

Data Preparation: From Raw to Revamped!

David Martin

2/26/2020

Before we get started

Set R Global Options

1. Go to Tools -> Global Options -> Code -> Make sure **Soft-Wrap R Source Files** is **checked**.
2. Go to Tools -> Global Options -> RMarkdown -> Make sure **Show Output Inline for all RMarkdown Documents** is **unchecked**.

Today's class

Today we are going to walk through how you go from disparate datasets to one unified dataset. In previous weeks we've worked with a dataset that was complete and clean, however, it did not start that way. This dataset was actually pulled together from three separate sources.

Albemarle County Office of Geographic Data Services

<http://www.albemarle.org/departments.asp?department=gds&relpage=3914Parcels>

Real Estate Information - Parcel Level Data

https://gisweb.albemarle.org/gisdata/CAMA/GIS_View_Redacted_ParcelInfo_TXT.zip

This file contains information about the parcel itself such as owner information, deed acreage value, and assessed value.

Real Estate Information - Card Level Data

Card Level Data refers to property information organized by particular residential dwellings or commercial units (e.g. building details and outbuilding information) on a given property. These tables can be linked to the Parcel Level Data table (**ParcelID** field) via the **TMP** field.

Card Level Data: https://gisweb.albemarle.org/gisdata/CAMA/GIS_CardLevelData_new_TXT.zip This file includes data such as year built, finished square footage, number of rooms, and condition.

Other characteristics: https://gisweb.albemarle.org/gisdata/CAMA/CityView_View_OtherParcelCharacteristics_TXT.zip This file contains other parcel information that is managed in our development tracking system (e.g. Zoning, School Districts, Jurisdictional Areas, etc.).

```
# install.packages("tidyverse")
# install.packages("lubridate")
library(tidyverse)
```

```
## -- Attaching packages ----- tidyverse 1.3.0
```

```
## v ggplot2 3.2.1      v purrr 0.3.3
## v tibble 2.1.3       v dplyr 0.8.4
## v tidyr 1.0.2        v stringr 1.4.0
## v readr 1.3.1        v forcats 0.4.0
```

```
## -- Conflicts ----- tidyverse_conflict_
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()     masks stats::lag()
```

```
library(lubridate)
```

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

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

```
ourGoal <- read_csv("data/albemarle_homes_2020.csv")
```

```
## Parsed with column specification:
## cols(
##   .default = col_double(),
##   tmp = col_character(),
##   usecode = col_character(),
##   condition = col_character(),
##   cooling = col_character(),
##   lastsaledate1 = col_character(),
##   esdistrict = col_character(),
##   msdistrict = col_character(),
##   hsdistrict = col_character(),
##   lastsaledate = col_character(),
##   condition2 = col_character()
## )
```

```
## See spec(...) for full column specifications.
```

Reading in our TXT Files

```
card_raw <- read_tsv("data/GIS_CardLevelData_new.txt")
```

```
## Parsed with column specification:
## cols(
##   .default = col_character(),
##   CardNum = col_double(),
##   VisionID = col_double(),
##   BID = col_double()
## )

## See spec(...) for full column specifications.

parcel_raw <- read_tsv("data/GIS_View_Redacted_ParcelInfo .txt")
```

```
## Parsed with column specification:
## cols(
##   .default = col_character(),
##   LotSize = col_double(),
##   LotSizeJan1 = col_double(),
##   LastSalePrice = col_double(),
##   Cards = col_double(),
##   UniqueField = col_double()
## )
## See spec(...) for full column specifications.
```

```
other_raw <- read_tsv("data/CityView_View_OtherParcelCharacteristics.txt")
```

```
## Parsed with column specification:
## cols(
##   .default = col_character(),
##   CensusBlockGroup = col_double(),
##   CensusTract = col_double(),
##   LandUsePrimaryStructuresNumber = col_double(),
##   LandUsePrimaryDwellingUnitsNumber = col_double(),
##   LanduseSecondaryStructuresNumber = col_double(),
##   LandUseSecondaryDwellingUnitsNumber = col_double(),
##   LanduseMinorStructuresNumber = col_double(),
##   LandUseMinorDwellingUnitsNumber = col_double()
## )
## See spec(...) for full column specifications.
```

Data Investigation

Before beginning making changes, we need to identify what we want in the end. We want to be able to look at the value of the house (parcel), the actual makeup of the house and land (card), and other pertinent information about the area in which the house resides (other). We are going to start by looking at the card dataset to see which variables we want to keep in regards to the house/land.

```
# Use the following functions to look for variables we want to keep or drop.
names(card_raw)
glimpse(card_raw)
```

There are a few things to note here. First, we want to make sure we keep the TMP variable since that is how we will join it to other datasets. Second, there is a lot of information here, perhaps more than we need. Finally, all the data was read in as character data, even things like number of stores and total rooms. We will want to change that.

