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The tools used in this tutorial are elemental functions of R as such is it not expected that this tutorial will become out-of-date.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.

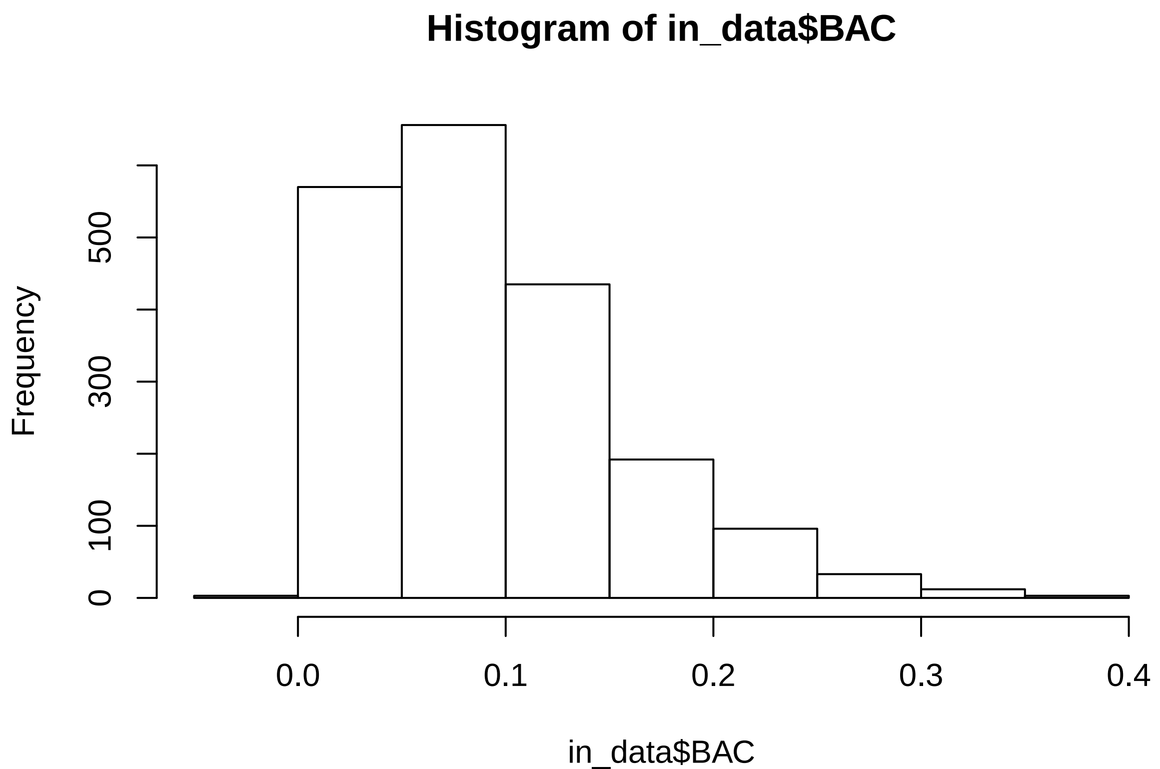
5.1 Commands Used

- hist
- read.csv
- row.names
- head
- nrow
- ncol

```r
library(ggplot2)
# 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)
```

```
##   X         BAC        age            collection_date drink_type
## 1 0 0.038199737 1954-11-19 2018-03-19T15:39:06.677432        Cab
## 2 1 0.008565803        621 2018-03-19T15:39:06.681675 Warsteiner
## 3 2 0.034301832 1987-10-31 2018-03-19T15:39:06.682148       Vino
## 4 3 0.091937048 1957-07-20 2018-03-19T15:39:06.682514      shots
## 5 4 0.060092061 1968-04-17 2018-03-19T15:39:06.682789     Scotch
## 6 5 0.190182940        464 2018-03-19T15:39:06.683122  Bud Light
##   num_drinks    sex  units volume_consumed   weight
## 1        1.6      M                 8.0000 200.4800
## 2        0.5      F metric        177.2587 227.8115
## 3        0.8      F                 4.0000 170.7000
## 4        2.1 Female                 3.1500 144.3300
## 5        2.0      F                 3.0000 166.2700
## 6        5.9 Female     SI       2097.1914 242.7423
```

```r
# 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`

- qplot

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

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

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

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 `|`.

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

