A Basic Data Science Workflow

Developing a clean and easy analysis workflow takes a really, really long time. In this post, I outline the workflow that I have developed over the last few years.

August 18, 2017

Rebecca Barter

Developing a seamless, clean workflow for data analysis is harder than it sounds, especially because this is something that is almost never explicitly taught. Apparently we are all just supposed to "figure it out for ourselves". For most of us, when we start our first few analysis projects, we basically have no idea how we are going to structure all of our files, or even what files we will need to make. As we try more and more things in our analysis (perhaps generating a large number of unnecessary files called `analysis2.R`, `next-analysis.R`, `analysis-writeup.Rmd`, `data_clean.csv`, `regression_results.csv` and `data_all.csv` along the way), we find that our project folder gets jumbled and confusing. The frustration when we come back to the project 6 months later and can't remember which file contained the code that lead to the final conclusions is all too real.

This is why I have decided to describe in (possibly too much) detail the data cleaning workflow that I have somehow ended up with. I find this workflow to be particularly useful when dealing with messy (and possibly large) datasets that need several cleaning steps. Note that I did not develop this workflow in conjunction with any resources, I simply figured out what worked best for me via trial-and-error (a process which took 5 years and is definitely still ongoing). There will be several other resources out there on the internet describing "optimal workflows", and these are definitely worth a read too (although a quick google found surprisingly few with the level of detail needed for a beginner). The key is figuring out a workflow that works best for ***you***. That may be similar to mine, or it may not be.

If you decide to keep reading (perhaps because you too suffer from messy-project-syndrome and want some relief), by the end of this post you will know far too much about me and how I spend my time. As you will discover, I am particularly thorough when I clean data, and can spend hours simply making sure that I know what is in the data and moulding it so that it exactly adheres precisely to the format that I consider "clean".

In this post I will describe my thought process as I download, clean and prepare for analysis the data from the 2016 **American Time Use Survey (ATUS)**. I have written my process in sufficient detail such that you can follow along if you'd like to.

The American Time Use Survey is a yearly survey administered by the U.S. Census Bureau and sponsored by the Bureau of Labor Statistics. As with all surveys, it is probably good practice to first get an idea of what kind of population its respondents are supposed to represent. According to their website, the survey is sent to a randomly selected individual from each household in a set of eligible households chosen so as to represent a range of demographic characteristics. The set of eligible households consist of those who have completed their final month of the Current Population Survey (a monthly survey of households conducted by the Bureau of Census for the Bureau of Labor Statistics).

This survey data has been featured heavily on Nathan Yau's blog, Flowing Data, which is where I became aware of it (thanks Nathan!).

Obtaining the data from the website

The ATUS data can be downloaded from the American Time Use Survey Extract Builder which is maintained by the Minnesota Population Center at the University of Minnesota.

To actually obtain the data, you need to click on "Build an Extract" in the left-hand "Data" menu item (or click on "Get Data" inside the "Create an Extract" box). While I was initially confused about what to do once I got to the page with the drop-down menus asking me to "Select Variables", I decided to just go crazy and start clicking. I soon discovered that an "extract" refers to the subset of the data corresponding to whichever variables I like. Once inside each of these drop-down menu entries I needed to click on the yellow plus symbols under "cart" to add the variables to my extract.

After selecting the variables I wanted, I clicked on "Select Samples" and selected only the year 2016. I then went to my cart and clicked on "Create Data Extract" and I was taken to a page where I had to choose my data format. I **changed the data format to .csv** and submitted my extract by clicking on "Submit Extract". Note that you need to create an account to download your selected data, but this only takes a second. Once your data has been created (they will email you when it is ready; this should only take about a minute), you can refresh the page and download the CSV data file!

It is also a good idea to download the basic codebook by right clicking on the link and selecting "Save Link As" (which will tell us what each of the variables mean). I saved the file as a .txt file rather than whatever .cbk (the default) is.

In case you're interested, the variables I selected are listed at the end of this post.

Setting up the project directory

Now that I have successfully downloaded the data (a file called `atus_00002.csv.gz`), I am ready to set up my project directory. This involved a few quick steps.

1. I made a directory called `ATUS/`. This is where my project is going to live.
2. Within `ATUS/`, I made two empty sub-directories `R/` and `data/`
3. In the `R/` sub-directory I make two empty .R files called `load.R` and `clean.R`.
4. I then move the downloaded data and codebook files into `data/`

If you're following along, my working directory now looks like this:

```
data/
  atus_00002.csv.gz
  atus_00002.txt
R/
  clean.R
  load.R
```

It should be pretty obvious what `load.R` and `clean.R` are going to be for: they will be for loading the data and then cleaning the data (shocking, I know!).

