Exploratory Data Analysis with R

Data Cleaning - Part I

Xuemao Zhang
East Stroudsburg University

October 3, 2022

Outline

- Dealing with missing data
- Tables and Tabulations
- Recoding Variables
- String functions
- Base R String functions

Data Cleaning

- In general, data cleaning is a process of investigating your data for inaccuracies, or recoding it in a way that makes it more manageable.
- MOST IMPORTANT RULE - LOOK AT YOUR DATA!
- We have seen some data cleaning techniques, such as renaming variables and removing missing values.

Useful checking functions

- `skimr::skim()` function to check data quality
- `is.na` - is `TRUE` if the data in a cell is missing, is `FALSE` otherwise
- `!` - negation (NOT)
  - if `is.na(x)` is `TRUE`, then `!is.na(x)` is `FALSE`
- `all` takes in a `logical` and will be `TRUE` if ALL are `TRUE`
  - `all(!is.na(x))` - are all values of `x` NOT NA
- `any` will be `TRUE` if ANY are true
  - `any(is.na(x))` - do we have any `NA`'s in `x`?
- `complete.cases` - returns `TRUE` if EVERY value of a row is NOT `NA`
  - very stringent condition; it will be `FALSE` for missing one value (even if not important)

Dealing with missing data: Missing data types

One of the most important aspects of data cleaning is missing values.

Types of "missing" data:

- `NA` : general missing data
- `NaN` : stands for "**N**ot **a** **N**umber", happens when you do 0/0.
- `Inf` and `-Inf` : Infinity, happens when you divide a positive number (or negative number) by 0.

Finding Missing data

Each missing data type has a function that returns `TRUE` if the data is missing:

- `NA` : `is.na`
- `NaN` : `is.nan`
- `Inf` and `-Inf` : `is.infinite`
  - `is.finite` returns `FALSE` for all missing data and `TRUE` for non-missing

Missing Data with Logicals

- One important aspect (esp with subsetting) is that logical operations return `NA` for `NA` values.
- The following missing value could be `> 2` or not, but we don't know. So `R` says there is no `TRUE` or `FALSE`, so that is missing.

```
x = c(0, NA, 2, 3, 4)
x > 2
```

```
## [1] FALSE    NA FALSE  TRUE  TRUE
```

Missing Data with Logicals

- What to do? What if we want if `x > 2` and `x` isn't `NA`?
- Don't do `x != NA`, do `x > 2` and `x` is NOT `NA`:

```
x != NA    #x==NA
```

```
## [1] NA NA NA NA NA
```

```
x > 2 & !is.na(x)
```

```
## [1] FALSE FALSE FALSE  TRUE  TRUE
```

Missing Data with Logicals

- What about seeing if a value is equal to multiple values? You can do `(x == 0 | x == 2) & !is.na(x)`, but that is not efficient.

```
(x == 0 | x == 2) # has NA
```

```
## [1]  TRUE    NA  TRUE FALSE FALSE
```

```
(x == 0 | x == 2) & !is.na(x) # No NA
```

```
## [1]  TRUE FALSE  TRUE FALSE FALSE
```

Missing Data with Logicals: %in%

- The %in% operator:

```
x %in% c(0, 2)    # It NEVER has NA and returns logical
```

```
## [1]  TRUE FALSE  TRUE FALSE FALSE
```

- Reads "return TRUE if x is in 0 or 2".

```
x %in% c(0, 2, NA)  # NEVER has NA and returns logical
```

```
## [1]  TRUE  TRUE  TRUE FALSE FALSE
```

Filtering and tibbles

- dplyr::filter removes missing values, have to keep them if you want them:

```
library(dplyr)
df = tibble(x = x) #make x a data frame
df %>% filter(x > 2)
```

```
## # A tibble: 2 x 1
##       x
##   <dbl>
## 1     3
## 2     4
```

```
filter(df, between(x, -1, 3) | is.na(x))
```

```
## # A tibble: 4 x 1
##       x
##   <dbl>
## 1     0
## 2    NA
## 3     2
## 4     3
```

```
#'dplyr::between' check if values in a numeric vector fall in specified ran
```