```
##     X        BAC        age            collection_date drink_type
## 13 12 0.05480940 1970-10-21 2018-03-19T15:39:06.684620        IPA
## 14 13 0.04199972 1992-10-09 2018-03-19T15:39:06.684793      vidka
## 32 31 0.01605389 1968-09-05 2018-03-19T15:39:06.687594        Cab
## 37 36 0.06380654 1961-06-19 2018-03-19T15:39:06.688531        IPA
## 40 39 0.01486070        545 2018-03-19T15:39:06.689001      shots
## 41 40 0.17429455 1989-05-20 2018-03-19T15:39:06.689192       Beer
##    num_drinks  sex  units volume_consumed   weight
## 13        2.6 Male metric        923.0617 318.5457
## 14        2.0 Male                 3.0000 203.9700
## 32        0.5 Male                 2.5000 203.4800
## 37        3.1 Male metric       1101.6464 302.3016
## 40        0.6 Male                 0.9000 208.1500
## 41        6.1 Male                73.2000 162.0200
```

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.

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

```
## [1] 13 14 32 37 40 41
```

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`.

```
# 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
## 1  0.03819974    M
## 9  0.13954625 meal
```

```
## 13 0.05480940 Male
## 14 0.04199972 Male
## 15 0.08648585 male
## 16 0.17228833    M
## 22 0.10091553 meal
## 23 0.07260159 meal
## 26 0.12902471    M
## 28 0.09735920 meal
```

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 value and only the selected rows will be updates.

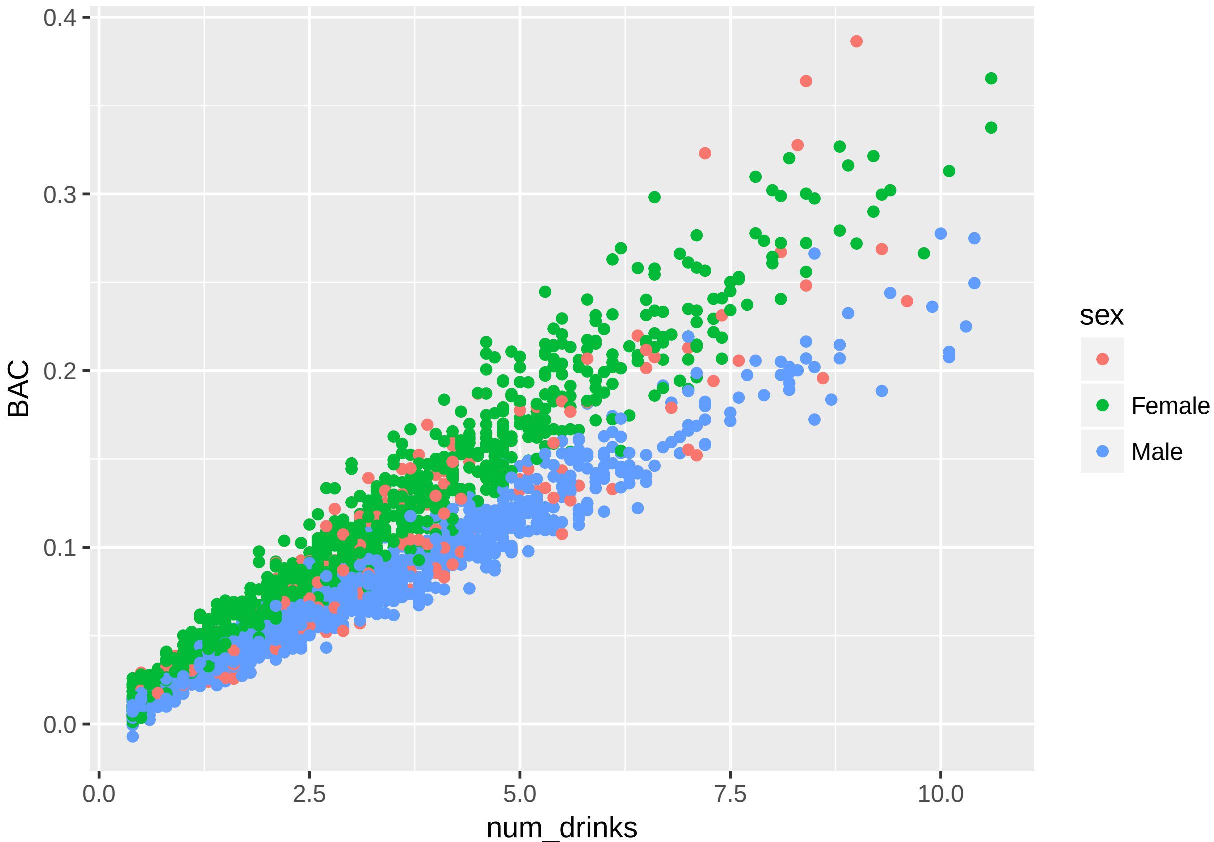
```r
wrk_data[which(wrk_data$sex %in% male_values), "sex"] = "Male"

