Workshop 2: Data and ETL

Myles D. Garvey, Ph.D

Winter, 2020

1 Learning Objectives

1. To understand the purpose of extract transform load (ETL)
2. To understand how to read in data and work with it in R
3. To understand how to read and write data in R
4. To understand the various data-storing formats in R
5. To understand how R can be used to perform ETL

2 Workshop Description and Submission

In this workshop, you will learn how to use R's functionality for loading and writing data from/to different types of files. We will first explore the common types of data formats, and how we can read these different formats using R. We will also explore the different types of data that you are likely to come across in practice. In the second tutorial, we will review through a few fundamental practices for understanding your data by ways of visual and descriptive analysis. In the third tutorial you will learn how to write data to files using R, as well as how to compress data into certain types of data formats. In the last tutorial we will review through the principles of ETL, and you will learn how to aggregate multiple data sets into one.

You will submit to blackboard a .PDF document. Complete each exercise in this workshop. These are easy and small, and so it should not take you very long. In a word document, type up the question number, the original question text, and your response in R code. Save your word document as a .PDF and submit it to the submission link on Blackboard. Please note that failure to submit this as a .PDF will result in a 0 on the workshop.

3 Tutorial 1: Loading Data into R and Different Data Types

When we seek to analyze data sets in R, it goes without saying that the data must first be "imported" into R. At the very technical level, what this simply means is that information is being sent from your hard drive (or, another piece of hardware) to your RAM. Needless to say, if you have insufficient RAM, then R will most likely crash, or, your computer will slow. Hence, the first principle of analyzing data: ensure you have enough RAM on your computer. For example, if you only have 4GB of RAM, but your data file is 16GB, then it should go without saying that you WILL have problems loading this file. What is the solution? We will explore this question more in a later lecture.

As for the data itself, when we "load" it into R, the data could be in one of a variety of potential formats. Usually, the extension of the file easily tells us the format. However, if our file is corrupt (that is, someone accidentally used a different extension when they saved the file), then we will run into problems loading our data. There are many types of formats that organize and store information, including but not limited to XLS, CSV, TSV, TXT, JSON, XML, EDI, PDF, and many many more! So how can we come to understand what lies within our data if there are so many possible ways in which the data is stored? Typically, we organize types of data formats into one of two fundamental types: structured and unstructured. The type of format (structured or unstructured) our data is in helps guide us to understand how we should go about loading it into R.

Structured data formats are formats that organize data in such a way that easily allows us to extract the information using different libraries within R. Examples include XML, EDI, JSON, CSV, XLS (to a certain extent), and TSV. In the case of XLS, CSV, and TSV, data is usually organized in a table format, where each row represents a single entry, and each column represents a single attribute. In the case of JSON and XML, data is usually organized by the use of map/key pairs or the use of tags, where each observation is given a unique identifier of some kind, and all of the attributes about the individual are stored in tags or some other type of data structure. We will see the structure of these soon. We reserve a discussion of working with unstructured data for a future lecture, specifically when we cover textual analytics.

Let us start to load data that is in these different formats into R. Usually, when we load data into R, it is loaded into one of two resulting data structures: a data frame or a tibble. A data frame is R's basic data structure that allows us to easily work with data in a table format. The tibble format on the other hand is a newer format that allows for easier data processing leveraging SQL-like commands right within R. Depending on the type of packages we use to load information in R, it will most likely load the data as a data frame or a tibble.

Part of your module includes data files that have multiple formats. First, we will work with the csv. CSV files stand for comma separated value files. The structure of a basic csv file should be as follows. On the first line, include the headers of each column of data separated by a comma. Each data observation is represented on its own line in the file, where the data for each variable for a specific observation is separated by commas.

In order to load the csv file, we can use R's base command, `read.csv`, to load the data into R as a data frame. Before we do, we need to ensure that our data file is in our working directory. If it is not, we need to specify the full address of the file location. It is recommended that you place the file in the working directory. If you do not know the location of this, you can use the following to obtain it. After we load it, we can explore the initial structure of the data as follows:

```
> getwd()
[1] "C:/Users/myles/Documents"
> setwd("C:/Users/myles/Downloads/data_mining/mod_2")
> getwd()
[1] "C:/Users/myles/Downloads/data_mining/mod_2"
```

```
> data<-read.csv("household_survey.csv")
> str(data)
'data.frame':	5916 obs. of  10 variables:
 $ NEWID   : int  3639434 3639444 3639454 3639504 3639544 3639564 3639594 3639614
    3639624 3639634 ...
 $ AGE_REF : int  36 57 57 22 71 67 55 72 65 65 ...
 $ AGE2    : int  36 57 NA 27 NA 68 52 NA NA 63 ...
 $ BATHRMQ : int  3 1 1 2 1 1 1 2 1 1 ...
 $ BEDROOMQ: int  5 3 1 3 3 3 2 3 3 3 ...
 $ BLS_URBN: int  1 1 1 1 1 1 1 1 1 1 ...
 $ FAM_SIZE: int  4 3 1 3 1 2 3 1 1 2 ...
 $ INC_RANK: num  0.0964 0.3435 0.1532 0.4964 0.2817 ...
 $ RENTEQVX: int  2770 500 NA 1200 300 3500 1600 2100 900 1200 ...
 $ VEHQ    : int  2 2 0 3 1 3 1 1 1 3 ...

> head(data)
    NEWID AGE_REF AGE2 BATHRMQ BEDROOMQ BLS_URBN FAM_SIZE
1 3639434      36   36       3        5        1        4
2 3639444      57   57       1        3        1        3
3 3639454      57   NA       1        1        1        1
4 3639504      22   27       2        3        1        3
5 3639544      71   NA       1        3        1        1
6 3639564      67   68       1        3        1        2
   INC_RANK RENTEQVX VEHQ
1 0.0963827     2770    2
2 0.3434733      500    2
3 0.1532240       NA    0
4 0.4964219     1200    3
5 0.2817358      300    1
6 0.7539039     3500    3
```

Listing 1: Loading a CSV file into R

A few observations are worthy of note. First, we always ensure that our files to load are in the working directory. It is advisable, for organization purposes, to have one folder for each distinct project. For our purposes, we will just have all files in one working folder for the current module. Second, if you are on a windows machine, notice how the working directory is specified. Notable is the reversal of the slash from the traditional Windows usage of "
" to "/".

Next, it should be noted that not all CSV files are separated by commas. Sometimes, the authors of a data file will separate these using other characters. These characters are called *delimiters*, and they help separate data. If you come across a file (like in the exercise below) where it uses a different character, you will need to specify this when you use the function `read.csv`. Likewise, not all data files have a *header*, that is, a first row indicating the names of the columns. It is always advisable to manually add this into the file, or, at the very least, to do so once the data is loaded into R.

