Tutorial 2: Merging

Leah Brooks

January 24, 2022

Contents

Today's tutorial is about merging. Merging means combining more than one dataframe using a common variable.

We teach merging early on because you will likely need to merge data for almost any project. We devote an entire tutorial to merging because merging can frequently go wrong. When it does, your data become garbage, and all products that come from that data are also garbage.

This tutorial uses small example datasets to show you how merging works. It builds in examples of "merges gone wrong" to point out the type of problems for which you should be on the lookout. For your homework, you'll find some additional data and merge it in to a county-level dataset.

We end with a discussion of variable types in R, with an important note about factor variables. Factor variables are strange but important, and you need to be at least slightly familiar with them to use R successfully.

Next class we begin with graphs!

A. Load Data

As you did last week, create a R script for this class. Write all your commands in the R script (recall, a file with R commands ending in .R). You can run all of the program at once (code -> run region -> run all), or just selected lines.

For the purposes of this tutorial, you may wish to run selected lines. At the end, be sure that your entire script works – this will show whether your logic follows throughout the entire program.

We'll begin by loading two very small datasets, examining them, and merging them. These are student-level datasets, and here are the links for what we'll call `students1` and `students2`.

Like we did last class, we'll use `read.csv()` to open these files, and we'll use other commands to help us understand the file. This is a small file, and you can understand this file without these commands just by opening it in Excel. However, you will need to deal with files larger than you can see in Excel – which is why we learn these tools. We practice them on a small dataset so you can easily see what does and doesn't work.

Let's begin by loading `students1`, printing the data, and then seeing what variable names the data have.

```
# load the first student data
student1 <- read.csv("H:/pppa_data_viz/2019/tutorial_data/lecture02/2019-01-27_fake_student_data.csv")

# how many students?
student1
```

```
##   First_name last_name GWID gpa degree remaining_sem
## 1      elpis    josefa  G37 2.9    mpa             1
## 2    longina   nedelya  G12 3.9    mpp             4
## 3   richelle    bjoern  G08 3.4    mpp             1
## 4    mozghan      mara  G62 3.5    mpa             2
## 5       runa   marcelo  G14 3.8    mpp             3
```

```
dim(student1)
```

```
## [1] 5 6
```

```
# which variables?
names(student1)
```

```
## [1] "First_name"    "last_name"     "GWID"          "gpa"
## [5] "degree"        "remaining_sem"
```

We can see from the data print-out and from `dim()` that this dataset has five rows and six variables (look at last week's tutorial if you need a refresher on `dim()`). The command `names()` tells us the variable names in this dataset. Note that there are two ways you could identify a student: by name (first and last) and by GWID.

Let's now load the second students dataset, `students2`.

```
# load the second student data
student2 <- read.csv("H:/pppa_data_viz/2019/tutorial_data/lecture02/2019-01-27_fake_student_data_part2.

# how many students?
dim(student2)
```

```
## [1] 5 4
```

```
# what variables?
names(student2)
```

```
## [1] "First_name" "last_name"  "GWID"       "age"
```

This new dataset also has five rows (students), so it is possible that the datasets could match perfectly (by which I mean a one-to-one match). The `names()` command tells us that this dataset has three variables that we can see overlap with the variables in `students1`: `First_name`, `last_name` and `GWID`.

B. Merging

B.1. Basic Merge

Let's start by merging these two datasets by the `GWID` variable. "Merging by" a variable means matching a row in dataset A with a row in dataset B when both observations have the same value for the merging variable. In this class we'll use exact matches. Fancier data work can do things like "fuzzy matches."

Before we merge, however, recall that both datasets have, in addition to `GWID`, `First_name` and `last_name`. If we merge by `GWID`, then the new dataframe will have first and last name from `student1` and first and last name from `student2`. It is **very bad** data management practice to have duplicate variables in a dataset (in addition, R may also not let you have two variables with the same name in the same dataframe). Duplicate variables in the same dataframe leads to confusion (which one is right?) and is wasteful from a data storage point of view. So we will begin by keeping just `GWID` and `age` from students2 before merging. Below we use syntax introduced last class; if it seems unfamiliar, return to last class's tutorial on subsetting.

```
# make student2 w/o duplicate variables
student2.nd <- student2[,c("GWID","age")]
```

Now we want to merge `students1` with the newly created `student2.nd`. R's basic merge command has four key components that I show below in generic form:

```
# do the merge
new.dataframe <- merge(x = DATAFRAME1,
                       y = DATAFRAME2,
                       by = VARIABLE COMMON IN BOTH DATASETS,
                       all = TRUE/FALSE)
```

Use `x =` and `y =` to specify the input datasets. Use `by =` to specify the variable on which you'd like to merge, putting it in quotes. In this method, this variable name needs to exist in both datasets (we relax this requirement later in the tutorial).