# 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"
```

```r
# SUCCESS! If we examine the unique set of values for `sex` in wrk_data we see that Male and Female
# are now standardized.
unique(wrk_data$sex)
```

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

If you aren't careful you may have missed that there was a value in the `sex` column of `""` or nothing. Since we suspect that `sex` has impact on on the BAC with all things equal, let's plot `num_drinks` vs `BAC` and color the data points based on `sex`.

```r
qplot(num_drinks, BAC, colour = sex,
   data = wrk_data)
```



Whoa, there appears to be a clear difference between `Male` and `Female`. Given this difference we have two choices about how to handle the missing `sex` values. The simple option would be to simply drop the missing values. We will do this right before we fit our model. If you are interested in seeing how one might impute these values, see the appendix.

6.2 Cleaning `units`

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.

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

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

```
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"
```

6.3 `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?

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

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

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")`
- `lubridate`

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.

```r
# Load the datetime utilities libraries
library(lubridate)
```

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

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

```r
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(wrk_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] "2018-03-19"
```

Why does this matter?

Suppose 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("2018-05-19") - wrk_data$collection_date[1]
```

```
## Time difference of 61 days
```

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

7.5 A Brief Aside on Regular Expressions (Regex)

WIP

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!

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

```
##  [1] "1954-11-19" "621"        "1987-10-31" "1957-07-20" "1968-04-17"
##  [6] "464"        "1975-04-01" "1955-10-21" "1984-01-13" "1978-07-18"
```

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 `YYYY-MM-DD`. In regex for R, this is `'^(\\d{4})-(\\d{2})-(\\d{2})'`.

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

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

```
## [1] "1954-11-19" "621"        "1987-10-31" "1957-07-20" "1968-04-17"
## [6] "464"
```

```
# If we run grep along the column, it returns the row number of the matching columns
grep('^(\\d{4})-(\\d{2})-(\\d{2})', wrk_data$age, perl = TRUE)[1:10]
```

```
## [1]  1  3  4  5  7  8  9 10 11 13
```

```
# Let's make a vector of these
birthdate_rows = grep(
                    '^(\\d{4})-(\\d{2})-(\\d{2})',
                    in_data$age,
                    perl = TRUE)

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