Last, notice that the `str` function helps us understand the structure of our data. It lists the type (in our case, it is a data frame), as well as the variable names, the type (an `int`,`num`,`char`, etc), and some of the first few values of each variable. We equally can few some of the first few observations by using the `head` function. All of this can also be applied to a TSV (tab separated value) or TXT file. The only differences between this type of file and a CSV is (1) values are separated by TAB characters and (2) the extension is .tsv/.txt instead of .csv. Notice that our steps remain almost identical, with the notable exception of using the function `read.table`, which is a more general form of `read.csv`:

```
> data_tab<-read.table("household_survey.txt",sep="\t",header=TRUE)
> str(data_tab)
'data.frame': 5916 obs. of  10 variables:
 $ NEWID   : int  3639434 3639444 3639454 3639504 3639544 3639564 3639594 3639614
   3639624 3639634 ...
 $ AGE_REF : int  36 57 57 22 71 67 55 72 65 65 ...
 $ AGE2    : int  36 57 NA 27 NA 68 52 NA NA 63 ...
 $ BATHRMQ : int  3 1 1 2 1 1 1 2 1 1 ...
 $ BEDROOMQ: int  5 3 1 3 3 3 2 3 3 3 ...
 $ BLS_URBN: int  1 1 1 1 1 1 1 1 1 1 ...
 $ FAM_SIZE: int  4 3 1 3 1 2 3 1 1 2 ...
 $ INC_RANK: num  0.0964 0.3435 0.1532 0.4964 0.2817 ...
 $ RENTEQVX: int  2770 500 NA 1200 300 3500 1600 2100 900 1200 ...
 $ VEHQ    : int  2 2 0 3 1 3 1 1 1 3 ...
> head(data_tab)
    NEWID AGE_REF AGE2 BATHRMQ BEDROOMQ BLS_URBN FAM_SIZE
1 3639434      36   36       3        5        1        4
2 3639444      57   57       1        3        1        3
3 3639454      57   NA       1        1        1        1
4 3639504      22   27       2        3        1        3
5 3639544      71   NA       1        3        1        1
6 3639564      67   68       1        3        1        2
   INC_RANK RENTEQVX VEHQ
1 0.0963827     2770    2
2 0.3434733      500    2
3 0.1532240       NA    0
4 0.4964219     1200    3
5 0.2817358      300    1
6 0.7539039     3500    3
```

Listing 2: Loading a TSV/TXT file into R

Notice that in the previous two cases, our data were loaded into R as a data frame. However, this is not always the case due to the type of package that we use to load the data in the first place. For example, let us load a similar file, namely `household_survey.xlsx`, into R, which has two spreadsheets: the first with the data, and the second with what is called *metadata* (which is data about data), which indicates what each column represents. We will do so using the `readxl` library. We can install this package using the command `install.packages("readxl")`. Once installed, you can load it into R with `library(readxl)`. Once installed and loaded, we can leverage the functions in this package to read in excel files:

```
> install.packages("readxl")
Installing package into  C :/Users/myles/Documents/R/win-library/3.6
(as   lib   is unspecified)
trying URL 'https://cran.rstudio.com/bin/windows/contrib/3.6/readxl_1.3.1.zip'
Content type 'application/zip' length 1524644 bytes (1.5 MB)
downloaded 1.5 MB

package   readxl   successfully unpacked and MD5 sums checked

The downloaded binary packages are in
  C:\Users\myles\AppData\Local\Temp\RtmpM1dm4U\downloaded_packages
```

```

> library(readxl)
Warning message:
package    readxl    was built under R version 3.6.2

> data_excel<-read_excel("household_survey.xlsx")
> str(data_excel)
Classes 'tbl_df', 'tbl' and 'data.frame': 5916 obs. of  10 variables:
 $ NEWID   : num  3639434 3639444 3639454 3639504 3639544 ...
 $ AGE_REF : num  36 57 57 22 71 67 55 72 65 65 ...
 $ AGE2    : num  36 57 NA 27 NA 68 52 NA NA 63 ...
 $ BATHRMQ : num  3 1 1 2 1 1 1 2 1 1 ...
 $ BEDROOMQ: num  5 3 1 3 3 3 2 3 3 3 ...
 $ BLS_URBN: num  1 1 1 1 1 1 1 1 1 1 ...
 $ FAM_SIZE: num  4 3 1 3 1 2 3 1 1 2 ...
 $ INC_RANK: num  0.0964 0.3435 0.1532 0.4964 0.2817 ...
 $ RENTEQVX: num  2770 500 NA 1200 300 3500 1600 2100 900 1200 ...
 $ VEHQ    : num  2 2 0 3 1 3 1 1 1 3 ...
> data_excel
# A tibble: 5,916 x 10
    NEWID AGE_REF  AGE2 BATHRMQ BEDROOMQ BLS_URBN FAM_SIZE
    <dbl>   <dbl> <dbl>   <dbl>    <dbl>    <dbl>    <dbl>
 1 3.64e6      36    36       3        5        1        4
 2 3.64e6      57    57       1        3        1        3
 3 3.64e6      57    NA       1        1        1        1
 4 3.64e6      22    27       2        3        1        3
 5 3.64e6      71    NA       1        3        1        1
 6 3.64e6      67    68       1        3        1        2
 7 3.64e6      55    52       1        2        1        3
 8 3.64e6      72    NA       2        3        1        1
 9 3.64e6      65    NA       1        3        1        1
10 3.64e6      65    63       1        3        1        2
# ... with 5,906 more rows, and 3 more variables:
#   INC_RANK <dbl>, RENTEQVX <dbl>, VEHQ <dbl>
```

Listing 3: Loading an Excel File into R

We notice that when we load in the file, we obtain a *tibble*, which is different than our data frame from earlier. We will learn more about tibbles in the text mining lecture. For now, we will simply convert our tibble to an ordinary data frame, which is indicated below. Also notice that the `read_excel` command will read in, by default, the first spreadsheet in the file. However, we can use a variety of functions and inputs to specify different spreadsheets to load into R, such as:

```
> excel_sheets("household_survey.xlsx")
[1] "data"      "variables"

> metadata_excel<-read_excel("household_survey.xlsx",sheet="variables")
> str(metadata_excel)
Classes   tbl  _ d f  ,    tbl    and 'data.frame': 9 obs. of  2 variables:
 $ NEWID                  : chr  "AGE_REF" "AGE2" "BATHRMQ" "BEDROOMQ" ...
 $ The survey taker's id: chr  "The age of the survey taker" "The age of the
    survey taker's spouse" "The number of bathrooms" "The number of bedrooms" ...
> metadata_excel
# A tibble: 9 x 2
  NEWID    `The survey taker's id`
  <chr>    <chr>
```

```
1 AGE_REF  The age of the survey taker
2 AGE2     The age of the survey taker's spouse
3 BATHRMQ  The number of bathrooms
4 BEDROOMQ The number of bedrooms
5 BLS_URBN If the home is urban or rural
6 FAM_SIZE The number of individuals in the household
7 INC_RANK The income percentile of the household
8 RENTEQVX The perceived rent of the home
9 VEHQ     The number of vehicles owned by someone in the home

> data_excel<-data.frame(data_excel)
> metadata_excel<-data.frame(metadata_excel)
> str(data)
'data.frame': 5916 obs. of  10 variables:
 $ NEWID   : int  3639434 3639444 3639454 3639504 3639544 3639564 3639594 3639614
    3639624 3639634 ...
 $ AGE_REF : int  36 57 57 22 71 67 55 72 65 65 ...
 $ AGE2    : int  36 57 NA 27 NA 68 52 NA NA 63 ...
 $ BATHRMQ : int  3 1 1 2 1 1 1 2 1 1 ...
 $ BEDROOMQ: int  5 3 1 3 3 3 2 3 3 3 ...
 $ BLS_URBN: int  1 1 1 1 1 1 1 1 1 1 ...
 $ FAM_SIZE: int  4 3 1 3 1 2 3 1 1 2 ...
 $ INC_RANK: num  0.0964 0.3435 0.1532 0.4964 0.2817 ...
 $ RENTEQVX: int  2770 500 NA 1200 300 3500 1600 2100 900 1200 ...
 $ VEHQ    : int  2 2 0 3 1 3 1 1 1 3 ...
> str(metadata_excel)
'data.frame': 9 obs. of  2 variables:
 $ NEWID                 : chr  "AGE_REF" "AGE2" "BATHRMQ" "BEDROOMQ" ...
 $ The.survey.taker.s.id: chr  "The age of the survey taker" "The age of the
    survey taker's spouse" "The number of bathrooms" "The number of bedrooms" ...
```