Finally, `all` = lets you modify R's behavior when merging. If you omit this part, R will just keep observations in the first dataset. I strongly suggest that you keep all observations. If they don't merge properly (more below on what this means) you should generally understand why – so it's important to know whether the merge is successful or not.

So for our particular example, we specify the merge as below:

```
# do the merge
students <- merge(x = student1,
                  y = student2.nd,
                  by = "GWID",
                  all = TRUE)
```

How many observations do you expect this new dataframe to have? Five – the same number as in each input dataframe. How many variables? The six variables in `students1` plus one (`age`) from `students2`. You might object, saying that `students2` has two variables – and that's right, but one of them is a duplicate of one in `students1` (`GWID`).

More generally, when you're doing a merge, you should know how many observations you expect in the final product before you merge. You should then check to be sure that the total number of observations fits your expectations. Not doing this is data malpractice and can cause serious errors. Many times things that you think should work do not. It is better to verify that things work, rather than assume that they do.

Use the commands below to check on whether your merged dataframe has the five observations and seven variables:

```
# check on things in new dataframe
dim(students)
```

```
## [1] 5 7
```

```
names(students)
```

```
## [1] "GWID"          "First_name"    "last_name"     "gpa"
## [5] "degree"        "remaining_sem" "age"
```

```
str(students)
```

```
## 'data.frame':    5 obs. of  7 variables:
##  $ GWID         : chr  "G08" "G12" "G14" "G37" ...
##  $ First_name   : chr  "richelle" "longina" "runa" "elpis" ...
##  $ last_name    : chr  "bjoern" "nedelya" "marcelo" "josefa" ...
##  $ gpa          : num  3.4 3.9 3.8 2.9 3.5
##  $ degree       : chr  "mpp" "mpp" "mpp" "mpa" ...
##  $ remaining_sem: int  1 4 3 1 2
##  $ age          : int  40 25 24 22 25
```

We see that the new dataframe has 5 observations (good!), and seven variables (good!). If you compare `names(students1)` to `names(students)`, you see one additional variable: `age`. Also good.

I include the command `str()` so you have another tool for examining a dataframe. This tool tells you the number of observations in a dataframe, the variables, and some hint of the values of those variables.

B.2. Merging by two variables

Sometimes it is necessary to merge by two variables, rather than one. In our small dataframe, first name information is sufficient to identify a student. This is not generally true, however. (And frequently names are insufficient to uniquely identify a student, which is why schools use ID numbers!) But for purposes of example, we will do a merge by two variables.

When else might you want to merge by two variables? Think back to the county data from last class. County FIPS codes are not unique across states. For example, multiple states have county `001`. Thus to merge properly, we need to match by both state FIPS code (e.g., `11` for DC) and county FIPS code (`001` for DC). State FIPS plus county FIPS do uniquely identify counties.

To start this merge, let's first make a dataset that has first name, last name and age from `students2`.

```
# make student2 w/o gwid
student2.nd2 <- student2[,c("First_name","last_name","age")]
```

With this new dataframe, we do a merge on two variables. We denote them `c("var1","var2")`.

```
# do the merge
students.t2 <- merge(x=student1,
                     y=student2.nd2,
                     by=c("First_name","last_name"),
                     all = TRUE)
```

Let's check like before. Does it have 5 observations?

```
# check on size
dim(students.t2)
```

```
## [1] 6 7
```

No! What went wrong?

B.3. Figuring out problems in a merge

The best place to start when figuring out problems in a merge is to make a marker for each of your datasets. This way you know which observation comes from which dataframe. Having this variable allows you to pick out observations without a match. Many times, when you view the non-merged observations, you can diagnose the problem.