```
##  [1] "1954-11-19" "1987-10-31" "1957-07-20" "1968-04-17" "1975-04-01"
##  [6] "1955-10-21" "1984-01-13" "1978-07-18" "1962-02-22" "1970-10-21"
```

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] "1954-11-19" "621"        "1987-10-31" "1957-07-20" "1968-04-17"
## [6] "464"
```

```
wrk_data$age = in_data$age

# Compute months_between, convert to strings, update the dataframe
wrk_data[birthdate_rows, "age"] =
                          as.character(
                          months_between(
                            in_data[birthdate_rows, "collection_date"],
                            in_data[birthdate_rows, "age"]
                            )
                          )

# Convert the now-uniform column to numeric values
wrk_data[, 'age'] = as.integer(wrk_data$age)

# Convert months to decimal years
wrk_data[, 'age'] = wrk_data[, 'age']/12

head(wrk_data[, 'age'])
```

```
## [1] 63.33333 51.75000 30.41667 60.66667 49.91667 38.66667
```

```r
head(wrk_data)
```

```
##   X         BAC      age collection_date  drink_type num_drinks    sex
## 1 0 0.038199737 63.33333      2018-03-19        Wine        1.6   Male
## 2 1 0.008565803 51.75000      2018-03-19        Beer        0.5 Female
## 3 2 0.034301832 30.41667      2018-03-19        Wine        0.8 Female
## 4 3 0.091937048 60.66667      2018-03-19 Hard Liquor        2.1 Female
## 5 4 0.060092061 49.91667      2018-03-19 Hard Liquor        2.0 Female
## 6 5 0.190182940 38.66667      2018-03-19        Beer        5.9 Female
##      units volume_consumed   weight
## 1 imperial          8.0000 200.4800
## 2   metric        177.2587 227.8115
## 3 imperial          4.0000 170.7000
## 4 imperial          3.1500 144.3300
## 5 imperial          3.0000 166.2700
## 6   metric       2097.1914 242.7423
```

8 Units Conversions

Data is often collected in different places under different measurement systems. In this section we will look at how the reader might be able to figure out what units were used and how to perform the unit conversions.

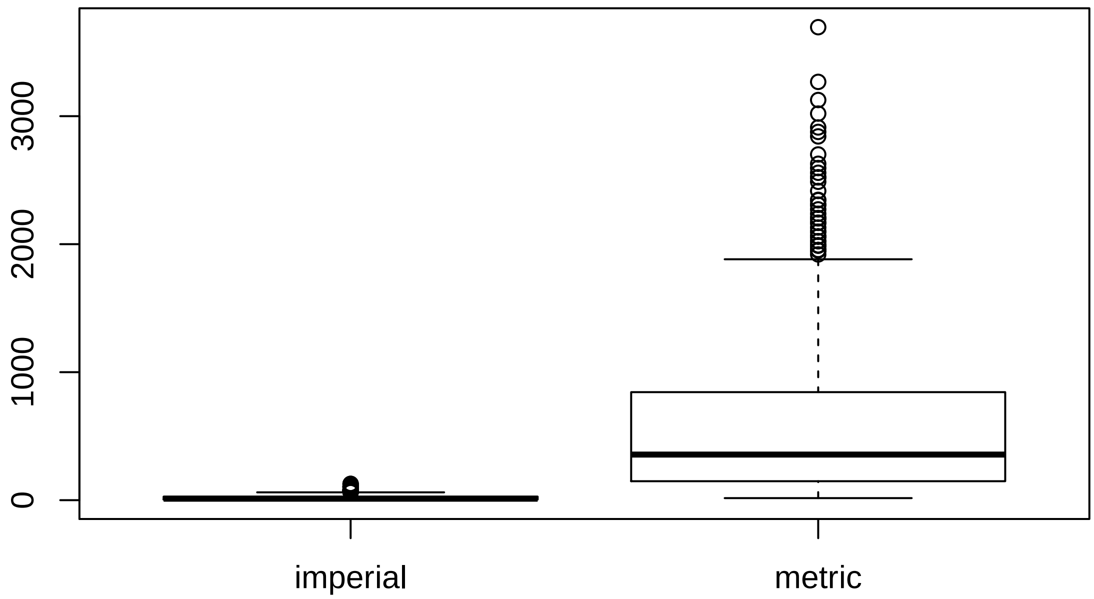
8.1 Commands Used

- as.factor
- aggregate

8.2 Unit Discovery and Metric to Imperial Conversions

We know that there is a discriminator column called "units" which tells us what unit system was used. The two measurement values in the dataset are `weight` and `volume_consumed`. If we create box-whisker plot for `volume_consumed` for each unit system we can see the difference clearly.

```r
# Box and Whisker plot of volume_consumed by unit type
plot(as.factor(wrk_data$units), in_data[, "volume_consumed"])
```



