MODULE 2

READING AND WRITING DATA

Reading Data from Text Files

One of the important formats to store a file is in a text file. R provides various methods that one can read data from a text file. **read.delim()**: This method is used for reading "tab-separated value" files (".txt"). By default, point (".") is used as decimal point.

- ***Syntax:*** *read.delim(file, header = TRUE, sep = "\t", dec = ".", …)*
- Eg: # Read a text file using read.delim()

```
myData = read.delim("Testing.txt", header = FALSE)
        print(myData)
```

read.delim2(): This method is used for reading "tab-separated value" files (".txt"). By default, point (",") is used as decimal points.

- ***Syntax:*** *read.delim2(file, header = TRUE, sep = "\t", dec = ",", …)*
- # Read a text file using read.delim()

  ```
  myData1 = read.delim2("Test2.txt", header = FALSE)    print(myData1)
  ```

file.choose(): In R it's also possible to choose a file interactively using the function **file.choose()**, and if you're a beginner in R programming then this method is very useful for you.

- myFile = read.delim(file.choose(), header = FALSE)

```
# If you use the code above in Rstudio
```

  - # you will be asked to choose a file

```
print(myFile)
```

Reading CSVs

The best way to read data from a CSV file1 is to use **read.table()**

We can also use **read.csv**, which is a wrapper around **read.table** with the sep argument preset to a comma (,). The result of using **read.table** is a data.frame. The first argument to read.table is the full path of the file to be loaded. The file can be sitting on disk or even the Web.

- tomato <- **read.table**(file=”TomatoFirst.csv”, header=TRUE, sep=",")
- head(tomato)

- data <- read.csv("tested.csv", header = FALSE, sep = "\t")
- print(data)

Exporting Data

When a program is terminated, the entire data is lost. Storing in a file will preserve one’s data even if the program terminates. If one has to enter a large number of data, it will take a lot of time to enter them all. However, if one has a file containing all the data, he/she can easily access the contents of the file using a few commands in R. One can easily move his data from one computer to another without any changes. So those files can be stored in various formats. It may be stored in .txt(tab-separated value) file, or in a tabular format i.e .csv(comma-separated value) file or it may be on the internet or cloud. R provides very easy methods to export data to those files

Exporting data to a text file

One of the important formats to store a file is in a text file. R provides various methods that one can export data to a text file.

- **write.table()**:

The R base function write.table() can be used to export a data frame or a matrix to a text file.

Syntax: write.table(x, file, append = FALSE, sep = ” “, dec = “.”, row.names = TRUE, col.names = TRUE)

- **write_tsv()**:

This write_tsv() method is also used for to export data to a tab separated (“\t”) values by using the help of **readr** package.

Exporting data to a csv file

Another popular format to store a file is in a csv(comma-separated value) format. R provides various methods that one can export data to a csv file.

- **write.table()**:

The R base function write.table() can also be used to export a data frame or a matrix to a csv file.

Syntax: write.table(x, file, append = FALSE, sep = ” “, dec = “.”, row.names = TRUE, col.names = TRUE)

- **write.csv()**:

This write.csv() method is recommendable for exporting data to a csv file. It uses “.” for the decimal point and a comma (“, ”) for the separator.

- **write.csv2()**:

This method is much similar as write.csv() but it uses a comma (“, ”) for the decimal point and a semicolon (“;”) for the separator.

Reading from Databases

Databases arguably store the vast majority of the world’s data. Most of these, whether they be PostgreSQL, MySQL, Microsoft SQL Server or Microsoft Access, can be accessed either through various drivers, typically via an Open Data Base Connectivity (ODBC) connection. The most popular open-source databases have packages such as RPostgreSQL and RMySQL. Other databases without a specific package can make use of the more generic, and aptly named, RODBC package. Database connectivity can be difficult, so the DBI package was written to create a uniform experience while working with different databases.

Setting up a database is beyond the scope of this book so we use a simple SQLite database though these steps will be similar for most databases. First, we download the database file2 using download.file.

```
> download.file("http://www.jaredlander.com/data/diamonds.db",
+ destfile = "data/diamonds.db", mode='wb')
```

Since SQLite has its own R package, RSQLite, we use that to connect to our database, otherwise we would use RODBC.

```
> library(RSQLite)
```

To connect to the database we first specify the driver using dbDriver. The function's main argument is the type of driver, such as "SQLite" or "ODBC."

```
> drv <- dbDriver('SQLite')
> class(drv)

[1] "SQLiteDriver"

attr(,"package")

[1] "RSQLite"
```

We then establish a connection to the specific database with dbConnect. The first argument is the driver. The most common second argument is the DSN3 connection string for the database, or the path to the file for SQLite databases. Additional arguments are typically the database username, password, host and port.

```
> con <- dbConnect(drv, 'data/diamonds.db')
> class(con)
[1] "SQLiteConnection"
attr(,"package")
[1] "RSQLite"
```

Now that we are connected to the database we can learn more about the database, such as the table names and the fields within tables, using functions from the DBI package.

```
> dbListTables(con)
[1] "DiamondColors" "diamonds" "sqlite_stat1"
> dbListFields(con, name='diamonds')
[1] "carat" "cut" "color" "clarity" "depth" "table"
[7] "price" "x" "y" "z"
> dbListFields(con, name='DiamondColors')
[1] "Color" "Description" "Details"
```

At this point we are ready to run a query on that database using dbGetQuery. This can be any valid SQL query of arbitrary complexity. dbGetQuery returns an ordinary data.frame, just like any other.

```
> # simple SELECT * query from one table
> diamondsTable <- dbGetQuery(con,
+ "SELECT * FROM diamonds",
+ stringsAsFactors=FALSE)
> # simple SELECT * query from one table
> colorTable <- dbGetQuery(con,
+ "SELECT * FROM DiamondColors",
+ stringsAsFactors=FALSE)
> # do a join between the two tables
> longQuery <- "SELECT * FROM diamonds, DiamondColors
WHERE
diamonds.color = DiamondColors.Color"
> diamondsJoin <- dbGetQuery(con, longQuery,
stringsAsFactors=FALSE)
```

We can easily check the results of these queries by viewing the resulting data.frames.

```
>head(diamondsTable)

carat cut color clarity depth table price x y z
1 0.23 Ideal E SI2 61.5 55 326 3.95 3.98 2.43
2 0.21 Premium E SI1 59.8 61 326 3.89 3.84 2.31
3 0.23 Good E VS1 56.9 65 327 4.05 4.07 2.31
4 0.29 Premium I VS2 62.4 58 334 4.20 4.23 2.63
5 0.31 Good J SI2 63.3 58 335 4.34 4.35 2.75
6 0.24 Very Good J VVS2 62.8 57 336 3.94 3.96 2.48

> head(colorTable)
Color Description Details
1 D Absolutely Colorless No color
2 E Colorless Minute traces of color
3 F Colorless Minute traces of color
4 G Near Colorless Color is dificult to detect
5 H Near Colorless Color is dificult to detect
6 I Near Colorless Slightly detectable color
> head(diamondsJoin)
carat cut color clarity depth table price x y z
1 0.23 Ideal E SI2 61.5 55 326 3.95 3.98 2.43
2 0.21 Premium E SI1 59.8 61 326 3.89 3.84 2.31
3 0.23 Good E VS1 56.9 65 327 4.05 4.07 2.31
4 0.29 Premium I VS2 62.4 58 334 4.20 4.23 2.63
5 0.31 Good J SI2 63.3 58 335 4.34 4.35 2.75
6 0.24 Very Good J VVS2 62.8 57 336 3.94 3.96 2.48
Color Description Details
1 E Colorless Minute traces of color
2 E Colorless Minute traces of color
3 E Colorless Minute traces of color
4 I Near Colorless Slightly detectable color
5 J Near Colorless Slightly detectable color
6 J Near Colorless Slightly detectable color
```

While it is not necessary, it is good practice to close the ODBC connection using dbDisconnect, although it will close automatically when either R closes or we open another connection using dbConnect. Only one connection may be open at a time.

Missing Data- NA, NULL

Missing data is typically represented by the special values NA or NULL. These values indicate the absence of a value or an unknown value for a particular observation in a dataset. Dealing with missing data is an important aspect of data analysis, as it can impact the accuracy and reliability of the results.

