

Loading Data into R

This tutorial is an attempt to demonstrate how to load data into R from files, URLs and database.

Loading Data from Files

Data frame is used to store data in R. To create a data frame use `data.frame()` command as follows.

```
> d <- data.frame(a1=c(5,3), a2=c('a','b'))
```

To examine or view the data frame, type

```
> print(d)
```

Here we are loading data from a file (a csv file named `cardio.csv`) using `read.table()` and store the data into a data frame called `cardio_df`. We specify two arguments, `sep` and `header`. Argument `sep` is used to specify the character that separates the values of the dataset. Argument `header` is used to indicate the first line of the dataset contains the names of the columns (attributes).

```
> cardio_df <- read.table('cardio.csv', sep=';', header=T)
```

We can examine the class of object `cardio_df`.

```
> class(cardio_df)
```

As you can see the class is data frame.

R has a command to show the distribution of each attribute of the dataset.

```
> summary(cardio_df)
```

The dimension of the data frame can be examined by using `dim()`.

```
> dim(cardio_df)
```

We can also load data using `read.csv()` and `read.csv2()`. These two functions are identical to `read.table` except for the defaults where the default separator for `read.csv()` is comma (,) while the default separator for `read.csv2()` is semicolon (;).

Check the field names of the data frame.

```
> names(cardio_df)
```

Rename the second column to "maintenance".

```
> names(cardio_df)[9] <- 'glucose'
```

To check the data type of the attributes. You can check if the data type of the attributes are correct or not.

```
> str(cardio_df)
```

Attribute "gender" should be a categorical data. However, it is shown that the data type is integer (numerical). Use `factor` to convert to categorical data. `Factor` is used to represent categorical data.

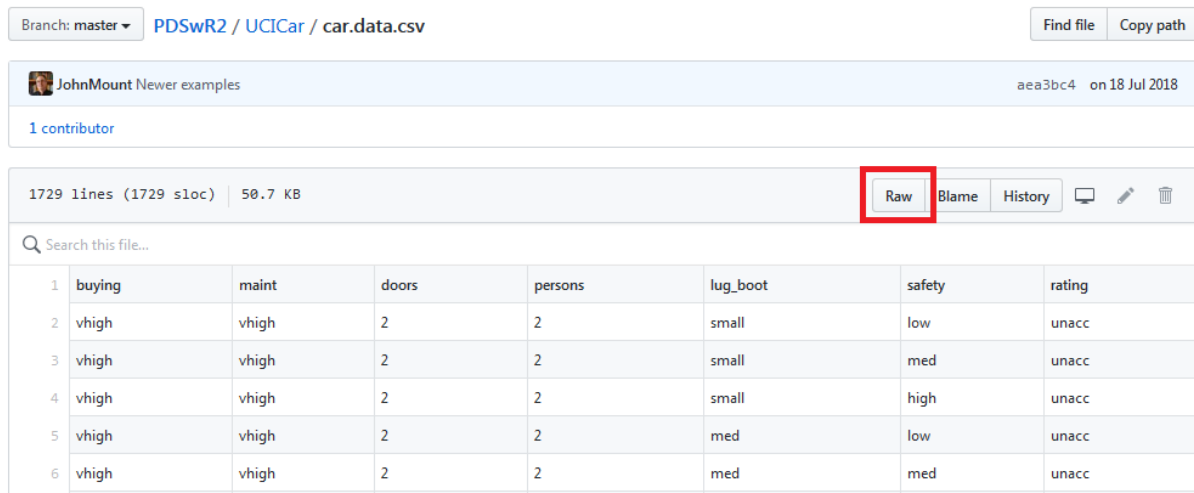
```
> cardio_df$gender <- as.factor(cardio_df$gender)
```

Loading Data from HTTP Sources

To read data directly from HTTP style URLs, we need to find the correct URL in order to avoid any page redirect i.e. GitHub. Assuming we want to load data from the following page.