Missing Data with Operations

- Similarly with logicals, operations/arithmetic with `NA` will result in `NA`s:

```
x + 2;
```

```
## [1]  2 NA  4  5  6
```

```
x * 2;
```

```
## [1]  0 NA  4  6  8
```

Tables and Tabulations: Useful checking functions

- `unique` - gives you the unique values of a variable
- `table(x)` - will give a one-way table of `x`
  - `table(x, useNA = "ifany")` - will have row NA
- `table(x, y)` - will give a cross-tab of `x` and `y`

Creating One-way Tables

- Here we will use `table` to make tabulations of the data. Look at `?table` to see options for missing data.

```
unique(x);
```

```
## [1]  0 NA  2  3  4
```

```
table(x);
```

```
## x
## 0 2 3 4
## 1 1 1 1
```

```
table(x, useNA = "ifany");
```

```
## x
##    0    2    3    4 <NA>
##    1    1    1    1    1
```

Creating One-way Tables

- `useNA = "ifany"` will not have NA in table heading if no `NA`.

```
table(c(0, 1, 2, 3, 2, 3, 3, 2,2, 3), useNA = "ifany");
```

```
##
## 0 1 2 3
## 1 1 4 4
```

- You can set `useNA = "always"` to have it always have a column for `NA`

```
table(c(0, 1, 2, 3, 2, 3, 3, 2,2, 3), useNA = "always");
```

```
##
##    0    1    2    3 <NA>
##    1    1    4    4    0
```

Tables with Factors

- If you use a `factor`, all levels will be given even if no exist!
  - May be wanted or not

```
fac = factor(c(0, 1, 2, 3, 2, 3, 3, 2,2, 3), levels = 1:4);
fac;
```

```
##  [1] <NA> 1    2    3    2    3    3    2    2    3
## Levels: 1 2 3 4
```

```
tab = table(fac);
tab;
```

```
## fac
## 1 2 3 4
## 1 4 4 0
```

```
tab[ tab > 0 ]; #show the table with frequency >0 only
```

```
## fac
## 1 2 3
## 1 4 4
```

Creating Two-way Tables

- A two-way table. If you pass in 2 vectors, `table` creates a 2-dimensional table.

```
x1=c(0, 0, 0, 1, 1, 2, 3, 2, 3, 3, 2, 2,3,3,2,3,3);
y1=c(0, 0, 1, 1, 2, 3, 3, 2, 3, 3, 4, 4,3,2,2,1,1);
tab <- table(x1, y1, useNA = "ifany");
tab;
```

```
##     y1
## x1   0 1 2 3 4
##    0 2 1 0 0 0
##    1 0 1 1 0 0
##    2 0 0 2 1 2
##    3 0 2 1 4 0
```

Finding Row or Column Totals

- `margin.table` finds the marginal sums of the table.
- `margin` is 1 for rows, 2 for columns in general in `R`. Here is the column sums of the table:

```
margin.table(tab, 2);
```

```
## y1
## 0 1 2 3 4
## 2 4 4 5 2
```

Proportion Tables

- `prop.table` finds the marginal proportions of the table. Think of it dividing the table by it's respective marginal totals. If `margin` not set, divides by overall total.

```
prop.table(tab);
```

```
##    y1
## x1           0          1          2          3          4
##   0 0.11764706 0.05882353 0.00000000 0.00000000 0.00000000
##   1 0.00000000 0.05882353 0.05882353 0.00000000 0.00000000
##   2 0.00000000 0.00000000 0.11764706 0.05882353 0.11764706
##   3 0.00000000 0.11764706 0.05882353 0.23529412 0.00000000
```

```
prop.table(tab,2) * 100;
```

```
##    y1
## x1    0   1   2   3   4
##   0 100  25   0   0   0
##   1   0  25  25   0   0
##   2   0   0  50  20 100
##   3   0  50  25  80   0
```