Listing 4: Using the readxl package in R

Notice that when the other spreadsheet was loaded, it defaulted the first two entries as the column names. Obviously, we do not want this. We will review how to correct this in the next tutorial when we discuss the basics of data cleaning. Moving forward, we also have other structured data formats that are not so straightforward. They are structured in a different way than our typical table format. The reason may be that multiple pieces of data which are more complex may belong to some observations but not others. For example, a Twitter Tweet is a very complicated set of observations, with Twitter returning over 80 different attributes per observation.

Each attribute may itself be more complex. For example, one attribute that Twitter returns is a list of hashtags used in the Tweet. A list of keywords, however, is difficult to represent in table format. If each row represents a single Tweet, then we need to somehow list all the hashtags in a single entry (the common approach is to use a delimiter that absolutely will not appear elsewhere in the data, which is itself difficult to not have as an instance). In situations where our data is still structured, but just very complex (also known as multi-dimensional), we often opt for storing our data not in a table format, but in a different format. The two prevailing formats used to store and represent this information is the JSON (Javascript Object Notation) and XML (Extensible Markup Language) formats.

For example, in the `pizza.json` file (which can be opened in notepad in windows or text edit on mac) we have a collection of data that is stored in an unfamiliar format. In practice, many web APIs return, and even necessitate to be queried within, the JSON format. The JSON format is a way that we can organize highly complex and multi-dimensional data that allows us to move "beyond" the basic table format. However, it is difficult to analyze such a format, and we often find ourselves taking information that is stored in this format and converting it into our basic table format. For example, let us look at a much smaller instance in our file:

```
{
  "data": {
    "search": {
      "total": 547,
      "business": [
        {
          "name": "Mercato Tomato Pie",
          "url": "https://www.yelp.com/biz/mercato-tomato-pie-newark?adjust_creative=yu71TTzPRNWvhkXpS1DYaw&utm_campaign=yelp_api_v3&utm_medium=api_v3_graphql&utm_source=yu71TTzPRNWvhkXpS1DYaw",
          "location": {
            "address1": "212 Market St",
            "city": "Newark",
            "state": "NJ"
          },
          "rating": 4
        },
        {
          "name": "Robert’s Pizzeria",
          "url": "https://www.yelp.com/biz/roberts-pizzeria-newark-3?adjust_creative=yu71TTzPRNWvhkXpS1DYaw&utm_campaign=yelp_api_v3&utm_medium=api_v3_graphql&utm_source=yu71TTzPRNWvhkXpS1DYaw",
          "location": {
            "address1": "63 New St",
            "city": "Newark",
            "state": "NJ"
          },
          "rating": 4
        }
        ]
    }
  }
}
```

Listing 5: JSON Formatted Data

JSON works on the idea of an *object*. Think of an object as a structure in its own right. The easiest way to think about an object is like an entry in a table, where one of the entries can be a table (i.e. another object) itself. Each object is defined by a list of attributes and values. The attributes have names, and the values are assigned to the names (much like variable names and values). JSON works on the idea of embedded objects. Take a look in the code above. It opens with an open brace. This is the way that we indicate we are starting the definition of a new "object". Inside the brace is text in quotations, followed by a colon, followed by either another open brace (to indicate a new object), or a straight value.

We can see that there is a single object on the outer most portion of the text (this represents a "response" object). This object has a single attribute, namely that of "data". This attribute is assigned to another object which also has a single attribute, named "search". This attribute is assigned to another object with two attributes: "total" and "business". The "total" attribute is assigned a value of 547, while the "business" attribute is assigned to a list (indicated by the open brackets) of other objects. Notice how the objects in the list are similar. Each object in the list has 4 attributes, namely "name", "url", "location", and "rating". Each attribute, except location, is assigned to a string or numerical value. The location attribute is assigned to another object which itself has 3 attributes, namely "address1", "city", and "state".

This may all look different, but rest assured, it is a much more logical way to represent our information. Furthermore, you will see that when you want to obtain data from an Application Programming Interface (API), which is a very common way to quickly obtain publicly available data these days, many of the APIs return the data to you in this format. However, when we go to conduct analysis on the data, we may need to convert our data to the more traditional table format (this is part of the "transform" function in the ETL paradigm). Luckily, we can easily read in JSON formatted files into R using the

```
>install.packages("jsonlite")
>library(jsonlite)
> data_json<-read_json("pizza.json")
> names(data_json)
[1] "data"
> names(data_json$data)
[1] "search"
> names(data_json$data$search)
[1] "total"    "business"
> length(data_json$data$search$business)
[1] 50
> data_json$data$search$business[[1]]
$name
[1] "Mercato Tomato Pie"

$url
[1] "https://www.yelp.com/biz/mercato-tomato-pie-newark?adjust_creative=
    yu71TTzPRNWvhkXpS1DYaw&utm_campaign=yelp_api_v3&utm_medium=api_v3_graphql&utm_
    source=yu71TTzPRNWvhkXpS1DYaw"

$location
$location$address1
[1] "212 Market St"

$location$city
[1] "Newark"

$location$state
[1] "NJ"


$rating
[1] 4
```

Listing 6: Reading in a JSON file.

What the package `jsonlite` will do is read in the data from a JSON file and organize all of the data in an R *list*(if you have not come across an R list before, I recommend you review through it here, which has a pretty good introductory breakdown of them). Notice how the list is embedded, just as it is in the JSON file. That is, the main object embeds the data object, which embeds the search object, which embeds the business object (which is a list of individual businesses). We then pull (using the double bracket notation) the first object in the list, which is another list of attributes specific to a single company.