The first and second tasks we can complete right now by using the select function.

```
# After looking through the variables, we pick out the ones that we want (Would also use a codebook, et
vars <- c("TMP", "CardNum", "YearBuilt", "YearRemodeled",
  "UseCode", "Condition", "NumStories", "FinSqFt", "Cooling", "FP_Open", "Bedroom", "FullBath", "HalfBa

# We can now use the vector we created above as input into our select function.
card <- card_raw %>%
  select(vars)
```

Using Mutate to change/add things to your dataset.

So, we've already culled our card dataset down from 35 to 14 variables. However, we still have some issues with all the data being characters and a few other issues. To make changes to your data, like adding a new variable/column or changing the variable type, we will be utilizing the mutate function.

However, we are simply going to make a small change at first by using mutate to add a simple column.

```
#Let's make sure that we know what dataset our data is coming from.
card <- card %>%
  mutate(source = "card")
```

Now that we have started on our card dataset, we are going to move over to the parcel and other datasets and go through similar motions.

```
glimpse(parcel_raw)
glimpse(other_raw)
```

Exercise

I have selected the variables below for you to keep in our two datasets.

```
parcel_vars <- c("ParcelID", "Owner", "LotSize", "PropName", "LandValue", "LandUseValue", "Improvements"
other_vars <- c("ParcelID", "ESDistrict", "MSDistrict", "HSDistrict" , "CensusTract")
```

I want you to:

1. Create two new datasets `parcel` and `other` by selecting the chosen variables from the raw datasets

```
parcel <- parcel_raw %>%
  select(parcel_vars)

other <- other_raw %>%
  select(other_vars)
```

Then:

2. Add variables to `parcel` and `other` naming the `source` of the data.

```
parcel <- parcel %>%  
  mutate(source = "parcel")  
  
other <- other %>%  
  mutate(source = "other")
```

3. There is no question 3...
-

Turning many into one (the power of the merge)

We now have 3 datasets with the variables that we want. We've gone from having over 110 variables to around 30 variables. Now, the goal is to take our 3 datasets and create 1 dataset that we will work on moving forward. To do this, we are going to use a series of merges using `join` functions. There are several types of `join` functions in R, all depending on your specific needs.

There are several types of joins (merges) that we can use. The one you use depends upon your specific needs.

`inner_join()`: return all rows from x where there are matching values in y, and all columns from x and y. If there are multiple matches between x and y, all combination of the matches are returned.

`left_join()`: return all rows from x, and all columns from x and y. Rows in x with no match in y will have NA values in the new columns. If there are multiple matches between x and y, all combinations of the matches are returned.

`right_join()`: return all rows from y, and all columns from x and y. Rows in y with no match in x will have NA values in the new columns. If there are multiple matches between x and y, all combinations of the matches are returned.

`full_join()`: return all rows and all columns from both x and y. Where there are not matching values, returns NA for the one missing.

However, before attempting to join our data, we need to investigate the variable we are going to use to uniquely identify observations in each of the datasets.

```
n_distinct(card$TMP) #Multiple cards are possible, so duplicates are ok.  
n_distinct(parcel$ParcelID) # Checks out  
n_distinct(other$ParcelID) # Almost Checks out  
  
# Quickly checking on those duplicates in other  
tmp <- other %>% group_by(ParcelID) %>%  
  summarize(dups = n()) %>% filter(dups > 1)  
other %>% filter(ParcelID %in% tmp$ParcelID)  
# They should not be trouble in our merge
```

We are going to start by combining our `parcel` and `other` datasets together. Since there, in theory, only one parcel in each of the datasets, we are going to use a one-to-one join where a Parcel in one dataset should only match up to one row in the other dataset.

```

#We are going to use a left join to join parcel and other together. If a match is not found, then obser
parcel_other <- parcel %>%
  left_join(other, by = "ParcelID")
#We now have 47856 observations and 17 columns

# View(parcel_other)
# Let's sort out the source.x, source.y fiasco. If two columns exist with the same name in two datasets

parcel_other <- parcel_other %>%
  select(-source.y) %>%
  select(everything(), "source" = "source.x")

