The data.table Package for Data Cleaning and Preparation in R

Ben Lim, Huan Qin

December 9, 2016

Contents

Introduction

In class, we mostly used SAS to prepare and clean data. However, as a full-fledged programming language, `R` also has its own data cleaning and preparation capabilities, especially with user-contributed packages. The main one that this project aims to showcase is the `data.table` package (Dowle and Srinivasan 2016), and how it can be used to prepare data efficiently. We will compare this with `R`'s native data preparation capabilities, and compare the difference in computational speed and programming difficulty.

This project will also show other tasks that `R` is capable of executing with the right packages, such as scraping data from the internet, and geospatial data visualization.

The `data.table` package

First released in 2006, this package is the highlight of this project. It introduces a new class of `R` object, called `data.table`, which is an extension of the `data.frame` object. The syntax for code in the `data.table` package is written to resemble `SQL` syntax, which makes programming much easier. At the same time, the code is written in the programming language `C` for speed. It is able to work quickly even with large datasets, reading it in and performing most operations within seconds. Coupled with other packages such as `reshape2` (Wickham 2007), this equips `R` with the data preparation capabilities comparable with top line software such as `SAS` and `SQL`.

Dataset and objectives to be accomplished

The data that we will use for illustration purposes is the county unemployment data by the Bureau of Labor Statistics (BLS). The BLS releases monthly unemployment data at the state, county and even metropolitan levels at a rather timely manner, with a time lag of at most 2 or 3 months.

The county-level data that we are looking at (http://download.bls.gov/pub/time.series/la/la.data.64.County) is rather large (1.5 million observations, 300MB), which makes it a good candidate to compare the time savings that are possible with the `data.table` package. Being geographically indexed, it also allows us to showcase the knowledge and tools needed when working with such data.

The following tasks are reasonable within an applied statistician's line of work, and so we will work towards these as an end-goal to illustrate the data preparation process:

1. Identify the county that each entry in the dataset is associated with, and add in the appropriate county ID
2. Identify the state that each county is associated with, and use the state abbreviation to identify each state
3. Consolidate the various measurements for each month-year-county combination into a single entry
4. Reconstruct state and federal-level unemployment rates, using only county-level data
5. Reformat county-level unemployment into wide format: with counties as the columns, and month-year as the rows (this is useful for certain analyses, such as PCA)
6. Reformat federal-level unemployment with the months as columns and years as rows (useful to identify seasonal trends)
7. Create a heat map of the most recent county-level unemployment rates

Reading in the data

As we know, `read.table` reads in data as a `data.frame` object. Similarly, the `fread` function reads in data as a `data.table` object. We will define `datdt` to be the data read in the format of a `data.table`, and `datdf` to be the data in `data.frame` form.

To save time in reading in the data, we use the argument `colClasses` in the statement, which plays a similar role to `input` in SAS. This saves the time that `R` uses to identify which classes that each variable belongs to. Also, since missing values are coded as "-", we use the argument `na.strings` in the `data.frame` environment; as for `data.table`, we read that particular column in as a `character` variable (since its `na.strings` option merely coerces the column the character anyway):

```
library(data.table)
setwd("~/U Iowa/STAT5400")
system.time(datdt <- fread("la.data.64.County.txt", header = TRUE,
      colClasses = c("character", "factor", "factor", "character", "factor")))
```

```
##
Read 61.6% of 4465684 rows
Read 4465684 rows and 5 (of 5) columns from 0.270 GB file in 00:00:03
```

```
##    user  system elapsed
##    2.77    0.07    2.84
```

```
system.time(datdf <- read.table("la.data.64.County.txt", header = TRUE,
      colClasses = c("character", "factor", "factor", "numeric", "factor"),
      sep="\t", na.strings = "-"))
```

```
##    user  system elapsed
##   23.96    0.74   24.86
```

