Data Cleaning Tutorials

Eilif Mikkelsen

Contents

1 Introduction

In an academic setting data sets are often presented as clean and orderly sets of information. For most real-world applications this couldn't be farther from the truth. When talking to data scientists across industries it is commonly noted that the vast majority of their time is spent cleaning and structuring data before they can even consider models.

In this tutorial we will cover some core data cleaning methods. This is by no means exhaustive however it should leave the reader with a strong set of data manipulation skills. For context, I wrote this document to hone my data manipulation skills in R after spending years working with the Python Pandas data library which provides data structures and features similar to the R `data.frame` object. Periodically, comments detailing the `pandas` equivalents will be included looking something like `# Py/Pandas: df.head()`.

This document was originally written in R Markdown, a format for generating mixed format documents combining code and text. Where it seems useful, I will include links to relevent packages or functions. Here is a link to a cheat sheet on the R Markdown syntax.

2 What is Presumed About the Reader

- The reader has a very basic understanding of simple programming concepts.

- The reader is has cursory knowledge of the `data.frame` tabular data structure and its methods.
- The reader remembers that arrays in R are 1-indexed rather than 0-indexed.

I will use `=` rather than `<-` out of habit from my work in Python. So far as I can tell, they are equivalent.

3 Setup

Here is the output information on the operating system and R version used to generate this tutorial. The tools used in this tutorial are elemental functions of R as such is it not expected that this tutorial will become out-of-date. If you find issues, please feel free to edit this file and submit a pull request to this repository. Where specific packages are required, additional notes will be made.

```
version
```

```
##                _
## platform       x86_64-apple-darwin13.4.0
## arch           x86_64
## os             darwin13.4.0
## system         x86_64, darwin13.4.0
## status
## major          3
## minor          3.3
## year           2017
## month          03
## day            06
## svn rev        72310
## language       R
## version.string R version 3.3.3 (2017-03-06)
## nickname       Another Canoe
```

4 Topics Covered

- Cleaning string categories
- Context aware string categorization
- Context aware unit standardization
- Datetime parsing

5 Initial Look

I once had a professor that said that no matter how good and experienced and smart we are, there is no substitute for viewing and plotting your data!

Let's start by loading and looking at the first few rows of the data. The ix columns is an arbitrary row number that was created along with the dataset. We will use this to uniquely identify rows. Additionally, we want all string columns to be treated as characters rather than "factors", a special R data type.