Recoding to missing

- Sometimes people code missing data in weird or inconsistent ways.

```
ages = c(23,21,44,32,57,65,-999,54);
range(ages);
```

```
## [1] -999   65
```

- How do we change the -999 to be treated as missing?

```
ages[ages == -999] = NA;
range(ages);
```

```
## [1] NA NA
```

```
range(ages,na.rm=TRUE);
```

```
## [1] 21 65
```

```
# na.rm=TRUE will ignore NA's in the calculation
```

Recoding from missing

- What if you were the person that code missing values as -999?

```
ages;
```

```
## [1] 23 21 44 32 57 65 NA 54
```

```
is.na(ages);
```

```
## [1] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE
```

```
ages[is.na(ages)] = -999;
ages
```

```
## [1]   23   21   44   32   57   65 -999   54
```

Read in the UFO dataset

- Read in data

```
library(readr)
ufo = read_csv("../data/ufo_sightings.csv")
```

```
## Rows: 80332 Columns: 11
## -- Column specification --------------------------------
## Delimiter: ","
## chr (8): date_time, city_area, state, country, ufo_shape, describ
## dbl (3): encounter_length, latitude, longitude
##
## i Use `spec()` to retrieve the full column specification for this
## i Specify the column types or set `show_col_types = FALSE` to qui
```

```
problems(ufo)  # Retrieve parsing problems
```

```
## # A tibble: 0 x 5
## # ... with 5 variables: row <int>, col <int>, expected <chr>, act
## #   file <chr>
```

Read in the UFO dataset

```
library(dplyr)
glimpse(ufo)
```

```
## Rows: 80,332
## Columns: 11
## $ date_time                 <chr> "10/10/1949 20:30", "10/10/194
## $ city_area                 <chr> "san marcos", "lackland afb",
## $ state                     <chr> "tx", "tx", NA, "tx", "hi", "t
## $ country                   <chr> "us", NA, "gb", "us", "us", "u
## $ ufo_shape                 <chr> "cylinder", "light", "circle",
## $ encounter_length          <dbl> 2700, 7200, 20, 20, 900, 300,
## $ described_encounter_length <chr> "45 minutes", "1-2 hrs", "20 s
## $ description               <chr> "This event took place in earl
## $ date_documented           <chr> "4/27/2004", "12/16/2005", "1/
## $ latitude                  <dbl> 29.88306, 29.38421, 53.20000,
## $ longitude                 <dbl> -97.941111, -98.581082, -2.916
```

Read in the UFO dataset

```
library(skimr)
skim(ufo)
```

Checking for logical conditions

- Recall that
  - `any()` - checks if there are any `TRUE`s
  - `all()` - checks if ALL are true

```
any(is.na(ufo$state)); # are there any NAs?
```

```
## [1] TRUE
```

```
table(is.na(ufo$state)); # are there any NAs?
```

```
##
## FALSE  TRUE
## 74535  5797
```

Recoding Variables: base R

- For example, let's say gender was coded as Male, M, m, Female, F, f. Using Excel to find all of these would be a matter of filtering and changing all by hand or using if statements.
- In R, you can simply do something like this.

```
data$gender[data$gender %in% c("Male", "M", "m")] <- "Male"
```

Example of Cleaning: more complicated

- Sometimes though, it's not so simple. That's where functions that find patterns come in very useful.

```
set.seed(7); # random sample below - make sure same every time
gender <- sample(c("Male", "mAle", "MaLe", "M",
                   "MALE", "Ma", "FeMAle", "F",
                   "Woman", "Man", "Fm", "FEMALE"),
                 1000, replace = TRUE);
```

```
table(gender)
```

```
## gender
##      F FeMAle FEMALE     Fm      M     Ma   mAle   Male   MaLe
##     78     79     93     81     85     83     96     83     79
##  Woman
##     76
```

Useful String Functions

Useful String functions

- `toupper()`, `tolower()` - uppercase or lowercase your data
- `str_trim()` (in the `stringr` package) or `trimws` in base
  - will trim whitespace
- `nchar()` - get the number of characters in a string
- `paste()` - paste strings together with a space
- `paste0()` - paste strings together with no space as default

