   2.544   0.0225 *

Latitude     -12.039      5.953  -2.022   0.0613 .
Multiple R-squared:  0.2143,  Adjusted R-squared:  0.1619
F-statistic:  4.09 on 1 and 15 DF,  p-value: 0.06134
library(car)
Anova(model, type="II")              # shows p-value for effects in model
Response: Species
          Sum Sq Df F value  Pr(>F)
Latitude  1096.6  1  4.0903 0.06134 .
Residuals 4021.4 15
```

```
int =  model$coefficient["(Intercept)"]
slope =model$coefficient["Latitude"]

plot(Species ~ Latitude,
     data = Data,
     pch=16,
     xlab = "Latitude",
     ylab = "Species")

abline(int, slope,
       lty=1, lwd=2, col="blue")
```




```
hist(residuals(model),
     col="darkgray")
```





Conclusion:

Henceforth, we have successfully studied Basics of correlation and regression in R.

Experiment 10

Aim:

K-means Clustering in R.

Theory:

K Means Clustering is an unsupervised learning algorithm that tries to cluster data based on their similarity. Unsupervised learning means that there is no outcome to be predicted, and the algorithm just tries to find patterns in the data. In k means clustering, we have the specify the number of clusters we want the data to be grouped into. The algorithm randomly assigns each observation to a cluster, and finds the centroid of each cluster. Then, the algorithm iterates through two steps:

- Reassign data points to the cluster whose centroid is closest.
- Calculate new centroid of each cluster.

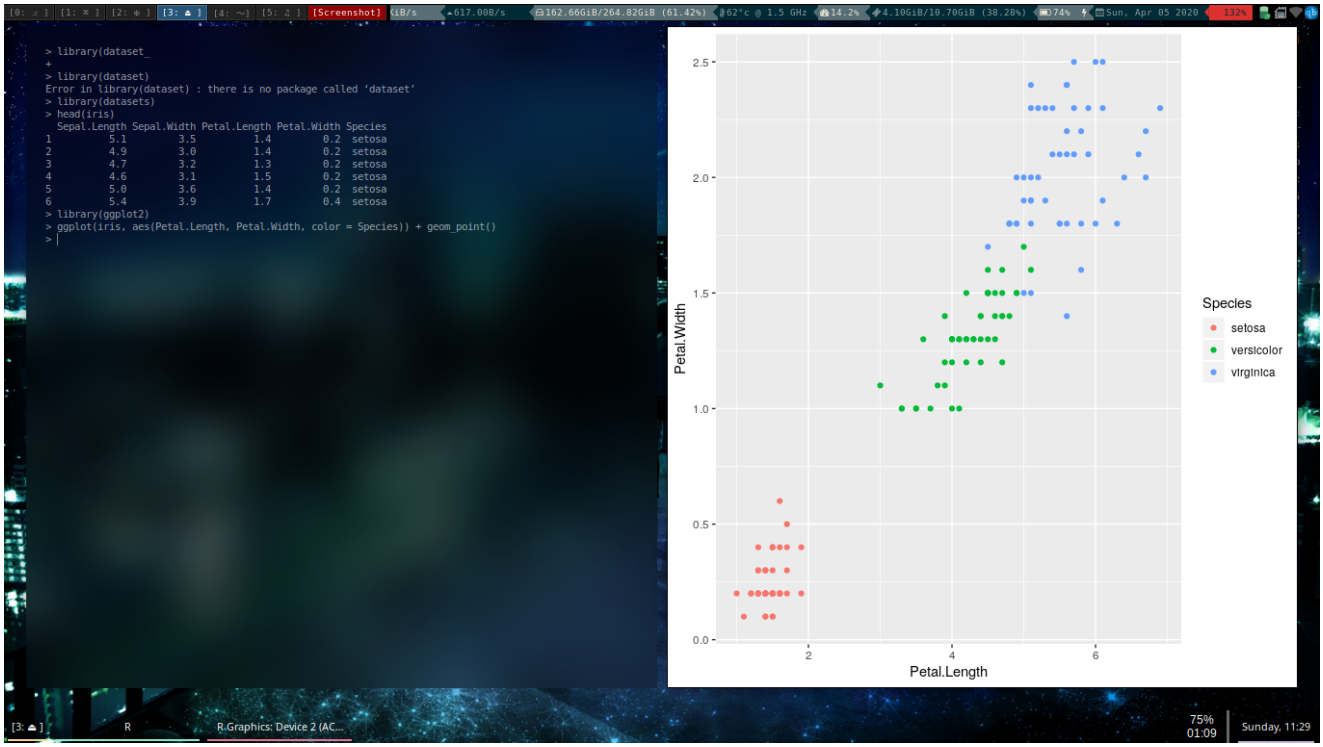
These two steps are repeated till the within cluster variation cannot be reduced any further. The within cluster variation is calculated as the sum of the euclidean distance between the data points and their respective cluster centroids.

Exploring the data

The `iris` dataset contains data about sepal length, sepal width, petal length, and petal width of flowers of different species. Let us see what it looks like