NA (Not Available):

1.NA is used in R to represent missing or unavailable data.
2. It is a reserved value in R that signifies the absence of a value for a particular data point.
3. NA can occur in different data types, including numeric, character, logical, and factors.
4. NA is often used in functions and operations that require handling missing values, such as calculations or data manipulations.

Example

```
# Creating a vector with missing values (NA)
x <-c(1, 2, NA, 4, 5)
# Checking for NA values
is.na(x)
```

NULL:

NULL is a special object in R that represents the absence of any object or value.
1.It is commonly used to indicate the absence of a variable or an empty object.
2.Unlike NA, which is a placeholder for missing data, NULL indicates the absence of any data or object itself.
3.NULL is often used in programming contexts where the absence of an object is needed, such as removing variables or freeing up memory.

Example

```
# Creating a NULL object
y <-NULL
# Checking if an object is NULL
is.null(y)
```

Handling Missing Data in R:

Checking for Missing Values:

The function is.na() is used to identify NA values in a dataset. It returns a logical vector indicating the presence of missing values.

Dealing with Missing Values:

Depending on the analysis and the nature of missing data, there are several approaches to handle missing values, including:

1. Removing observations with missing values using functions like na.omit().

If the missing values are relatively few and randomly distributed, removing observations with missing values can be a viable option. The na.omit() function removes rows with any missing values from a data frame.

Example

```
# Create a data frame with missing values
df <-data.frame(A = c(1, 2, NA, 4, 5),
B = c(NA, 2, 3, 4, NA))
# Remove rows with missing values
df_clean <-na.omit(df)
```

2. Replacing missing values with appropriate values, such as the mean, median, or mode, using functions like na.mean(), na.aggregate(), or impute() from the imputeTS package.

When the missing values are few and replacing them with summary statistics is reasonable, mean, median, or mode can be summary statistics is reasonable, mean, median, or mode can be used as replacements.

Functions like mean(), median(), and Mode() (custom function) can be used along with the is.na() function to replace missing values.

Example

```
# Replace missing values with mean
df$A[is.na(df$A)] <-mean(df$A, na.rm = TRUE)
# Replace missing values with median
df$B[is.na(df$B)] <-median(df$B, na.rm = TRUE)
```

3. Utilizing advanced techniques like multiple imputation or model-based imputation for more sophisticated missing data handling.

Example

```
# Install and load the imputeTS package
install.packages("imputeTS")
library(imputeTS)
# Impute missing values using linear interpolation
df_imputed <-na.interpolation(df)
```

It's essential to handle missing data appropriately, as omitting or mishandling missing values can lead to biased results or loss of valuable information in the analysis. The choice of handling missing data depends on the specific context and the goals of the analysis.

Combining data sets

Sometimes the data we need might be in different datasets and we will have to combine those datasets to get the results we need.

Pasting Together Data Structures:
R provides several functions that allow you to paste together multiple data structures into a single structure.
Paste
The simplest of these functions is paste. The paste function allows you to concatenate multiple character vectors into a single vector. (If you concatenate a vector of another type, it will be coerced to a character vector first.)

```
> x <- c("a","b","c","d","e")
> y <- c("A","B","C","D","E")
> paste(x,y)
[1] "a A" "b B" "c C" "d D" "e E"
```

By default, values are separated by a space; you can specify another separator (or none at all) with the sep argument:

```
> paste(x,y,sep="-")
[1] "a-A" "b-B" "c-C" "d-D" "e-E"
```

If you would like all of the values in the returned vector to be concatenated with each other (to return just a single value), then specify a value for the collapse argument. The value of collapse will be used as the separator in this value:

```
> paste(x,y,sep="-",collapse="#")
[1] "a-A#b-B#c-C#d-D#e-E"
```

rbind and cbind

Sometimes, you would like to bind together multiple data frames or matrices. You can do this with the rbind and cbind functions. The cbind function will combine objects by adding columns. You can picture this as combining two tables horizontally.

As an example, let's start with the data frame for the top five salaries in the NFL in 2008:*

```
> top.5.salaries
   name.las  name.first   team       position   salary
1 Manning   Peyton        Colts      QB          18700000
2 Brady       Tom             Patriots   QB          14626720
3 Pepper     Julius          Panthers DE         14137500
4 Palmer     Carson         Bengals   QB         13980000
5 Manning   Eli                Giants     QB         12916666
```

Now, let's create a new data frame with two more columns (a year and a rank):

```
> year <- c(2008,2008,2008,2008,2008)
> rank <- c(1,2,3,4,5)
> more.cols <- data.frame(year,rank)
> more.cols

   year   rank
1 2008   1
2 2008   2
3 2008   3
4 2008   4
5 2008   5
```

Finally, let's put together these two data frames:

```
> cbind(top.5.salaries,more.cols)
  name.last    name.first      team      position   salary       year   rank
1 Manning      Peyton           Colts      QB         18700000  2008   1
2 Brady          Tom               Patriots   QB        14626720  2008   2
3 Pepper       Julius             Panthers  DE        14137500  2008   3
```

```
4 Palmer      Carson          Bengals   QB      13980000  2008   4
5 Manning     Eli             Giants     QB     12916666  2008   5
```

The rbind function will combine objects by adding rows. You can picture this as combining two tables vertically.

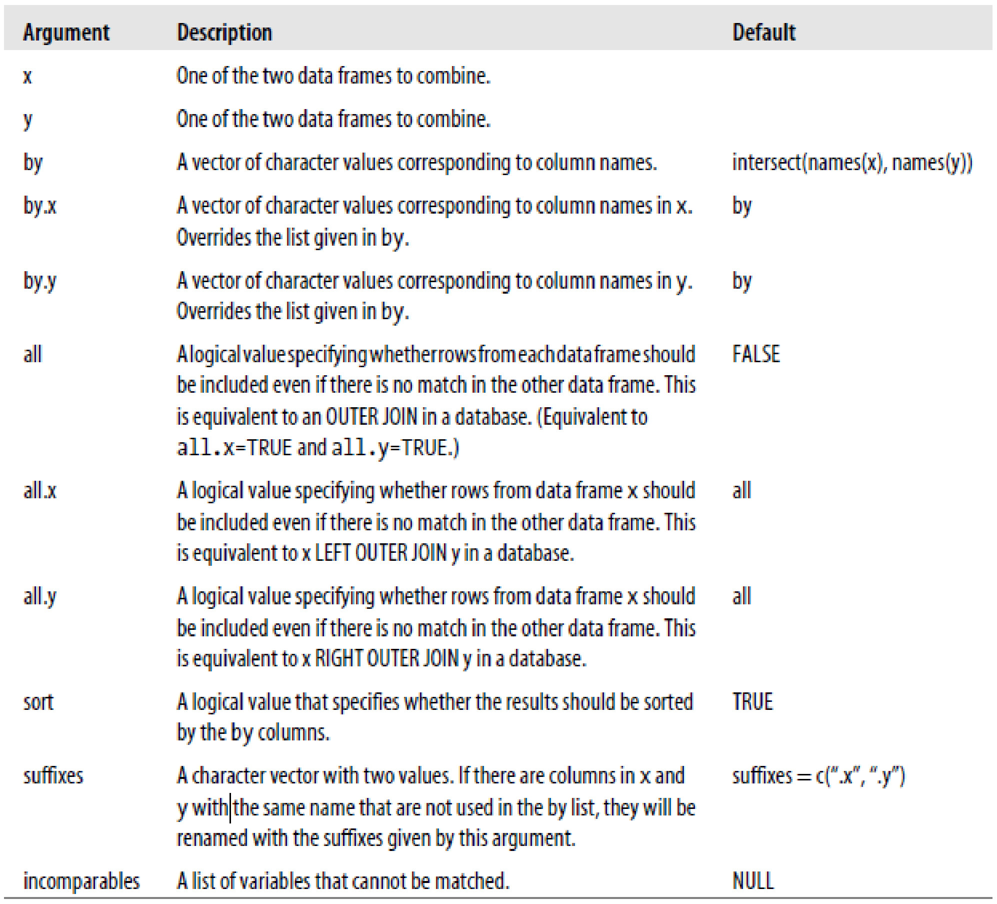
Merging Data by Common Fields:

Suppose that you wanted to show batting statistics for each player along with his name and age. To do this, you would need to merge data from the two tables. In R, you can do this with the merge function:

```
> batting <- dbGetQuery(con, "SELECT * FROM Batting")
> master <- dbGetQuery(con, "SELECT * FROM Master")
> batting.w.names <- merge(batting,master)
```

By default, merge uses common variables between the two data frames as the merge keys. So, in this case, we did not have to specify any more arguments to merge.

```
merge(x, y, by = , by.x = , by.y = , all = , all.x = , all.y = ,
sort = , suffixes = , incomparables = , ...)
```



Argument	Description	Default
x	One of the two data frames to combine.	
y	One of the two data frames to combine.	
by	A vector of character values corresponding to column names.	intersect(names(x), names(y))
by.x	A vector of character values corresponding to column names in x. Overrides the list given in by.	by
by.y	A vector of character values corresponding to column names in y. Overrides the list given in by.	by
all	A logical value specifying whether rows from each data frame should be included even if there is no match in the other data frame. This is equivalent to an OUTER JOIN in a database. (Equivalent to `all.x`=TRUE and `all.y`=TRUE.)	FALSE
all.x	A logical value specifying whether rows from data frame x should be included even if there is no match in the other data frame. This is equivalent to x LEFT OUTER JOIN y in a database.	all
all.y	A logical value specifying whether rows from data frame x should be included even if there is no match in the other data frame. This is equivalent to x RIGHT OUTER JOIN y in a database.	all
sort	A logical value that specifies whether the results should be sorted by the by columns.	TRUE
suffixes	A character vector with two values. If there are columns in x and y with the same name that are not used in the by list, they will be renamed with the suffixes given by this argument.	suffixes = c(".x", ".y")
incomparables	A list of variables that cannot be matched.	NULL

Data Transformations in R

Data transformations refer to the process of modifying the structure, format, or values of a dataset to make it suitable for analysis or to meet specific requirements. These transformations can involve changing variable types, reordering data, creating new variables, or applying mathematical or statistical operations to the data.

There are various techniques and functions available in R for performing data transformations. Here are some commonly used ones:

1. Changing Variable Types:
   as.numeric(): Converts variables to numeric format.
   as.character(): Converts variables to character format. as.factor(): Converts variables to factor format.
2. Reordering and Sorting Data:
   order(): Sorts a vector or a data frame by one or more variables.
   sort(): Sorts a vector or a data frame.
   arrange() (from the dplyr package): Sorts rows of a data frame based on one or more variables.
3. Creating New Variables:
   Using assignment (<-or =) to create new variables based on existing variables or calculations.
   mutate() (from the dplyr package): Adds new variables to a data frame based on calculations using existing variables.
4. Mathematical and Statistical Operations:
   Arithmetic operations: +, -, *, /, etc.
   Statistical functions: mean(), sum(), min(), max(), sd(), var(), etc.
5. Reshaping Data:
   reshape() or melt() (from the reshape2 package): Restructures data from wide to long format or vice versa.
   gather() or pivot_longer() (from the tidyr package): Converts data from wide to long format.
   spread() or pivot_wider() (from the tidyr package): Converts data from long to wide format.
6. Data Aggregation and Grouping:

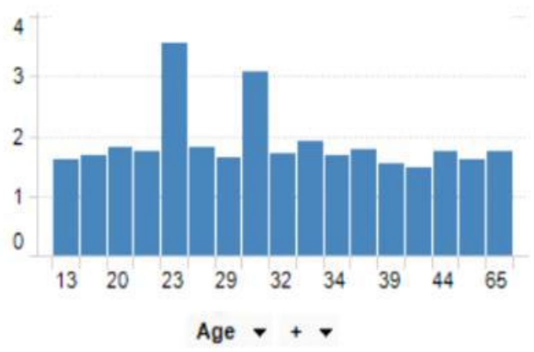
aggregate(): Performs aggregation operations (e.g., sum, mean) on subsets of data based on one or more grouping variables.
group_by() (from the dplyr package): Groups data by one or more variables for subsequent analysis.

Binning Data

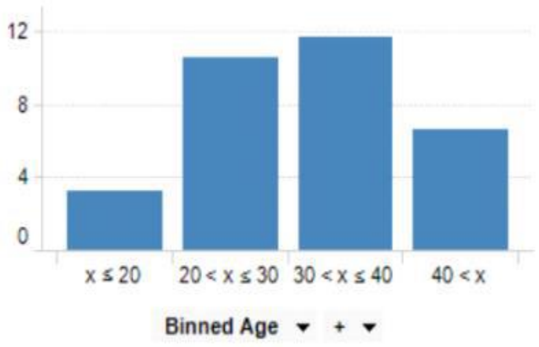
Data binning, or bucketing, is a process used to minimize the effects of observation errors. It is the process of transforming numerical variables into their categorical counterparts.

In other words, binning will take a column with continuous numbers and place the numbers in “bins” based on ranges that we determine. This will give us a new categorical variable feature
Binning is a way to group a number of more or less continuous values into a smaller number of "bins". For example, if you have data about a group of people, you might want to arrange their ages into a smaller number of age intervals

A special use case of this binning method is grouping values that are misspelt or differ due to other reasons. For example, if a column contains values like “apple” and “appel”, or “UK” and “United Kingdom”, you can group these values into bins.

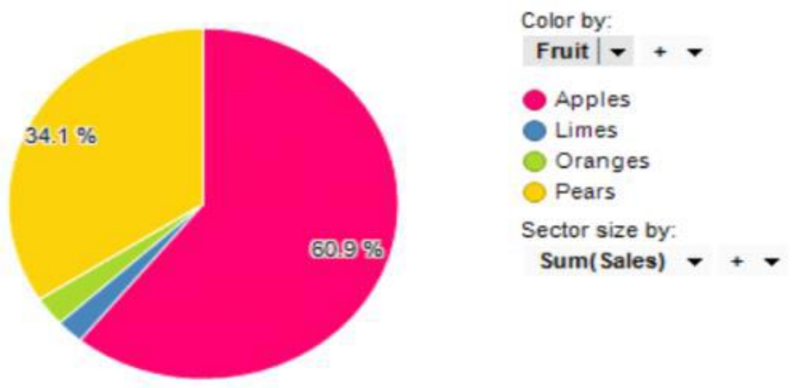


By binning the age of the people into a new column, data can be visualized for the different age groups instead of for each individual.

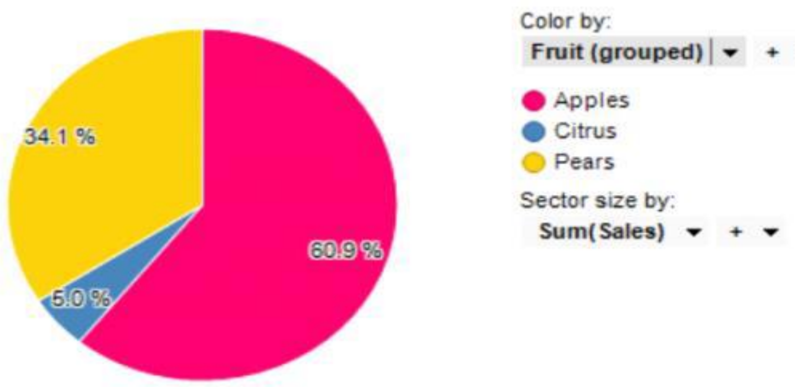


Example of binning categorical data

The pie chart shows sales per apples, limes, oranges and pears.



Below oranges and limes have been grouped into a bin called "Citrus".



Advantages of binning

.

Improves the accuracy of predictive models by reducing noise or non-linearity in the dataset.

2.

Helps identify outliers and invalid and missing values of numerical variables.