While I will later start conducting my analysis in an `eda.Rmd` file, I usually don't want to do the initial data cleaning in this file as it can be long and annoying to have to scroll past. Instead, I prefer to have separate scripts containing functions for loading and cleaning the data which I will later call in my `eda.Rmd` file.

Loading the data: `load.R`

Time to start work in the `load.R` file! The first thing I want to do is attempt to load in the data. Sometimes this is painless and easy, and sometimes this is a nightmare (prompting a session of frantic googling on how to load obscure data types into R).

The first lines of code I write in `load.R` is as follows:

```
# open zipped file for reading
unz <- gzfile("../data/atus_00002.csv.gz")
# load in the data
time_use_orig <- read.csv(unz)
```

when I run it in the console, I am pleasantly surprised to find that it works without any issue.

I then take my first look at the data in the console using the `dim()` command to identify the dimension of the data and the `head()` command to view the first 6 rows.

```
dim(time_use_orig)
```

```
## [1] 10493    64
```

```
head(time_use_orig)
```

```
##         CASEID YEAR NUMCONTACTS_CPS8  HRHH
## 1 2.01601e+13 2016                1 3.509
## 2 2.01601e+13 2016                1 2.025
## 3 2.01601e+13 2016                2 1.243
## 4 2.01601e+13 2016                1 2.041
## 5 2.01601e+13 2016                1 2.203
## 6 2.01601e+13 2016                0 1.021
##   HH_SIZE FAMINCOME HH_NUMKIDS HH_NUMADUL
## 1       3         3          0
## 2       2         6          0
## 3       4         4          2
## 4       4         8          3
## 5       2        13          0
## 6       5         5          4
##   RACE HISPAN ASIAN MARST AGE_CPS8 SEX_CP
## 1  100    100   999     1       62
## 2  110    100   999     1       69
## 3  110    100   999     6       24
## 4  100    100   999     4       31
## 5  100    100   999     1       59
## 6 [illegible]
```

It is pretty clear that everything is coded numerically and the variable names are fairly meaningless to a human. Fortunately, the codebook explains all. I spend some time browsing it.

The `loadData()` function

To make things simple in the long-run, I turn the above commands into a reusable function called `loadData()`. This function will have only one argument that specifies the path of the data in the local directory (relative to the `load.R` file). I usually set the default path to be the actual path for my setup.

```
# a function to load in the data
loadData <- function(path_to_data = "../data/
  # open zipped file for reading
  unz <- gzfile(path_to_data)
  # load in the data
  read.csv(unz)
}
```

To test my function, I simply run in my console by typing

```
time_use_orig <- loadData()
```

and look at the output of `head(time_use_orig)`.

Obviously such a function is a bit redundant in this setting: it is just as easy to write `read.csv(gzfile("../data/atus_00002.csv.gz"))` in my eventual `eda.Rmd` file as it is to write `loadData("../data/atus_00002.csv.gz")`. The reason I keep the `load.R` file in this case is because this is just my default workflow. I always load in my data using a function called `loadData`. In some situations, there are many, many things that need to be done in order to load the data, meaning that my `loadData` function can be fairly complicated. For example, sometimes column names need to be read in separately and then attached to the data, and sometimes I need to play with the format of the data to get R to play nice.

Cleaning the data: `clean.R`

Next, I need to make some decisions about whether to keep the data in its raw, ugly form, or to spend some time making my life easier in the long-run by converting the column names to human-readable versions and converting the numeric codes for each variable to text descriptive characters or factors.

I also need to ensure that missing values are coded as `NA`s and that the class of each variable is what I would expect. For example, when I looked at the `head()` of the data above, I noticed that the `CASEID` variable is printed as a numeric in scientific notation, which is not ideal. IDs should probably be factors or characters (I go back and forth a lot on which I prefer)!

In `clean.R` I start work on a function called `cleanData()`. Like `loadData()`, the function `cleanData()` is always a part of my workflow.