```
library(datasets)
head(iris)
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa
```

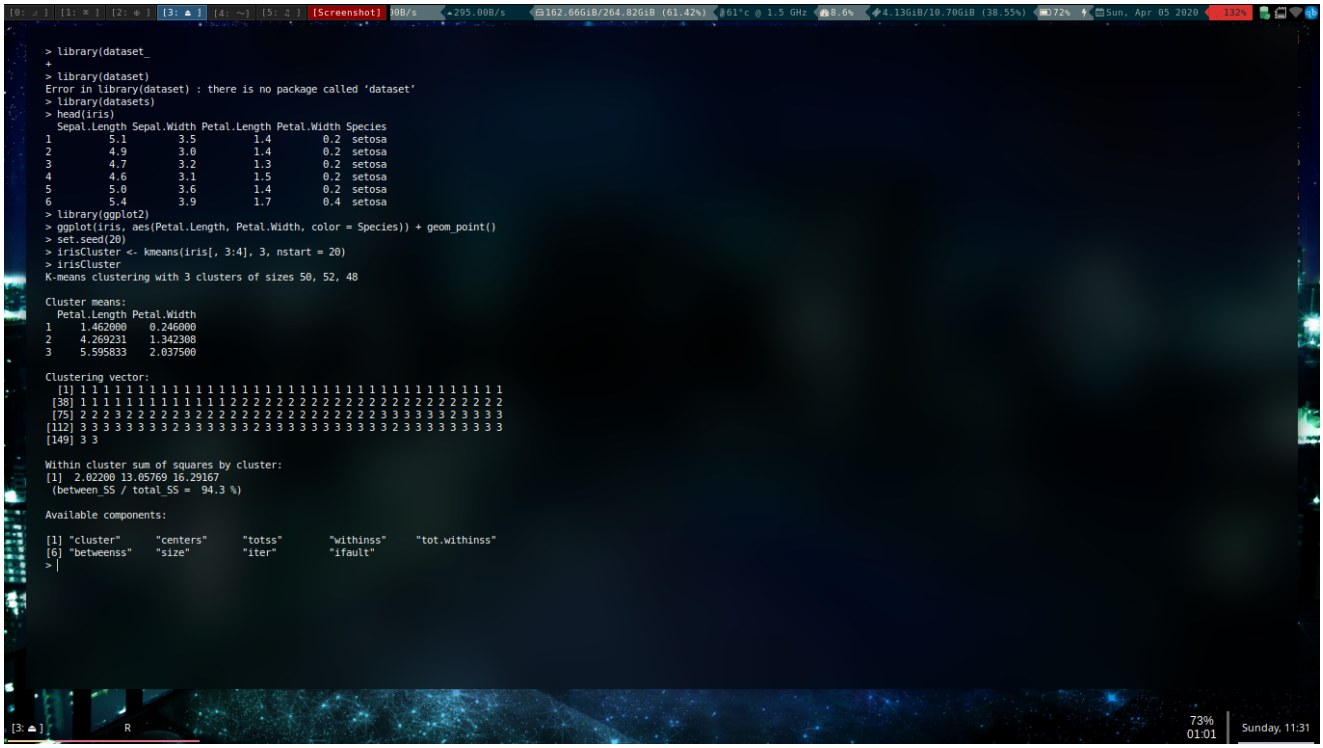
```
library(ggplot2)
ggplot(iris, aes(Petal.Length, Petal.Width, color = Species)) + geom_point()
```



Since we have seen the data, now we shall start and create a cluster of them.