Pasting strings with paste and paste0

- Paste can be very useful for joining vectors together.

```
paste("Visit", 1:5, sep = "_")
```

```
## [1] "Visit_1" "Visit_2" "Visit_3" "Visit_4" "Visit_5"
```

```
paste("Visit", 1:5, sep = "_", collapse = ";")
```

```
## [1] "Visit_1;Visit_2;Visit_3;Visit_4;Visit_5"
```

```
paste("To", "is going be the ", "we go to the store!", sep = "day ")
```

```
## [1] "Today is going be the day we go to the store!"
```

```
# and paste0 can be even simpler see ?paste0
paste0("Visit",1:5)
```

```
## [1] "Visit1" "Visit2" "Visit3" "Visit4" "Visit5"
```

Paste Depicting How Collapse Works

```
paste(1:5);
```

```
## [1] "1" "2" "3" "4" "5"
```

```
paste(1:5, collapse = " ");
```

```
## [1] "1 2 3 4 5"
```

The `stringr` package

Like `dplyr`, the `stringr` package:

- Makes some things more intuitive
- Is different than base R
- Has a standard format for most functions
  - the first argument is a string which is like the first argument is a `data.frame` in `dplyr`

Substringing

Package `stringr`:

- `str_sub(x, start, end)`
  - substrings from position start to position end
- `str_split(string, pattern)`
  - splits strings up and returns a list!

Splitting String: stringr

- In stringr, strsplit splits a vector on a string into a list

```
library(stringr)
x <- c("I really", "like writing", "R code programs")
x  #length(x)
```

```
## [1] "I really"        "like writing"    "R code programs"
```

```
y <- str_split(x, pattern = " ")  # returns a list
y
```

```
## [[1]]
## [1] "I"      "really"
##
## [[2]]
## [1] "like"    "writing"
##
## [[3]]
## [1] "R"        "code"     "programs"
```

Splitting String: use a fixed expression

- One special case is when you want to split on a period ".". In regular expressions . means **ANY** character, so

```
str_split("I.like.strings", ".");
```

```
## [[1]]
##  [1] "" "" "" "" "" "" "" "" "" "" "" "" "" "" ""
```

```
str_split("I.like.strings", fixed("."));
```

```
## [[1]]
## [1] "I"       "like"    "strings"
```

```
# fixed Compares literal bytes in the string
```

Extracting from a string using purrr

- The purrr package allows you to more easily interface with lists.

https://purrr.tidyverse.org/

- The main function family for this is map()
  - The map functions transform their input by applying a function to each element of a list or atomic vector and returning an object of the same length as the input
- map_chr() takes a list and returns a character vector

```
library(purrr)
map_chr(y, first)  # y is a list
```

```
## [1] "I"    "like" "R"
```

```
map_chr(y, nth, 2)
```

```
## [1] "really"  "writing" "code"
```

```
map_chr(y, last)
```

```
## [1] "really"   "writing"  "programs"
```

'Find' functions: `stringr`

- `str_detect`, `str_subset`, `str_replace`, and `str_replace_all` search for matches to argument pattern within each element of a character vector: they differ in the format of and amount of detail in the results.

- `str_detect` - returns `TRUE` if `pattern` is found
- `str_subset` - returns only the strings which pattern were detected
  - convenient wrapper around `x[str_detect(x, pattern)]`
- `str_extract` - returns only strings which pattern were detected, but ONLY the **pattern**
- `str_replace` - replaces `pattern` with `replacement` the first time
- `str_replace_all` - replaces `pattern` with `replacement` as many times matched

Let's look at modifier for `stringr`

```
?modifiers
```

```
## starting httpd help server ... done
```

- `fixed` - match everything exactly
- `regexp` - default - uses **reg**ular **exp**ressions: http://www.regular-expressions.info/reference.html)%3C
- `ignore_case` is an option to not have to use `tolower`