`data.table` is clearly faster. Unfortunately, it wasn't able to read in the factor variables correctly:

```
str(datdt)
```

```
## Classes 'data.table' and 'data.frame':   4465684 obs. of  5 variables:
##  $ series_id     : chr  "LAUCN010010000000003" "LAUCN010010000000003" "LAUCN010010000000003" "LAUCN0
##  $ year          : chr  "1990" "1990" "1990" "1990" ...
##  $ period        : chr  "M01" "M02" "M03" "M04" ...
##  $ value         : chr  "6.4" "6.6" "5.8" "6.6" ...
##  $ footnote_codes: chr  "" "" "" "" ...
##  - attr(*, ".internal.selfref")=<externalptr>
```

So we will need to convert the character classes manually. defining `colClasses` did save some time in reading in the data though.

```
system.time(datdt[, c("year", "period", "value", "footnote_codes")
  := list(as.factor(year), as.factor(period), as.numeric(value),
      as.factor(footnote_codes))])
```

```
## Warning: NAs introduced by coercion
##    user  system elapsed
##    1.45    0.13    1.58
```

This is syntax that is unique to `data.table`: instead of the usual `datdt[,year]<- as.factor(year)` etc., updating the column is instead done *within* the `[]`, first defining the columns to be updated, then using `:=` to define how to update each of them (using a list). This saves having to refer back to the original `data.table` every time (using the `$` sign with `data.frame`), and condenses code as well.

Now let's take a look at the data that we have. To save typing, we will define `head3` to look at the first three rows of a table:

```
head3 <- function(x){head(x, 3)}
head3(datdt)
```

```
##                series_id year period value footnote_codes
## 1: LAUCN010010000000003 1990    M01   6.4
## 2: LAUCN010010000000003 1990    M02   6.6
## 3: LAUCN010010000000003 1990    M03   5.8
```

```
str(datdt)
```

```
## Classes 'data.table' and 'data.frame':   4465684 obs. of  5 variables:
##  $ series_id     : chr  "LAUCN010010000000003" "LAUCN010010000000003" "LAUCN010010000000003" "LAUCN0
##  $ year          : Factor w/ 27 levels "1990","1991",..: 1 1 1 1 1 1 1 1 1 1 ...
##  $ period        : Factor w/ 13 levels "M01","M02","M03",..: 1 2 3 4 5 6 7 8 9 10 ...
##  $ value         : num  6.4 6.6 5.8 6.6 6 7 6 6.7 7.2 7.1 ...
##  $ footnote_codes: Factor w/ 4 levels "","c","n","p": 1 1 1 1 1 1 1 1 1 1 ...
##  - attr(*, ".internal.selfref")=<externalptr>
```

```
head3(datdf)
```

```
##                         series_id year period value footnote_codes
## 1 LAUCN010010000000003           1990    M01   6.4
## 2 LAUCN010010000000003           1990    M02   6.6
## 3 LAUCN010010000000003           1990    M03   5.8
```

```
str(datdf)
```

```
## 'data.frame':    4465684 obs. of  5 variables:
##  $ series_id     : chr  "LAUCN010010000000003          " "LAUCN010010000000003          " "LAUCN0100
##  $ year          : Factor w/ 27 levels "1990","1991",..: 1 1 1 1 1 1 1 1 1 1 ...
##  $ period        : Factor w/ 13 levels "M01","M02","M03",..: 1 2 3 4 5 6 7 8 9 10 ...
##  $ value         : num  6.4 6.6 5.8 6.6 6 7 6 6.7 7.2 7.1 ...
##  $ footnote_codes: Factor w/ 4 levels "          ","c",..: 1 1 1 1 1 1 1 1 1 1 ...
```

So it seems now the data is in the classes that we need.

Getting a unique identifier: FIPS codes and the {`substr`} `function`

Despite being county-level data, there is no label for each county within the columns. The only column that seems to resemble an ID is the `series_id` column, so we will need to extract a unique identifier from it.

This is where knowledge of how counties are indexed is useful. In the US, each county is assigned a unique 5-number code known as the FIPS code. The first two numbers represent the state that the county is located in: the state code, whereas the last 3 number uniquely identify the county within the state.