5.1 Commands Used

- `hist`
- `read.csv`

- row.names
- head
- nrow
- ncol

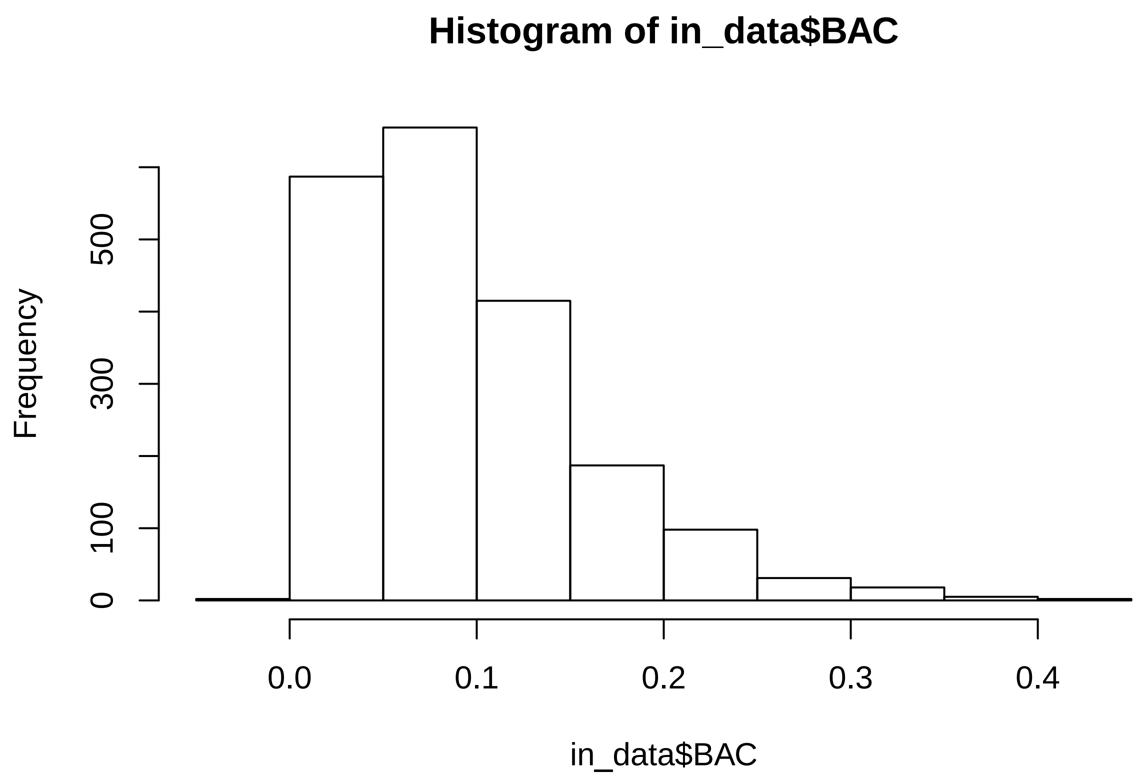
```
# Loading the CSV
# Py/Pandas: pd.read_csv('./horrible_data.csv', index=0)
in_data = read.csv('./horrible_data.csv', stringsAsFactors=FALSE)

# Let's duplicate the data so we can keep a copy of the original. When we update data we will update wr
wrk_data = cbind(in_data)
# Setting the row names to the `ix` column
row.names(in_data) = in_data$ix

# Py/Pandas: df.head()
head(in_data)
```

```
##   ix        BAC     age            collection_date             drink_type
## 0  0 0.05725438  1/7/66 2018-01-29T12:21:46.232643                 Merlot
## 1  1 0.15269702 8/19/68 2018-01-29T12:21:46.233977 Franziskaner Weissbier
## 2  2 0.05564855      50 2018-01-29T12:21:46.234572                   Wine
## 3  3 0.03288376     671 2018-01-29T12:21:46.235095                   Wine
## 4  4 0.03005566  4/2/60 2018-01-29T12:21:46.235582                    Vin
## 5  5 0.02840692 9/14/74 2018-01-29T12:21:46.236121                whiskey
##   num_drinks    sex  units volume_consumed   weight
## 0        2.4      M                12.0000 177.8200
## 1        4.6      F metric       1634.5534 237.3310
## 2        2.5   Male                12.5000 185.4400
## 3        0.9 female metric        134.6799 265.0237
## 4        1.7   meal     SI        253.6016 315.3754
## 5        0.6 Female metric         28.0873 225.4100
```

```
# Let's use the super handy built-in to generate a histogram
hist(in_data$BAC)
```



```r
# Let's also look at the overall shape of the dataset.
nrow(in_data)
```

```
## [1] 2000
```

```r
ncol(in_data)
```

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

6 Standardizing Text (string) Columns

When faced with text variables, figuring out how to group and organize the values into usable data can be extremely time consuming and difficult.

Immediately upon observing the first 6 rows of the data we see that there are several variations of the same category for both the `units` and `sex` column. The goal is to identify what values in the column correspond to what categories. The `drink_type` column also needs to be cleaned in a similar manner but it requires additional discussion.

I used this review of R subsetting methods to brush up for this tutorial. Be sure to also remeber that a dataframe has two dimensions, rows and columns. Let `rdf` be an R data frame. If we wish to select the first column for all observations (rows) we would call `rdf[, 1]` as dataframe selecting follows [row, column] convention.

6.1 Commands Used:

- `unique`
- `which`
- `c`

Let's print out the list of unique items in the sex column

```r
sex_vals = unique(in_data$sex)
sex_vals
```

```
## [1] "M"      "F"      "Male"   "female" "meal"   "Female" "femeal" ""
## [9] "male"
```