```

We are now going to take the card dataset and merge it onto the combined parcel dataset. Since the card dataset can have multiple observations that belong to a single parcel, this will be considered a many-to-one join. We are going to use a full-join for this merge, as we want to keep all data, matching or not.

```

#We could rename TMP to parcelID, however, you can also handle the differences in primary key names as
homes <- card %>% full_join(parcel_other, by = c("TMP" = "ParcelID"))

#We now have 51,823 observations and 30 variables.
#You can check the result of the merge by looking at the source.x and source.y variables. If missing, t

homes %>%
  filter(is.na(source.x)) %>%
  tally() #Shortcut for summarize(n())
#We have 9124 observations that are only found in the parcel dataset
#Most of these missing end up being because of 0 improvements.

homes %>%
  filter(is.na(source.y)) %>%
  tally()
#We only have 1 observation that was in the card dataset and not in the parcel dataset.

#Let's do a final filter here and get rid of the source.x and source.y columns.
homes <- homes %>%
  filter(!is.na(source.y) & !is.na(source.x)) %>%
  select(-c(source.y, source.x))

#51823 observations. Still around 20000 observations away from our goal dataset. More to do!

```

Details, details, details.

Now the tedious work begins. This is where we have to go through and ensure that the data we have is in the shape and form that we want it to be in. One way to start is just to investigate the data as we did before.

```

names(homes)
glimpse(homes)

```

One thing we could do, if desired, is change the variable names to all lower case. This could, in the future, make it a bit easier to minimize mistakes when mis-capitalizing variable names.

```
#To change the case of variables names and/or values we can use the str_to_lower(), str_to_upper(), str_to_sentence(), str_to_title()
str_to_upper(names(homes))
names(homes) <- str_to_lower(names(homes))
names(homes)

str_to_sentence("HERE Is an example")
str_to_title("HERE Is an example")
```

We are close on the variables, but we still have some cleaning up to do in regards to the value of our variables. Here are a few things we have (make believe with me) identified as important for our data.

1. We only want residential homes records (not businesses, apartment complexes, etc.).
2. We want properties with individual households own and can accrue wealth or properties that can be rented to individual households

```
homes %>%
  count(usecode) %>%
  print(n = Inf)

#Now search through all the values and pick out the right one, I'll wait...
#or you can just use these values.
res <- c("Duplex", "Single Family", "Single Family-Rental")

#We can use the res vector much like we do with a select, however, this time it will be in a filter statement

#The %in% operator allows you to look for a series of values, instead of having to use multiple ORs.
homes <- homes %>%
  filter(usecode %in% res)

#We are not down to less than 34000 observations.
```

Much earlier, I mentioned about the issue with all of our variables being seen as character variables. This obviously is not ideal, so we need to use a special version of mutate to handle our mis-classified variables.

This will look quite similar to the use of %in% above. We create a vector of variables we want to change and then we use the mutate_at function to change them.

```
#We could go wild and change every variable to a number, but that may not look right.
homes %>%
  mutate_if(is.character, as.numeric)

#It is better to have a bit of control and use a specific list of variables.
numvar <- c("yearbuilt", "yearremodeled", "numstories", "finsqft", "bedroom", "fullbath", "halfbath", "basement")

homes <- homes %>%
  mutate_at(numvar, as.numeric)

#This function also works well for converting variables to factors. Factors provide discrete levels to a variable
facvar <- c("condition", "cooling", "esdistrict", "msdistrict", "hsdistrict", "censustract")

homes <- homes %>%
  mutate_at(facvar, as.factor)
```

```
#Finally, to make life easier on ourselves, we want to convert sale date to an actual object. This can
homes <- homes %>%
  mutate(lastsaledate = as.Date(lastsaledate1, "%m/%d/%Y"))
```

Into the Weeds

For today, the main purpose of this dataset is to look at the total value (`totalvalue`), the square footage (`finsqft`), and lot size (`lotsize`). Since these are the key variables for us to look at, we need to ensure that we have very reliable data for these variables.

```
# Let's look at the total value.
summary(homes$totalvalue)
# Some numbers jump out, like a 200 million dollar home , 37 missing values, and houses that 0 dollars

#We know we want to filter out NAs.
homes <- homes %>%
  filter(!is.na(totalvalue))

#The ones that are 0 may not be as obvious.
homes %>% # check 0s
  filter(totalvalue == 0) %>%
  select(usecode, finsqft, lotsize, landvalue, improvementsvalue)

#Let's get rid of those too
homes <- homes %>%
  filter(totalvalue > 0)
```