Here we create the marker variable in each dataset before we merge. Note that the variables always takes the value 1.

```
# make id variables for each dataset
student2.nd2$s2 <- 1
student1$s1 <- 1
```

Now, re-do the merge with these revised dataframes:

```
# re-do merge
students.t2 <- merge(x=student1,
                     y=student2.nd2,
                     by=c("First_name","last_name"),
                     all = TRUE)
dim(students.t2)
```

```
## [1] 6 9
```

```
students.t2
```

```
##   First_name last_name GWID gpa degree remaining_sem s1 age s2
## 1      elpis    josefa  G37 2.9    mpa             1  1  22  1
## 2    longina   nedelya  G12 3.9    mpp             4  1  25  1
## 3    mozghan      mara  G62 3.5    mpa             2  1  25  1
## 4   richelle    bjoern  G08 3.4    mpp             1  1  40  1
## 5       runa   marcelo  G14 3.8    mpp             3  1  NA NA
## 6       runa  marcelo  <NA>  NA   <NA>            NA NA  24  1
```

We still have the same problem of six observations (and we should, since we haven't yet done anything to fix it).

Now clean up the marker variables to make `<NA>` values zero. We do this using the (very useful) `ifelse` command. The syntax for this command is

```
output$var <- ifselse(test = IF CONDITION,
                      yes = [DO IF TRUE],
                      no = [OTHERWISE DO THIS])
```

The first part of this command is the test. You tell R to evaluate a statement. If the statement after `test =` is true, then R assigns to variable `var` in dataframe `output` the value following the `yes` option. If the statement after `test =` is false, then R assigns the variable `var` the value after the `no =` option.

In the `ifelse()` function below, we are going to use another R function that evaluates whether an observation is missing or not. The function `is.na()` returns `TRUE` or `FALSE` depending on whether the observation is missing or not. Here is a small example of this function. First we create a dataframe called `example`. The column (or variable) `v1` has a missing value; `v2` does not.

```
example <- data.frame(v1 = c(1,3, NA),
                      v2 = c(4,5, 6))
# this reporting if there are any variables in v1 missing
is.na(example$v1)
```

```
## [1] FALSE FALSE  TRUE
```

```
# this reporting if there are any variables in v2 missing
is.na(example$v2)
```

```
## [1] FALSE FALSE FALSE
```

```
# and now we can figure out how many
table(is.na(example$v1))
```

```
##
## FALSE  TRUE
##     2     1
```

```
table(is.na(example$v2))
```

```
##
## FALSE
##     3
```

The output of `is.na()` is only `TRUE` or `FALSE`, and there is only one false, since only on observation (row in the dataframe) is missing. The output of the `table()` command reports the number of `TRUE` and `FALSE` in each variable. Look back at Tutorial 1 if you would like a reminder about this function.

Now we apply the logic of this command to the students data. The first command below says that if `studentst2$s1` is missing, then make `studentst2$s1` 0; otherwise, leave `studentst2$s1` its original value. Note that the condition we evaluate in the `test =` portion of the function has a conditional equals sign (`==`).

```
# make marker variables zero
students.t2$s1 <- ifelse(test = is.na(students.t2$s1)==TRUE,
                         yes = 0,
                         no = students.t2$s1)
students.t2$s2 <- ifelse(test = is.na(students.t2$s2)==TRUE,
                         yes = 0,
                         no = students.t2$s2)
```

Now we'll use the newly created markers `s1` and `s2` to see what's going on. How many observations are both in the first dataset (`s1=1`) and in the second (`s2=1`)? There are two ways to do this, now that we've made our markers. First, you can also print just observations where the merge doesn't work – where `s1+s2 != 2`, using the same subsetting logic as before.

```
# check on it
students.t2[which(students.t2$s1 + students.t2$s2 != 2),]
```

```
##   First_name last_name GWID gpa degree remaining_sem s1 age s2
## 5       runa   marcelo  G14 3.8    mpp             3  1  NA  0
## 6       runa  marcelo  <NA>  NA   <NA>            NA  0  24  1
```

The printed out subset shows that the problem observations are Runa Marcelo. In this dataframe, where there are no missing data, we could also use missing values in any variable to pick out the problem. However, this is generally not the case, which is why the markers are so useful.