Knowing this, and after looking through the various accompanying description files, it is not too hard to figure out that the first 5 numbers after the letters characterize the FIPS code:

- When it changes, the first two numbers usually remain constant.
- It has many 0's after it. (At finer levels of division, these will not be 0)

Another piece of information embeeded in `series_id` is the `measure` code. According to the description files, this is a 2-number code which tells us what `value` is describing: an unemployment rate (03), total unemployment (04), total employed (05), or total labor force (06). Looking at the `value` variable while paying attention to `series_id`, it is not too hard to figure out that `measure` is the last 2 numbers of `series_id`.

So we will use the `substr` function in `R` to extract these pieces of information. The syntax is

```
substr(x, start, stop)
```

Where `x` is the string vector of which a substring is desired, `start` is the start position of the substring desired, and stop is the end position.

```
datdt[, c("fips", "measure") := list(as.factor(substr(series_id, start = 6, stop = 10)),
    as.factor(substr(series_id, start = 19, stop = 20)))]

datdf$fips <- as.factor(substr(datdf$series_id, start = 6, stop = 10))
datdf$measure <- as.factor(substr(datdf$series_id, start = 19, stop = 20))
```

Consolidating data: `reshape2` and the `dcast` function

Now that we have the `measure` codes, we will need to consolidate all of them for each year-period-county combination. This is where the `dcast` function from the `reshape2` package comes in handy; it converts data from long format to wide format. The basic syntax is

```
dcast(dat, formula, ...)
```

Where `dat` is the object to be cast into wide format, and `formula` is the type of casting desired. The basic syntax of formula is

```
V1 + V2 + v3~V4 + V5
```

Where the left-hand side denotes the variables that are used to identify the rows, and the right-side denotes the variables whose values are desired. When more than one value is cast onto a single cell, a aggregate function, `fun.aggregate` will need to be defined as an optional argument to obtain a single value from the various values. Note that `dcast` is faster for `data.table` than for `data.frame`.

```r
system.time(datdt<-dcast(datdt,year+period+fips~measure))
```

```
##    user  system elapsed
##    1.64    0.08    1.71
```

```r
system.time(datdf <- dcast(datdf, year+period+fips~measure))
```

```
##    user  system elapsed
##    4.60    0.46    5.08
```

```r
colnames(datdt)[4:7]<-c("rate","unemployment","employment","total")
colnames(datdf)[4:7]<-c("rate","unemployment","employment","total")
#measure codes are "03" - "06" - changing them to more meaningful names
str(datdt)
```

```
## Classes 'data.table' and 'data.frame':   1116421 obs. of  7 variables:
##  $ year        : Factor w/ 27 levels "1990","1991",..: 1 1 1 1 1 1 1 1 1 1 ...
##  $ period      : Factor w/ 13 levels "M01","M02","M03",..: 1 1 1 1 1 1 1 1 1 1 ...
##  $ fips        : Factor w/ 3222 levels "01001","01003",..: 1 2 3 4 5 6 7 8 9 10 ...
##  $ rate        : num  6.4 6.2 7.1 12.4 5.2 11.4 13.6 6.5 4.8 8.1 ...
##  $ unemployment: num  1050 2865 793 910 965 ...
##  $ employment  : num  15469 43013 10412 6457 17720 ...
##  $ total       : num  16519 45878 11205 7367 18685 ...
##  - attr(*, ".internal.selfref")=<externalptr>
##  - attr(*, "sorted")= chr  "year" "period" "fips"
```

So now we have the data in the form that we need.

Replacing state codes with state abbreviations

Scraping a web table for state abbreviations: the rvest package

To reconstruct state rates from the county rates, we will need the state that each county belongs to. The state codes are the first two numbers of the FIPS code, but for them to be more readable we will also need the corresponding state abbreviations.