Conditional indexing allows for the selection of rows based on matching column conditions. For those readers familiar with general programming methods, this is essentially the creation of a binary mask. Here we will look for the rows where the `sex` column equals "Male". We can make these masks as elaborate as required by combining boolean criteria with `&` and `|`.

```r
# Selecting rows where
head(in_data[which(in_data$sex == "Male"), ])
```

```
##    ix        BAC age            collection_date            drink_type
## 2   2 0.05564855  50 2018-01-29T12:21:46.234572                  Wine
## 13 13 0.09123496 746 2018-01-29T12:21:46.237715 Franziskaner Weissbier
## 20 20 0.04443760  53 2018-01-29T12:21:46.238809                Scotch
## 22 22 0.01751559 774 2018-01-29T12:21:46.239154            Warsteiner
## 24 24 0.06256844 584 2018-01-29T12:21:46.239470                   IPA
## 41 41 0.09546701 556 2018-01-29T12:21:46.242135                Merlot
##    num_drinks  sex  units volume_consumed   weight
## 2         2.5 Male                 12.5000 185.4400
## 13        3.6 Male                 43.2000 171.7100
## 20        1.4 Male                  2.1000 167.3100
## 22        0.4 Male metric         140.5923 280.8853
## 24        3.1 Male                 37.2000 231.9700
## 41        3.3 Male metric         488.3116 238.5470
```

What is the which command doing? It is applying the boolean filters provided and returning the row numbers/names of the rows for which the filters return TRUE. If we print out the results of the which command we can see this.

```r
head(which(in_data$sex == "Male"))
```

```
## [1]  3 14 21 23 25 42
```

We see from the output that this dataset has two correct options for `sex`, Male and Female. We also observe that the dataset contains a variety of abbreviations and typos for the two categories. To standardize the values such that all observations that should be recorded as Male are updated accordingly. We will perform the same operation for Female.

To acomplish this task, we will need to use the `%in%` operator. The `%in%` operator checks if a value is in the supplied set. In a pseudocode example let `s = {1, 3, 6}`. The evaluation of `1 in s` returns `TRUE` while `2 in s` returns `FALSE`.

```r
# A vector of all the values that should be classified as "Male"
male_values = c("Male", "meal", "male", "M")

# Now we will use the `%in%` operator to select all rows where the `sex` column value is in the set of
# possible values.
head(in_data[which(in_data$sex %in% male_values), c("BAC", "sex")], n = 10)
```

```
##           BAC  sex
## 0  0.057254381    M
## 2  0.055648547 Male
## 4  0.030055655 meal
## 9  0.055791317 meal
## 13 0.091234964 Male
## 14 0.043404374 male
## 15 0.005654698    M
## 18 0.075756832    M
## 20 0.044437597 Male
## 21 0.017537210    M
```

Now that we know how to select the rows that need to be updates, we can update wrk_data directly. We can set a column of the filtered dataframe to a column and only the selected rows will be updates.

```r
wrk_data[which(wrk_data$sex %in% male_values), "sex"] = "Male"

# Let's do the same thing for "Female"
female_values = c("F", "female", "Female", "femeal")
wrk_data[which(wrk_data$sex %in% female_values), "sex"] = "Female"

# SUCCESS! If we examine the unique set of values for `sex` in wrk_data we see that the only option for
unique(wrk_data$sex)
```

```
## [1] "Male"   "Female" ""
```

Alright! We have learned how to update rows/columns in a dataframe using boolean masking. The `units` column has similar issues. I won't go through the explaination again but the steps are the same as those for `sex` above.

```r
# Let's print out the list of unique items in the units column
unique(in_data$units)
```

```
## [1] ""       "metric" "SI"
```

```r
metric_values = c("metric", "SI")
wrk_data[which(wrk_data$units %in% metric_values), "units"] = "metric"

# It's bad practice to have the absense of information imply a definitive value.
# I know (because I made the dataset) that all values that are not marked as metric
# use the America/imperial units.
wrk_data[which(!(wrk_data$units %in% metric_values)), "units"] = "imperial"

# Let's check our work.
unique(wrk_data$units)
```

```
## [1] "imperial" "metric"
```

6.2 `drinks_type`: A Demonstration in Data Context

Before I go into what the values for this column mean, let's start by just looking at them. What do you notice?

```r
unique(wrk_data$drink_type)
```

```
##  [1] "Merlot"                    "Franziskaner Weissbier"
##  [3] "Wine"                      "Vin"
##  [5] "whiskey"                   "Warsteiner"
##  [7] "shots"                     "Vino"
##  [9] "Scotch"                    "IPA"
## [11] "Beer"                      "vidka"
## [13] "Bud Light"              "Cab"
```