When I eventually start the `eda.Rmd` file, I will load and clean the data like this:

```
# define the loadData() function
source("load.R")
# define the cleanData() function
source("clean.R")
```

```
# load the raw data
time_use_orig <- loadData("../data/atus_0000
# clean the data
time_use <- cleanData(time_use_orig)
```

The `cleanData()` function

The `cleanData()` function will actually call three separate functions, each performing a single task. These functions are

- `renameColumns()` : an optional part of my workflow that changes the column names of each of my columns so that I can actually understand what they mean.
- `convertMissing()` : a function which converts missing values to `NA`
- `convertClass` : a function which sets factor variables to factors, sets character variables to characters, etc

Making columns human-readable: `renameColumns()`

I hate reading column names that are all-caps, use ridiculous abbreviations and generally don't adhere to my definition of "aesthetically pleasing". Thus, whenever possible, I tend to convert my column names to human-readable versions. This is fairly tedious whenever the data has more than around 10 variables or so, but the process itself of renaming the variables is a very effective way of ensuring that you have a good idea of which variables are even in the data.

A word of caution: it is extremely important to check that you have correctly renamed the variables, since it is very easy to assign the wrong name to a variable, resulting in misleading conclusions.

Obviously this step is not practical if you have more than 100 or so variables (although I once did it with a dataset that had around 300 variables!). In addition, if I will at some point need to present the data to people who are very familiar with the original variable names, I won't do any renaming either.

In this case, however, I have no particular allegiance to the original variable names and I want to make it as clear as possible (to myself, at least) what they mean.

To change the variable names, the `renameColumns()` function will leverage the `dplyr` function, `select()` . Note that I also drop a few variables at this stage that I decided weren't interesting.

```
library(dplyr)
renameColumns <- function(data) {
  data <- data %>% select(id = CASEID,
                          year = YEAR,
                          # number of attempt
                          num_contacts = NUM
                          state = STATEFIP,
                          household_size = HH
                          family_income = FAM
                          num_children = HH_N
                          num_adults = HH_NUM
                          age = AGE,
                          sex = SEX,
                          race = RACE,
                          marital_status = MA
                          education_level = E
                          education_years = E
                          employment_status =
                          occupation_category
                          occupation_industry
```

I then test the function out by writing

```
time_use <- renameColumns(time_use_orig)
```

in the console, and looking at the output of `head(time_use)`.

Now, I am fully aware that this function I have just written is not generalizable to alternate subsets of the data variables. This will be one of only two places where I will need to change things if I want to re-run the analysis on a different subset of variables (the second place will be when I explicitly convert numeric variables to their character counterparts). I'm facing a trade-off between generalizability of my pipeline and having human-readable data. If I were intending to repeat this analysis on different variables, I would either remove the part of the workflow where I rename the variables (as well as the part where I convert numeric variables to meaningful factors later on), or I would set the variable names as an argument in the `renameColumns()` function (but sadly, `select()` doesn't play very nicely with variables read in as character strings, so I try to avoid this).

Recoding missing values as `NA`: `convertMissing()`

If you took a look at the codebook, you will have noticed that there are many different ways to say that data is missing. This can be very problematic.

The most common way to code missingness in this data is to code it as `99` `999`, `9999`, etc (depending on whether the entries for the variable are two, three, four or more digit numbers, respectively). These entries are referred to in the codebook as `NIU (Not in universe)`. Other types of missing values are recorded such as `996` corresponding to `Refused`, `997` corresponding to `Don't know` and `998` corresponding to `Blank`.

I now need to decide what to convert to `NA`, keeping in mind that I need to be particularly careful for the variables with many different types of missingness (such as people who refused to answer, didn't know or simply left the entry blank). I decide to take a look to see how widespread these types of missingness are by running the following code in the console:

```
# identify how many times informative missing
informative_missing <- sapply(time_use,
                              function(x) sum
# print out only the non-zero values
informative_missing[informative_missing != 0]
```

```
##          weekly_earning hours_worked_hour
##                       1
##      time_spent_leisure       time_spent_
##                       1
```

Since these types of missingness are extremely rare, I decide to simply lump them in with all of the other `NA` values.

