Data Cleaning

Data Wrangling in R

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!

Useful checking functions

- `is.na` - is `TRUE` if the data 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
  - `FALSE` 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 take 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. Think about it, the data could be > 2 or not 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
```

```
[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 == 1 | 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
```

what to do?

Missing Data with Logicals: %in%

Introduce the `%in%` operator:

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

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

reads "return `TRUE` if `x` is in 0 or 2". (Like `inlist` in Stata).

Missing Data with Logicals: %in%

NEVER has NA, even if you put it there (BUT DON'T DO THIS):

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

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

```
x %in% c(0, 2) | is.na(x)
```

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

Filtering and tibbles

Filter removes missing values, have to keep them if you want them:

```
df = tibble(x = x)
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
```

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") # will not
```

```
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
```

Creating One-way Tables

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)
tab = table(fac)
tab
```

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

```
tab[ tab > 0 ]
```

```
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.

```
tab <- table(c(0, 0, 0, 1, 1,2, 3, 2, 3, 3, 2,2, 3,3,2,3,3)
             c(0, 0,1, 1, 2, 3, 3, 2, 3, 3, 4, 4, 3,2,2,1,1
              useNA = "always")
```

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)
```

```
   0    1    2    3    4 <NA>
   2    4    4    5    2    0
```

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.

```r
prop.table(tab)
```

```
                0          1          2          3
  0    0.11764706 0.05882353 0.00000000 0.00000000 0.000000
  1    0.00000000 0.05882353 0.05882353 0.00000000 0.000000
  2    0.00000000 0.00000000 0.11764706 0.05882353 0.117647
  3    0.00000000 0.11764706 0.05882353 0.23529412 0.000000
  <NA> 0.00000000 0.00000000 0.00000000 0.00000000 0.000000
```

```r
prop.table(tab,1) * 100
```

```
              0        1        2        3        4      <N
  0    66.66667 33.33333  0.00000  0.00000  0.00000  0.0000
  1     0.00000 50.00000 50.00000  0.00000  0.00000  0.0000
```

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
```

Recoding to missing

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

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

```
[1] NA NA
```

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

```
[1] 21 65
```

Recoding from missing

What if you were the person that coded the -999

```
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 from RStudio Cloud or download from: http://sisbid.github.io/Module1/data/ufo/ufo_data_complete.csv.gz

```
ufo = read_csv("../data/ufo/ufo_data_complete.csv")
```

```
Parsed with column specification:
cols(
  datetime = col_character(),
  city = col_character(),
  state = col_character(),
  country = col_character(),
  shape = col_character(),
  `duration (seconds)` = col_double(),
  `duration (hours/min)` = col_character(),
  comments = col_character(),
  `date posted` = col_character(),
  latitude = col_character(),
  longitude = col_double()
)
```

Data cleaning "before" R

You saw warning messages when reading in this dataset.

```
p = problems(ufo)
p
```

```
# A tibble: 199 x 5
     row col   expected   actual     file
   <int> <chr> <chr>      <chr>      <chr>
 1   877 <NA>  11 columns 12 columns '../data/ufo/ufo_data_
 2  1712 <NA>  11 columns 12 columns '../data/ufo/ufo_data_
 3  1814 <NA>  11 columns 12 columns '../data/ufo/ufo_data_
 4  2857 <NA>  11 columns 12 columns '../data/ufo/ufo_data_
 5  3733 <NA>  11 columns 12 columns '../data/ufo/ufo_data_
 6  4755 <NA>  11 columns 12 columns '../data/ufo/ufo_data_
 7  5388 <NA>  11 columns 12 columns '../data/ufo/ufo_data_
 8  5422 <NA>  11 columns 12 columns '../data/ufo/ufo_data_
 9  5613 <NA>  11 columns 12 columns '../data/ufo/ufo_data_
10  5848 <NA>  11 columns 12 columns '../data/ufo/ufo_data_
# ... with 189 more rows
```

Data cleaning "before" R