Binning implementation in R

Using cut() function

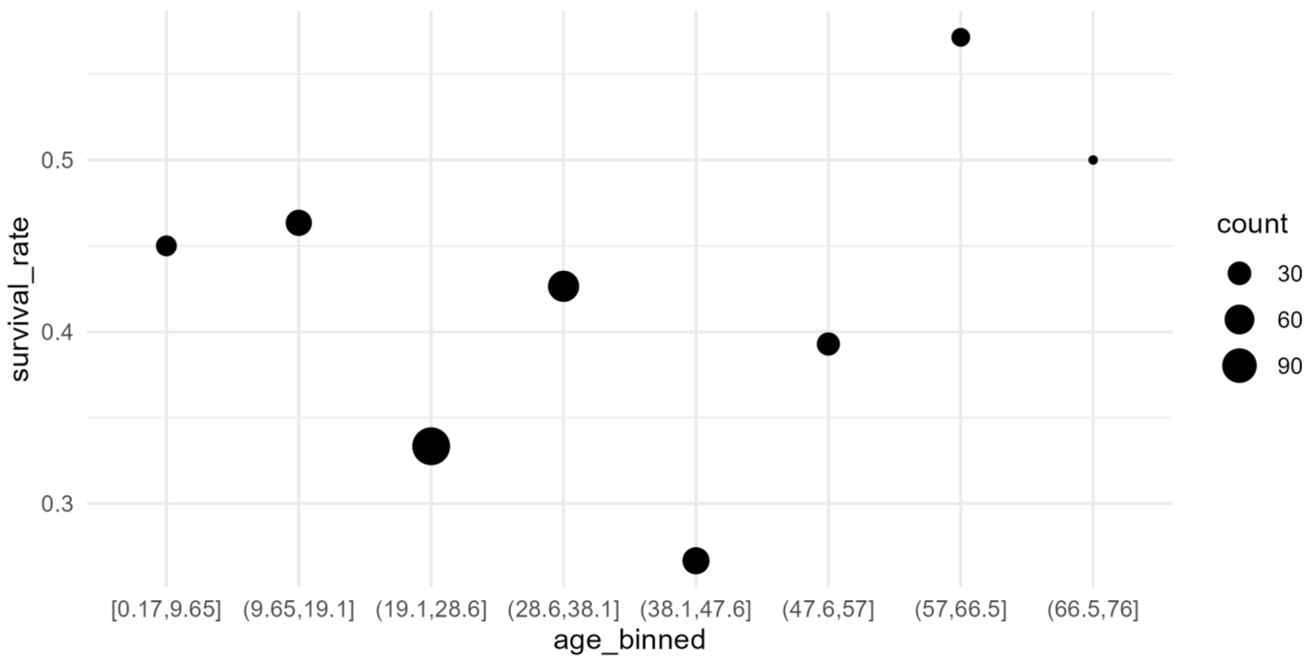
```
library(dplyr)

#perform binning with custom breaks
df %>% mutate(new_bin = cut(variable_name, breaks=c(0, 10, 20, 30)))

#perform binning with specific number of bins
df %>% mutate(new_bin = cut(variable_name, breaks=3))
```