As the author of the original dataset, I can disclose that the volume units for `metric` are milliliters while the `imperial` units are ounces. The best way to examine the relationship between `metric` and `imperial` units is by using data aggregation. Aggregating operations require a set of columns to sum/count/mean based on a categorical variable. We can see the relationship between units by taking ratio of the average volume per unit type. We know that there are 29.6 ml in an ounce. The ratio should be right around this number. We can do the same for the weight where the `metric` units are kg and the `imperial` units are lbs.

```
# Let's aggregate the volume and weight numbers by their respective units and take the mean
volume_means = aggregate(
                    wrk_data[, c("volume_consumed", "weight")],
                    list(units = wrk_data$units),
                    mean)
volume_means
```

```
##      units volume_consumed   weight
## 1 imperial        19.51505 176.9783
## 2   metric       586.98502 259.3690
```

```
# Ratio of the metric volume mean to the imperial volume mean
volume_means[which(volume_means$units == "metric"),  "volume_consumed"] /
  volume_means[volume_means$units == "imperial",  "volume_consumed"]
```

```
## [1] 30.07858
```

```
# Ratio of the metric weight mean to the imperial weight mean
volume_means[which(volume_means$units == "metric"),  "weight"] /
  volume_means[volume_means$units == "imperial",  "weight"]
```

```
## [1] 1.465541
```

The computed BAC in this dataset used `imperial` units so we we must convert the `metric` values back to `imperial`.

```
# Convert volume - ml to oz
wrk_data[which(wrk_data$units == 'metric'), "volume_consumed"] =
  wrk_data[which(wrk_data$units == 'metric'), "volume_consumed"]/29.6

head(wrk_data[which(wrk_data$units == 'metric'), "volume_consumed"])
```

```
## [1]  5.988469 70.851060  3.641609 70.863386 24.999332 43.214149
```

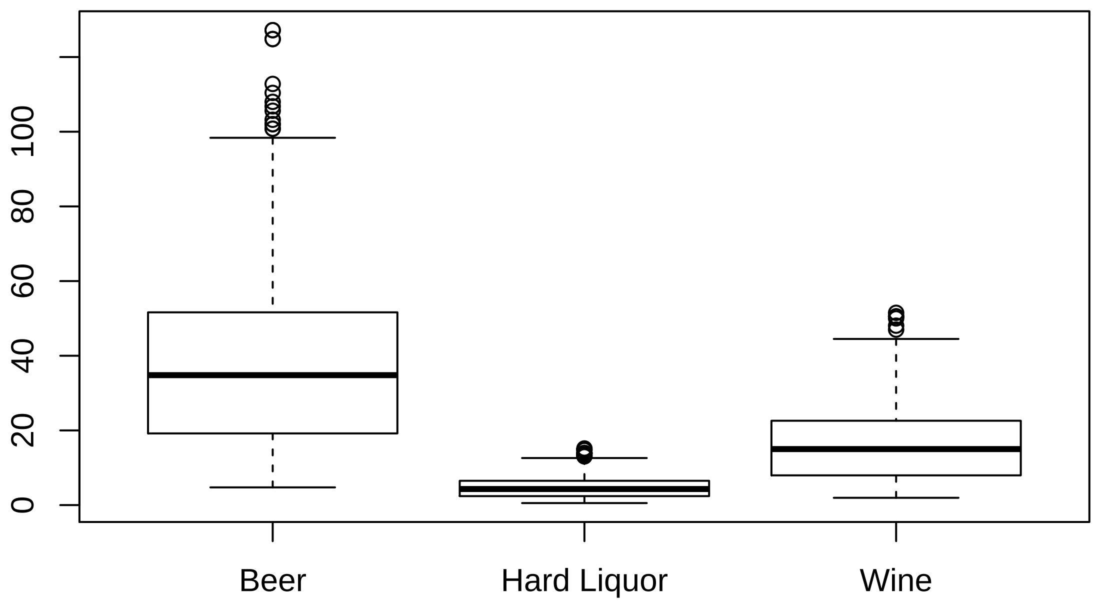
```r
# Convert weight - kg to lbs
wrk_data[which(wrk_data$units == 'metric'), "weight"] =
  wrk_data[which(wrk_data$units == 'metric'), "weight"]/1.453592

head(wrk_data[which(wrk_data$units == 'metric'), "weight"])
```

```
## [1] 156.7231 166.9948 150.1663 178.5105 158.8878 179.4409
```

8.3 Convert `volume_consumed` to alcohol consumed

As we all learned in health class, different types of alcoholic beverage contain different quantities of alcohol. If we plot the `volume_consumed` for each of the `drink_types` we can clearly see the difference in volume consumed depending on the type of drink.

```r
plot(as.factor(wrk_data$drink_type), wrk_data$volume_consumed)
```



Since the number of "drinks" is supplied in the dataset, we do not need to convert the volume to ounces. The best way to do this would be to multiply the volumes by their ABV (Alcohol by Volume) percentages.