Alternatively, use the two-way `table` command to check. Above we used a one-way `table()` command, which does a one-variable frequency. Here we do a two variable frequency.

```
# check on it
table(students.t2$s1,students.t2$s2)
```

```
##
##     0 1
##   0 0 1
##   1 1 4
```

The (unhelpfully unlabled) top row of this table shows values for the first variable in the command, `students.t2$s1`. The first column shows the values of the second variable in the table command, `students.t2$s2`. No observations have `s1 = 0` and `s2 = 0` – this makes sense, since it is equivalent to saying that no observations come not from `students1` (1 if the observation comes from `students1`) and no observations come from `students2` (1 if the observation comes from `students2`).

The table also tells us that there are four observations where `s1=1` and `s2=1`. There is one observation where `s1 = 1` and `s2 = 0`, and one observation where `s1 = 0` and `s2 = 1`. This method is very good for larger dataframes. It can help you understand if you have a merging problem, and whether it is a big or little problem.

Why doesn't the Marcelo observation merge? There's a hint in the print out: one `marcelo` is longer than the other. We can check whether this is the case by creating new variables that are the length of the name (where this is the length in characters). The command `nchar()` counts the number of characters in a string. If the variable is a factor (as these names are; more on factors later), you need to tell R to treat it as a character variable, using `as.character()`.

```
# look at the length of the merging variables
students.t2$first.length <- nchar(as.character(students.t2$First_name))
students.t2$last.length <- nchar(as.character(students.t2$last_name))
students.t2
```

```
##   First_name last_name GWID gpa degree remaining_sem s1 age s2 first.length
## 1      elpis    josefa  G37 2.9    mpa             1  1  22  1            5
## 2    longina   nedelya  G12 3.9    mpp             4  1  25  1            7
## 3    mozghan      mara  G62 3.5    mpa             2  1  25  1            7
## 4   richelle    bjoern  G08 3.4    mpp             1  1  40  1            8
## 5       runa   marcelo  G14 3.8    mpp             3  1  NA  0            4
## 6       runa  marcelo  <NA>  NA   <NA>            NA  0  24  1            4
##   last.length
## 1           6
## 2           7
## 3           4
## 4           6
## 5           7
## 6           8
```

Looking at the output, we can see that in the fifth observation the last name has length 7; in the sixth observation, last name has length 8. It has a space after "marcelo"! You can fix this using a `ifelse()` command on the `student2` dataset for this particular observation. This is for your homework.

More generally, this is one of the many types of problems that can plague merges. Be vigilant when merging data! This section has given you the tools to figure out when a merge is not working as expected.

C. Many-to-1 merges

So far we have done 1:1 merges, where each observation in each dataset has one or zero corresponding (where corresponding means a match of the "by' ' merging variable) observations in the other dataset. Now we'll try a many to one match, where many observations in a dataset correspond to one observation in the merging dataset.

C.1. Merge in attributes by degree program

For this example, we'll merge in attributes of the degree program. Each degree program has some features (perhaps number of students, mean time to graduation) that you may want to add to the individual student records. Begin by downloading these data, and then load them and look at them.

```
# bring in degree program info
dp <- read.csv("H:/pppa_data_viz/2019/tutorial_data/lecture02/2019-01-27_program_data_1.csv")
dp
```

```
##   deg grad_rate
## 1 mpp      0.95
## 2 mpa      0.96
```

```
names(dp)
```

```
## [1] "deg"       "grad_rate"
```

Note that this degree program data has a different variable name for the degree program (`deg`) than the students dataset (`degree`). We want to merge by this variable and that's ok: we just let R know that the merging variable has a different name in dataframes x and y as below using `by.x` and `by.y`.

Before merging, how many observations do you expect these data to have?

```
# merge with student data
students.wd <- merge(x=students,
                     y=dp,
                     by.x="degree",
                     by.y="deg",
                     all = TRUE)
dim(students.wd)
```

```
## [1] 5 8
```

```
names(students.wd)
```

```
## [1] "degree"        "GWID"          "First_name"    "last_name"
## [5] "gpa"           "remaining_sem" "age"           "grad_rate"
```

After merging, we find that the dataset has the expected 5 observations, and that the new dataframe has a graduation rate variable. All good.