Example

```
> #binning
> library(tidyverse)
> titanic <- read_csv('C:\\Users\\soorya\\Documents\\tested.csv')
> cut_number(titanic$Age,
+           n=8)
[1] (32,39]   (39,48]   (48,76]   (24,27]
  [5] (21,24]   [0.17,18] (27,32]   (24,27]
  [9] [0.17,18] (18,21]   <NA>      (39,48]
 [13] (21,24]   (48,76]   (39,48]   (21,24]
 [17] (32,39]   (18,21]   (24,27]   (39,48]
 [21] (48,76]   [0.17,18] <NA>      (18,21]
 [25] (39,48]   (48,76]   (21,24]   (21,24]
 [29] (39,48]   <NA>      (48,76]   (21,24]
 [33] (32,39]   <NA>      (27,32]   (18,21]
 [37] <NA>      (18,21]   (24,27]   <NA>
 [41] (32,39]   <NA>      (39,48]   (27,32]
 [45] (39,48]   (24,27]   (39,48]   <NA>
 [49] (48,76]   (32,39]   (21,24]   (24,27]
 [53] (18,21]   (27,32]   <NA>      [0.17,18]
 [57] (32,39]   (24,27]   <NA>      (32,39]
 [61] [0.17,18] (27,32]   [0.17,18] (21,24]
 [65] [0.17,18] <NA>      [0.17,18] (39,48]
 [69] (27,32]   (48,76]   (21,24]   (18,21]
 [73] (27,32]   (27,32]   (32,39]   (32,39]
 [77] <NA>      (48,76]   (27,32]   (21,24]
 [81] [0.17,18] (48,76]   (48,76]   <NA>
 [85] <NA>      <NA>      (24,27]   [0.17,18]
 [89] <NA>      [0.17,18] (21,24]   <NA>
 [93] (24,27]   <NA>      (24,27]   (24,27]
 [97] (48,76]   (27,32]   (18,21]   (32,39]
[101] (39,48]   (24,27]   <NA>      (24,27]
[105] [0.17,18] (27,32]   (18,21]   <NA>
[109] <NA>      (18,21]   (39,48]   <NA>
[113] (32,39]   (18,21]   (48,76]   [0.17,18]
[117] <NA>      [0.17,18] (32,39]   (27,32]
[121] [0.17,18] <NA>      (32,39]   (27,32]
[125] <NA>      [0.17,18] (21,24]   <NA>
[129] (39,48]   (21,24]   (27,32]   (48,76]
[133] <NA>      <NA>      (39,48]   (21,24]
[137] (24,27]   (24,27]   (21,24]   (39,48]
[141] [0.17,18] (32,39]   (48,76]   (27,32]
[145] (39,48]   (27,32]   <NA>      (21,24]
[149] <NA>      (27,32]   (21,24]   <NA>
[153] (48,76]   (32,39]   [0.17,18] (21,24]
[157] (27,32]   (21,24]   (39,48]   (24,27]
[161] <NA>      [0.17,18] (24,27]   <NA>
[165] (39,48]   (24,27]   (39,48]   [0.17,18]
[169] <NA>      (21,24]   <NA>      (24,27]
[173] (21,24]   <NA>      (39,48]   [0.17,18]
[177] (18,21]   (48,76]   (32,39]   (48,76]
[181] (27,32]   (32,39]   [0.17,18] <NA>
[185] (24,27]   (39,48]   (18,21]   [0.17,18]
[189] <NA>      (39,48]   (32,39]   <NA>
[193] [0.17,18] (48,76]   [0.17,18] (32,39]
[197] [0.17,18] [0.17,18] (21,24]   <NA>
[201] <NA>      [0.17,18] (39,48]   [0.17,18]
[205] (24,27]   <NA>      (32,39]   (21,24]
[209] (32,39]   (24,27]   (27,32]   <NA>
[213] [0.17,18] (48,76]   (32,39]   (39,48]
[217] <NA>      (48,76]   (48,76]   <NA>
[221] (27,32]   (18,21]   (21,24]   (18,21]
[225] (48,76]    <NA>       (21,24]    <NA>
[229] (39,48]    (32,39]    [0.17,18] (18,21]
[233] (18,21]    <NA>       (32,39]    (18,21]
[237] (48,76]    (18,21]    [0.17,18] (39,48]
[241] (48,76]    (39,48]    (39,48]    <NA>
[245] <NA>       (39,48]    (21,24]    (39,48]
[249] (27,32]    <NA>       [0.17,18] (18,21]
[253] (24,27]    (21,24]    (32,39]    <NA>
[257] <NA>       (27,32]    (18,21]    (18,21]
[261] (32,39]    (18,21]    (27,32]    [0.17,18]
[265] (27,32]    <NA>       <NA>       <NA>
[269] <NA>       [0.17,18] (39,48]    <NA>
[273] (24,27]    <NA>       <NA>       (18,21]
[277] (27,32]    (39,48]    (27,32]    (21,24]
[281] (21,24]    [0.17,18] <NA>       [0.17,18]
[285] [0.17,18] (32,39]    <NA>       (21,24]
[289] <NA>       <NA>       <NA>       (27,32]
[293] <NA>       (48,76]    (32,39]    (24,27]
[297] [0.17,18] <NA>       (27,32]    (27,32]
[301] (27,32]    <NA>       (39,48]    (21,24]
[305] <NA>       (48,76]    (27,32]    [0.17,18]
[309] (48,76]    (39,48]    [0.17,18] (21,24]
[313] <NA>       (32,39]    (48,76]    [0.17,18]
[317] (48,76]    (18,21]    (24,27]    (21,24]
[321] (24,27]    (24,27]    (24,27]    (32,39]
[325] (32,39]    (21,24]    [0.17,18] (39,48]
[329] (27,32]    (18,21]    (39,48]    (32,39]
[333] <NA>       (18,21]    (24,27]    (27,32]
[337] (27,32]    (32,39]    (24,27]    <NA>
[341] [0.17,18] (27,32]    <NA>       (48,76]
[345] <NA>       [0.17,18] (24,27]    (32,39]
[349] (21,24]    (27,32]    (39,48]    (24,27]
[353] [0.17,18] (48,76]    [0.17,18] (48,76]
[357] (48,76]    <NA>       <NA>       (27,32]
[361] [0.17,18] (21,24]    (27,32]    (24,27]
[365] (24,27]    <NA>       <NA>       (21,24]
[369] (39,48]    (27,32]    (18,21]    (27,32]
[373] (48,76]    (39,48]    (48,76]    (39,48]
[377] (21,24]    (18,21]    (48,76]    [0.17,18]
[381] <NA>       (24,27]    <NA>       (18,21]
[385] <NA>       (21,24]    (21,24]    (48,76]
[389] (18,21]    [0.17,18] (21,24]    (48,76]
[393] [0.17,18] (39,48]    (27,32]    [0.17,18]
[397] (21,24]    (39,48]    (21,24]    (27,32]
[401] (27,32]    (32,39]    (21,24]    [0.17,18]
[405] (39,48]    (18,21]    (21,24]    (48,76]
[409] <NA>       [0.17,18] <NA>       (32,39]
[413] (27,32]    <NA>       (32,39]    (32,39]
[417] <NA>       <NA>
8 Levels: [0.17,18] (18,21] ... (48,76]
> cut_number(titanic$Age,
+            n=8)
  [1] (32,39]    (39,48]    (48,76]    (24,27]
  [5] (21,24]    [0.17,18] (27,32]    (24,27]
  [9] [0.17,18] (18,21]    <NA>       (39,48]
 [13] (21,24]    (48,76]    (39,48]    (21,24]
 [17] (32,39]    (18,21]    (24,27]    (39,48]
 [21] (48,76]    [0.17,18] <NA>       (18,21]
 [25] (39,48]    (48,76]    (21,24]    (21,24]
 [29] (39,48]    <NA>       (48,76]    (21,24]
 [33] (32,39]    <NA>       (27,32]    (18,21]
 [37] <NA>       (18,21]    (24,27]    <NA>
 [41] (32,39]    <NA>       (39,48]    (27,32]
 [45] (39,48]    (24,27]    (39,48]    <NA>
 [49] (48,76]    (32,39]    (21,24]    (24,27]
 [53] (18,21]    (27,32]    <NA>       [0.17,18]
 [57] (32,39]    (24,27]    <NA>       (32,39]
 [61] [0.17,18]  (27,32]    [0.17,18]  (21,24]
 [65] [0.17,18]  <NA>       [0.17,18]  (39,48]
 [69] (27,32]    (48,76]    (21,24]    (18,21]
 [73] (27,32]    (27,32]    (32,39]    (32,39]
 [77] <NA>       (48,76]    (27,32]    (21,24]
 [81] [0.17,18]  (48,76]    (48,76]    <NA>
 [85] <NA>       <NA>       (24,27]    [0.17,18]
 [89] <NA>       [0.17,18]  (21,24]    <NA>
 [93] (24,27]    <NA>       (24,27]    (24,27]
 [97] (48,76]    (27,32]    (18,21]    (32,39]
[101] (39,48]    (24,27]    <NA>       (24,27]
[105] [0.17,18]  (27,32]    (18,21]    <NA>
[109] <NA>       (18,21]    (39,48]    <NA>
[113] (32,39]    (18,21]    (48,76]    [0.17,18]
[117] <NA>       [0.17,18]  (32,39]    (27,32]
[121] [0.17,18]  <NA>       (32,39]    (27,32]
[125] <NA>       [0.17,18]  (21,24]    <NA>
[129] (39,48]    (21,24]    (27,32]    (48,76]
[133] <NA>       <NA>       (39,48]    (21,24]
[137] (24,27]    (24,27]    (21,24]    (39,48]
[141] [0.17,18]  (32,39]    (48,76]    (27,32]
[145] (39,48]    (27,32]    <NA>       (21,24]
[149] <NA>       (27,32]    (21,24]    <NA>
[153] (48,76]    (32,39]    [0.17,18]  (21,24]
[157] (27,32]    (21,24]    (39,48]    (24,27]
[161] <NA>       [0.17,18]  (24,27]    <NA>
[165] (39,48]    (24,27]    (39,48]    [0.17,18]
[169] <NA>       (21,24]    <NA>       (24,27]
[173] (21,24]    <NA>       (39,48]    [0.17,18]
[177] (18,21]    (48,76]    (32,39]    (48,76]
[181] (27,32]    (32,39]    [0.17,18]  <NA>
[185] (24,27]    (39,48]    (18,21]    [0.17,18]
[189] <NA>       (39,48]    (32,39]    <NA>
[193] [0.17,18]  (48,76]    [0.17,18]  (32,39]
[197] [0.17,18]  [0.17,18]  (21,24]    <NA>
[201] <NA>       [0.17,18]  (39,48]    [0.17,18]
[205] (24,27]    <NA>       (32,39]    (21,24]
[209] (32,39]    (24,27]    (27,32]    <NA>
[213] [0.17,18]  (48,76]    (32,39]    (39,48]
[217] <NA>       (48,76]    (48,76]    <NA>
[221] (27,32]    (18,21]    (21,24]    (18,21]
[225] (48,76]    <NA>       (21,24]    <NA>
[229] (39,48]    (32,39]    [0.17,18]  (18,21]
[233] (18,21]    <NA>       (32,39]    (18,21]
[237] (48,76]    (18,21]    [0.17,18]  (39,48]
[241] (48,76]    (39,48]    (39,48]    <NA>
[245] <NA>       (39,48]    (21,24]    (39,48]
[249] (27,32]    <NA>       [0.17,18]  (18,21]
[253] (24,27]    (21,24]    (32,39]    <NA>
[257] <NA>       (27,32]    (18,21]    (18,21]
[261] (32,39]    (18,21]    (27,32]    [0.17,18]
[265] (27,32]    <NA>       <NA>       <NA>
[269] <NA>       [0.17,18]  (39,48]    <NA>
[273] (24,27]    <NA>       <NA>       (18,21]
[277] (27,32]    (39,48]    (27,32]    (21,24]
[281] (21,24]    [0.17,18]  <NA>       [0.17,18]