```
ufo[p$row, ]
```

```
# A tibble: 199 x 11
   datetime city  state country shape `duration (seco~ `dur
   <chr>    <chr> <chr> <chr>   <chr>            <dbl> <chr
 1 10/1/20~ <NA>  <NA>  <NA>    <NA>                 0 <NA>
 2 10/14/2~ <NA>  <NA>  <NA>    <NA>                 0 <NA>
 3 10/14/2~ <NA>  nv    <NA>    <NA>                 0 ligh
 4 10/17/2~ <NA>  tx    <NA>    <NA>                 0 oval
 5 10/20/2~ <NA>  ct    <NA>    <NA>                 0 egg
 6 10/24/2~ <NA>  sd    <NA>    <NA>                 0 sphe
 7 10/27/2~ <NA>  ny    <NA>    <NA>                 0 <NA>
 8 10/27/2~ <NA>  mi    <NA>    <NA>                 0 ligh
 9 10/28/2~ <NA>  <NA>  <NA>    <NA>                 0 <NA>
10 10/29/2~ <NA>  oh    <NA>    <NA>                 0 <NA>
# ... with 189 more rows, and 4 more variables: comments <c
#   posted` <chr>, latitude <chr>, longitude <dbl>
```

Data cleaning "before" R

Let's just drop those rows for now

```
ufo = ufo[-p$row,]
```

Checking for logical conditions

- `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
81268  7408
```

Recoding Variables

Example of Recoding: 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:

```
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.

```
table(gender)
```

```
gender
     F FeMAle FEMALE     Fm      M     Ma   mAle   Male   M
    80     88     76     87     99     76     84     83
   Man  Woman
    84     71
```

String functions

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
```

```
[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

```r
paste(1:5)
```

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

```r
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
- Is used on forums for answers
- Has a standard format for most functions
  - the first argument is a string like first argument is a `data.frame` in `dplyr`

Splitting/Find/Replace and Regular Expressions

- R can do much more than find exact matches for a whole string
- Like Perl and other languages, it can use regular expressions.
- What are regular expressions?
  - Ways to search for specific strings
  - Can be very complicated or simple
  - Highly Useful - think "Find" on steroids

A bit on Regular Expressions

- http://www.regular-expressions.info/reference.html
- They can use to match a large number of strings in one statement
- `.` matches any single character
- `*` means repeat as many (even if 0) more times the last character
- `?` makes the last thing optional
- `^` matches start of vector `^a` - starts with "a"
- `$` matches end of vector `b$` - ends with "b"

Substringing

`stringr`

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

Splitting String: stringr

In base R, strsplit splits a vector on a string into a list

```
x <- c("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"
```

Using a fixed expression

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

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

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

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

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

Let's extract from `y`

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

https://purrr.tidyverse.org/

The main function family for this is `map()`

Let's extract from y

map_chr() takes a list and returns a character vector

```
map_chr(y, first) # on the fly
```

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

```
map_chr(y, nth, 2) # on the fly
```

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

```
map_chr(y, last) # on the fly
```

```
[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`

- `fixed` - match everything exactly
- `regexp` - default - uses **reg**ular **exp**ressions
- `ignore_case` is an option to not have to use `tolower`

‘Find’ functions: Finding Indices

These are the indices where the pattern match occurs:

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

```
[1]  1728 61579
```

'Find' functions: Finding Logicals

These are the indices where the pattern match occurs:

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

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

‘Find’ functions: finding values, stringr and dplyr

```r
str_subset(ufo$comments, "two aliens")
```

```
[1] "((HOAX??))  two aliens appeared from a bright light to
[2] "Witnessed two aliens walking along baseball field fenc
```

```r
ufo %>% filter(str_detect(comments, "two aliens"))
```

```
# A tibble: 2 x 11
  datetime city  state country shape `duration (seco~ `dura
  <chr>    <chr> <chr> <chr>   <chr>            <dbl> <chr>
1 10/14/2~ yuma  va    us      form~              300 5 min
2 7/1/200~ nort~ ct    <NA>    unkn~               60 1 min
# ... with 4 more variables: comments <chr>, `date posted`
#   latitude <chr>, longitude <dbl>
```

Showing differnce in str_extract

str_extract extracts just the matched string