Regular Expressions

Further reading

- R for Data Science, Chapter 14: https://r4ds.had.co.nz/strings.html
- Regular expressions in R vignette: https://cran.r-project.org/web/packages/stringr/vignettes/regular-expressions.html
  - Introduction to stringr: https://cran.r-project.org/web/packages/stringr/vignettes/stringr.html
- Regular expressions cheat sheet by Microsoft: https://download.microsoft.com/download/c/3/e/c3ef6850-d455-478a-afbe-b89e57df8569/Regular_Expressions_Cheat_Sheet.pdf

'Find' functions: Finding Indices

- These are the indices where the pattern match occurs.

```
which(str_detect(ufo$description, "two aliens"))
```

```
## [1]  1588 55518
```

```
str_detect(ufo$description, "two aliens") %>% head()
```

```
## [1] FALSE FALSE FALSE FALSE FALSE FALSE
```

```
#it returns a logic vector
```

'Find' functions: finding values, stringr and dplyr

```
str_subset(ufo$description, "two aliens")
```

```
## [1] "((HOAX??))  two aliens appeared from a bright light to peacefully i
## [2] "Witnessed two aliens walking along baseball field fence."
```

```
#it returns strings with the pattern
ufo %>% filter(str_detect(description, "two aliens"))
```

```
## # A tibble: 2 x 11
##   date_t~1 city_~2 state country ufo_s~3 encou~4 descr~5 descr~6 date_~7
##   <chr>    <chr>   <chr> <chr>   <chr>     <dbl> <chr>   <chr>   <chr>
## 1 10/14/2~ yuma    va    us      format~     300 5 minu~ ((HOAX~ 4/27/2~
## 2 7/1/200~ north ~ ct    <NA>    unknown      60 1 minu~ Witnes~ 10/19/~
## # ... with 1 more variable: longitude <dbl>, and abbreviated variable na
## #   1: date_time, 2: city_area, 3: ufo_shape, 4: encounter_length,
## #   5: described_encounter_length, 6: description, 7: date_documented,
## #   8: latitude
```

Showing differnce in `str_extract`

- `str_extract` extracts just the **matched string**

```
ss = str_extract(ufo$description, "two aliens")
class(ss)
```

```
## [1] "character"
```

```
head(ss)
```

```
## [1] NA NA NA NA NA NA
```

```
ss[ !is.na(ss)] #two mathed strings are returned
```

```
## [1] "two aliens" "two aliens"
```

Showing differnce in `str_extract`

- Look for any comment that **starts with** "aliens"
  - ^ start of string
  - . The dot matches a single character, without caring what that character is.
  - * Match-zero-or-more characters

```
str_subset(ufo$description, "^aliens.*")
```

```
## [1] "aliens speak german???" "aliens in srilanka"
```

Using Regular Expressions

- That contains space then ship maybe with stuff in between.
  - .? Match-zero-or-one character

```
str_subset(ufo$description, "space.?ship") %>% head(7)
```

```
## [1] "I saw the cylinder shaped looked like a spaceship hovring ab
## [2] "description of a spaceship spotted over Birmingham Alabama i
## [3] "A space ship was descending to the ground"
## [4] "On Monday october 3&#44 2005&#44 I spotted two spaceships in
## [5] "Me and my daughter seen the most beautiful shiney spaceship.
## [6] "I saw a Silver space ship rising into the early morning sky
## [7] "Saw a space ship hanging over the southern (Manzano) portion
```

Ordering

```
sort(c("1", "2", "10")) #  not sort correctly (order simply ranks th
```

```
## [1] "1"  "10" "2"
```

```
order(c("1", "2", "10"));
```

```
## [1] 1 3 2
```

So we must change a string (containing numbers only) into a numeric to order.