We can leverage many tools in R to convert this file to a CSV. However, this necessitates knowledge of the underlying data. Unfortunately, since data can be structured in many ways, it is usually up to the user to define the procedure for converting from JSON to CSV (or another table format). We will expore this later. As an example of how we can gather all of our data, the code below will gather all of the names of our businesses in our data set:

```
> businesses<-data_json$data$search$business
> business_names<-c()
> for(business in businesses){
+   business_name<-business$name
+   business_names<-c(business_names,business_name)
+}
> business_names
 [1] "Mercato Tomato Pie"                  "Robert's Pizzeria"
 [3] "Blaze Fast-Fire'd Pizza"             "Queen Pizza II"
 [5] "Gina's Pizzeria"                     "Mulberry Street Pizza"
 [7] "Two Brothers Pizza"                  "Napoli Pizza"
 [9] "Golden Pizzeria"                     "Ah' Pizz - Harrison"
[11] "Big Nick's Pizzeria"                 "Ramon's Pizza and Wings"
[13] "Pizzeria Sao Paulo"                  "Brazilian Pizza"
[15] "Tony's Pizzeria"                     "Dickie Dee's"
[17] "Francesca's Pizza & Pasta"           "Star Tavern"
[19] "Queen Pizza"                         "Pizza Village Cafe"
[21] "Sabatino's"                          "Papa Pat's Pizza"
[23] "Uptop Pizzeria"                      "Nino's Pizza & Restaurant"
[25] "Pizza Land"                          "Comet Pizzeria"
[27] "Queen Pizza & Deli"                  "Frank's Pizzeria"
[29] "Bread of Life Pao da Vida Bakery"    "Eli's Pizza Pasta"
[31] "Gencarelli's Pizzeria & Restaurant"  "Bivio Panificio"
[33] "Pizzalino's"                         "Pizza Town"
[35] "Giovanni Pizza Pasta & Grill"        "Urban Bricks Pizza"
[37] "Pizza 2000"                          "Vinny's Pizzeria"
[39] "Mama's Pizza"                        "Saffron Kebab & Pizza"
[41] "Johnny Napkins"                      "Joe's Restaurant & Pizzeria"
[43] "La Pizza"                            "Jersey Fried Chicken"
[45] "Mozzarella"                          "La Sicilia"
[47] "Andros Diner"                        "Pizza Hut"
[49] "New York Fried Chicken & Pizza"      "Starlite Restaurant & Pizza"
```

Listing 7: Getting all of the business names in our data

And notice, there is a shorter way to accomplish this using the `lapply` function (I will leave it up to you as a self-exercise to understand this function):

```
>businesses <-data_json$data$search$business
>business_names<-lapply(businesses,function(x){
   x$name
})
>business_names<-unlist(business_names)
```

Listing 8: Using lapply to extract all business names

A simliar format which is often used in web-based applications that return data is XML. While XML is old, and is slowly being replaced with JSON and other formats (Base 64 is a recent newcomer, but we will not discuss this either here), it is still widely used. Take a look at the structure of the `pizza.xml` file:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<root>
   <data>
      <search>
         <businesses>
            <business>
               <location>
                  <address1>212 Market St</address1>
                  <city>Newark</city>
                  <state>NJ</state>
               </location>
               <name>Mercato Tomato Pie</name>
               <rating>4</rating>
               <url>https://www.yelp.com/biz/mercato-tomato-pie-newark?adjust_
    creative=yu71TTzPRNWvhkXpS1DYaw&amp;utm_campaign=yelp_api_v3&amp;utm_medium=api
    _v3_graphql&amp;utm_source=yu71TTzPRNWvhkXpS1DYaw</url>
            </business>
...................................................
             <business>
               <location>
                  <address1>993 Pleasant Valley Way</address1>
                  <city>West Orange</city>
                  <state>NJ</state>
               </location>
               <name>Starlite Restaurant &amp; Pizza</name>
               <rating>3</rating>
               <url>https://www.yelp.com/biz/starlite-restaurant-and-pizza-west-
    orange?adjust_creative=yu71TTzPRNWvhkXpS1DYaw&amp;utm_campaign=yelp_api_v3&amp;
    utm_medium=api_v3_graphql&amp;utm_source=yu71TTzPRNWvhkXpS1DYaw</url>
            </business>
         </businesses>
         <total>547</total>
      </search>
   </data>
</root>
```

Listing 9: The pizza data but in XML format

Notice the structure of this file format. It is similar to the JSON format, but slightly different. Data is stored inside *tags*. We always indicate information inside an open and closing tag. For example, the city "Newark" is stored in-between (or inside) the open tag `<city>` and the closing tag `</city>`. Closing tags will always have the slash. We can represent *objects* by creating a tag (for example, the tag business represents a business object) and putting other tags inside (or in between) the object's tags. So the business object will have location, a name, a rating, and a url tags. The values of each, respectively, are again stored between the tags.

Notice again that the location tag is an object instead of a number or character, and so what is stored in these tags are again other tags, namely address1, city, and state. Put differently, XML and JSON are very similar to each other, with the only distinction being in how the meta-data (i.e. the variable/attribute names) are organized). Once we load XML into R and convert it to a list, we can take similar extraction methods to convert it over to CSV/Vector format, something again we will see in a later tutorial. We can read in an XML file using the package `xml2`:

```
> install.packages("xml2")
> library(xml2)
> data_xml<-read_xml("pizza.xml")
> data_xml<-as_list(data_xml)
> names(data_xml)
[1] "root"
> names(data_xml$root)
[1] "data"
> names(data_xml$root$data)
[1] "search"
> names(data_xml$root$data$search)
[1] "businesses" "total"
> data_xml$root$data$search$businesses[[1]]
$location
$location$address1
$location$address1[[1]]
[1] "212 Market St"

$location$city
$location$city[[1]]
[1] "Newark"

$location$state
$location$state[[1]]
[1] "NJ"

$name
$name[[1]]
[1] "Mercato Tomato Pie"

$rating
$rating[[1]]
[1] "4"

$url
$url[[1]]
[1] "https://www.yelp.com/biz/mercato-tomato-pie-newark?adjust_creative=
    yu71TTzPRNWvhkXpS1DYaw&utm_campaign=yelp_api_v3&utm_medium=api_v3_graphql&utm_
    source=yu71TTzPRNWvhkXpS1DYaw"
```

Listing 10: Loading in an XML file into R.

Exercises

1. Find a way to load in the file `household_survey_delim.csv`. First inspect it to understand how it should be loaded. Then, ensure it is loaded correctly by using the correct commands. I would recommend looking up the base command online, and understand the inputs so that you can properly adjust it as needed.

2. Write R code to extract the ratings and urls of each business. Ensure these are stored in their own respective vectors.

4 Tutorial 2: Cleaning and Understanding Your Data

When you are exploring and eventually analyzing your data, you must ensure one thing remains true: your data is *clean*. So what does it mean to have "clean" data? Ideally, a data-set ready for basic analysis should **not** have any of the following problems:

- Missing Values
- Different values for categorical variables
- Characters in numerical variables
- One value for each observation and variable

Furthermore, specific to R, your data also needs to be in a certain format (depending on the type of analysis you are doing). If you are doing a basic descriptive analysis in a data frame, then you need to ensure that each column is appropriately indicated as a character, factor, or number. Furthermore, you need to ensure that all numerical columns are actually numerical, and you will need to know how to handle or work with information that is blank (in R, it automatically designates this type of data as "NA", and you must know how to handle these types of situations).

Just like analyzing data, the cleaning of data is an art. There is no one single procedure for you to take. It necessitates making tough decisions between loosing data as well as keeping it, and, potentially manipulating it (in an ethical way, of course). No one can "teach you" how to clean data. This is a skill that comes from many years of experience working with data. With this said, there are fundamental principles for cleaning (mostly those enumerated above). First, load in the data set from earlier into R. The very first action we would like to take is to ensure that all the variables are being brought into R in the correct manner:

```
> data<-read.csv("household_survey.csv")
> str(data)
'data.frame': 5916 obs. of  10 variables:
 $ NEWID   : int  3639434 3639444 3639454 3639504 3639544 3639564 3639594 3639614
    3639624 3639634 ...
 $ AGE_REF : int  36 57 57 22 71 67 55 72 65 65 ...
 $ AGE2    : int  36 57 NA 27 NA 68 52 NA NA 63 ...
 $ BATHRMQ : int  3 1 1 2 1 1 1 2 1 1 ...
 $ BEDROOMQ: int  5 3 1 3 3 3 2 3 3 3 ...
 $ BLS_URBN: int  1 1 1 1 1 1 1 1 1 1 ...
 $ FAM_SIZE: int  4 3 1 3 1 2 3 1 1 2 ...
 $ INC_RANK: num  0.0964 0.3435 0.1532 0.4964 0.2817 ...
 $ RENTEQVX: int  2770 500 NA 1200 300 3500 1600 2100 900 1200 ...
 $ VEHQ    : int  2 2 0 3 1 3 1 1 1 3 ...
```

Listing 11: Understanding the structure of our data.

Notice that that we have 10 variables, all of which appear to be numerical. However, this is not necessarily the case. The variable `BLS_URBN` is actually a categorical variable, which indicates whether or not the individual lives in a rural or urban portion of the United States. Yet, R is treating this variable as a numerical variable, because the categories are *coded*, that is, the categories are labels not with names but with numbers. With this stated, we can ensure that any intended categorical variables are categorical by leveraging the `as.factor` or just simply `factor` functions. This will convert what R considers a numerical variable into a categorical variable:

```
> data$BLS_URBN<-as.factor(data$BLS_URBN)
> str(data)
'data.frame': 5916 obs. of  10 variables:
 $ NEWID   : int  3639434 3639444 3639454 3639504 3639544 3639564 3639594 3639614
   3639624 3639634 ...
 $ AGE_REF : int  36 57 57 22 71 67 55 72 65 65 ...
 $ AGE2    : int  36 57 NA 27 NA 68 52 NA NA 63 ...
 $ BATHRMQ : int  3 1 1 2 1 1 1 2 1 1 ...
 $ BEDROOMQ: int  5 3 1 3 3 3 2 3 3 3 ...
 $ BLS_URBN: Factor w/ 2 levels "1","2": 1 1 1 1 1 1 1 1 1 1 ...
 $ FAM_SIZE: int  4 3 1 3 1 2 3 1 1 2 ...
 $ INC_RANK: num  0.0964 0.3435 0.1532 0.4964 0.2817 ...
 $ RENTEQVX: int  2770 500 NA 1200 300 3500 1600 2100 900 1200 ...
 $ VEHQ    : int  2 2 0 3 1 3 1 1 1 3 ...
```

Listing 12: Ensuring that our categorical variables are not considered numerical.

Next, we need to find potential missing values, and figure out what to do with them. There is no hard fast way to accomplish this. Usually, we always strive to keep information we already have in the data, if we can. This means that we would need to *impute* a number to a variable whose value is missing. In other situations, like ours, we may need to simply split the data into more than one data set. For our example, we can run the `summary` function on our data set to determine if there are any missing values, which is indicated by "NA". Running the code below, we can see that the variables `AGE2`, `BATHRMQ`, `BEDROOMQ`, and `RENTEQVX` have NA values. That is, there is missing information in these columns:

```
> summary(data)
     NEWID             AGE_REF           AGE2          BATHRMQ
 Min.   :3639434   Min.   :16.00   Min.   :18.0   Min.   :0.00
 1st Qu.:3694423   1st Qu.:37.00   1st Qu.:38.0   1st Qu.:1.00
 Median :3758252   Median :53.00   Median :52.0   Median :2.00
 Mean   :3752624   Mean   :52.17   Mean   :51.6   Mean   :1.72
 3rd Qu.:3820314   3rd Qu.:65.00   3rd Qu.:63.0   3rd Qu.:2.00
 Max.   :3860671   Max.   :88.00   Max.   :88.0   Max.   :8.00
                                   NAs   :2978   NAs   :55
    BEDROOMQ      BLS_URBN    FAM_SIZE         INC_RANK
 Min.   :0.000   1:5517   Min.   : 1.000   Min.   :0.0001603
 1st Qu.:2.000   2: 399   1st Qu.: 1.000   1st Qu.:0.2563882
 Median :3.000            Median : 2.000   Median :0.5011178
 Mean   :2.803            Mean   : 2.403   Mean   :0.5018925
 3rd Qu.:3.000            3rd Qu.: 3.000   3rd Qu.:0.7500923
 Max.   :8.000            Max.   :11.000   Max.   :1.0000000
 NAs   :57
    RENTEQVX         VEHQ
 Min.   :   1   Min.   : 0.000
 1st Qu.:1000   1st Qu.: 1.000
 Median :1500   Median : 2.000
 Mean   :1665   Mean   : 1.839
 3rd Qu.:2000   3rd Qu.: 2.000
 Max.   :6442   Max.   :14.000
 NAs   :2160
```

Listing 13: Running summary to find missing values in our data.

Given that our sample size is large enough, we can sacrifice some of this data by simply deleting it, namely for the `BATHRMQ` and `BEDROOMQ` variables. Notice, however, that a fairly large number of observations has "NA" for the AGE2 variable (the spouse's age) and the RENTEQVC variable. This is due to the fact that not everyone has a spouse in their home, and not everyone may have known what their rent would be on their home. Therefore, we can slightly adjust the data by removing the second age variable and replacing it with a "married" variable. We can do the same by replacing the rent data with an indicator variable that indicates if the person knows the value of their home. We can also split off the observations of those who are married and know their rent into a second data set, which would represent a more refined portion of the population of interest. We will reserve doing this for a later section. The removal of variables and general "cleaning" process looks like the following:

```
> data<-data[!is.na(data$BATHRMQ),]
> data<-data[!is.na(data$BEDROOMQ),]
> MARRIED<-rep(0,nrow(data))
> MARRIED[which(!is.na(data$AGE2))]=1
> KNOWS_VALUE<-rep(0,nrow(data))
> KNOWS_VALUE[which(!is.na(data$RENTEQVX))]=1
> data<-data.frame(data,MARRIED=as.factor(MARRIED),KNOWS_VALUE=as.factor(KNOWS_
   VALUE))
> data<-data[,c("NEWID","AGE_REF","BATHRMQ","BEDROOMQ","BLS_URBN","FAM_SIZE","INC_
   RANK","VEHQ","MARRIED","KNOWS_VALUE")]
> summary(data)
     NEWID             AGE_REF         BATHRMQ         BEDROOMQ      BLS_URBN
 Min.   :3639434   Min.   :16.00   Min.   :0.00   Min.   :0.000   1:5464
 1st Qu.:3694453   1st Qu.:38.00   1st Qu.:1.00   1st Qu.:2.000   2: 395
 Median :3758262   Median :53.00   Median :2.00   Median :3.000
 Mean   :3752614   Mean   :52.36   Mean   :1.72   Mean   :2.803
 3rd Qu.:3820316   3rd Qu.:65.00   3rd Qu.:2.00   3rd Qu.:3.000
 Max.   :3860671   Max.   :88.00   Max.   :8.00   Max.   :8.000
    FAM_SIZE         INC_RANK              VEHQ         MARRIED  KNOWS_VALUE
 Min.   : 1.000   Min.   :0.0001603   Min.   : 0.000   0:2928   0:2118
 1st Qu.: 1.000   1st Qu.:0.2627644   1st Qu.: 1.000   1:2931   1:3741
 Median : 2.000   Median :0.5053216   Median : 2.000
 Mean   : 2.411   Mean   :0.5055027   Mean   : 1.853
 3rd Qu.: 3.000   3rd Qu.:0.7527321   3rd Qu.: 2.000
 Max.   :11.000   Max.   :1.0000000   Max.   :14.000
```

Listing 14: Cleaning up missing values and slightly adjusting our data. Notice how there are no more NAs after cleaning which indicates every observation has a value for every variable.

Last, we want to ensure that each numerical variable is actually numerical (there are no letters put in there by accident). We can accomplish this by inspecting each variable through a descriptive analysis. Once the data is clean (or at least partially clean), we can conduct a descriptive analysis in R. This type of analysis mostly deals with summarizing information about our numerical and categorical variables, as well as looking at potential correlations or associations between the variables via cross tabulations, correlation coefficients, and visualizations. We start a descriptive analysis by computing count tables and relative frequency tables in R for our categorical variables. For our numerical variables, we compute and report on measures of central tendency (mean, median, mode), measures of dispersion (range, min, max, variance, standard deviation, mean absolute deviation), and measures of distribution (mean, variance, skewness, kurtosis, quartiles, etc). We accomplish all of our descriptive analysis of our data below:

```
> table(data$BLS_URBN)/nrow(data)

         1          2 
0.93258235 0.06741765 
> table(data$MARRIED)/nrow(data)

       0        1 
0.499744 0.500256 
> table(data$KNOWS_VALUE)/nrow(data)

        0         1 
0.3614951 0.6385049 
> numerical_variables<-data[,c("AGE_REF","BATHRMQ","BEDROOMQ","FAM_SIZE","INC_RANK
    ","VEHQ")]
> apply(numerical_variables,2,var)
     AGE_REF      BATHRMQ     BEDROOMQ     FAM_SIZE     INC_RANK 
307.19847817   0.57195377   1.19086713   2.02361423   0.08159852 
        VEHQ 
  2.15200975 
> apply(numerical_variables,2,sd)
   AGE_REF    BATHRMQ   BEDROOMQ   FAM_SIZE   INC_RANK       VEHQ 
17.5270784  0.7562763  1.0912686  1.4225380  0.2856546  1.4669730 
```

Listing 15: Running a basic Descriptive Analysis in R on a simple data set.

Furthermore, we can look at potential associations between our variables. This can be done by computing tables. More simplistically, we can create plots that illustrate the different potential associations between the data. First we run code to compute the correlation matrix on the numerical variables. Next, we look at averages of values split across different categories for our categorical variables. Last, we look at cross-tabulations to understand potential associations between two categorical variables:

```
> cor(numerical_variables)
             AGE_REF   BATHRMQ  BEDROOMQ   FAM_SIZE   INC_RANK       VEHQ
AGE_REF   1.00000000 0.0737906 0.0398201 -0.2969729 -0.1431544 0.02259579
BATHRMQ   0.07379060 1.0000000 0.6040733  0.2097062  0.2840587 0.28144646
BEDROOMQ  0.03982010 0.6040733 1.0000000  0.3873933  0.2922886 0.35525881
FAM_SIZE -0.29697294 0.2097062 0.3873933  1.0000000  0.2446864 0.27108114
INC_RANK -0.14315437 0.2840587 0.2922886  0.2446864  1.0000000 0.33362051
VEHQ      0.02259579 0.2814465 0.3552588  0.2710811  0.3336205 1.00000000
> table(data$BLS_URBN,data$MARRIED)

       0    1
  1 2745 2719
  2  183  212
> table(data$KNOWS_VALUE,data$MARRIED)

       0    1
  0 1506  612
  1 1422 2319
> table(data$KNOWS_VALUE,data$BLS_URBN)

       1    2
  0 2023   95
```

```
 1 3441   300
```

Listing 16: Looking at correlations and cross tabulations.

As you can see, the `cor` function will return a correlation matrix, which indicates the correlation between the variable in the row and the column. Recall that correlation is a measure of *linear association* (NOTE: Association is not equivalent with causation. Likewise, note the emphasis placed on the word linear. Correlation does not mean any association, but more specifically a linear association). We also have from the cross tabulation (using the `table` function) a count of the number of observations that were in a category in the first variable and simultaneously in a category in the second variable. We are not only restricted to descriptive analysis by way of numbers and tables, but we can further gain understanding by visualizing our data. This can be done by looking a bar plots for categorical variables, histograms and box-and-whisker plots for numerical variables, scatter plots for two numerical variables, box-and-whisker plots for categorical and numerical variables, and spine plots for two categorical variables. All of these visuals give us a sense of what our data looks like:

```
> #Draw Individual Histograms
> par(mfrow=c(2,3))
> hist(numerical_variables$AGE_REF)
> hist(numerical_variables$BATHRMQ)
> hist(numerical_variables$BEDROOMQ)
> hist(numerical_variables$FAM_SIZE)
> hist(numerical_variables$INC_RANK)
> hist(numerical_variables$VEHQ)

#Draw Individual Box and Whisker Plots
> par(mfrow=c(2,3))
> boxplot(numerical_variables$AGE_REF)
> boxplot(numerical_variables$BATHRMQ)
> boxplot(numerical_variables$BEDROOMQ)
> boxplot(numerical_variables$FAM_SIZE)
> boxplot(numerical_variables$INC_RANK)
> boxplot(numerical_variables$VEHQ)

#Draw scatter plots between all numerical variables
> par(mfrow=c(1,1))
> plot(numerical_variables)

#If it is not clear enough, you can always convert the numerical variable to a
    categorical variable and plot a bunch of box plots:
>plot(as.factor(numerical_variables$FAM_SIZE),numerical_variables$INC_RANK)

#You can also take the numerical variable, and cut it into intervals, creating a
    box plot of the dependent variable for each interval of the independent
    variable:
> plot(cut(numerical_variables$AGE_REF,breaks=50),numerical_variables$INC_RANK)