This is where we scrape an online table that contains the list of all abbreviations and the corresponding state codes. The `rvest` package (Wickham 2016) comes in handy when scraping HTML tables:

```r
library(rvest)
```

```
## Loading required package: xml2
```

```r
theurl <- "https://www.mcc.co.mercer.pa.us/dps/state_fips_code_listing.htm"
file <- read_html(theurl) #reads the html page into R
tables <- html_nodes(file, "table") #looks for objects of type "table" in HTML
statecodes <- as.data.frame(html_table(tables[1], fill = TRUE), header = TRUE)
#extracts out the html table as a readable format in R
statecodes <- statecodes[-1, ] #deletes header
#The table is divided into two columns; we will need to combine them
```

```
codetable <- data.frame(codes = c(statecodes$X2, statecodes$X5),
                        abbr = c(statecodes$X1, statecodes$X4),
                        names = c(statecodes$X3, statecodes$X6))
codetable <- codetable[-28,] #removes the blank line in the table
```

Matching indices: the `match` function

Now we have the table that we need. We can now extract the state codes from the FIPS code, and replace them with the appropriate abbreviations. Essentially, since our state abbreviations are factor variables, we will need to replace the levels of our state codes factor variable with the appropriate state abbreviations.

For this we will need the `match` function. The basic syntax is

```
match(x, y)
```

This function is a index retrieval function. For each element in `x`, this function returns the position of the corresponding element in `y`. Since the way that the scraped data is ordered doesn't necessarily match the way the levels of the factor variable is ordered, this is useful:

```
datdt[, state := as.factor(substr(fips, start = 1, stop =2))]
```

```
## Warning in `[.data.table`(datdt, , `:=`(state, as.factor(substr(fips, start
## = 1, : Invalid .internal.selfref detected and fixed by taking a (shallow)
## copy of the data.table so that := can add this new column by reference.
## At an earlier point, this data.table has been copied by R (or been created
## manually using structure() or similar). Avoid key<-, names<- and attr<-
## which in R currently (and oddly) may copy the whole data.table. Use set*
## syntax instead to avoid copying: ?set, ?setnames and ?setattr. Also, in
## R<=v3.0.2, list(DT1,DT2) copied the entire DT1 and DT2 (R's list() used to
## copy named objects); please upgrade to R>v3.0.2 if that is biting. If this
## message doesn't help, please report to datatable-help so the root cause can
## be fixed.
```

```
datdf$state <- as.factor(substr(datdf$fips, start = 1, stop = 2))
#extracts state code from FIPS code

levels(datdt$state) <- codetable$abbr[match(levels(datdt$state), codetable$codes)]
levels(datdf$state) <- codetable$abbr[match(levels(datdf$state), codetable$codes)]
#matches each state code in the factor variable to its corresponding abbreviation
```

Reconstructing state rates from county rates

`data.table` syntax: `dat[i, j, by]`

Now we are ready to reconstruct state rates from the county rates. We will need to sum up the total unemployment in each county by state, and `data.table` syntax makes it very easy to do this.

The `data.table` syntax `dat[i, j, by]` is very intuitive: `i` defines which rows are to be subsetted (can be defined using rows numbers or logical conditions), `j` defines which values are to be used as columns, and `by` defines which identifying variables are used to evaluate the expression in `j`.

For each part, we use `.()` to define a concatenation of variables, similar to how `c()` is used in base `R`.

```
statedatdt<-datdt[,.(unemployment=sum(unemployment, na.rm=TRUE),
                     total=sum(total, na.rm=TRUE)),
                  by=.(state,year,period)]
#creates a new data.table - by summing unemployment and total labor force
#Use the combination of state, year and period as grouping variable for summing
statedatdt[, c("rate", "employment")
           := list(round(100*unemployment/total, 1), (total-unemployment))]
#reconstructs unemployment rate and employed people using available info
head3(statedatdt)
```

```
##    state year period unemployment   total rate employment
## 1:    AL 1990    M01       130333 1879961  6.9    1749628
## 2:    AK 1990    M01        21882  258638  8.5     236756
## 3:    AZ 1990    M01        98394 1749597  5.6    1651203
```

data.frame syntax: melt and using the fun.aggregate argument in dcast

For `data.frame`, ideally we would want to do a similar "sum by" operation, but this is not available. Instead, we use a not-so-efficient alternative afforded by the `reshape2` package.