We can also do something similar to square footage and lot size.

```
# finsqft
summary(homes$finsqft) # 19 NAs, some 0s, and up to up to 34K sqft

homes %>% # check NAs (I want to see the full Owner/Name so save in tmp data frame)
  filter(is.na(finsqft)) %>%
  select(yearbuilt, finsqft, owner, lotsize, propname, improvementsvalue)

# some commercial, or condo-ish (remove below)
homes %>% # check 0s
  filter(finsqft == 0) %>%
  select(yearbuilt, finsqft, owner, lotsize, propname, improvementsvalue) %>%
  arrange(desc(improvementsvalue))

# outside of the first 3, these don't seem to have improvements valued
# highly enough to be a home (remove below)

# check the high end
homes %>%
  filter(finsqft > 5000) %>%
  select(yearbuilt, usecode, finsqft, totalrooms, owner, lotsize:totalvalue, cards) %>%
  arrange(desc(finsqft)) # limit this, e.g., < 10K
# some of these are not single family homes (some probably are -- Howie, Jaffray, etc.)
```



```

# remove rows with finsqft >= 10000, = 0, or missing
homes <- homes %>%
  filter(!is.na(finsqft) & finsqft > 0 & finsqft < 10000)

# lotsize
summary(homes$lotsize)

homes %>% # check 0s
  filter(lotsize == 0) %>%
  arrange(finsqft)
# almost certainly incorrect (given landvalue has positive values), ah well...

homes %>% # check high end
  filter(lotsize > 250) %>%
  arrange(desc(lotsize))
# may farms (also golf course, school, ashlaun, etc.)
# ... many with mutiple properties on one assessment

# remove records with 2 or more cards associated with parcel
homes <- homes %>%
  filter(cards < 2)

```

After running those filters, we now have the same number of observations in our datasets.

Fixing up the rest

This is quite the process, huh? Let's keep moving...

The first thing we are going to do is create an age variable, since we may want to bin things by age.

```

# yearbuilt
summary(homes$yearbuilt)

# more likely to use this as age than year, create age of home
homes <- homes %>%
  mutate(age = 2019 - yearbuilt)

summary(homes$age)

# There are over 952 missing values here, instead of dropping those observations or leaving them missing

# impute median value within census tract for missing
tract_age <- homes %>%
  group_by(censustract) %>%
  summarize(med_age = round(median(age, na.rm = TRUE)))

#We are going to join the med_age values back to our dataset.
homes <- left_join(homes, tract_age, by = "censustract")

#We can then use an if_else function to fill in those that were missing
homes <- homes %>%
  mutate(age = if_else(is.na(age), med_age, age))

```

```
summary(homes$age) #no more nas
```

The next step we are going to take is to relevel a factor variable. When we look at the condition of the house, the factor will automatically set the factor up in alphabetical order. This is fine at times, but sometimes you want the factor to be ordered in your own specific way.

```
# Looking at the condition of the house
homes %>%
  count(condition)

# re-order levels of factor by our own defined order
cond_levels <- c("Substandard", "Poor", "Fair", "Average", "Good", "Excellent", "Unknown", "NULL")

# We are going to create a new condition variable
homes <- homes %>%
  mutate(condition2 = fct_relevel(condition, cond_levels))

summary(homes$condition2)

# combine unknown and null into none and relevel
# This can be done using the handy fct_collapse function and then using the fct_relevel function and mo
homes <- homes %>%
  mutate(condition2 = fct_collapse(condition2,
                                     None = c("Unknown", "NULL")),
         condition2 = fct_relevel(condition2, "None", after = 0))
summary(homes$condition2)
```

The rest of the cleaning done on these datasets is shown below. These tasks tend to be repeats of things we have done earlier, but they will be good examples for you moving forward.