What we have here is known in the industry as a “hot mess”. If you didn’t guess, the values are a mix of alocholic beverage brands and types in a variety of languages. When encountering this sort of data, it’s often worth taking the 20 mins. to go an Google each item. I am not well read in the boozes of the world. A quick Google search reveals that “Warsteiner” and “Franziskaner Weissbier” are beer. Once we know what everything is we can generate our list of categories and assign the options to a specific category as we did with `sex` and `units` The resulting code would look something like this.

```
# Vectors for each type of alcoholic beverage.
wine_options = c("Merlot", "Vino", "Vin", "Cab")
beer_options = c("Bud Light", "IPA", "Beer", "Franziskaner Weissbier", "Warsteiner")
hard_liquor_options = c("shots", "Scotch", "whiskey", "vidka")

wrk_data[which(wrk_data$drink_type %in% wine_options), "drink_type"] = "Wine"

wrk_data[which(wrk_data$drink_type %in% beer_options), "drink_type"] = "Beer"

wrk_data[which(wrk_data$drink_type %in% hard_liquor_options), "drink_type"] = "Hard Liquor"

unique(wrk_data$drink_type)
```

```
## [1] "Wine"        "Beer"        "Hard Liquor"
```

7 DateTime Parsing and Computation

In this section we will clean up the `collection_date` and `age` columns, discuss the importance of correct data typing, and demonstrate datetime operations in R.

7.1 Commands Used

- `anytime`
- `typeof`
- `as.POSIXct`
- `as.Date`

7.2 Packages Used

- `anytime`. This can be installed from your favorite R session using `install.packages("anytime")`

7.3 A Brief Introduction to Why Data Types Matter

Every column in the loaded dataframe `in_data` has a data type. Without going too far into what this means know that the type of a column influences how R, and the code we write for it, process the data. Imagine we have the data value `01776`. This set of digits/characters could mean a variety of completely unrelated things. For example `01776` could be:

- The string Zip Code for Sudbury, Massachusetts
- The year in which the Declaration of Independence of the United States of America was signed but for some reason the number was padded with a zero.
- An arbitrary integer measurement where the padded zero was included to indicate that the maximum value is 99999.

If this value was a Zip Code then we must tell R that under no circumstance should the numerical value be considered. The string `01778` is a code of characters, not a number. If the value was a year, we may consider forcing R to treat it as a string, date, or integer depending on the application. If the value was a measurement, we would want it to be considered a float or an integer.

Before we can perform operations on entire columns we must first unify the data types within the column.

There is a lot more than can be said on data types and "type safety" in programming. If you're interested in reading more about this be prepared for a fierce internet debate between programming communities.

Back to the examples at hand…

7.4 Setting data type for `collection_date`

In this example dataset we have two columns that need additional attention to their types, `collection_date`, the data of data collection, and `age`, the age of the participant at the time of the study. If we examine the `collection_date` column we can observe that the values are "string encoded dates" or a representation of a date and time using a standard string format. Here, the dates are represented using the ISO 8601 date format. We will use the `anytime` package which is excellent for automatically parsing standard date formats into the datetime datatypes in R. We will also be using `lubridate` and a custom function.

```
# Load the anytime library
library(anytime)

# Examine the type of the collection_date column. Note that R currently classifies it as "character"
typeof(in_data$collection_date)
```

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

```
# Use anytime to convert the column to POSIXct times.
wrk_data[,'collection_date'] = anydate(in_data$collection_date)

# Note that the type has changed to "double", don't worry, this is really a datetime.
typeof(wrk_data$collection_date)
```

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

```
wrk_data$collection_date[1]
```

```
## [1] "2018-01-29"
```

Why does this matter? Supposed I wanted to compute the time between a value in the column and some other date. I could try to type out the dates as strings and pray that the compute knows what I mean. This won't work because the subtraction of two string types has no idea what you intended to do! You could also transform the dates from your colloquial format to some relative time unit like "fortnights since 1/1/2000" but this will quickly become cumbersome and error prone. By having two values that are typed as datetime, R knows about all the leap years, time zones, and other quirks and is able to correctly perform time operations without any effort from the user.

```
# Let's just type out the dates and try to subtract them.... ERROR
# "2011-03-25" - "2001-03-25"

# Now let's try a computation of two datetime objects.
as.Date("2018-03-19") - wrk_data$collection_date[1]
```

```
## Time difference of 49 days
```

Tada! Using datetime objects gives a correct and useful result.