First we will use the opposite of `dcast` - the `melt` function to get the data back into long format. Then we will use `dcast` again, but this time without `fips` as an identifying variable. This causes all measurements with the same state to be grouped together in the same cell. Since a single cell only admits one measurement, we use `fun.aggregate = sum` to obtain that single measurement.

```
statedatdf <- melt(datdf, id.vars = c("year", "period", "state", "fips"),
                   measure.vars = c("rate", "unemployment", "employment", "total"),
                   variable.name = "measure", value.name = "value")
str(statedatdf)#check the state of the data.frame after melting
```

```
## 'data.frame':    4465684 obs. of  6 variables:
##  $ year   : Factor w/ 27 levels "1990","1991",..: 1 1 1 1 1 1 1 1 1 1 ...
##  $ period : Factor w/ 13 levels "M01","M02","M03",..: 1 1 1 1 1 1 1 1 1 1 ...
##  $ state  : Factor w/ 52 levels "AL","AK","AZ",..: 1 1 1 1 1 1 1 1 1 1 ...
##  $ fips   : Factor w/ 3222 levels "01001","01003",..: 1 2 3 4 5 6 7 8 9 10 ...
##  $ measure: Factor w/ 4 levels "rate","unemployment",..: 1 1 1 1 1 1 1 1 1 1 ...
##  $ value  : num  6.4 6.2 7.1 12.4 5.2 11.4 13.6 6.5 4.8 8.1 ...
```

```
statedatdf <- dcast(statedatdf, year+period+state~measure,
                    value.var="value", fun.aggregate = sum)
statedatdf$rate <- round(100*statedatdf$unemployment/statedatdf$total, 1)
str(statedatdf)#confirm this is in the format we need
```

```
## 'data.frame':    18044 obs. of  7 variables:
##  $ year        : Factor w/ 27 levels "1990","1991",..: 1 1 1 1 1 1 1 1 1 1 ...
##  $ period      : Factor w/ 13 levels "M01","M02","M03",..: 1 1 1 1 1 1 1 1 1 1 ...
##  $ state       : Factor w/ 52 levels "AL","AK","AZ",..: 1 2 3 4 5 6 7 8 9 10 ...
##  $ rate        : num  6.9 8.5 5.6 7.5 5.7 6.2 5.3 4.6 5.5 5.9 ...
##  $ unemployment: num  130333 21882 98394 83151 854161 ...
##  $ employment  : num  1749628 236756 1651203 1029087 14099741 ...
##  $ total       : num  1879961 258638 1749597 1112238 14953902 ...
```

As we saw, `data.table` syntax is easier to use.

Constructing federal unemployment rates

Now we want a table that list federal unemployment rates, within years as rows and months as columns. Using dcast this is fairly easy to do:

```
feddatdt<-datdt[,.(unemployment=sum(unemployment, na.rm=TRUE),
                   total=sum(total, na.rm=TRUE)),
                by=.(year,period)]
#sums unemployment across each county
feddatdt[, rate := round(100*unemployment/total, 1)]
#calculates federal unemployment
head3(feddatdt)
```

```
##    year period unemployment     total rate
## 1: 1990    M01      7568905 125355936  6.0
## 2: 1990    M02      7432850 125516011  5.9
## 3: 1990    M03      7008944 125943166  5.6
```

```
fedymdt<-dcast(feddatdt,year~period,value.var="rate")
#gets years in rows and months in columns
head3(fedymdt)
```

```
##    year M01 M02 M03 M04 M05 M06 M07 M08 M09 M10 M11 M12 M13
## 1: 1990 6.0 5.9 5.6 5.4 5.3 5.5 5.6 5.6 5.6 5.6 5.9 6.1 5.7
## 2: 1991 7.1 7.4 7.2 6.6 6.8 7.1 6.9 6.7 6.6 6.6 6.8 7.0 6.9
## 3: 1992 8.2 8.3 7.9 7.3 7.4 8.0 7.8 7.5 7.4 7.0 7.2 7.1 7.6
```

```
#for data.frame we use same trick as earlier
feddatdf<-melt(datdf, id.vars = c("year", "period", "state", "fips"),
               measure.vars = c("rate", "unemployment", "employment", "total"),
               variable.name = "measure", value.name = "value")
feddatdf<-dcast(feddatdf,year+period~measure,value.var="value",fun.aggregate=sum)
feddatdf$rate <- round(100*feddatdf$unemployment/feddatdf$total, 1)
head3(feddatdf)
```

```
##   year period rate unemployment employment     total
## 1 1990    M01  6.0      7568905  117787031 125355936
## 2 1990    M02  5.9      7432850  118083161 125516011
## 3 1990    M03  5.6      7008944  118934222 125943166
```

```
fedymdf<-dcast(feddatdf,year~period,value.var="rate")
head3(fedymdf)
```

```
##   year M01 M02 M03 M04 M05 M06 M07 M08 M09 M10 M11 M12 M13
## 1 1990 6.0 5.9 5.6 5.4 5.3 5.5 5.6 5.6 5.6 5.6 5.9 6.1 5.7
## 2 1991 7.1 7.4 7.2 6.6 6.8 7.1 6.9 6.7 6.6 6.6 6.8 7.0 6.9
## 3 1992 8.2 8.3 7.9 7.3 7.4 8.0 7.8 7.5 7.4 7.0 7.2 7.1 7.6
```