Next, I want to identify which variables have missing values coded as a varying number of 9s. Since the missing values are always supposed to correspond to the maximum, I printed out the maximum of each variable.

```
# print out the maximum of each column
```

```
sapply(time_use, max)
```

```
##                              id
##                    2.016121e+13
##                    num_contacts
##                    8.000000e+00
##                  household_size
##                    1.300000e+01
##                    num_children
##                    9.000000e+00
##                             age
##                    8.500000e+01
##                            race
##                    4.000000e+02
##                 education_level
##                    4.300000e+01
##               employment_status
##                    5.000000e+00
##             occupation_industry
##                    9.999000e+03
##            hours_usually_worked
##                    9.999000e+03
```

I notice here that there are several variables with missing values as their maxima such as `occupation_industry` with a maximum of `9999`, and `hourly_wage` with a maximum of `999.99`. Since I don't really want to manually convert these missing values to `NA`, I decide to automate it using the `mutate_if()` function from the `dplyr` package. First I write a few helper functions in the `clean.R` file for calculating the maximum of a vector and for identifying specific values in a vector.

```
# Helper function for identifying missing va
equalFun <- function(x, value) {
  x == value
}

# Helper function for identifying if the max
maxFun <- function(x, value) {
  max(x, na.rm = T) == value
}
```

The first argument of `mutate_if()` is a function which returns a Boolean value specifying which columns to select. The second argument is wrapped in `funs()` and itself is a function which specifies what to do to each column. `if_else(equalFun(., 99), NA_integer_, .)` can be read aloud as "If the value is equal to 99, convert it to a `NA` of integer type, otherwise do nothing" (the `.` serves as a placeholder for the data, like `x` in `function(x)`).

```
convertMissing <- function(data) {
  # convert missing values to NA
  data <- data %>%
    # mutate all missing values coded as 99
    mutate_if(function(x) maxFun(x, 99),
              funs(if_else(equalFun(., 99),
    # mutate all missing values coded as 999
    mutate_if(function(x) maxFun(x, 999),
              funs(if_else(equalFun(., 999),
    # mutate all missing values coded as 999
    mutate_if(function(x) maxFun(x, 9999),
              funs(if_else(equalFun(., 9999)
    # mutate all missing values coded as 999
    mutate_if(function(x) maxFun(x, 999.99),
              funs(if_else(equalFun(., 999.9
    # mutate all missing values coded as 999
    mutate_if(function(x) maxFun(x, 99999.99
              funs(if_else(equalFun(., 99999
    # mutate all missing values coded as 998
    mutate_if(function(x) maxFun(x, 998),
```

It took some playing around with running the body of the function in the console (with `data` defined as `time_use`) to get it to run without errors (I was getting errors to do with `NA` values and realized that I needed to add `na.rm = T` in the `maxFun()` function).

Once the body runs in the console, I then check to make sure that the complete function worked as expected by running it in the console and checking out the summary of the output.

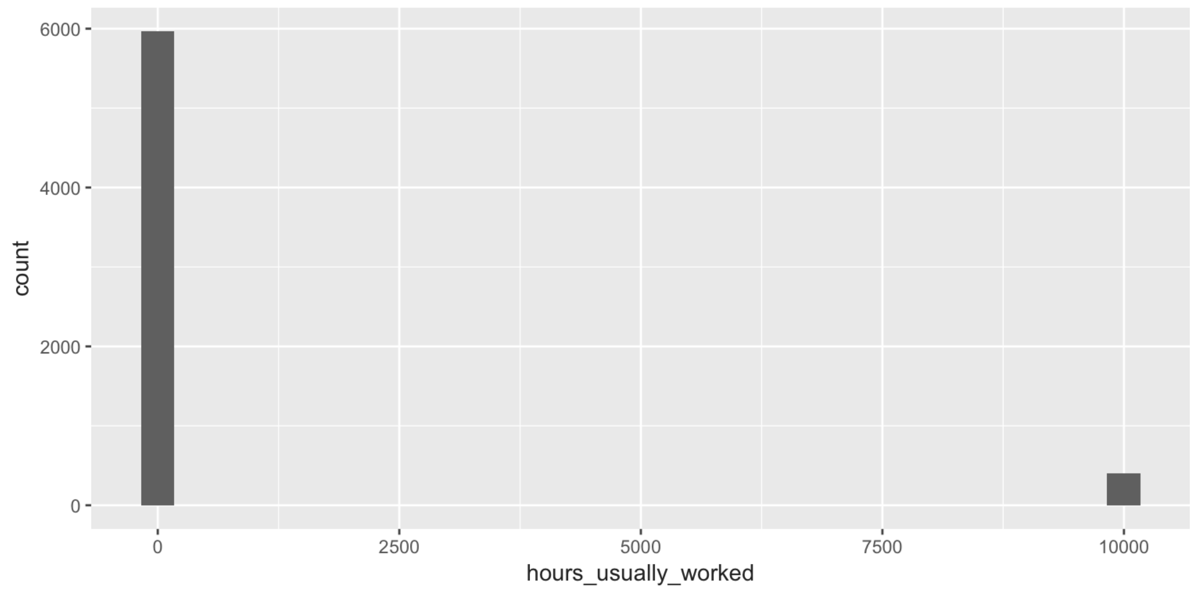
```
# convert the missing values to NAs
time_use <- convertMissing(time_use)
# check out the summary
summary(time_use)
```

```
##        id                 year         num_
##  Min.   :2.016e+13   Min.   :2016   Min.
##  1st Qu.:2.016e+13   1st Qu.:2016   1st Qu
##  Median :2.016e+13   Median :2016   Median
##  Mean   :2.016e+13   Mean   :2016   Mean
##  3rd Qu.:2.016e+13   3rd Qu.:2016   3rd Qu
##  Max.   :2.016e+13   Max.   :2016   Max.
##
##  household_size   family_income    num_ch
##  Min.   : 1.000   Min.   : 1.00   Min.
##  1st Qu.: 1.000   1st Qu.: 8.00   1st Qu.
##  Median : 2.000   Median :12.00   Median
##  Mean   : 2.657   Mean   :10.85   Mean
##  3rd Qu.: 4.000   3rd Qu.:14.00   3rd Qu.
##  Max.   :13.000   Max.   :16.00   Max.
##
##       age              sex              race
##  Min.   :15.00   Min.   :1.000   Min.   :
##  1st Qu.:35.00   1st Qu.:1.000   1st Qu.:
##  Median :49.00   Median :2.000   Median :
```