```
set.seed(20)
irisCluster <- kmeans(iris[, 3:4], 3, nstart = 20)
irisCluster
K-means clustering with 3 clusters of sizes 46, 54, 50

Cluster means:
  Petal.Length Petal.Width
1     5.626087    2.047826
2     4.292593    1.359259
3     1.462000    0.246000

Clustering vector:
  [1] 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
 [35] 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 [69] 2 2 2 2 2 2 2 2 2 1 2 2 2 2 2 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 1 1
[103] 1 1 1 1 2 1 1 1 1 1 1 1 1 1 1 1 1 2 1 1 1 2 1 1 2 2 1 1 1 1 1 1 1 1 1 1
[137] 1 1 2 1 1 1 1 1 1 1 1 1 1 1

Within cluster sum of squares by cluster:
[1] 15.16348 14.22741  2.02200
 (between_SS / total_SS =  94.3 %)

Available components:

[1] "cluster"      "centers"      "totss"        "withinss"
[5] "tot.withinss" "betweenss"    "size"         "iter"
[9] "ifault"
```

Since we know that there are 3 species involved, we ask the algorithm to group the data into 3 clusters, and since the starting assignments are random, we specify `nstart = 20`. This means that R will try 20 different random starting assignments and then select the one with the lowest within cluster variation. We can see the cluster centroids, the clusters that each data point was assigned to, and the within cluster variation.



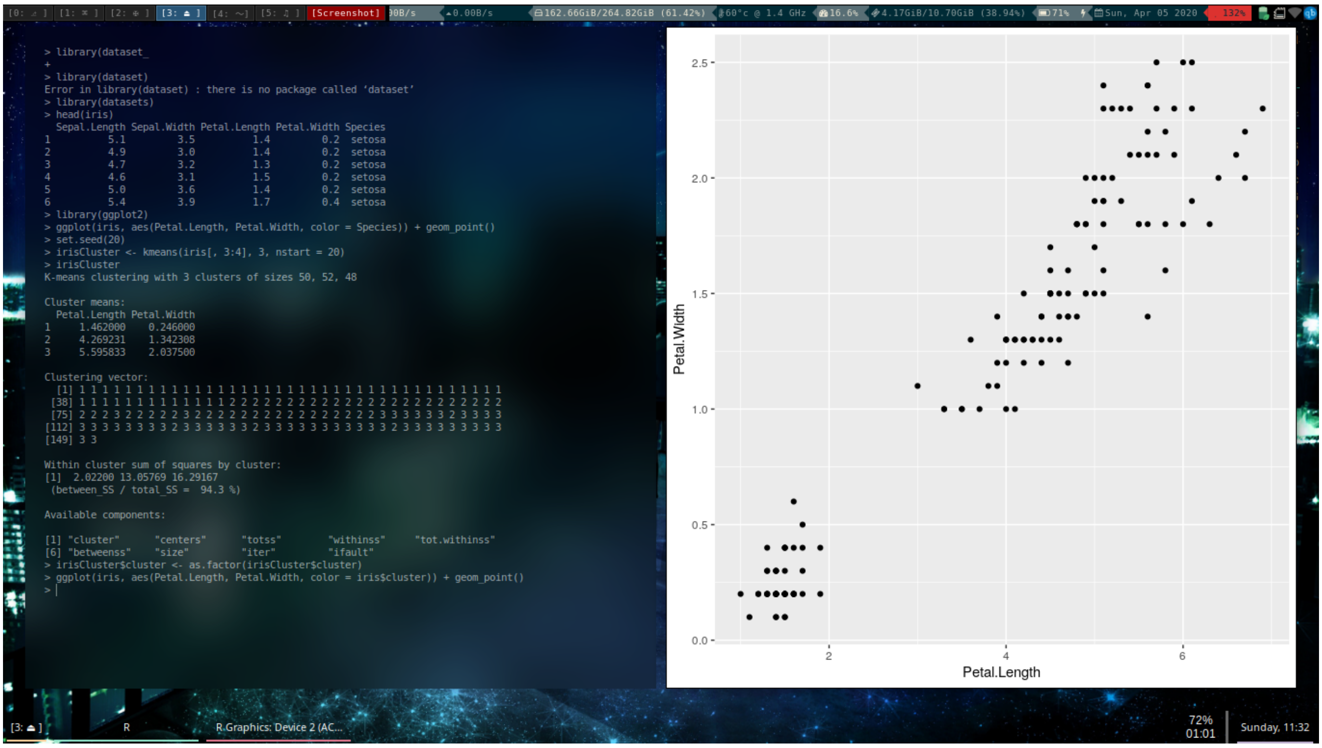
Let us compare the clusters with the species.