9 Final Preparations and Modeling!

```r
# Make a copy to use in the appendix
impute_data = wrk_data
```

```r
# Remove all rows where sex is blank
wrk_data = wrk_data[!(wrk_data$sex == ""),]

head(wrk_data)
```

```
##   X         BAC      age collection_date  drink_type num_drinks    sex
## 1 0 0.038199737 63.33333      2018-03-19        Wine        1.6   Male
## 2 1 0.008565803 51.75000      2018-03-19        Beer        0.5 Female
```

```
## 3 2 0.034301832 30.41667      2018-03-19        Wine        0.8 Female
## 4 3 0.091937048 60.66667      2018-03-19 Hard Liquor        2.1 Female
## 5 4 0.060092061 49.91667      2018-03-19 Hard Liquor        2.0 Female
## 6 5 0.190182940 38.66667      2018-03-19        Beer        5.9 Female
##      units volume_consumed   weight
## 1 imperial        8.000000 200.4800
## 2   metric        5.988469 156.7231
## 3 imperial        4.000000 170.7000
## 4 imperial        3.150000 144.3300
## 5 imperial        3.000000 166.2700
## 6   metric       70.851060 166.9948
```

```
model_data = wrk_data
model_data = model_data[, !(names(model_data) %in% c("collection_date", "units", "volume_consumed", "X"
```