Scrolling through the summary, I notice a few peculiarities. In particular, there are several variables that have stupidly large values. For example the maximum value for `hours_usually_worked` is `9995` (this didn't appear in the codebook!). I decided to look at a histogram of this variable to see how typical this value is. I ran the following code in the console:

```
library(ggplot2)
ggplot(time_use) + geom_histogram(aes(x = hou
## `stat_bin()` using `bins = 30`. Pick bette
```

```
## Warning: Removed 4119 rows containing non-
```



From the histogram, it is fairly clear that there is an additional type of missing value (405 samples have a value of `9995`) that was not mentioned in the documentation. I then go back and update my `convertMissing()` function to include this extra missing value.

```
convertMissing <- function(data) {
  # convert missing values to NA
  data <- data %>%
    # mutate all missing values coded as 99 
    mutate_if(function(x) maxFun(x, 99),
              funs(if_else(equalFun(., 99), N
    # mutate all missing values coded as 999
    mutate_if(function(x) maxFun(x, 999),
              funs(if_else(equalFun(., 999),
    # mutate all missing values coded as 999
    mutate_if(function(x) maxFun(x, 9999),
              funs(if_else(equalFun(., 9999)
    # mutate all missing values coded as 999
    mutate_if(function(x) maxFun(x, 999.99),
              funs(if_else(equalFun(., 999.9
    # mutate all missing values coded as 999
    mutate_if(function(x) maxFun(x, 99999.99
              funs(if_else(equalFun(., 99999
    # mutate all missing values coded as 998
    mutate_if(function(x) maxFun(x, 998),
```

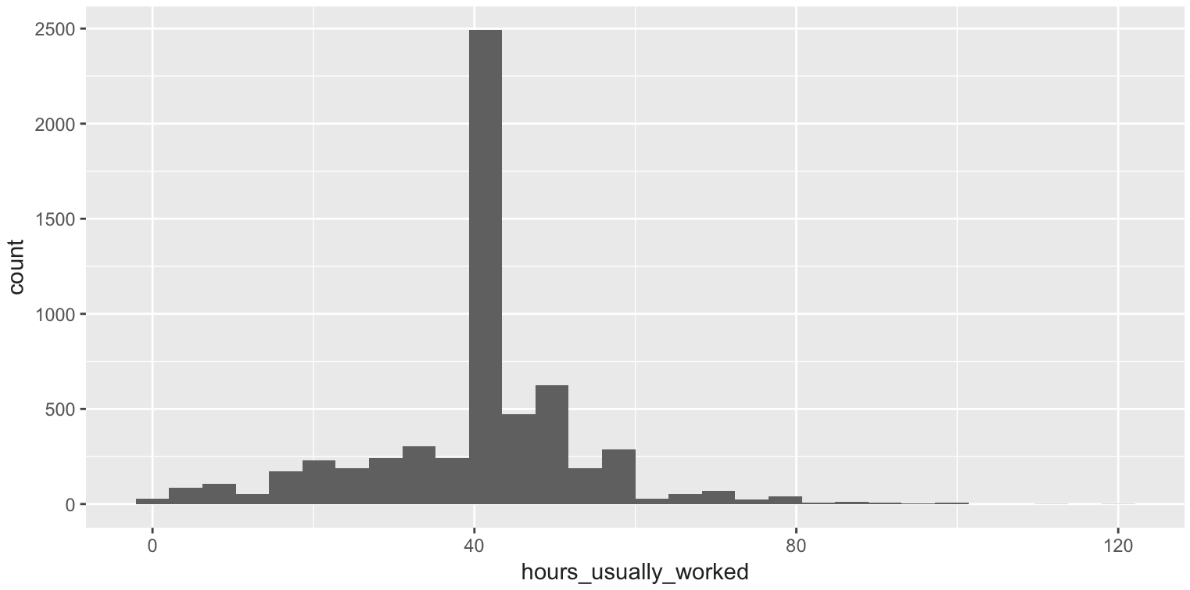
Next, I ran the `convertMissing()` function again the data and re-made the histogram to make sure that everything was going smoothly.

```
# re-run the renameColumns() function
time_use <- renameColumns(time_use_orig)
# convert missing values to NA
time_use <- convertMissing(time_use)
```

```
# re-make the histogram
ggplot(time_use) + geom_histogram(aes(x = hou
```

```
## `stat_bin()` using `bins = 30`. Pick bette
```