Run this code chunk:

```
# yearremodeled, numstories, cooling, fp_open, bedroom, fullbath,
# halfbath, totalrooms, landvalue, landusevalue, improvementsvalue,
# lastsaleprice, esdstrict, msdstrict, hsdistrict, censustract

# yearremodeled -> remodel indicator
summary(homes$yearremodeled) # NA not remodeled; not sure about 2 or 8
homes <- homes %>%
  mutate(remodel = if_else(!is.na(yearremodeled), 1, 0))

# numstories
table(homes$numstories) # realize I don't know what this means; was expecting 1, 2, 3, etc.. Let's drop
homes <- homes %>% select(-numstories)

# cooling
table(homes$cooling) # fix factor -- assume 00, M1, and NULL are no air
homes <- homes %>%
  mutate(cooling = fct_collapse(cooling,
                                "No Central Air" = c("00", "M1", "NULL")))
```

```

# fp_open (these are characters)
table(homes$fp_open) # make a binary indicator, 0 and Null are none
homes <- homes %>%
  mutate(fp = if_else(fp_open %in% c("0", "NULL"), 0, 1))

# bedroom, fullbath, halfbath, totalrooms
table(homes$bedroom) # 103 homes with no bedroom is a likely error
table(homes$fullbath) # 168 homes with no full bath is a likely error
table(homes$halfbath) # ok
table(homes$totalrooms) # 479 homes with no rooms is a likely error

# landvalue
summary(homes$landvalue) # no missing, some 0s
homes %>% filter(landvalue == 0) %>% count(lotsize) # 13 total, only one with 0 lotsize

# landusevalue
summary(homes$landusevalue) # no missing
homes <- homes %>% # create binary indicator for land use (land generates revenue)
  mutate(landuse = if_else(landusevalue > 0, 1, 0)) %>%
  select(-landusevalue) # remove variable
table(homes$landuse)

# improvementsvalue
summary(homes$improvementsvalue) # no missing, some 0s

# create a tmp file with ImprovementsValue == 0, arrange the file by finsqft
tmp <- homes %>%
  filter(improvementsvalue == 0) %>%
  arrange(finsqft)
# several new buildings with common values (e.g., condo/th dev?)

# lastsaleprice
summary(homes$lastsaleprice)
tmp <- homes %>%
  filter(lastsaleprice == 0) %>%
  arrange(lastsaledate)
# first ~ 700 records, last sale date is prior to year built

homes %>%
  mutate(datecheck = if_else(yearbuilt > as.integer(year(lastsaledate)), 1, 0)) %>%
  filter(datecheck == 1 | yearbuilt == 0) %>%
  tally()
# suggests yearbuilt (or lastsaledate) is wrong for at least 3472 records

#.....
# Clean up and save ----
# remove a few additional variables -- these were for examining the data
homes <- homes %>%
  select(-c(cardnum, fp_open, owner, propname, cards, med_age))

```

We are finally there. If you look at your homes object and compare it to the goal dataset, we should have

the same number of observations and variables. If you don't, don't fret, this is all just practice. Below, I am showing a way that you could compare two datasets to see just how well we did.

```
diff_list <- mapply(setdiff, homes, ourGoal)
names(diff_list[lengths(diff_list) != 0])

#Only differences are from conversions occurring in the csv file and issues with dates. Other than that
homes %>%
  select(tmp, lastsaledate, lastsaledate1) %>%
  glimpse()

ourGoal %>%
  select(tmp, lastsaledate, lastsaledate1) %>%
  glimpse()
```

Appendix

The following functions were not needed in this workshop, but they are ones that I find useful.

```
# The separate function allows you to separate out several pieces of information stored in a single cell
# For example you could turn 2/26/2020 into 3 columns for month, day and year.
# In the example below, we have a Grade column that has both the grade, plus some additional info. Maybe
head(card_raw$Grade)
card_raw %>%
  select(Grade) %>%
  separate(col = Grade, into = c("grade", "grade_info"), sep = ": ")
#Essentially, this function is going to split your variable everytime it sees the separator (in this case)
#If this is of interest to you, look into regular expressions because they can provide for you several
```

The final function I want to show is slightly out of date and has been replaced by the `pivot_wider`, `pivot_longer` functions. However, these functions are currently quite picky about the version of Rstudio you are running and work in quite a similar fashion.

The spread and gather functions allow you to take your data from wide to long or long to wide, reshaping your complete dataset. There are times where you want to have every observation represent a single thing, like one state, and have the columns represent a large amount of information, like average house price from 1992-2016. This would make the data quite wide. You also might want to break your dataset into a state-year format, where you would have many observations and very few columns. This would be a long dataset.

```
state <- rep(c("VA", "NC", "MD", "WV", "DC"), 5)
year <- c(rep(2012, 5), rep(2013, 5), rep(2014, 5), rep(2015, 5), rep(2016, 5))
value <- c(round(rnorm(n = (5*5), mean = 200000, sd = 50000), 0))

#This will be considered a long dataset
house_long <- as.data.frame(cbind(state, year, value))
house_long

#Let us now create a wide dataset using the spread function
house_wide <- spread(data = house_long, key = year, value = value)
house_wide

#We can take it back to long using the gather function
gather(data = house_wide, key = "year", value = "value", c(2:6))
```