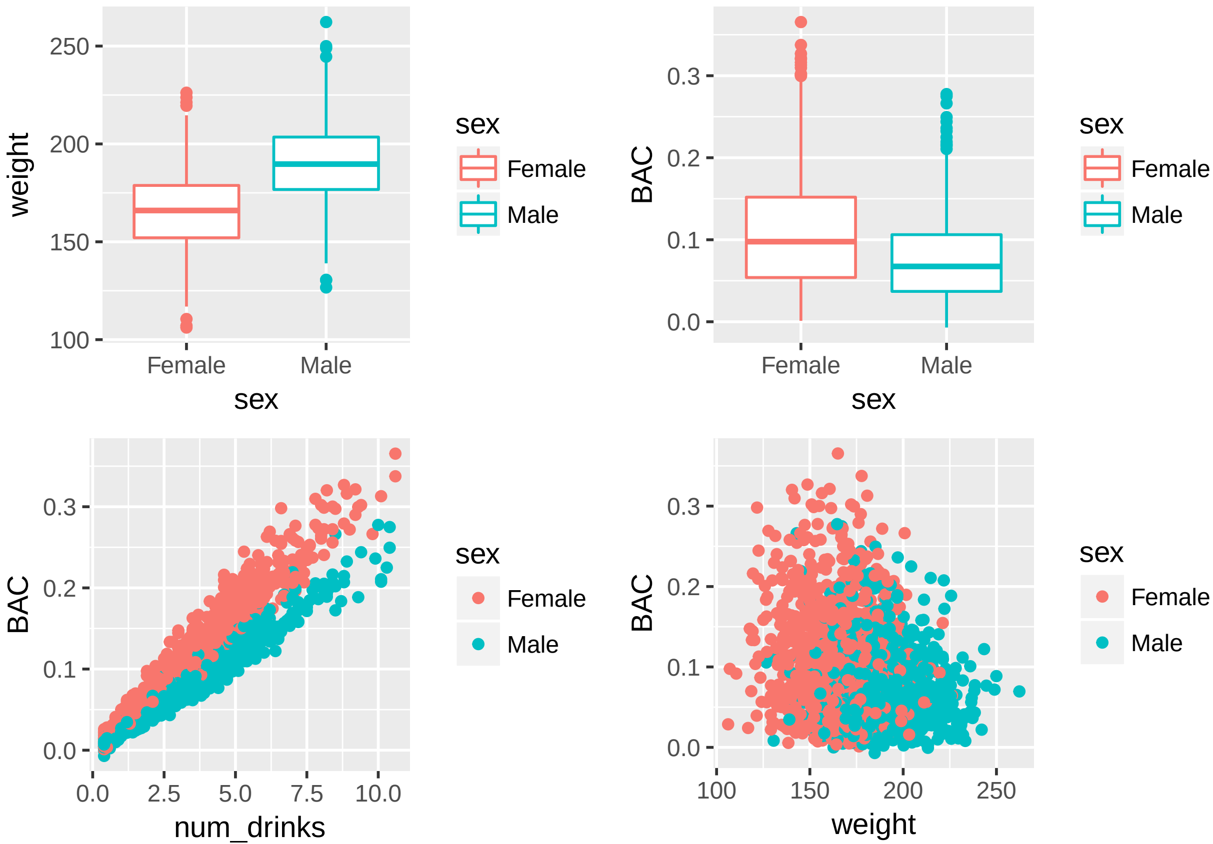
9.1 Looking at Clean Data

This serves both to understand the dataset and demonstrate various plotting methods. Specifically, let's look at how the variables relate to `sex` and `bac`

```
library(ggpubr)
```

```
## Loading required package: magrittr
```

```
# Box plot of sex and weight
p1 = qplot(sex, weight, color = sex, data = model_data, geom = "boxplot")

# Box plot of sex and BAC
p2 = qplot(sex, BAC, color = sex, data = model_data, geom = "boxplot")

# Scatter plot of num_drinks and BAC
p3 = qplot(num_drinks, BAC, colour = sex,
   data = model_data)

# Scatter plot of weight and BAC
p4 = qplot(weight, BAC, color=sex, data = model_data)

ggarrange(p1, p2, p3, p4)
```



9.2 Fit a Linear Regression Model

```
model_data$sex = ifelse(model_data$sex == "Male", 1, 0)
lm1 = lm(BAC ~ ., data = model_data)

summary(lm1)
```

```
##
## Call:
## lm(formula = BAC ~ ., data = model_data)
##
## Residuals:
##       Min        1Q    Median        3Q       Max
## -0.054357 -0.007777 -0.000741  0.006698  0.068529
##
## Coefficients:
##                Estimate Std. Error t value Pr(>|t|)
## (Intercept)  9.775e-02  2.962e-03  33.000   <2e-16 ***
## age          3.741e-05  2.779e-05   1.346    0.178
## num_drinks   2.910e-02  1.753e-04 166.018   <2e-16 ***
## sex         -1.953e-02  7.137e-04 -27.367   <2e-16 ***
## weight      -5.026e-04  1.536e-05 -32.721   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
```

```
## Residual standard error: 0.01283 on 1802 degrees of freedom
## Multiple R-squared:  0.9572, Adjusted R-squared:  0.9571
## F-statistic: 1.007e+04 on 4 and 1802 DF,  p-value: < 2.2e-16
```

10 Appendix

10.1 Imputing the mising sex values

A fairly easy model to use for the purpose is logistic regression. These models produce a binary output useful for simple classification problems like imputing the `sex` based on some of the other variables.

```r
# Set sex as a binary variable
impute_data[which(impute_data$sex == "Male"), 'sex'] = 1
impute_data[which(impute_data$sex == "Female"), 'sex'] = 0

# Set the data type to numeric
impute_data$sex = as.numeric(impute_data$sex)
```

```r
# Use all rows where `sex` is available as the training data.
# We will predict all of the missing observations
train = impute_data[which(impute_data$sex != ""),]
missing = impute_data[which(is.na(impute_data$sex)),]
missing = missing[, (names(missing) %in% c('num_drinks', 'BAC'))]

# Fit
model <- glm (sex ~ num_drinks + BAC, data = train, family = binomial)
summary(model)
```

```
##
## Call:
## glm(formula = sex ~ num_drinks + BAC, family = binomial, data = train)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -3.1922  -0.2783   0.0050   0.3462   3.4988
##
## Coefficients:
##              Estimate Std. Error z value Pr(>|z|)
## (Intercept)    0.3304     0.1417   2.331   0.0197 *
## num_drinks     6.7290     0.3521  19.111   <2e-16 ***
## BAC         -239.0240    12.3687 -19.325   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 2504.74  on 1806  degrees of freedom
## Residual deviance:  943.98  on 1804  degrees of freedom
## AIC: 949.98
##
## Number of Fisher Scoring iterations: 7
```