[285] [0.17,18]  (32,39]    <NA>       (21,24]
[289] <NA>       <NA>       <NA>       (27,32]
[293] <NA>       (48,76]    (32,39]    (24,27]
[297] [0.17,18]  <NA>       (27,32]    (27,32]
[301] (27,32]    <NA>       (39,48]    (21,24]
[305] <NA>       (48,76]    (27,32]    [0.17,18]
[309] (48,76]    (39,48]    [0.17,18] (21,24]
[313] <NA>       (32,39]    (48,76]    [0.17,18]
[317] (48,76]    (18,21]    (24,27]    (21,24]
[321] (24,27]    (24,27]    (24,27]    (32,39]
[325] (32,39]    (21,24]    [0.17,18] (39,48]
[329] (27,32]    (18,21]    (39,48]    (32,39]
[333] <NA>       (18,21]    (24,27]    (27,32]
[337] (27,32]    (32,39]    (24,27]    <NA>
[341] [0.17,18] (27,32]    <NA>       (48,76]
[345] <NA>       [0.17,18] (24,27]    (32,39]
[349] (21,24]    (27,32]    (39,48]    (24,27]
[353] [0.17,18] (48,76]    [0.17,18] (48,76]
[357] (48,76]    <NA>       <NA>       (27,32]
[361] [0.17,18] (21,24]    (27,32]    (24,27]
[365] (24,27]    <NA>       <NA>       (21,24]
[369] (39,48]    (27,32]    (18,21]    (27,32]
[373] (48,76]    (39,48]    (48,76]    (39,48]
[377] (21,24]    (18,21]    (48,76]    [0.17,18]
[381] <NA>       (24,27]    <NA>       (18,21]
[385] <NA>       (21,24]    (21,24]    (48,76]
[389] (18,21]    [0.17,18] (21,24]    (48,76]
[393] [0.17,18] (39,48]    (27,32]    [0.17,18]
[397] (21,24]    (39,48]    (21,24]    (27,32]
[401] (27,32]    (32,39]    (21,24]    [0.17,18]
[405] (39,48]    (18,21]    (21,24]    (48,76]
[409] <NA>       [0.17,18] <NA>       (32,39]
[413] (27,32]    <NA>       (32,39]    (32,39]
[417] <NA>       <NA>
8 Levels: [0.17,18] (18,21] ... (48,76]
> table()
Error in table() : nothing to tabulate
> cut_number(titanic$Age,
+             n=8)
  [1] (32,39]    (39,48]    (48,76]    (24,27]
  [5] (21,24]    [0.17,18] (27,32]    (24,27]
  [9] [0.17,18] (18,21]    <NA>       (39,48]
 [13] (21,24]    (48,76]    (39,48]    (21,24]
 [17] (32,39]    (18,21]    (24,27]    (39,48]
 [21] (48,76]    [0.17,18] <NA>       (18,21]
 [25] (39,48]    (48,76]    (21,24]    (21,24]
 [29] (39,48]    <NA>       (48,76]    (21,24]
 [33] (32,39]    <NA>       (27,32]    (18,21]
 [37] <NA>       (18,21]    (24,27]    <NA>
 [41] (32,39]    <NA>       (39,48]    (27,32]
 [45] (39,48]    (24,27]    (39,48]    <NA>
 [49] (48,76]    (32,39]    (21,24]    (24,27]
 [53] (18,21]    (27,32]    <NA>       [0.17,18]
 [57] (32,39]    (24,27]    <NA>       (32,39]
 [61] [0.17,18] (27,32]    [0.17,18] (21,24]
 [65] [0.17,18] <NA>       [0.17,18] (39,48]
 [69] (27,32]    (48,76]    (21,24]    (18,21]
 [73] (27,32]    (27,32]    (32,39]    (32,39]
 [77] <NA>       (48,76]    (27,32]    (21,24]
 [81] [0.17,18] (48,76]    (48,76]    <NA>
 [85] <NA>       <NA>       (24,27]    [0.17,18]
 [89] <NA>       [0.17,18] (21,24]    <NA>
 [93] (24,27]    <NA>       (24,27]    (24,27]
 [97] (48,76]    (27,32]    (18,21]    (32,39]
[101] (39,48]    (24,27]    <NA>       (24,27]
[105] [0.17,18] (27,32]    (18,21]    <NA>
[109] <NA>       (18,21]    (39,48]    <NA>
[113] (32,39]    (18,21]    (48,76]    [0.17,18]
[117] <NA>       [0.17,18] (32,39]    (27,32]
[121] [0.17,18] <NA>      (32,39]   (27,32]  
[125] <NA>      [0.17,18] (21,24]   <NA>     
[129] (39,48]   (21,24]   (27,32]   (48,76]  
[133] <NA>      <NA>      (39,48]   (21,24]  
[137] (24,27]   (24,27]   (21,24]   (39,48]  
[141] [0.17,18] (32,39]   (48,76]   (27,32]  
[145] (39,48]   (27,32]   <NA>      (21,24]  
[149] <NA>      (27,32]   (21,24]   <NA>     
[153] (48,76]   (32,39]   [0.17,18] (21,24]  
[157] (27,32]   (21,24]   (39,48]   (24,27]  
[161] <NA>      [0.17,18] (24,27]   <NA>     
[165] (39,48]   (24,27]   (39,48]   [0.17,18]
[169] <NA>      (21,24]   <NA>      (24,27]  
[173] (21,24]   <NA>      (39,48]   [0.17,18]
[177] (18,21]   (48,76]   (32,39]   (48,76]  
[181] (27,32]   (32,39]   [0.17,18] <NA>     
[185] (24,27]   (39,48]   (18,21]   [0.17,18]
[189] <NA>      (39,48]   (32,39]   <NA>     
[193] [0.17,18] (48,76]   [0.17,18] (32,39]  
[197] [0.17,18] [0.17,18] (21,24]   <NA>     
[201] <NA>      [0.17,18] (39,48]   [0.17,18]
[205] (24,27]   <NA>      (32,39]   (21,24]  
[209] (32,39]   (24,27]   (27,32]   <NA>     
[213] [0.17,18] (48,76]   (32,39]   (39,48]  
[217] <NA>      (48,76]   (48,76]   <NA>     
[221] (27,32]   (18,21]   (21,24]   (18,21]  
[225] (48,76]   <NA>      (21,24]   <NA>     
[229] (39,48]   (32,39]   [0.17,18] (18,21]  
[233] (18,21]   <NA>      (32,39]   (18,21]  
[237] (48,76]   (18,21]   [0.17,18] (39,48]  
[241] (48,76]   (39,48]   (39,48]   <NA>     
[245] <NA>      (39,48]   (21,24]   (39,48]  
[249] (27,32]   <NA>      [0.17,18] (18,21]  
[253] (24,27]   (21,24]   (32,39]   <NA>     
[257] <NA>      (27,32]   (18,21]   (18,21]  
[261] (32,39]   (18,21]   (27,32]   [0.17,18]
[265] (27,32]   <NA>      <NA>      <NA>     
[269] <NA>      [0.17,18] (39,48]   <NA>     
[273] (24,27]   <NA>      <NA>      (18,21]  
[277] (27,32]   (39,48]   (27,32]   (21,24]  
[281] (21,24]   [0.17,18] <NA>      [0.17,18]
[285] [0.17,18] (32,39]   <NA>      (21,24]  
[289] <NA>      <NA>      <NA>      (27,32]  
[293] <NA>      (48,76]   (32,39]   (24,27]  
[297] [0.17,18] <NA>      (27,32]   (27,32]  
[301] (27,32]   <NA>      (39,48]   (21,24]  
[305] <NA>      (48,76]   (27,32]   [0.17,18]
[309] (48,76]   (39,48]   [0.17,18] (21,24]  
[313] <NA>      (32,39]   (48,76]   [0.17,18]
[317] (48,76]   (18,21]   (24,27]   (21,24]  
[321] (24,27]   (24,27]   (24,27]   (32,39]  
[325] (32,39]   (21,24]   [0.17,18] (39,48]  
[329] (27,32]   (18,21]   (39,48]   (32,39]  
[333] <NA>      (18,21]   (24,27]   (27,32]  
[337] (27,32]   (32,39]   (24,27]   <NA>     
[341] [0.17,18] (27,32]   <NA>      (48,76]  
[345] <NA>      [0.17,18] (24,27]   (32,39]  
[349] (21,24]   (27,32]   (39,48]   (24,27]  
[353] [0.17,18] (48,76]   [0.17,18] (48,76]  
[357] (48,76]   <NA>      <NA>      (27,32]  
[361] [0.17,18] (21,24]   (27,32]   (24,27]  
[365] (24,27]   <NA>      <NA>      (21,24]  
[369] (39,48]   (27,32]   (18,21]   (27,32]  
[373] (48,76]   (39,48]   (48,76]   (39,48]  
[377] (21,24]    (18,21]    (48,76]    [0.17,18]
[381] <NA>       (24,27]    <NA>       (18,21]
[385] <NA>       (21,24]    (21,24]    (48,76]
[389] (18,21]    [0.17,18] (21,24]    (48,76]
[393] [0.17,18] (39,48]    (27,32]    [0.17,18]
[397] (21,24]    (39,48]    (21,24]    (27,32]
[401] (27,32]    (32,39]    (21,24]    [0.17,18]
[405] (39,48]    (18,21]    (21,24]    (48,76]
[409] <NA>       [0.17,18] <NA>       (32,39]
[413] (27,32]    <NA>       (32,39]    (32,39]
[417] <NA>       <NA>
8 Levels: [0.17,18] (18,21] ... (48,76]
>   table()
Error in table() : nothing to tabulate
> cut_number(titanic$Age,
+            n=8) %>%
+   table()
.
[0.17,18]   (18,21]   (21,24]   (24,27]
       54        32        45        36
  (27,32]   (32,39]   (39,48]   (48,76]
       45        38        43        39
> titanic <- titanic %>%
+   drop_na(Age)
> titanic <- titanic %>%
+   mutate(age_binned = cut_number(Age,
+                              n=8))
> View(titanic)
> titanic <- titanic %>%
+   mutate(age_binned = cut_interval(Age,
+                              n=8))
> View(titanic)
> count(titanic, age_binned)
# A tibble: 8 × 2
  age_binned      n
  <fct>       <int>
1 [0.17,9.65]    20
2 (9.65,19.1]    41
3 (19.1,28.6]   114
4 (28.6,38.1]    68
5 (38.1,47.6]    45
6 (47.6,57]      28
7 (57,66.5]      14
8 (66.5,76]       2
> titanic_summary <- titanic %>%
+   group_by(age_binned) %>%
+   summarise(survival_rate = mean(Survived),
+             count = n())
> titanic_summary
# A tibble: 8 × 3
  age_binned  survival_rate count
  <fct>               <dbl> <int>
1 [0.17,9.65]         0.45     20
2 (9.65,19.1]         0.463    41
3 (19.1,28.6]         0.333   114
4 (28.6,38.1]         0.426    68
5 (38.1,47.6]         0.267    45
6 (47.6,57]           0.393    28
7 (57,66.5]           0.571    14
8 (66.5,76]           0.5       2
> ggplot(titanic_summary, aes(x =age_binned,
+                             y= survival_rate,
+                             size=count))
> ggplot(titanic_summary, aes(x =age_binned,
+                              y= survival_rate,
+                              size=count))+
+   geom_point()
```