```
table(irisCluster$cluster, iris$Species)
    setosa versicolor virginica
  1      0          2        44
  2      0         48         6
  3     50          0         0
```

As we can see, the data belonging to the `setosa` species got grouped into cluster 3, `versicolor` into cluster 2, and `virginica` into cluster 1. The algorithm wrongly classified two data points belonging to `versicolor` and six data points belonging to `virginica`.

We can also plot the data to see the clusters:

```
irisCluster$cluster <- as.factor(irisCluster$cluster)
ggplot(iris, aes(Petal.Length, Petal.Width, color = iris$cluster)) +
geom_point()
```



Conclusion:

Henceforth, we have successfully studied K-means clustering on iris dataset in R.

Experiment 11

Aim:

Classification on large and noisy dataset with R - logistic regression and naive bayes.

Theory:

Logistic regression is yet another technique borrowed by machine learning from the field of statistics. It's a powerful statistical way of modeling a binomial outcome with one or more explanatory variables. It measures the relationship between the categorical dependent variable and one or more independent variables by estimating probabilities using a logistic function, which is the cumulative logistic distribution.

Logistic regression is an instance of classification technique that you can use to predict a qualitative response. More specifically, logistic regression models the probability that belongs to a particular category.

That means that, if you are trying to do gender classification, where the response falls into one of the two categories, `male` or `female`, you'll use logistic regression models to estimate the probability that belongs to a particular category.

Who survived the Titanic disaster?

For illustration, we'll be working on one of the most popular data sets in machine learning: **Titanic**. It's fairly small in size and a variety of variables will give us enough space for creative feature engineering and model building. This data set has been taken from Kaggle. I hope you know that model building is the last stage in machine learning. Therefore, we'll be doing quick data exploration, pre-processing, and feature engineering before implementing Logistic Regression.

```
#set working directory
 path <- "~/SoopaProject/R/"
 setwd(path)

#load libraries and data
 library (data.table)
 library (plyr)
 library (stringr)
 train <- fread("train.csv",na.strings = c("","  ",NA,"NA"))
 test <- fread("test.csv",na.strings = c("","  ",NA,"NA"))
```


```
#check missing values
> colSums(is.na(train))
 colSums(is.na(test))
```

We see that variable `Age` and `Cabin` have missing values. Interestingly, the variable `Fare` has one missing value only in the test set. Let's start exploring each variable individually. For numeric variables, we'll use the `summary` function. For character / factor variables, we'll use table for exploration:

```
#Quick Data Exploration
 summary(train$Age)
 summary(test$Age)

 train[,.N/nrow(train),Pclass]
 test[,.N/nrow(test),Pclass]

 train [,.N/nrow(train),Sex]
 test [,.N/nrow(test),Sex]

 train [,.N/nrow(train),SibSp]
 test [,.N/nrow(test),SibSp]

 train [,.N/nrow(train),Parch]
 test [,.N/nrow(test),Parch] #extra 9

 summary(train$Fare)
 summary(test$Fare)

 train [,.N/nrow(train),Cabin]
 test [,.N/nrow(test),Cabin]

 train [,.N/nrow(train),Embarked]
 test [,.N/nrow(test),Embarked]
```

Following are the insights we can derive from the data exploration above:

1. The variable `Fare` is skewed (right) in nature. We'll have to log transform it such that it resembles a normal distribution.
2. The variable `Parch` has one extra level (9) in the test set as compared to the train set. We'll have to combine it with its mode value.

A smart way to make modifications in train and test data is by combining them. This way, you'll save yourself from writing some extra lines of code. I suggest you follow every line of code carefully and simultaneously check how every line affects the data.