Helpfully, you can see from the output of `names()` that R does not keep `degree` and `deg`: one is enough! (Both would be redundant and confusing.)

C.2. Another many-to-one merge example

That merge went very smoothly. Here's a (more typical) example when things don't go as you'd hoped. Download this new program csv file, and load the data.

```
# bring in degree program info
dp2 <- read.csv("H:/pppa_data_viz/2019/tutorial_data/lecture02/2019-01-27_program_data_2.csv")
names(dp2)
```

```
## [1] "degree"    "grad_rate"
```

```
dp2
```

```
##   degree grad_rate
## 1    mpp      0.95
## 2    mpa      0.96
## 3   enrp      0.95
```

Again, how many observations do we expect after the merge?

Now here is the merge:

```r
# merge with student data
students.wd2 <- merge(x=students,
                      y=dp2,
                      by="degree",
                      all = TRUE)
dim(students.wd2)
```

```
## [1] 6 8
```

This output has one more observation than we expect. Why?

We look for the answer following merging logic. I know that the dataframe `students` has a value for the variable `first_name` in every row (this is the same thing as saying "for every observation"). Below I tell R to print any observation from the merged data that does not have a value for the variable `First_name`, using `is.na()`.

```r
students.wd2[which(is.na(students.wd2$First_name) == TRUE),]
```

```
##   degree GWID First_name last_name gpa remaining_sem age grad_rate
## 1   enrp <NA>       <NA>      <NA>  NA            NA  NA      0.95
```

I can get rid of this extra row – we have no ENRP students in the `students` dataframe – by using a subsetting command as we learned last class. In words, this command says "keep only observations from `students.wd2` where `GWID` is not missing. We are comforted that the final product has the five observations we expect.

```r
# keep only true students
students.wd2 <- students.wd2[which(is.na(students.wd2$GWID) == FALSE),]
dim(students.wd2)
```

```
## [1] 5 8
```

It is good practice to only ever drop observations from a merge when you know what you're dropping and why.

D. Using the merged data

The point of merging data is to do something with the newly created data.

Here are some examples of things you can do with this newly merged dataframe.

```r
# what is the average age in the class?
av.age <- mean(x = students.wd2$age,
               na.rm = TRUE)
av.age
```

```
## [1] 27.2
```

```r
# standard deviation of age?
std.age <- sd(x = students.wd2$age,
              na.rm = TRUE)
std.age
```

```
## [1] 7.259477
```

```r
# how many people by degree program?
table(students.wd2$degree)
```

```
##
## mpa mpp
##   2   3
```

E. Data types in R

Variables in R may be one of six types

1. `numeric`
2. `integer`
3. `character`
4. `logical`
5. `complex`
6. `raw`

We will use the first four in this class, and we cover them in this tutorial. We will also cover a special case of the integer variable type, called a factor. The remaining two variable types (raw and complex) are likely not going to be of use to you (or me) in the foreseeable future.

E.1. Numeric variable

A numeric variable is one where all the observations of the variable are either numbers of missing (`NA`). For example, `v1` below is a numeric variable, but `v2` is not. You can evaluate whether a variable is numeric by using `is.numeric()`, which returns `TRUE` if numeric and `FALSE` if not.

```r
e1 <- data.frame(v1 = c(1.2, 2.3, 3.4),
                 v2 = c(4.5, "c", 5.6))
is.numeric(e1$v1)
```

```
## [1] TRUE
```

```r
is.numeric(e1$v2)
```

```
## [1] FALSE
```

You can also take a look at the structure of the dataframe using `str()` to evaluate variable types.

```r
str(e1)
```

```
## 'data.frame':    3 obs. of  2 variables:
##  $ v1: num  1.2 2.3 3.4
##  $ v2: chr  "4.5" "c" "5.6"
```

E.2. Integer variable

An integer variable is a special case of a numeric variable – it is a numeric variable where each value is an integer. Because integers do not have decimals and therefore take up less memory, R prefers to store numeric variables as integers if possible.