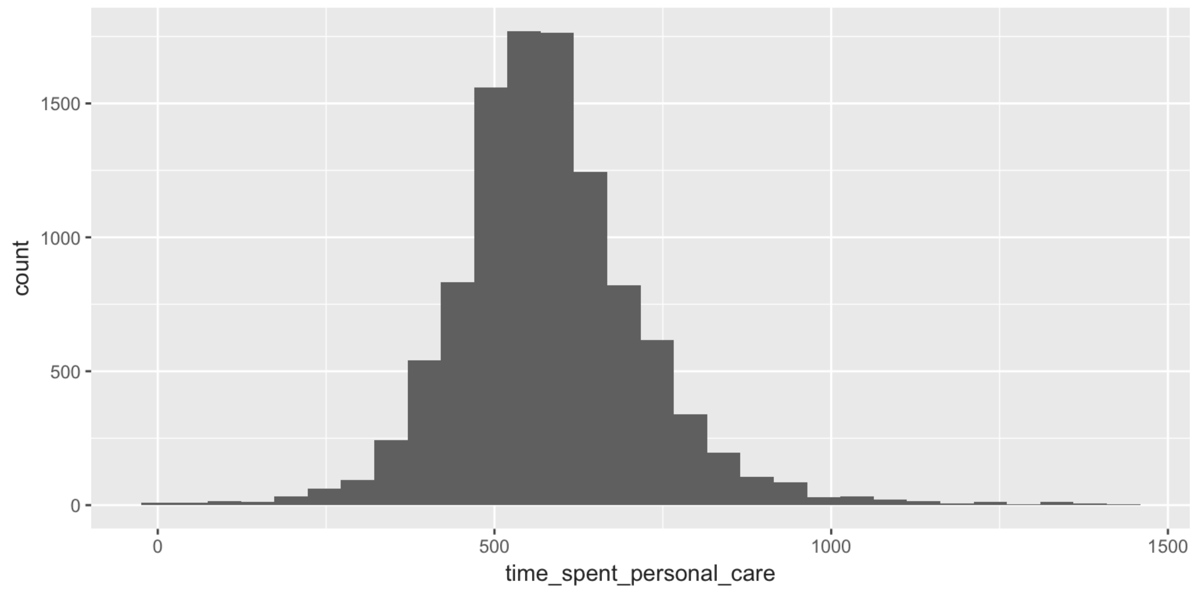
```
## Warning: Removed 4524 rows containing non-
```



Now that that was sorted out, it occurred to me that I wasn't sure what kind of scale the `time_spent` variables were on (is it hours spent in the last week? In the last month? The last year? Perhaps it is minutes spent over the last day? It probably should have occurred to me to ask this earlier, but it didn't. Whatever… I'm asking it now! After spending some time perusing the internet for a while, I found this tablewhich summarised the average hours spend ***per day*** on a range of activities. For example, it said that on average, people spend 9.58 hours per day on "personal care activities". The histogram below shows the distribution of values for the `time_spent_personal_care` .

```
ggplot(time_use) + geom_histogram(aes(x = tir
```

```
## `stat_bin()` using `bins = 30`. Pick bette
```



The mean value in the data is 580, which when divided by 60 gives 9.6. From this “evidence” I conclude that what the data contains is the **number of minutes spent per day**. Whether this is averaged over a week, or is based on one particular day, I honestly don’t know. But for my purposes, I’ll just take each value as the number of minutes spent on the activity on a “typical” day.