#Last, we can plot the categorical variables:
> plot(data$BLS_URBN,data$MARRIED)
> plot(data$BLS_URBN,data$KNOWS_VALUE)
> plot(data$MARRIED,data$KNOWS_VALUE)
```

Listing 17: Visualizing the data with various types of plots.

Exercises

1. Clean the data to create a new data set from the original one that only includes data points of married individuals who know the value of their rent. Include the variables `AGE_REF`, `AGE2`,`BATHRMQ`,`BEDROOMQ`,`FAM_SIZE`,`INC_RANK`,`VEHQ`,`RENTEQVX`. It is mandatory that there exist absolutely no "NA"s.

2. On the data that only includes married individuals who know the value of their home, conduct a full descriptive analysis. Ensure to find the quantiles (1st,2st [ie median], 3rd), min, max, variance, standard deviation, count tables, correlation table, and cross tabulations.

5 Tutorial 3: Extract-Transform-Load

The concept of Extract-Transform-Load (ETL) stems from the fact that organizations have many data bases in their respective firms. Each data base serves its own purpose. However, there are occasions where we will need to stitch information in one database with information in another database to report to a manager, or, to store in some other database in the organization. The sequence we often go through to accomplish this is the process of Extract-Transform-Load. We will illustrate one such implementation of the process by showing you how to combine information to craft a new set of data.

In your folder, find three files that you will need to load into R, namely the `dow_companies.csv` , `stocks.csv`, and the `tweets.json` files. Each of these files contain different types of data regarding all companies that are listed on the Dow Jones Industrial Average. The first file lists simple information about the industry, company name, what date they were added to the Dow, as well as other information regarding each company. The second file contains daily stock data for each of the stocks, which has information on the high, low, open, close, and volume. The third file is a JSON file that has Tweets on each company's Twitter Timeline.

This example is similar to what you will experience in practice. We have three different data sets (company info, stock information, social media data) from which we would like to the the data (Extract), convert it into a specific data structure (Transform), and subsequently print it out to a file, or simply analyze it (Load). In order to do so, we need to think about the unit of analysis in each data set and try to determine what should the unit of analysis be for the final data set. The final data set should comprise of 30 entries, where each column represents information regarding the company. We want to keep everything that was in the original data. However, we would like to append to this the average open, high, low, and close for each stock, the company's Twitter profile information (number of followers, etc), and the average number of characters, retweets, quotes, and other engagement and post information about the company's social media activity. First, we will illustrate the extraction:

```r
#Load the files into R
> library(jsonlite)

Attaching package:     jsonlite

The following object is masked from   package  :rtweet :

    flatten
```

```

Warning message:
package     jsonlite     was built under R version 3.6.1
> company_json<-read_json("tweets.json")
Warning message:
JSON string contains (illegal) UTF8 byte-order-mark!
> dow_companies<-read.csv("dow_companies.csv")
> stocks<-read.csv("stocks.csv")

#Convert JSON to a data frame.  This can be accomplished
#by iterating through each name of the company
>to_iterate<-names(company_json[[1]])
to_construct<-list()
for(i in to_iterate){
to_construct[[i]]<-unlist(lapply(company_json,function(x){x[[i]]}))
}
>tweets<-data.frame(to_construct)
> str(tweets)
'data.frame': 2805 obs. of  7 variables:
 $ screen_name    : Factor w/ 29 levels "3M","AmericanExpress",..: 1 1 1 1 1 1 1 1 1 1 ...
 $ status_id      : num  1.21e+18 1.21e+18 1.21e+18 1.21e+18 1.21e+18 ...
 $ text           : Factor w/ 2802 levels "'Tis the season ... <U+0001F912> <U+0001F637> https://t.co/zYXhxQcfFN",..: 648 1674 1260 773 1022 1259 534 206 1615 269 ...
 $ retweet_count  : int  0 2 0 0 0 0 0 0 3 0 ...
 $ quote_count    : Factor w/ 1 level "NA": 1 1 1 1 1 1 1 1 1 1 ...
 $ friends_count  : int  4998 4998 4998 4998 4998 4998 4998 4998 4998 4998 ...
 $ followers_count: int  1425626 1425626 1425626 1425626 1425626 1425626 1425626 1425626 1425626 1425626 ...
> str(stocks)
'data.frame': 94783 obs. of  8 variables:
 $ X       : Factor w/ 94783 levels "2007-01-03","2007-01-031",..: 1 29 57 85 113 141 169 197 225 253 ...
 $ Open    : num  77.5 78.4 77.9 77.4 78 ...
 $ High    : num  78.8 78.4 77.9 78 78.2 ...
 $ Low     : num  77.4 77.4 77 77 77.4 ...
 $ Close   : num  78.3 77.9 77.4 77.6 77.7 ...
 $ Volume  : num  3781500 2968400 2765200 2434500 1896800 ...
 $ Adjusted: num  55.5 55.3 54.9 55.1 55.1 ...
 $ Quote   : Factor w/ 30 levels "AAPL","AXP","BA",..: 17 17 17 17 17 17 17 17 17 17 ...
> str(dow_companies)
'data.frame': 30 obs. of  6 variables:
 $ Company   : Factor w/ 30 levels "3M","American Express",..: 1 2 3 4 5 6 7 21 8 9 ...
 $ Exchange  : Factor w/ 2 levels "NASDAQ","NYSE": 2 2 1 2 2 2 1 2 2 2 ...
 $ Symbol    : Factor w/ 30 levels "AAPL","AXP","BA",..: 17 2 1 3 4 6 5 15 8 30 ...
 $ Industry  : Factor w/ 15 levels "Aerospace manufacturer and Arms industry",..: 5 8 10 1 6 12 10 9 4 12 ...
 $ Date.Added: Factor w/ 19 levels "10/1/1928","10/30/1985",..: 17 16 8 5 12 4 15 5 9 1 ...
 $ Twitter.ID: Factor w/ 30 levels "","3M","AmericanExpress",..: 2 3 4 5 6 7 8 9 1
```

```
11 ...
```

Listing 18: Extracting information from each data set.

First, I will show you how to aggregate the data, since the unit of analysis is different accross the two data sets. What for I mean by "unit of analysis"? I simply mean the type of observation made. In the dataset `tweets`, the unit of analysis is a single tweet, which in the dataset `dow_companies`, the unit of analysis is a single company. We need to pick the unit. The `tweet` unit is smaller than the `dow_companies` unit. If we wish to convert the `tweet` unit to a company unit, we need to *aggregate* data. Put differently: there are multiple tweets for a single company. We need to somehow combine the data on the multiple tweets for a single company into single data points so that we can easily match this data with the data stored at the company level. If we were going the opposite way, we would need to repeat the same data for the company. Going from the smaller unit (tweet level) to the larger unit (company level), we aggregate the data by somehow combining all the info (either by averaging, finding the median, variance, min, max, etc). We do some cleaning of the data before hand, however:

```
> summary(tweets[,-3])
          screen_name     status_id          retweet_count       quote_count
 3M              : 100   Min.   :6.646e+17   Min.   :    0.00   NA:2805
 AmericanExpress: 100   1st Qu.:1.198e+18   1st Qu.:    0.00
 Boeing          : 100   Median :1.205e+18   Median :    3.00
 CaterpillarInc : 100   Mean   :1.191e+18   Mean   :   70.74
 Chevron         : 100   3rd Qu.:1.211e+18   3rd Qu.:   11.00
 Cisco           : 100   Max.   :1.213e+18   Max.   :38287.00
 (Other)         :2205
 friends_count    followers_count
 Min.   :    1   Min.   :      0
 1st Qu.:  241   1st Qu.: 206858
 Median : 1008   Median : 555461
 Mean   : 5601   Mean   :1653468
 3rd Qu.: 2563   3rd Qu.:1425626
 Max.   :61503   Max.   :8768393
#Quote Count is not important due to all NAs, so lets remove it:
> tweets<-tweets[,-5]
#Convert the text to a character count
>char_count<-as.numeric(sapply(as.character(tweets$text),nchar))
#Remove text, replace with character count:
>tweets<-tweets[,-3]
>tweets<-data.frame(tweets,char_count)
#Now aggregate by finding the average:
> dat_agg<-aggregate(tweets,list(screen_name=tweets$screen_name),mean)
#Now, clean and force fit the this data frame with the dow companies data frame:
> str(dat_agg)
'data.frame': 28 obs. of  7 variables:
 $ screen_name    : Factor w/ 28 levels "3M","AmericanExpress",..: 1 2 3 4 5 6 7 8
    9 10 ...
 $ screen_name    : num  NA NA NA NA NA NA NA NA NA NA ...
 $ status_id      : num  1.21e+18 1.21e+18 1.18e+18 1.21e+18 1.20e+18 ...
 $ retweet_count  : num  2.13 0.14 282.12 3.77 3.29 ...
 $ friends_count  : num  4998 19017 241 243 301 ...
 $ followers_count: num  1425626 874853 555461 132866 366280 ...
 $ char_count     : num  189 82.1 203.4 202.6 179.1 ...
> dat_agg<-dat_agg[,-2]
> names(dat_agg)[1]<-"Twitter.ID"
```

```
#Finally, combine the two data sets by using merge.
#This function will match values across both sets of
#data based on the "by" input, which in our case is Twitter.ID:
> dat_agg<-merge(dat_agg,dow_companies,"Twitter.ID")
> head(dat_agg)
       Twitter.ID    status_id retweet_count friends_count followers_count
1              3M 1.207419e+18          2.13          4998         1425626
2 AmericanExpress 1.210803e+18          0.14         19017          874853
3          Boeing 1.184170e+18        282.12           241          555461
4  CaterpillarInc 1.208086e+18          3.77           243          132866
5         Chevron 1.201050e+18          3.29           301          366280
6           Cisco 1.207852e+18         13.66          3133          686079
  char_count             Company Exchange Symbol
1     189.02                  3M     NYSE    MMM
2      82.09    American Express     NYSE    AXP
3     203.42              Boeing     NYSE     BA
4     202.57    Caterpillar Inc.     NYSE    CAT
5     179.11 Chevron Corporation     NYSE    CVX
6     181.36       Cisco Systems   NASDAQ   CSCO
                                 Industry Date.Added
1                             Conglomerate   8/9/1976
2                       Financial services  8/30/1982
3 Aerospace manufacturer and Arms industry  3/12/1987
4                   Construction and Mining   5/6/1991
5                       Petroleum industry  2/19/2008
6                   Information technology   6/8/2009
```

Listing 19: Transforming from the tweet level data to the company level via aggregating.

Exercises

3. Continue the merging process. Load in the file stocks.csv, aggregate the data based on the company and by finding the average for each value (open, high, low, close, volume), and last merge this aggregated data with the aggregated data we just created.

6 Tutorial 4: Compressing Data

Reading data into R is only slightly challenging. As we have discussed, reading data into R mainly depends on understanding which format it is in, and to subsequently identify the appropriate library to use to unpackage it. After, the heavy lifting is simply ensuring that the data is in the format it claims to be in, as well as some mild additional cleansing. Writing data, on the other hand, is a whole different challenge, one that we will explore more thoroughly in the next lecture. For now, we will briefly discuss a way to store data in a compressed format, as sometimes it is necessary to save on space, which can translate into saving money, especially if data is stored on cloud devices.

There are many ways to compress information. The fundamental mechanics of it are almost all the same. We try to assign mappings to groups of text/binary information, and store the mapped info and the map, hence reducing the overall size. For example, here is one (yet very simple) data compression method. Suppose we have the text: "hello there! how low can you go, and then find me!". We notice some repetition in permutations of 2-character sequences.

Notice that "he" appears twice. We can represent these two characters using a symbol, lets say the number "2". We have now have: "2llo t2re! how low can you go, and t2n find me!".

We also notice that "e!" appears twice, so replace this with a single character, say "3": "2llo t2r3 how low can you go, and t2n find m3". Also notice that "ow" and "nd" appears more than once, respectively, and assume we replace it with "5" and "4", respectively, so we have:"2llo t2r3 h5 l5 can you go, a4 t2n fi4 m3". Now, if we were to observe how much space this would occupy, we can see that the original text would occupy 52 bytes, while the compress text would only occupy 43 bytes. This equates to a 17% reduction in space (not including the space it takes to store the mapping).

There are many other clever ways to compress information, many of which we will ignore. Now that you understand the fundamental technical way of compressing information, let us discuss the non-technical way. Put simply, compressing data depends on the compressing algorithm you use. Many packages for R exist that do just this. The most fundamental way to compress data is to use the RDS file format. Taking our housing data from before, we can compress this using the following commands:

```
> data<-read.csv("household_survey.csv")
> saveRDS(data,"household_survey.rds")
> file.size("household_survey.csv")
[1] 220518
> file.size("household_survey.rds")
[1] 96386
```

Listing 20: Using R to compress the data frame.

Notice that this is over a 56% reduction in size. Yet another compression mechanism is the *brotli* compression, information of which can be found here. We can compress files and information in brotli using the brotli package. This is a bit more involved. We first need to convert the data to a pure character dataset. Next, we read it in as raw bytes, pass it through the compressor, and subsequently save it. Running the code below, we can see that we are able to achieve over a 71% reduction in size. Compression tools become important in practice when we are seeking to save on space when our data starts to become too big to handle in ordinary text files. This certainly can be the case when we are gathering social media data, article data, and other textual/picture/video data. Compression can help save us!

```
> data<-paste(readLines("household_survey.csv"),collapse="\n")
> data_raw<-charToRaw(data)
> data_compressed<-brotli_compress(data_raw)
> writeBin(data_compressed,"household_survey.brot")
> file.size("household_survey.csv")
[1] 220518
> file.size("household_survey.brot")
[1] 62005
```

Listing 21: Using R to compress the data frame using Brotli compression.

Exercises

4. Compress the JSON and XML files from before using RDS and Brotli. Compare the file sizes across all of them.