7.5 A Brief Aside on Regular Expressions (Regex)

7.6 Using Regex to Sort Out `age`

Looking at the first 10 values in the `age` column we can see immediatly that the type is mixed. Some of the values are the age in **months** while others are the birthday. Yikes!

```r
head(in_data$age, n = 10)
```

```
##  [1] "1/7/66"  "8/19/68" "50"      "671"     "4/2/60"  "9/14/74" "6/21/59"
##  [8] "389"     "549"     "35"
```

To select the rows where the `age` column contains a birthdate rather than an age, we need to match the values that follow the pattern `MM/DD/YYYY`. In regex for R, this is `'^(\\d{2}|\\d{2})\\/(\\d{1}|\\d{2})\\/(\\d{2})'`.

For these examples we will be exploring the features of the `lubridate` package.

```r
library(lubridate)
```

```
##
## Attaching package: 'lubridate'

## The following object is masked from 'package:base':
##
##     date
```

```r
# If we run grep along the column, it returns the row number of the matching columns
#grep('^(\\d{1}|\\d{2})\\/(\\d{1}|\\d{2})\\/(\\d{2})', in_data$age, perl = TRUE)

# Let's make a vector of these
birthdate_rows = grep('^(\\d{1}|\\d{2})\\/(\\d{1}|\\d{2})\\/(\\d{2})', in_data$age, perl = TRUE)

# Let's look at the values
in_data[birthdate_rows, "age"][1:10]
```

```
##  [1] "1/7/66"  "8/19/68" "4/2/60"  "9/14/74" "6/21/59" "5/31/82" "8/5/62"
##  [8] "1/7/54"  "5/1/55"  "1/13/62"
```

```r
# Here is what happens when we try to parse the MM/DD/YY format...
as.Date(in_data[birthdate_rows, "age"][1:10], format = '%d/%m/%y')
```

```
##  [1] "2066-07-01" NA           "2060-02-04" NA           NA
##  [6] NA           "2062-05-08" "2054-07-01" "2055-01-05" NA
```

```r
# And the same thing using a wrapper in lubridate
mdy(in_data[birthdate_rows, "age"][1:10])
```

```
##  [1] "2066-01-07" "2068-08-19" "2060-04-02" "1974-09-14" "2059-06-21"
##  [6] "1982-05-31" "2062-08-05" "2054-01-07" "2055-05-01" "2062-01-13"
```

UH OH! The dates are in the future! LET THIS BE A LESSON TO ALWAYS USE 4 YEAR DATES. If we know that the dates are all 1900s dates we can write our own function that converts the date strings to dates in the 1900s. I won't go too far into the syntax here but it should be simple enough to follow along.

```r
nineteen_hundreds_bday <- function(x, year=2017){
   # Parse as MM/DD/YY date
   x <- mdy(x)

   # Get the last two digits of the date
   m <- year(x) %% 100
```

```
    # Set the year to the 1900s year with that two digit suffix
    year(x) <- 1900+m
    return(x)
}

# Test it
nineteen_hundreds_bday(c("10/3/63", "12/26/73", "12/8/91"))
```

```
## [1] "1963-10-03" "1973-12-26" "1991-12-08"
```

We will also need a function to convert the time between the birthday and collection date to months.

```
months_between <- function(end_date, start_date) {
    end_dt <- as.POSIXlt(end_date)
    start_dt <- as.POSIXlt(start_date)

    # 12 times the elapsed years + the elapsed months intrayear
    return(12 * (end_dt$year - start_dt$year) + (end_dt$mon - start_dt$mon))
}

# Test it
months_between("2016-02-23", "2015-01-28")
```

```
## [1] 13
```

Phew! Now let's put it all together.

```
head(wrk_data$age)
```

```
## [1] "1/7/66"  "8/19/68" "50"      "671"     "4/2/60"  "9/14/74"
```

```
wrk_data$age = in_data$age

wrk_data[birthdate_rows, ]$age = as.character(months_between(in_data[birthdate_rows, "collection_date"]
wrk_data[, 'age'] = as.integer(wrk_data$age)
```

8 Including Plots

You can also embed plots, for example:

```
# {r pressure, echo=FALSE}
plot(pressure)
```

Note that the `echo = FALSE` parameter was added to the code chunk to prevent printing of the R code that generated the plot.