Ensuring each variable has the correct class: `convertClass()`

The final cleaning task involves converting categorical values to have a categorical variable class (such as a factor), and other things along these lines involving variable classes.

Recall that the person ID variable, `CASEID`, is currently coded as a numeric (which is printed in scientific notation). In general, it is good practice to code IDs as factors (or characters).

There are also many other variables that should be coded as factors: state, sex, race, marital_status, education_level, family_income, employment_status, occupation_category, and occupation_industry.

Now begins the part of my cleaning process that often takes the longest: I am going to convert each of these numeric variables not only to factors, but to ***meaningful***factors. I don’t want to make plots for genders 1 and 2, or for states 42 and 28; I want to make plots for males and females and for states Pennsylvania and Mississippi.

First, for each variable I need to define a data frame that stores the conversion from number to meaningful category. Fortunately, this information was found in the codebook, and I can copy and paste these code conversions into separate .txt files and save them in the `data/` folder: `states.txt`, `occupation_industry.txt`, `occupation_category.txt`, etc. I can then read them into R as tab-delimited text files.

In case you’re interested, after copying the subsets of the codebook, my project directory now looks like this:

```
data/
  atus_00002.csv.gz
  atus_00002.txt
  education_level.txt
  employment_status.txt
  family_income.txt
  marital_status.txt
  occupation_category.txt
  occupation_industry.txt
  race.txt
  sex.txt
  state.txt
```

```
R/
  clean.R
  load.R
```

I now start work on a `convertClass()` function which will be the third component of my `cleanData()` function. The first thing I do in `convertClass()` is convert the `id` variable to a factor. I then loop through each of the other factor variables to read in the code conversions from the .txt files, join the meaningful factors onto the original data frame using `left_join()` and remove the numeric version of the variable. The function that I wrote is presented below. I spent a while playing around in the console with various versions of the function below (always running code from the .R file rather than typing directly in the console itself).

```
convertClass <- function(data, path_to_codes
  # convert id to a factor
  data <- data %>% mutate(id = as.factor(id)
  # loop through each of the factor variables
  # factor then add to data frame
  for (variable in c("state", "occupation_inc
                     "education_level", "ra
                     "employment_status", '
    # identify the path to the code file
    path <- paste0(path_to_codes, variable, '
    # read in the code file
    codes <- read.table(path, sep = "\t")
    # remove the second column (the entries a
    codes <- codes[, -2]
    # convert the column names
    colnames(codes) <- c(variable, paste0(var
    # add the code to the original data frame
    data <- left_join(data, codes, by = varia
    # remove old variable and replace with ne
    data[, variable] <- data[, paste0(variab
```

After I was done, I tested out that the `convertClass()` did what I hoped by running the following code in the console:

```
# run the convertClass() function
time_use <- convertClass(time_use)
# compare the original variables with the mea
head(time_use)
```

```
##                id year num_contacts
## 1 20160101160045 2016            1
## 2 20160101160066 2016            1
## 3 20160101160069 2016            2 Distri
## 4 20160101160083 2016            1
## 5 20160101160084 2016            1
## 6 20160101160094 2016            0
##          family_income num_children num_adu
## 1   $7,500 to $9,999            0
## 2 $15,000 to $19,999            0
## 3 $10,000 to $12,499            2
## 4 $25,000 to $29,999            3
## 5 $60,000 to $74,999            0
## 6 $12,500 to $14,999            4
##                              race
## 1                      White only Married
## 2                      Black only Married
## 3                      Black only
## 4                      White only
## 5                      White only Married
```

Everything looks good! I add `renameColumns()` , `convertMissing()` and `convertClass()` to the `cleanData()` function. I'm finally done with the cleaning component of my workflow. I may have to come back and add additional steps as I make unpleasant discoveries in my analysis, but for now, I can move on.