<https://github.com/WinVector/PDSwR2/blob/master/UCICar/car.data.csv>

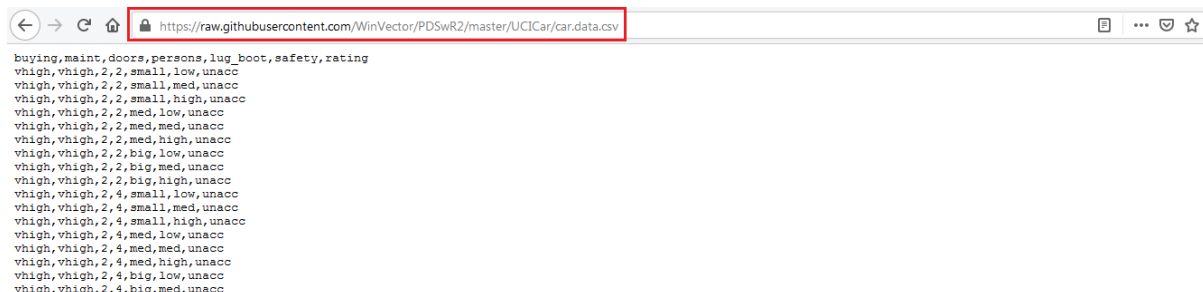
We need to find the URL that link directly to the dataset. To get the URL on GitHub page, click the Raw button as shown in the Figure.



The screenshot shows the GitHub interface for the file `car.data.csv` in the `UCICar` directory of the `PDSwR2` repository. The `Raw` button is highlighted with a red box. Below the file information, a table displays the first six rows of the dataset.

	buying	maint	doors	persons	lug_boot	safety	rating
1	vhhigh	vhhigh	2	2	small	low	unacc
2	vhhigh	vhhigh	2	2	small	med	unacc
3	vhhigh	vhhigh	2	2	small	high	unacc
4	vhhigh	vhhigh	2	2	med	low	unacc
5	vhhigh	vhhigh	2	2	med	med	unacc
6	vhhigh	vhhigh	2	2	med	med	unacc

It will bring us to a page of the dataset. Copy the URL and use it to load the dataset into R.



The screenshot shows a web browser displaying the raw data file URL: `https://raw.githubusercontent.com/WinVector/PDSwR2/master/UCICar/car.data.csv`. Below the URL, the first few lines of the dataset are visible as a text file.

```
buying,maint,doors,persons,lug_boot,safety,rating
vhhigh,vhhigh,2,2,small,low,unacc
vhhigh,vhhigh,2,2,small,med,unacc
vhhigh,vhhigh,2,2,small,high,unacc
vhhigh,vhhigh,2,2,med,low,unacc
vhhigh,vhhigh,2,2,med,med,unacc
vhhigh,vhhigh,2,2,med,high,unacc
vhhigh,vhhigh,2,2,big,low,unacc
vhhigh,vhhigh,2,2,big,med,unacc
vhhigh,vhhigh,2,2,big,high,unacc
vhhigh,vhhigh,2,4,small,low,unacc
vhhigh,vhhigh,2,4,small,med,unacc
vhhigh,vhhigh,2,4,small,high,unacc
vhhigh,vhhigh,2,4,med,low,unacc
vhhigh,vhhigh,2,4,med,med,unacc
vhhigh,vhhigh,2,4,med,high,unacc
vhhigh,vhhigh,2,4,big,low,unacc
vhhigh,vhhigh,2,4,big,med,unacc
```

First, we need to install the required package. Then include the library as follows.

```
> install.packages("RCurl")
```

```
> library(RCurl)
```

Create a variable to store the URL and a function that will receive the namespace to the dataset.

```
> urlBase <- 'https://raw.githubusercontent.com/WinVector/PDSwR2/master/'
```

```
> mkCon <- function(nm) { textConnection(getURL(paste(urlBase,nm,sep='')))} }
```

Use `read.table()` to load the dataset into a data frame.

```
> cars <- read.table(mkCon('UCICar/car.data.csv'), sep=',', header=T)
```

Loading Data from a Database

To load data from a database requires packages that allow us to connect to the database. The package that we need is depending on the database that we want to connect. In this lab, we will be using RMySQL. First, let's install the package.

```
> install.packages("RMySQL")  
> library(RMySQL)
```