Replace

- Let's say we wanted to sort the data set by latitude and longitude:

```
class(ufo$latitude);
```

```
## [1] "numeric"
```

```
any(is.na(ufo$latitude));
```

```
## [1] TRUE
```

```
which(is.na(ufo$latitude));
```

```
## [1] 43783
```

- Dropping bad observations

```
dim(ufo);
```

```
## [1] 80332    11
```

```
dropIndex = which(is.na(ufo$latitude) | is.na(ufo$longitude));
ufo_clean = ufo[-dropIndex,];
dim(ufo_clean);  # 1 observation is dropped
```

```
## [1] 80331    11
```

Ordering

```
ufo2 = ufo_clean;
#ufo2$latitude = as.numeric(ufo2$latitude)
#ufo2$longitude = as.numeric(ufo2$longitude)
ufo2 <- ufo2[order(ufo2$latitude, ufo2$longitude), ];
ufo2[1:5, c("date_time", "latitude", "longitude")];
```

```
## # A tibble: 5 x 3
##   date_time        latitude longitude
##   <chr>               <dbl>     <dbl>
## 1 5/15/1994 13:00    -82.9     -135
## 2 4/14/2002 22:22    -46.4      168.
## 3 10/23/2008 4:45    -46.2      170.
## 4 3/11/2009 0:00     -45.1      171.
## 5 3/28/2012 6:30     -45.0      169.
```

Special characters like money/$

```
money = tibble(group = letters[1:5],
  amount = c("$12.32", "$43.64", "$765.43", "$93.31", "$12.13"))

money %>% arrange(amount)
```

```
## # A tibble: 5 x 2
##   group amount
##   <chr> <chr>
## 1 e     $12.13
## 2 a     $12.32
## 3 b     $43.64
## 4 c     $765.43
## 5 d     $93.31
```

```
as.numeric(money$amount)
```

```
## Warning: NAs introduced by coercion
```

```
## [1] NA NA NA NA NA
```

Special characters like money/$

- One solution is replacing the $ sign with an empty string and convert to numeric:
  - `fixed` is used because $ in regular expression means the end of string

```
money$amountNum = as.numeric(str_replace(money$amount, fixed("$"), ""))
```

- A much easier way is using `readr::parse_number()`

```
money$amount = parse_number(money$amount);
money;
```

```
## # A tibble: 5 x 2
##   group amount
##   <chr>  <dbl>
## 1 a       12.3
## 2 b       43.6
## 3 c      765.
## 4 d       93.3
## 5 e       12.1
```

Base R versions: Substrings

- Base R
  - `substr(x, start, stop)` - substrings from position start to position stop
  - `strsplit(x, split)` - splits strings up - returns list!

Splitting String: base R

- In base R, `strsplit` like `stringr::str_split` splits a vector on a string into a list

```
x <- c("I really", "like writing", "R code programs");
y <- strsplit(x, split = " "); # returns a list
   #str_split(x, pattern = " ");
y;
```

```
## [[1]]
## [1] "I"      "really"
##
## [[2]]
## [1] "like"    "writing"
##
## [[3]]
## [1] "R"        "code"     "programs"
```

'Find' functions: base R

- `grep`: `grep`, `grepl`, `regexpr` and `gregexpr` search for matches to argument pattern within each element of a character vector: they differ in the format of and amount of detail in the results.
- `grep(pattern, x, fixed=FALSE)`, where:
  - pattern = character string containing a regular expression to be matched in the given character vector.
  - x = a character vector where matches are sought, or an object which can be coerced by as.character to a character vector.
  - If fixed=TRUE, it will do exact matching for the phrase anywhere in the vector (regular find)

'Find' functions: stringr compared to base R

Base R does not use these functions. Here is a "translator" of the `stringr` function to base R functions

- `str_detect` - similar to `grepl` (return logical)
  - `grep(value = FALSE)` is similar to `which(str_detect())`
- `str_subset` - similar to `grep(value = TRUE)` - return value of matched
- `str_replace` - similar to `sub` - replace one time
- `str_replace_all` - similar to `gsub` - replace many times

Important Comparisons

Base R:

- Argument order is `(pattern, x)`
- Uses option `(fixed = TRUE)`

`stringr`

- Argument order is `(string, pattern)`
- Uses function `fixed(pattern)`

License