E.3. Character variable

A character variable is one that has at least one non-numeric value in any observation. In other words, if you have 1000 observations of variable `v1`, and one of them has a letter or symbol, the entire variable will be non-numeric. For example, the variables `students$first_name` and `students$last_name` are character variables.

E.4. Logical variable

A logical variable takes only the values `TRUE` and `FALSE`. You can enter these variables yourself, or create them as an output from other variables, as we did with `is.na()` above. The `is.na()` function (and many others) returns only the two logical values.

E.5. Factor variable: special case of integer

Factor variables are a special case of the integer variable. In short, R creates factor variables for character variables that take on a limited number of values. For example, imagine a variable with a limited number of string outcomes, such as "good", "bad", and "ugly". R assigns a number to each of the string outcomes, so, for example, `1 = "bad"`, `2 = "good"`, and `3 = "ugly"` (by default, R creates factors in alphabetical order). R stores the numbers (not the strings) and also stores the numbers-to-strings mapping. This can save a lot of memory, especially when strings are long.

Factor variables are tricky, because they sometimes look like character variables. For example, if you ask R to make a table of a factor variable, R will display the character values, not the integer ones. For example, our `students.wd2` dataframe has some factor variables. We can get a sense of them by using the `str()` command. See what R outputs about 'degree' from the table command – it looks like a string, not the factor that it actually is.

```
# look at the variables here
str(students.wd2)
```

```
## 'data.frame':    5 obs. of  8 variables:
##  $ degree       : chr  "mpa" "mpa" "mpp" "mpp" ...
##  $ GWID         : chr  "G37" "G62" "G08" "G12" ...
##  $ First_name   : chr  "elpis" "mozghan" "richelle" "longina" ...
##  $ last_name    : chr  "josefa" "mara" "bjoern" "nedelya" ...
##  $ gpa          : num  2.9 3.5 3.4 3.9 3.8
##  $ remaining_sem: int  1 2 1 4 3
##  $ age          : int  22 25 40 25 24
##  $ grad_rate    : num  0.96 0.96 0.95 0.95 0.95
```

```
table(students.wd2$degree)
```

```
##
## mpa mpp
##   2   3
```

Apart from saving memory, why does R have factor variables? The UCLA website writes that "There are a number of advantages to converting categorical variables to factor variables. Perhaps the most important advantage is that they can be used in statistical modeling where they will be implemented correctly, i.e., they will then be assigned the correct number of degrees of freedom. Factor variables are also very useful in many different types of graphics.' '

As the people at UCLA said, and as we will see in future tutorials, it is helpful to have factors when you're looking to tabulate categorical data. But remember that although factors sort of look like numbers when you do `str()`, they are not numbers and can't be treated as such:

```
# can you add factors?
students.wd2$odd <- students.wd2$degree + students.wd2$GWID
str(students.wd2)
```

When we run this code, R gives an error message:

```
Error in students.wd2$degree + students.wd2$GWID :
  non-numeric argument to binary operator
```

We will return to factor variables when we make graphs, particularly focusing on factor variables' key role in labeling.

E.6. Why all these different types?

Types matter because functions apply only to certain variable types. For example, you cannot add strings. Similarly, you can't use `nchar()` to count the number of characters in a numeric variable.

F. PS 2: Try it yourself with bigger data

1. Use R programming commands to fix the one problematic observation in the `student2` dataframe and make it merge properly (by first and last name) with `student1`.
2. Find a dataset with county information and merge it either with the data we used last class, or with a larger county-level dataset for all counties in the US, 1910 to 2010. This second dataset is here, and the variable definitions are here.

It is sufficient for your new dataset to have just one year of information, or information for just one state or one year.

The final dataset could be at the county level, or at some other unit of observation. (For example, if you found a dataset about power plants, which had one observation per power plant with a county ID, you would end up with a power plant-level dataset.)

Make sure you explain the following:

- what your new data describe
- the source of your additional data
- how many observations the new data have
- how many observations you expect from the merge and why; if this seems off, fix it and explain what you did

3. Make three relevant summary statistics of your choice for these new merged data. What these output are depends on what you're merging in.

As with the first problem set, you need to turn in

- your R script
- your R console file output (for just the clean R script)
- and some text that explains answers to questions
- in one pdf
- to the google drive folder you created last class