Subsets

Sometimes we need only a part of dataset to do the data analysis. In this case, we need to use a subset of dataset. To do this we can use different methods.

Bracket Notation:

One way to take a subset of a data set is to use the bracket notation. You can select rows in a data frame by providing a vector of logical values. If you can write a simple expression describing the set of rows to select from a data frame, you can provide this as an index. For example, suppose that we wanted to select only batting data from 2008. The column batting.w.names$yearID contains the year associated with each row, so we could calculate a vector of logical values describing which rows to keep with the expression batting.w.names$yearID==2008. Now, we just have to index the data frame batting.w.names with this vector to select only rows for the year 2008:

> batting.w.names.2008 <- batting.w.names[batting.w.names$yearID==2008,]

```
> summary(batting.w.names.2008$yearID)
Min. 1st Qu. Median Mean 3rd Qu. Max.
2008 2008 2008 2008 2008 2008
```

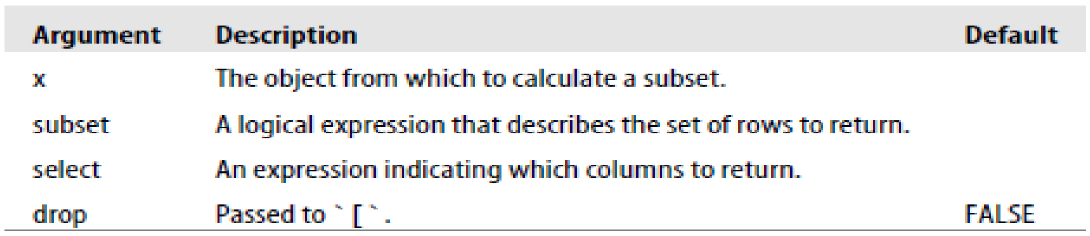
Similarly, we can use the same notation to select only certain columns. Suppose that we only wanted to keep some variables nameFirst, nameLast, AB, H, BB. We could provide these in the brackets as well:

```
> batting.w.names.2008.short <-
+ batting.w.names[batting.w.names$yearID==2008,
+ c("nameFirst","nameLast","AB","H","BB")]
```

Subset Function:

As an alternative, you can use the subset function to select a subset of rows and columns from a data frame (or matrix):

```
subset(x, subset, select, drop = FALSE, ...)
```

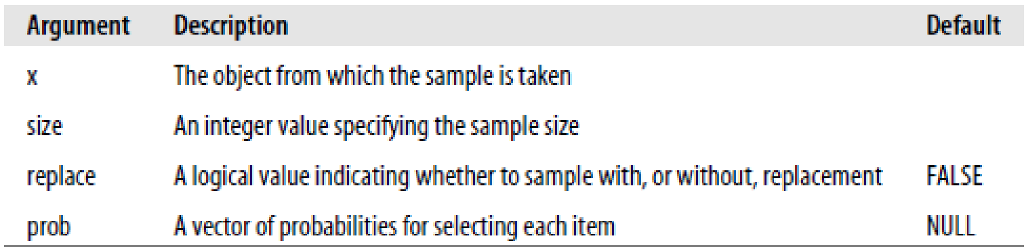


Argument	Description	Default
x	The object from which to calculate a subset.	
subset	A logical expression that describes the set of rows to return.	
select	An expression indicating which columns to return.	
drop	Passed to ` [ `.	FALSE

Random Sampling:

Often, it is desirable to take a random sample of a data set. Sometimes, you might have too much data (for statistical reasons or performance reasons). Other times, you simply want to split your data into different parts for modeling (usually into training, testing, and validation subsets). One of the simplest ways to extract a random sample is with the sample function. The sample function returns a random sample of the elements of a vector:

```
sample(x, size, replace = FALSE, prob = NULL)
```



Argument	Description	Default
x	The object from which the sample is taken	
size	An integer value specifying the sample size	
replace	A logical value indicating whether to sample with, or without, replacement	FALSE
prob	A vector of probabilities for selecting each item	NULL

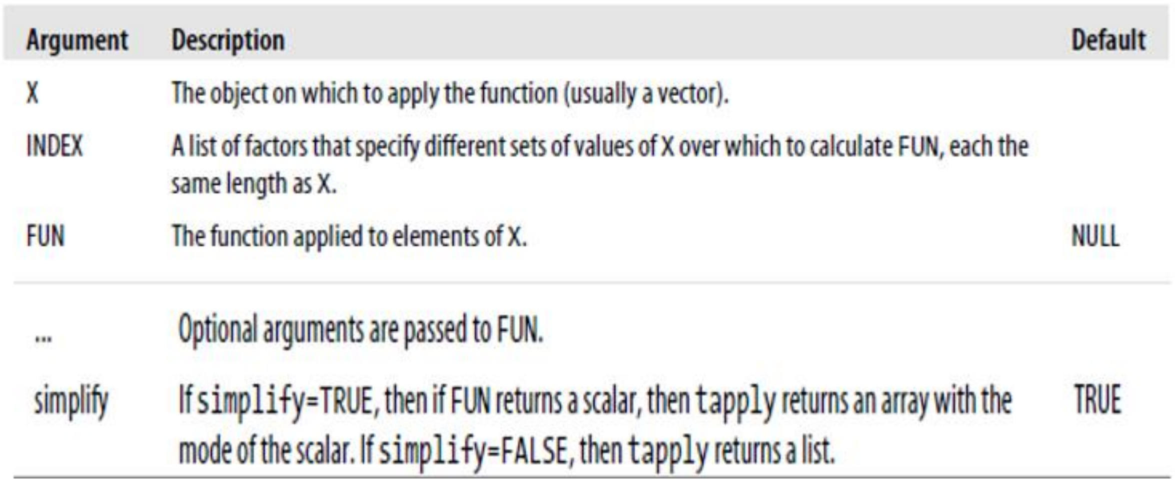
Summarizing Functions

Suppose that you wanted to know the average number of pages delivered to each user. To find the answer, you might need to look at every HTTP transaction (every request for content), grouping together requests into sessions and counting the number of requests. R provides a number of different functions for summarizing data, aggregating records together to build a smaller data set. The functions such as tapply and aggregate are used for summarizing.

tapply:

The tapply function is a very flexible function for summarizing a vector X. You can specify which subsets of X to summarize as well as the function used for summarization:

tapply(X, INDEX, FUN = , ..., simplify =)