Make a connection to the database as follows. We have to specify the name of the database, the username, the password and the hostname.

```
> con <- dbConnect(RMySQL::MySQL(), dbname = "tweater", host =  
  "courses.csrrinzqubik.us-east-1.rds.amazonaws.com", port = 3306, user =  
  "student", password = "datacamp")
```

Once the connection has been established, we can perform operations on the database such as listing the tables in the database.

```
> tables = dbListTables(con)  
> str(tables)
```

Here, we are read a specific table called users

```
> users <- dbReadTable(con, "users")  
> print(users)
```

Using tibble and dplyr

Tibble is one of the packages of **Tidyverse**. Tidyverse is a collection of R packages designed for data science. All packages share an underlying design philosophy, grammar, and data structures. The core packages includes in tidyverse: ggplot2, dplyr, tidyr, readr, purrr, tibble, stringr, forcats.

Install tidyverse package. This will install all the packages.

```
> install.packages("tidyverse")
> library(tidyverse)
```

A **tibble** is a modern class of data frame within R, which allows working with data easier. To create a tibble from an existing object, use `as_tibble()` as follows.

```
> data <- data.frame(a = 1:3, b = letters[1:3], c = Sys.Date() - 1:3)
> data_t1 <- as_tibble(data)
```

To create a new tibble from column vectors:

```
> data_t2 <- tibble(x = 1:5, y = 1, z = x ^ 2 + y)
```

Now, let's perform data manipulation on tibble using **dplyr**. We will be using an external dataset. To use the dataset, we need to install the package that contains the data. This data frame contains all 336,776 flights that departed from New York City in 2013.

```
> install.packages("nycflights13")
> library(nycflights13)
```

Type the following to display the tibble. Notice that tibbles show only the first 10 rows, and all the columns that fit on screen.

```
> flights
```

We can explicitly print the data frame and control the number of rows (n) and the width of the display. `width = Inf` will display all columns:

```
> flights %>% print(n = 20, width = Inf)
```

Filter rows to retrieve a subset of observations based on their values.

```
> filter(flights, month==2, day==1)
```

dplyr executes the filtering operation and returns a new data frame. dplyr functions never modify the original data frame. So to save the result, we need to use the assignment operator, `<-`.

```
> flight_feb1 <- filter(flights, month==2, day==1)
```

We can use filter with the following operator `>`, `>=`, `<`, `<=`, `!=` (not equal), and `==` (equal).

```
> flight_feb1 <- filter(flights, month==2, day>10)
```

Multiple filter() can be executed using logical operators: `&` is "and," `|` is "or," and `!` is "not."

```
> flight_oct_nov <- filter(flights, month==10 | month==11)
```

Similar output can be obtained as follows

```
> flight_oct_nov_dec <- filter(flights, month %in% c(10, 11, 12))
```

Function `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.

```
> arrange(flights, year, month, day)
```

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

```
> arrange(flights, desc(arr_delay))
```

To select certain columns

```
> select(flights, year, month, day)
```

Select all columns between year and day (inclusive)

```
> select(flights, year:day)
```

Select all columns except those from year to day (inclusive)

```
> select(flights, -(year:day))
```

There are a number of helper functions you can use within `select()`

- `starts_with("abc")` matches names that begin with "abc".

- `ends_with("xyz")` matches names that end with "xyz".

- `contains("ijk")` matches names that contain "ijk".

- `matches("(.)\\1")` selects variables that match a regular expression. This one matches any variables that contain repeated characters.

- `num_range("x", 1:3)` matches x1, x2, and x3.

```
> select(flights, starts_with("dep_"), starts_with("arr_"))
```

Rename a column e.g. `tailnum` becomes `tail_num`

```
> flights_ <- rename(flights, tail_num = tailnum)
```

`summarise()` creates a new data frame which the output will have a single row summarising all observations in the input. The output is just the mean of `dep_delay` observations. The parameter `na.rm` is set to true to ignore the missing values in the mean calculation.

```
> mean_dep_delay <- summarize(flights, dep_delay=mean(dep_delay,  
  na.rm=TRUE))
```

It doesn't look that useful unless if we use it with `group_by`. For example, if we apply `summarize()` on a `group_by` tibble, we can summarize the data e.g. mean of the observations and group them based on the grouping variables.