```
#combine data
> alldata <- rbind(train,test,fill=TRUE)
```

I suspect that `Ticket` notation could give us some information. For example, some ticket notation starts with alpha numeric, while others only have numbers. We'll capture this trend using a binary coded variable.

```
#New Variables
#Extract passengers title
> alldata [,title := strsplit(Name,split = "[,.]")]
> alldata [,title := ldply(.data = title,.fun = function(x) x[2])]
> alldata [,title := str_trim(title,side = "left")]

#combine titles
> alldata [,title := replace(title, which(title %in%
c("Capt","Col","Don","Jonkheer","Major","Rev","Sir")), "Mr"),by=title]
> alldata [,title := replace(title, which(title %in%
c("Lady","Mlle","Mme","Ms","the Countess","Dr","Dona")),"Mrs"),by=title]

#ticket binary coding
> alldata [,abs_col := strsplit(x = Ticket,split = " ")]
> alldata [,abs_col := ldply(.data = abs_col,.fun = function(x)length(x))]
> alldata [,abs_col := ifelse(abs_col > 1,1,0)]
```

Next, we'll impute missing values, transform `Fare` variable and remove an extra level from `Parch` variable. This will make our data ready for machine learning.

```
#Impute Age with Median
> for(i in "Age")
        set(alldata,i = which(is.na(alldata[[i]])),j=i,value =
median(alldata$Age,na.rm = T))

#Remove rows containing NA from Embarked
> alldata <- alldata[!is.na(Embarked)]

#Impute Fare with Median
> for(i in "Fare")
      set(alldata,i = which(is.na(alldata[[i]])),j=i,value =
median(alldata$Fare,na.rm = T))

#Replace missing values in Cabin with "Miss"
> alldata [is.na(Cabin),Cabin := "Miss"]

#Log Transform Fare
> alldata$Fare <- log(alldata$Fare + 1)

#Impute Parch 9 to 0
> alldata [Parch == 9L, Parch := 0]
```

The method of using for - set loop for imputing missing values works blazing fast on large data sets. In our case, the data set is small, hence it's difficult to note the difference. Now, our data set is ready. Let's implement Logistic Regression and check our model's accuracy.

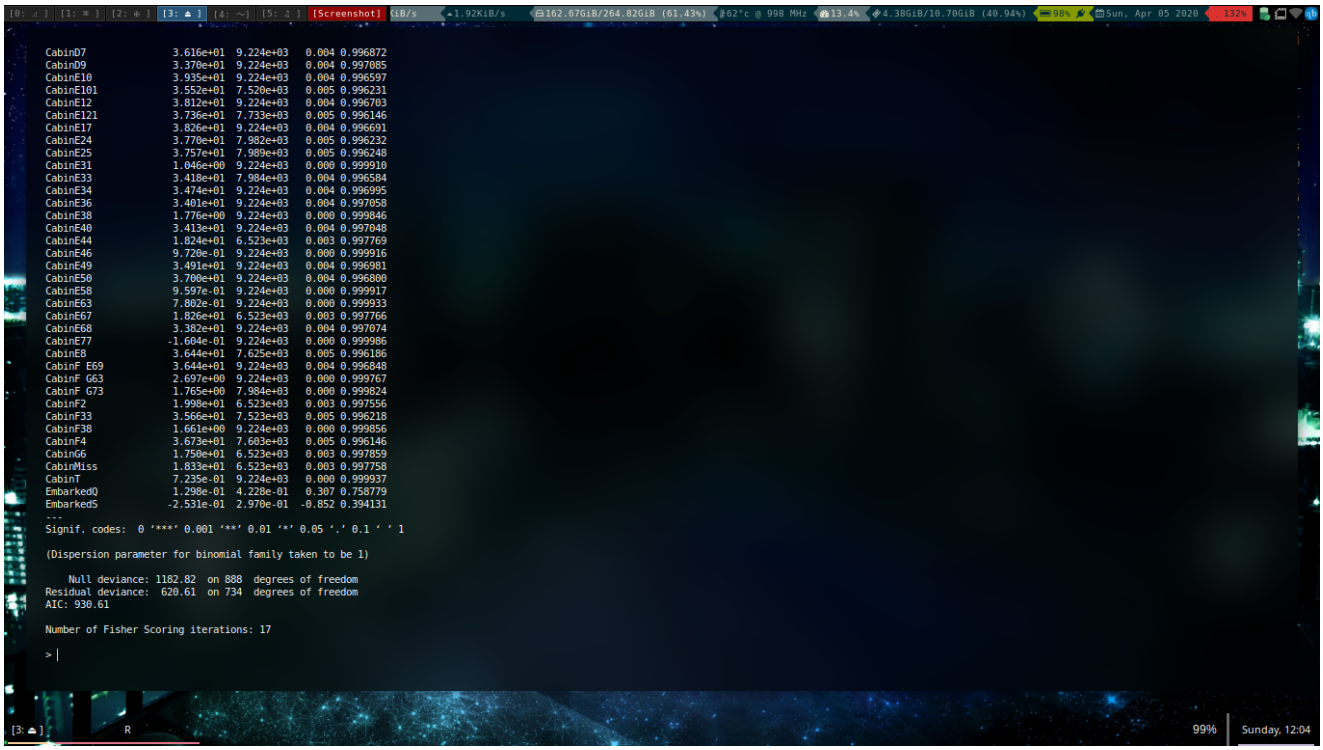
```
#Collect train and test
> train <- alldata[!(is.na(Survived))]
> train [,Survived := as.factor(Survived)]