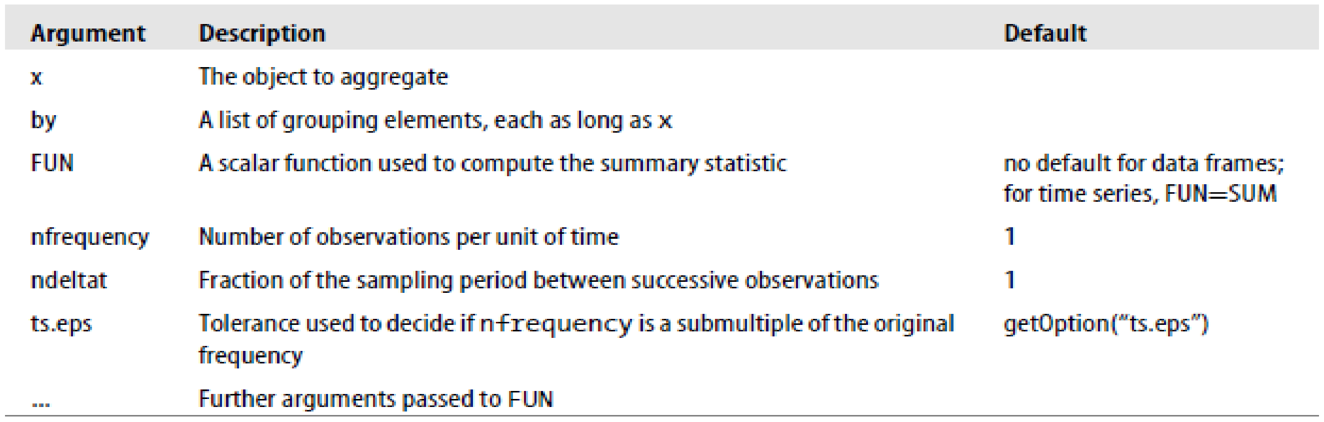
Argument	Description	Default
X	The object on which to apply the function (usually a vector).	
INDEX	A list of factors that specify different sets of values of X over which to calculate FUN, each the same length as X.	
FUN	The function applied to elements of X.	NULL
...	Optional arguments are passed to FUN.	
simplify	If `simplify=TRUE`, then if FUN returns a scalar, then `tapply` returns an array with the mode of the scalar. If `simplify=FALSE`, then `tapply` returns a list.	TRUE

Eg: tapply(X=batting.2008$HR,INDEX=list(batting.2008$teamID),FUN=sum)

aggregate:

Another option for summarization is the function aggregate. Here is the form of aggregate when applied to data frames:
aggregate(x, by, FUN, ...)
Aggregate can also be applied to time series and takes slightly different arguments:
aggregate(x, nfrequency = 1, FUN = sum, ndeltat = 1,
ts.eps = getOption("ts.eps"), ...)

Eg: > aggregate(x=batting.2008[,c("AB","H","BB","2B","3B","HR")],
+ by=list(batting.2008$teamID),FUN=sum)



Argument	Description	Default
x	The object to aggregate	
by	A list of grouping elements, each as long as x	
FUN	A scalar function used to compute the summary statistic	no default for data frames; for time series, FUN=SUM
nfrequency	Number of observations per unit of time	1
ndeltat	Fraction of the sampling period between successive observations	1
ts.eps	Tolerance used to decide if `nfrequency` is a submultiple of the original frequency	getOption("ts.eps")
...	Further arguments passed to FUN	

Data Cleaning

Data cleaning is an important step in any data analysis project. The data collected might contain insufficient data or there might be duplicate records of the data. Data cleaning doesn't mean changing the meaning of data. It means identifying problems caused by data collection, processing, and storage processes and modifying the data so that these problems don't interfere with analysis.

Data cleaning is the process of correcting or deleting inaccurate, damaged, improperly formatted, duplicated, or insufficient data from a dataset. Even if results and algorithms appear to be correct, they are unreliable if the data is inaccurate. There are numerous ways for data to be duplicated or incorrectly labeled when merging multiple data sources.

Steps for Data Cleaning:

1. Remove duplicate or irrelevant observations

Remove duplicate or pointless observations as well as undesirable observations from your dataset. The majority of duplicate observations will occur during data gathering. Duplicate data can be produced when you merge data sets from several sources, scrape data, or get data from clients or other departments. One of the most important factors to consider in this procedure is de-duplication. Those observations are deemed irrelevant when you observe observations that do not pertain to the particular issue you are attempting to analyze.

You might eliminate those useless observations, for instance, if you wish to analyze data on millennial clients but your dataset also includes observations from earlier generations.

This can improve the analysis's efficiency, reduce deviance from your main objective, and produce a dataset that is easier to maintain and use.

2. Fix structural errors
When you measure or transfer data and find odd naming practices, typos, or wrong capitalization, such are structural faults. Mis-labelled categories or classes may result from these inconsistencies. For instance, "N/A" and "Not Applicable" might be present on any given sheet, but they ought to be analyzed under the same heading.

3. Filter unwanted outliers
There will frequently be isolated findings that, at first glance, do not seem to fit the data you are analyzing. Removing an outlier if you have a good reason to, such as incorrect data entry, will improve the performance of the data you are working with.

However, occasionally the emergence of an outlier will support a theory you are investigating. And just because there is an outlier, that doesn't necessarily indicate it is inaccurate. To determine the reliability of the number, this step is necessary. If an outlier turns out to be incorrect or unimportant for the analysis, you might want to remove it.

4. Handle missing data
Because many algorithms won't tolerate missing values, you can't overlook missing data. There are a few options for handling missing data. While neither is ideal, both can be considered, for example:

Although you can remove observations with missing values, doing so will result in the loss of information, so proceed with caution.

Again, there is a chance to undermine the integrity of the data since you can be working from assumptions rather than actual observations when you input missing numbers based on other observations.

To browse null values efficiently, you may need to change the way the data is used.

Finding and Removing Duplicates

R provides some useful functions for detecting duplicate values.
Suppose that you accidentally included one stock ticker twice (say, GE) when you fetched stock quotes:

```
> my.tickers.2 <- c("GE","GOOG","AAPL","AXP","GS","GE")
> my.quotes.2 <- get.multiple.quotes(my.tickers.2, from=as.Date("2009-01-01"),
+ to=as.Date("2009-03-31"), interval="m")
```

R provides some useful functions for detecting duplicate values such as the duplica ted function. This function returns a logical vector showing which elements are duplicates of values with lower indices. Let's apply duplicated to the data frame my.quotes.2:

```
> duplicated(my.quotes.2)
[1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[12] FALSE FALSE FALSE FALSE TRUE TRUE TRUE
```

As expected, duplicated shows that the last three rows are duplicates of earlier rows.
You can use the resulting vector to remove duplicates:

```
> my.quotes.unique <- my.quotes.2[!duplicated(my.quotes.2),]
```

Alternatively, you could use the unique function to remove the duplicate values:

```
my.quotes.unique <- unique(my.quotes.2)
```

Sorting

In data analysis sorting and ranking functions are also play an important role sometimes.
To sort the elements of an object, use the sort function:

```
> w <- c(5,4,7,2,7,1)
> sort(w)
[1] 1 2 4 5 7 7
```

Add the decreasing=TRUE option to sort in reverse order:

```
> sort(w,decreasing=TRUE)
[1] 7 7 5 4 2 1
```

You can control the treatment of NA values by setting the na.last argument:

```
> length(w)
[1] 6
> length(w) <- 7
> # note that by default, NA.last=NA and NA values are not shown
> sort(w)
[1] 1 2 4 5 7 7
> # set NA.last=TRUE to put NA values last
> sort(w,na.last=TRUE)
[1] 1 2 4 5 7 7 NA
> # set NA.last=FALSE to put NA values first
> sort(w,na.last=FALSE)
[1] NA 1 2 4 5 7 7
```

To sort a data frame, you need to create a permutation of the indices from the data frame and use these to fetch the rows of the data frame in the correct order.

```
order(..., na.last = , decreasing = )
```

The order function takes a set of vectors as arguments. It sorts recursively by each vector, breaking ties by looking at successive vectors in the argument list. At the end, it returns a permutation of the indices of the vector corresponding to the sorted order. (The arguments na.last and decreasing work the same way as they do for sort). To see what this means, let's use a simple example. First, we'll define a vector with two elements out of order:

```
> v <- c(11, 12, 13, 15, 14)

> order(v)
[1] 1 2 3 5 4
```

This means "move row 1 to row 1, move row 2 to row 2, move row 3 to row 3, move row 4 to row 5, move row 5 to row 4." We can return a sorted version of v using an indexing operator:

```
> v[order(v)]
[1] 11 12 13 14 15
```

To sort a data frame, you need to create a permutation of the indices from the data frame and use these to fetch the rows of the data frame in the correct order. You can generate an appropriate permutation of the indices using the order function:

```
order(..., na.last = , decreasing = )
```