Below I print my final `clean.R` file

```
# filename: clean.R

# Main function for data cleaning stage
cleanData <- function(data) {
  # rename each of the columns to be human-re
  # ignore some of the useless columns (such
  data <- renameColumns(data)
  # convert missing data to NA
  data <- convertMissing(data)
  # convert integers to meaningful factors
  data <- convertClass(data)
  return(data)
}

# rename each of the columns to be human-read
renameColumns <- function(data) {
  data <- data %>% select(id = CASEID,
                          year = YEAR,
                          # number of attempt
                          num_contacts = NUM(
```

Analysis: `eda.Rmd`

Where I go from here depends strongly on what questions I want to ask. If I already know the category of questions I'm planning to ask, and, for example, I know that they fall into two groups, then I will probably make two .Rmd files, one for each question.

If, however, I just want to play around with the data for a while, as is the case here, I will make a .Rmd file called `eda.Rmd` (or something along those lines).

Sometimes I end up separating my initial exploration file into several separate files when I start to go down several diverging paths.

Regardless of the analysis I decide to conduct, each of my `.Rmd` files will start with the following code:

```
library(tidyverse)
source("R/load.R")
source("R/clean.R")

# laod the data
time_use_orig <- loadData()
```

```
# clean the data
time_use <- cleanData(time_use_orig)
```

The opportunities for analysis are wide open!

List of variables downloaded from ATUX-X

- CASEID (ATUS Case ID)
- YEAR (Survey year)
- NUMCONTACTS_CPS8 (Number of actual and attempted personal contacts)
- HRHHID_CPS8 (Household ID (CPS))
- HRHHID2_CPS8 (Household ID part 2 (CPS))
- STATEFIP (FIPS State Code)
- HH_SIZE (Number of people in household)
- FAMINCOME (Family income)
- HH_NUMKIDS (Number of children under 18 in household)
- HH_NUMADULTS (Number of adults in household)
- PERNUM (Person number (general))
- LINENO (Person line number)
- WT06 (Person weight, 2006 methodology)
- AGE (Age)
- SEX (Sex)
- RACE (Race)
- HISPAN (Hispanic origin)
- ASIAN (Asian origin)
- MARST (Marital status)
- AGE_CPS8 (Age (CPS))
- SEX_CPS8 (Sex (CPS))
- EDUC (Highest level of school completed)
- EDUCYRS (Years of education)
- SCHLCOLL (Enrollment in school or college)
- SCHLCOLL_CPS8 (Enrollment in school or college (CPS))
- EMPSTAT (Labor force status)
- OCC2 (General occupation category, main job)
- OCC (Detailed occupation category, main job)
- IND2 (General industry classification, main job)
- IND (Detailed industry classification, main job)
- FULLPART (Full time/part time employment status)
- UHRSWORKT (Hours usually worked per week)
- UHRSWORK1 (Hours usually worked per week at main job)
- UHRSWORK2 (Hours usually worked per week at other jobs)
- EARNWEEK (Weekly earnings)
- PAIDHOUR (Hourly or non-hourly pay)
- EARNRPT (Easiest way to report earnings)
- HOURWAGE (Hourly earnings)
- HRSATRATE (Hours worked at hourly rate)
- OTUSUAL (Usually receives overtime, tips, commission at main job)
- OTPAY (Weekly overtime earnings)
- UHRSWORKT_CPS8 (Hours usually worked per week (CPS))
- UHRSWORK1_CPS8 (Hours usually worked per week at main job (CPS))
- UHRSWORK2_CPS8 (Hours usually worked per week at other jobs (CPS))
- HRSWORKT_CPS8 (Hours worked last week (CPS))
- ACT_CAREHH (ACT: Caring for and helping household members)
- ACT_CARENHH (ACT: Caring for and helping non-household members)
- ACT_EDUC (ACT: Educational activities)
- ACT_FOOD (ACT: Eat and drinking)
- ACT_GOVSERV (ACT: Government services and civic obligations)
- ACT_HHACT (ACT: Household activities)
- ACT_HHSERV (ACT: Household services)
- ACT_PCARE (ACT: Personal care)
- ACT_PHONE (ACT: Telephone calls)

- ACT_PROFSERV (ACT: Professional and personal care services)
- ACT_PURCH (ACT: Consumer purchases)
- ACT_RELIG (ACT: Religious and spiritual activities)
- ACT_SOCIAL (ACT: Socializing, relaxing, and leisure)
- ACT_SPORTS (ACT: Sports, exercise, and recreation)
- ACT_TRAVEL (ACT: Traveling)
- ACT_VOL (ACT: Volunteer activities)
- ACT_WORK (ACT: Working and Work-related Activities)
- ERS_ASSOC (ERS: Activities associated with primary eating and drinking (travel and waiting))
- ERS_PRIM (ERS: Primary eating and drinking)

Categories | R | Workflow