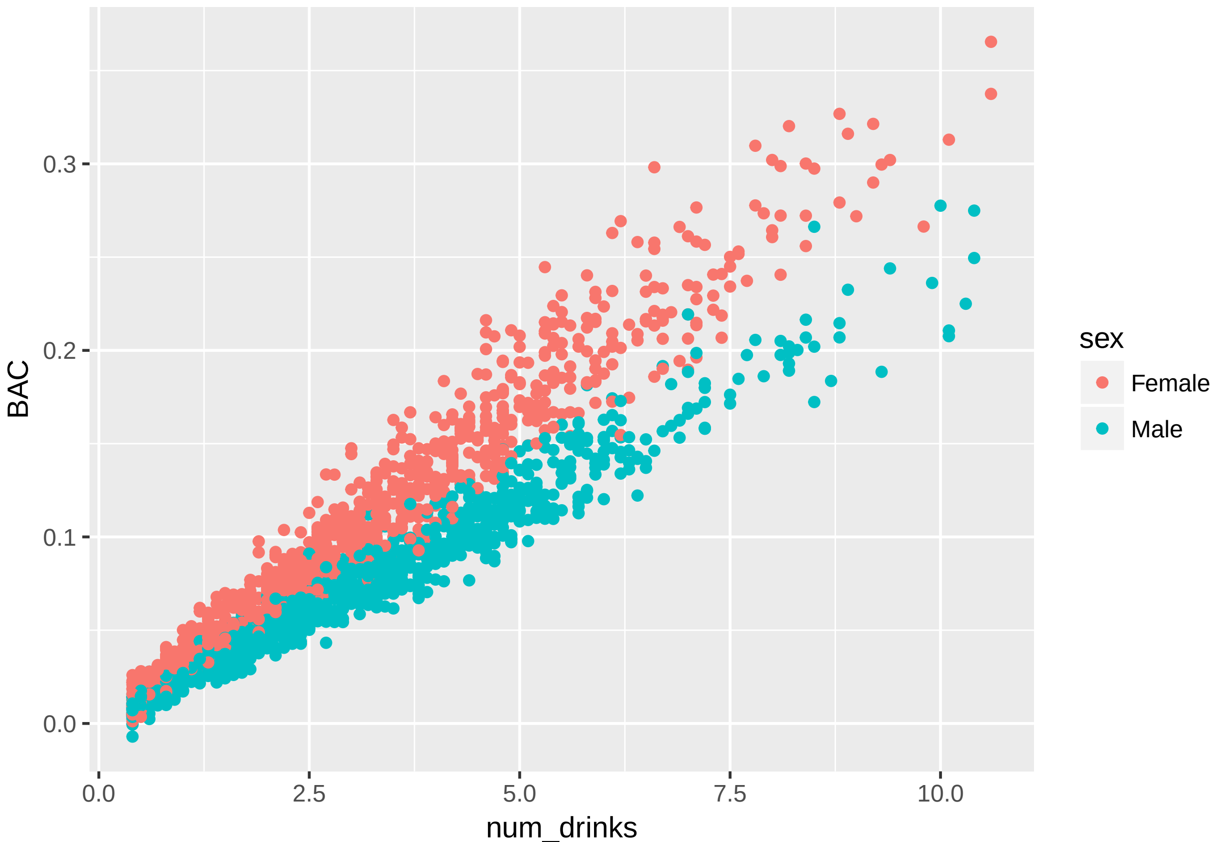
```r
# Make the prediction
prediction = predict(model, missing, type = 'response')
```

```r
range(prediction)
```

```
## [1] 2.220446e-16 9.999921e-01
```

```r
prediction = ifelse(prediction >= .5, "Male", "Female")

impute_data[names(prediction), 'sex'] = prediction
```

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



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