> test <- alldata[is.na(Survived)]
> test [,Survived := NULL]

#Logistic Regression
> model <- glm(Survived ~ ., family = binomial(link = 'logit'), data =
train[,-c("PassengerId","Name","Ticket")])
> summary(model)
```

In R, you can implement Logistic Regression using the `glm` function. Now, let's understand and interpret the crucial aspects of summary:

1. The glm function internally encodes categorical variables into n - 1 distinct levels.
2. Estimate represents the regression coefficients value. Here, the regression coefficients explain the change in log(odds) of the response variable for one unit change in the predictor variable.
3. Std. Error represents the standard error associated with the regression coefficients.
4. z value is analogous to t-statistics in multiple regression output. z value > 2 implies the corresponding variable is significant.
5. p value determines the probability of significance of predictor variables. With 95% confidence level, a variable having p < 0.05 is considered an important predictor. The same can be inferred by observing stars against p value.

In addition, we can also perform an ANOVA Chi-square test to check the overall effect of variables on the dependent variable.

```
#run anova
> anova(model, test = 'Chisq')
```



we can also perform an ANOVA Chi-square test to check the overall effect of variables on the dependent variable.

```
#run anova
> anova(model, test = 'Chisq')
```

(**TOP LEFT**)

Following are the insights we can collect for the output above:

1. In the presence of other variables, variables such as Parch, Cabin, Embarked, and abs_col are not significant. We'll try building another model without including them.
2. The AIC value of this model is 883.79.

Let's create another model and try to achieve a lower AIC value.

```
> model2 <- glm(Survived ~ Pclass + Sex + Age + SibSp + Fare + title, data =
train,family = binomial(link="logit"))
> summary(model2)
```

As you can see, we've achieved a lower AIC value and a better model. Also, we can compare both the models using the ANOVA test. Let's say our null hypothesis is that second model is better than the first model. p < 0.05 would reject our hypothesis and in case p > 0.05, we'll fail to reject the null hypothesis.

```
#compare two models
> anova(model,model2,test = "Chisq")
```

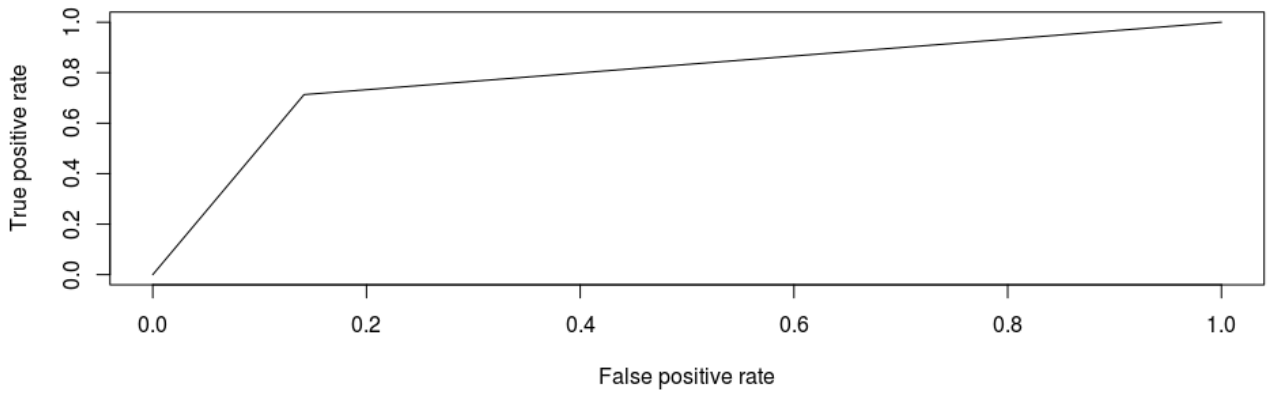
```
Analysis of Deviance Table