Constructing wide format datasets

The last thing we want is a dataset with the year-month combinations as the rows, and each county unemployment rate as the column. This is useful for those people who want to use the counties to estimate some sort of sample correlation matrix:

```
ymdatdt<-dcast(datdt,year+period~fips,value.var="rate")
ymdatdf <-dcast(datdf,year+period~fips,value.var="rate")
#gets counties on columns, and year+period as rows
```

Comparing total run time between `data.table` and `data.frame`

Finally, we will compare the total run time used to compare all of the operations done above. The code used is

```
ptm <- proc.time()
#insert all code here
proc.time()-ptm
```

Running time for data.table:

```
##
Read 78.4% of 4465684 rows
Read 4465684 rows and 5 (of 5) columns from 0.270 GB file in 00:00:03
```

```
## Warning: NAs introduced by coercion
```

```
## Warning in `[.data.table`(datdt, , `:=`(state, as.factor(substr(fips, start
## = 1, : Invalid .internal.selfref detected and fixed by taking a (shallow)
## copy of the data.table so that := can add this new column by reference.
## At an earlier point, this data.table has been copied by R (or been created
## manually using structure() or similar). Avoid key<-, names<- and attr<-
## which in R currently (and oddly) may copy the whole data.table. Use set*
## syntax instead to avoid copying: ?set, ?setnames and ?setattr. Also, in
## R<=v3.0.2, list(DT1,DT2) copied the entire DT1 and DT2 (R's list() used to
## copy named objects); please upgrade to R>v3.0.2 if that is biting. If this
## message doesn't help, please report to datatable-help so the root cause can
## be fixed.
```

```
##    user  system elapsed
##    8.03    0.59    8.78
```

Running time for data.frame:

```
##    user  system elapsed
##   34.09    1.89   36.46
```

Mapping the data: the `choroplethr` package

One last useful thing to have is the ability to map out the values of each county on a heat map. This is where the `choroplethr` package (Lamstein and Johnson 2016) is nifty:

```
factorasnumeric <- function(x){as.numeric(as.character(x))}
library(choroplethr)
```

```
## Loading required package: acs
## Loading required package: stringr
```

```
## Loading required package: plyr
```

```
## Loading required package: XML
```

```
##
## Attaching package: 'XML'
```

```
## The following object is masked from 'package:rvest':
##
##     xml
```