Here, we group by month – create a `group_by` tibble

```
> group_by_month <- group_by(flights, month)
```

Apply `summarize` on the `group_by` tibble to display the mean of the observations.

```
> summarize(group_by_month, dep_time=mean(dep_time, na.rm=TRUE),  
  arr_time=mean(arr_time, na.rm=TRUE), dep_delay=mean(dep_delay,  
  na.rm=TRUE))
```

Notice that we have to perform two steps to produce the output – we need to create an intermediate data frame (group_by_month). We can achieve the same output by using %>% operator. This operator is called **pipe** operator. It can be used to chain multiple functions together by taking the output of one function and insert it into the next.

```
> flights %>% group_by(month) %>% summarize(dep_time=mean(dep_time,  
  na.rm=TRUE), arr_time=mean(arr_time,  
  na.rm=TRUE), dep_delay=mean(dep_delay, na.rm=TRUE))
```

Here we are taking flights data frame and pass it to group_by() and the output of group_by() is passed to summarize(). Notice that the intermediate data frame is not created.

Dealing with Text Data

We need to install a few packages and load the packages

```
> install.packages('tidytext')
> install.packages('wordcloud')
> library(tidytext)
> library(ggplot2)
```

Let's define a string

```
> string <- c("Because I could not stop for Death -",
              "He kindly stopped for me -",
              "The Carriage held but just Ourselves -",
              "and Immortality")
```

A tibble is a modern class of data frame within R, available in the dplyr and tibble packages for text processing.

```
> text_df <- tibble(line = 1:4, text = string)
```

We can't filter out words or count which occur most frequently, since each row is made up of multiple combined words. We need to convert this so that it has one-token-per-document-per-row.

To tokenize the text, we use `unnest_tokens()` as follows.

```
> text_df %>% unnest_tokens(word, text)
```

Let's load the bbc text files.

```
> busn <- read.csv('BBC/business.txt', sep='\n', header=FALSE,
                  col.names='text')
> polc <- read.csv('BBC/politic.txt', sep='\n', header=FALSE, quote = "",
                  col.names='text')
```

Convert the data frames to tibble

```
> busn <- tibble(busn)
> polc <- tibble(polc)
> text_df = bind_rows(busn, polc)
```

Remove unwanted characters such as punctuation and some symbols.

```
> text_df <- text_df %>%
  mutate(clean_text = str_replace_all(text, "[:punct:]|Â£", "")) %>%
  select(clean_text)
```

Tokenize the text

```
> token_df <- text_df %>% unnest_tokens(word, clean_text)
```

To analyze the data, let's now count the words

```
> token_df %>% count(word, sort=TRUE)
```

We need to get rid of the “stop words”. The tidytext package has a way to filter out the common words that do not significantly contribute to the meaning of the overall text. The stop_words data frame is built into tidytext.

```
> data(stop_words)
```

Sample 40 random stop words

```
> stop_words %>% sample_n(40)
```

Now remove stop words from the document

```
> tidy_text <- token_df %>% anti_join(stop_words)
```

Let’s count the words. Notice words such "the" and "to" have been removed.

```
> tidy_text %>% count(word, sort=TRUE)
```

Let’s visualize using ggplot the common words

```
> tidy_text %>% count(word, sort=TRUE) %>% filter(n > 4) %>%  
  mutate(word=reorder(word, n)) %>%  
  ggplot(aes(word, n)) + geom_col() + xlab(NULL) + coord_flip()
```

We can add define words as stop words e.g. "ms", "yugansk" could be removed since they seems to carry no meaning. Plot the word count and notice that the words have been removed.

```
> tidy_text <- token_df %>%  
  filter(!word %in% c(stop_words$word, "yugansk", "ms"))
```

We can visualize the word cloud as follows.

```
> tidy_text %>% count(word) %>% with(wordcloud(word, n, max.words=100,  
  colors=brewer.pal(8, "Dark2")))
```

Exercises

Find all flights that had an arrival delay of three or more hours.

Find all flights that were operated by US Airways Inc., Delta Air Lines Inc., or Virgin America. Type airlines to view the carrier code.

Import the data from the other tables of the tweeter database

Print the data

Determine the relations among them

Comment 1012: what is the tweet message? Who post the tweet? Who post the comment?