Model 1: Survived ~ Pclass + Sex + Age + SibSp + Parch + Fare + Cabin +
    Embarked + title + abs_col
Model 2: Survived ~ Pclass + Sex + Age + SibSp + Fare + title
  Resid. Df Resid. Dev   Df Deviance Pr(&gt;Chi)
1       730     565.79
2       880     735.80 -150     -170    0.126
```

With p > 0.05, this ANOVA test also corroborates the fact that the second model is better than first model. Let's predict on unseen data now. Since, we can't evaluate a model's performance on test data locally, we'll divide the train set and use model 2 for prediction.

```
#partition and create training, testing data
> library(caret)
> split <- createDataPartition(y = train$Survived,p = 0.6,list = FALSE)

> new_train <- train[split]
> new_test <- train[-split]

#model training and prediction
> log_model <- glm(Survived ~ Pclass + Sex + Age + SibSp + Fare + title,
data = new_train[,-c("PassengerId","Name","Ticket")],family =
binomial(link="logit"))
> log_predict <- predict(log_model,newdata = new_test,type = "response")
> log_predict <- ifelse(log_predict > 0.5,1,0)
```

For now, I've set the probability threshold value as 0.5. Let's get the flavor of our model's accuracy. We'll use AUC-ROC score to determine model fit. Higher the score, better the model. You can also use confusion matrix to determine accuracy using `confusionMatrix` function from caret package.

```
#plot ROC
> library(ROCR)
> library(Metrics)
> pr <- prediction(log_predict,new_test$Survived)
> perf <- performance(pr,measure = "tpr",x.measure = "fpr")
> plot(perf) > auc(new_test$Survived,log_predict) #0.76343
```

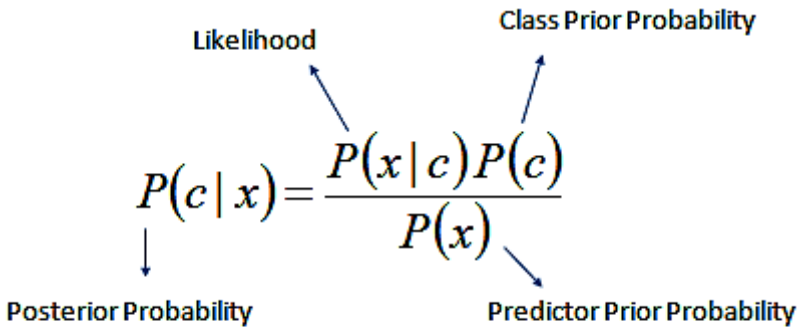


Our AUC score is 0.763. As said above, in ROC plot, we always try to move up and top left corner. From this plot, we can interpret that the model is predicting more negative values

incorrectly. To move up, let's increase our threshold value to 0.6 and check the model's performance.