```r
ss = str_extract(ufo$comments, "two aliens")
head(ss)
```

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

```r
ss[ !is.na(ss)]
```

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

- Look for anycomment that starts with "aliens"

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

```
[1] "aliens speak german???" "aliens exist"
[3] "aliens in srilanka"
```

Using Regular Expressions

That contains space then ship maybe with stuff in between

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

```
[1] "I saw the cylinder shaped looked like a spaceship hovr
[2] "description of a spaceship spotted over Birmingham Ala
[3] "A space ship was descending to the ground"
[4] "On Monday october 3&#44 2005&#44 I spotted two spacesh
[5] "Me and my daughter seen the most beautiful shiney spac
[6] "I saw a Silver space ship rising into the early mornin
[7] "Saw a space ship hanging over the southern (Manzano) p
```

Replace

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

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

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

```
sort(c("1", "2", "10")) #  not sort correctly (order simpl
```

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

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

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

Replace

So we must change the coordinates into a numeric:

```
head(ufo$latitude, 4)
```

```
[1] "29.8830556" "29.38421"   "53.2"       "28.9783333"
```

```
head(as.numeric(ufo$latitude), 4)
```

```
Warning in head(as.numeric(ufo$latitude), 4): NAs introduce
```

```
[1] 29.88306 29.38421 53.20000 28.97833
```

Dropping bad observations

```
dropIndex = which(is.na(as.numeric(ufo$latitude)) |
                        is.na(as.numeric(ufo$longitude)))
```

```
Warning in which(is.na(as.numeric(ufo$latitude)) |
is.na(as.numeric(ufo$longitude))): NAs introduced by coerci
```

```
ufo_clean = ufo[-dropIndex,]
dim(ufo_clean)
```

```
[1] 88675    11
```

Ordering

```r
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("datetime", "latitude", "longitude")]
```

```
# A tibble: 5 x 3
  datetime          latitude longitude
  <chr>                <dbl>     <dbl>
1 5/15/1994 13:00      -82.9     -135
2 4/14/2002 22:22      -46.4      168.
3 7/3/2002 22:35       -46.4      168.
4 10/23/2008 04:45     -46.2      170.
5 6/1/1998 22:00       -45.7      171.
```

Replacing and subbing: stringr

We can do the same thing (with 2 piping operations!) in dplyr

```
ufo_dplyr = ufo_clean
ufo_dplyr = ufo_dplyr %>% mutate(
  latitude = latitude %>% as.numeric,
  longitude = longitude %>% as.numeric) %>%
    arrange(latitude,longitude)
ufo_dplyr[1:5, c("datetime", "latitude", "longitude")]
```

```
# A tibble: 5 x 3
  datetime            latitude longitude
  <chr>                  <dbl>     <dbl>
1 5/15/1994 13:00        -82.9    -135
2 4/14/2002 22:22        -46.4     168.
3 7/3/2002 22:35         -46.4     168.
4 10/23/2008 04:45       -46.2     170.
5 6/1/1998 22:00         -45.7     171.
```

Special characters like money/$

```
money = tibble(group = letters[1:5],
  amount = c("$12.32", "$43.64", "$765.43", "$93.31", "$12.
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/$

In the past, we would recommend just replacing the $ sign with an empty string and convert to numeric:

```
money$amountNum = as.numeric(str_replace(money$amount, fixe
money %>% arrange(amountNum)
```

```
# A tibble: 5 x 3
  group amount  amountNum
  <chr> <chr>       <dbl>
1 e     $12.13       12.1
2 a     $12.32       12.3
3 b     $43.64       43.6
4 d     $93.31       93.3
5 c     $765.43     765.
```

Special characters like money/$

But now there are better helper functions for this:

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

This is way easier. . . "This drops any non-numeric characters before or after the first number. The grouping mark specified by the locale is ignored inside the number."

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` 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
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)` aka `(x, pattern)`
- Uses function `fixed(pattern)`