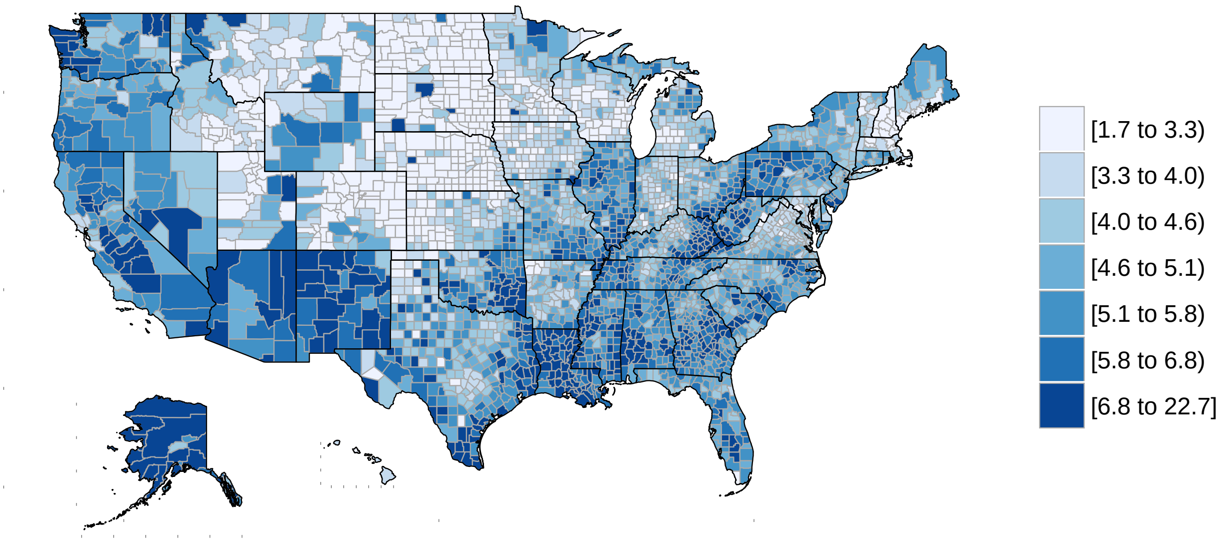
```
##
## Attaching package: 'acs'
```

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

```
mapdat<-datdt[(year=="2016"&period=="M09"),.(factorasnumeric(fips), rate)]
colnames(mapdat) <- c("region", "value")
#county_choropleth needs the first column to be named "region", and the second "value"
mapdat$region[mapdat$region==2158]<-2270
mapdat$region[mapdat$region==46102]<-46113
mapdat <- rbind(mapdat, data.frame(region = 51515, value = mapdat$value[mapdat$region==51019]))
county_choropleth(mapdat)
```

```
## Warning in super$initialize(map.df, user.df): Your data.frame contains
## the following regions which are not mappable: 72001, 72003, 72005, 72007,
## 72009, 72011, 72013, 72015, 72017, 72019, 72021, 72023, 72025, 72027,
## 72029, 72031, 72033, 72035, 72037, 72039, 72041, 72043, 72045, 72047,
## 72049, 72051, 72053, 72054, 72055, 72057, 72059, 72061, 72063, 72065,
## 72067, 72069, 72071, 72073, 72075, 72077, 72079, 72081, 72083, 72085,
## 72087, 72089, 72091, 72093, 72095, 72097, 72099, 72101, 72103, 72105,
## 72107, 72109, 72111, 72113, 72115, 72117, 72119, 72121, 72123, 72125,
## 72127, 72129, 72131, 72133, 72135, 72137, 72139, 72141, 72143, 72145,
## 72147, 72149, 72151, 72153
```

```
## Warning in self$bind(): The following regions were missing and are being
## set to NA: 15005
```


[1.7 to 3.3)
[3.3 to 4.0)
[4.0 to 4.6)
[4.6 to 5.1)
[5.1 to 5.8)
[5.8 to 6.8)
[6.8 to 22.7]

References

Dowle, Matt, and Arun Srinivasan. 2016. *Data.table: Extension of ‘data.frame‘*. https://CRAN.R-project.org/package=data.table.

Lamstein, Ari, and Brian P Johnson. 2016. *Choroplethr: Simplify the Creation of Choropleth Maps in R*. https://CRAN.R-project.org/package=choroplethr.

Wickham, Hadley. 2007. “Reshaping Data with the reshape Package.” *Journal of Statistical Software* 21 (12): 1–20. http://www.jstatsoft.org/v21/i12/.

———. 2016. *Rvest: Easily Harvest (Scrape) Web Pages*. https://CRAN.R-project.org/package=rvest.