```
> log_predict <- predict(log_model,newdata = new_test,type = "response")
> log_predict <- ifelse(log_predict > 0.6,1,0)

> pr <- prediction(log_predict,new_test$Survived)
> perf <- performance(pr,measure = "tpr",x.measure = "fpr")
> plot(perf)
> auc(new_test$Survived,log_predict)` ` #0.8008
```

Naïve Bayes Classifier



$$P(c \mid X) = P(x_1 \mid c) \times P(x_2 \mid c) \times \cdots \times P(x_n \mid c) \times P(c)$$

The ***Naïve Bayes classifier*** is a simple probabilistic classifier which is based on Bayes theorem but with strong assumptions regarding independence. Historically, this technique became popular with applications in email filtering, spam detection, and document categorization. Although it is often outperformed by other techniques, and despite the naïve design and oversimplified assumptions, this classifier can perform well in many complex real-world problems. And since it is a resource efficient algorithm that is fast and scales well, it is definitely a machine learning algorithm to have in your toolkit.

(**installing naivebayes**)

```
> install.package("naivebayes")
Error in install.package("naivebayes") :
  could not find function "install.package"
> install.packages("naivebayes")
Installing package into '/home/akuma/R/x86_64-pc-linux-gnu-library/3.4'
(as 'lib' is unspecified)
trying URL 'https://cloud.r-project.org/src/contrib/naivebayes_0.9.7.tar.gz'
Content type 'application/x-gzip' length 705994 bytes (689 KB)
==================================================
downloaded 689 KB

* installing *source* package 'naivebayes' ...
** package 'naivebayes' successfully unpacked and MD5 sums checked
** R
** inst
** preparing package for lazy loading
** help
*** installing help indices
*** copying figures
** building package indices
** installing vignettes
** testing if installed package can be loaded
* DONE (naivebayes)

The downloaded source packages are in
        '/tmp/RtmpdkVc47/downloaded_packages'
> library(naivebayes)
naivebayes 0.9.7 loaded

Attaching package: 'naivebayes'

The following object is masked from 'package:data.table':

    tables

>
> # Simulate example data
> n <- 100
> set.seed(1)
> data <- data.frame(class = sample(c("classA", "classB"), n, TRUE),
+                    bern = sample(LETTERS[1:2], n, TRUE),
+                    cat  = sample(letters[1:3], n, TRUE),
+                    logical = sample(c(TRUE,FALSE), n, TRUE),
+                    norm = rnorm(n),
+                    count = rpois(n, lambda = c(5,15)))
> train <- data[1:95, ]
> test <- data[96:100, -1]
> summary(train)
    class     bern    cat     logical             norm               count
 classA:50   A:44   a:42   Mode :logical   Min.   :-2.888921   Min.   : 1.000
```

```
** preparing package for lazy loading
** help
*** installing help indices
*** copying figures
** building package indices
** installing vignettes
** testing if installed package can be loaded
* DONE (naivebayes)

The downloaded source packages are in
        '/tmp/RtmpdkVc47/downloaded_packages'
> library(naivebayes)
naivebayes 0.9.7 loaded

Attaching package: 'naivebayes'

The following object is masked from 'package:data.table':

    tables

>
> # Simulate example data
> n <- 100
> set.seed(1)
> data <- data.frame(class = sample(c("classA", "classB"), n, TRUE),
+                    bern = sample(LETTERS[1:2], n, TRUE),
+                    cat  = sample(letters[1:3], n, TRUE),
+                    logical = sample(c(TRUE,FALSE), n, TRUE),
+                    norm = rnorm(n),
+                    count = rpois(n, lambda = c(5,15)))
> train <- data[1:95, ]
> test <- data[96:100, -1]
> summary(train)
    class     bern    cat     logical             norm               count
 classA:50   A:44   a:42   Mode :logical   Min.   :-2.888921   Min.   : 1.000
 classB:45   B:51   b:31   FALSE:42        1st Qu.:-0.562702   1st Qu.: 5.000
                    c:22   TRUE :53        Median :-0.005344   Median : 9.000
                                           Mean   : 0.015458   Mean   : 9.926
                                           3rd Qu.: 0.681308   3rd Qu.:15.500
                                           Max.   : 2.649167   Max.   :22.000

> summary(test)
 bern   cat     logical             norm               count
 A:2   a:3   Mode :logical   Min.   :-0.30582   Min.   : 4.0
 B:3   b:0   FALSE:1         1st Qu.:-0.05057   1st Qu.: 7.0
       c:2   TRUE :4         Median : 0.13445   Median :10.0
                            Mean   : 0.29976   Mean   : 8.8
                            3rd Qu.: 0.76560   3rd Qu.:11.0
                            Max.   : 0.95514   Max.   :12.0
>
```

Conclusion:

Henceforth, we have successfully understood the logistic regression and naive bayes.