Data Management II – Walkthough 2: Merging Data

Alix Ziff, Gaea Morales, Zachary Johnson, Marie Zaragoza, Natalie Song, and Ben Graham.

January 2023

Revised Walkthrough Data Management 2. Part 2: Merging Data

This walkthrough will cover merging datasets together in `R` using the `dplyr` and `tidyr` packages, both part of the `tidyverse`. We will work with data drawn from two sources often used in international relations research: *Correlates of War* and *World Development Indicators*. The two files you need are cow_cleaned.csv & wdi_milex_April2021.rds

Recall that in a "tidy" dataframe:

1). Each row is a unique observation (e.g., a country-year like the U.S. in 1985 or Zimbabwe in 2019)

2). Each column is a variable with information about the observations (e.g., information about the country-years)

3). There is only one type of observation in each dataframe (i.e. we don't have information about country-years and country-days in the same dataframe)

Before you get started, write a detailed header in your R script and save your R script. Then set your working directory to the Training Data folder.

Let's set our working directory and load our packages.

```r
#setting the working directory will differ based on where the training data are saved
library(tidyverse)
```

We're going to read in data from the World Development Indicators and name the data object *wdi*. Take a look at the data summary. Do the same for the Correlates of War data but name the object *cow*.

```r
getwd()
cow <- read.csv("cow_cleaned.csv")
wdi <- readRDS("wdi_milex_April2021.rds")

summary(wdi$year)
summary(cow$year)
```

Merging Data

Understanding rbind() & cbind() issues

In principle, we can merge data using the cbind() and rbind() functions that we are already familiar with.

```r
vec1 <- c(1, 2, 3) #creating an object with numerical values 1-3
vec2 <- c("a", "a", "b") #creating an object with character string a,a,b

cbind(vec1, vec2) #putting these two vectors together as variables into a dataframe

rbind(vec1, vec2) #putting these two vectors together as observations into a dataframe
```

Helpful Hint: Remember that rows are observations and columns are variables!

However, this requires the data to align in its dimensions. In addition, for cbind() the ordering of the information within rows has to be the exact same in each dataframe to achieve correct merging; for rbind() the ordering of the information within columns has to be the same in both dataframes to achieve the right output.

This is pretty intuitive, but let's consider what happens if the ordering of the information within rows isn't the same in each dataframe.

In dataframe 1, the information within each row is as follows: the first column is country name, the second is year, and the third is a measure of GDP. However in dataframe 2, the information is in a different order: the first column is country name, but the second is a measure of GDP, and the third is year. Therefore, in trying to bind these together, we would be mismatching information.

And what happens if one dataframe has more observations or more variables than the other?

```
dat1 <- cbind(vec1, vec2)
vec3 <- c(T, F, T, T) #adding another character string vector
cbind(dat1, vec3) #adding in vector 3 (with 4 values not 3) to the dat1 data frame
```

```
## Warning in cbind(dat1, vec3): number of rows of result is not a multiple of
## vector length (arg 2)
```

As we see, information gets dropped. The fourth observation in vector 3 gets omitted. The same things happens if we have the wrong number of columns for a row bind.

```
obs <- c("c", 4)
rbind(dat1, obs)
```

Sheesh, this world is bad enough to make a person buy STATA…

tidyr saves the day!

There will obviously be times in which we want to merge two datasets that don't perfectly align. Instead, we are going to use the `full_join()` function in tidyr to merge. `full_join()`, along with its companions, `inner_join()`, `left_join()`, and `right_join()` merge two dataframes together, based on one or more variables that:

1). Are present in both dataframes you want to merge together.

2). Uniquely identify each observation in each dataframe.

These two conditions intuitively make sense, because we need to make sure that R can find some way to match each observation (condition 1) and match them correctly (condition 2).

The trick in working with these join commands is to select the proper "by" variable(s). These are the variable(s) that R will use to match by, hence the name "by" variable(s).

To practice thinking about choosing the correct "by" variables, try to figure out what happens to our data when we do the following:

```
wrong <- full_join(wdi, cow, by = "ccode") #merging wdi and cow by ccode

alsowrong <- full_join(wdi, cow) #let R try to figure out what to merge by
```

```
## Joining, by = c("ccode", "year", "milex")
```

If you thought that we have way too many observations and duplicates, you would be right!

One way to think about the "by" variable is to determine which columns both dataframes have in common (i.e., columns with the same name) and for which we only want one version in the final merged dataset.

In the “wrong” dataframe, we are joining observations (i.e., rows) on only one of the columns that the two data frames have in common (the ‘ccode’). So R links observations from the two dataframes by matching ‘ccode’, keeps only one ‘copy’ of the ccode column, and then keeps all the data from the year and milex columns from both dataframes. R identifies which column came from where with .x and .y suffixes. The variables ending in .x are from the dataframe on the ‘left’ (listed first), and the ones ending in .y are from the dataframe on the ‘right’ (listed second).

Notice also that this results in 513,012 observations. This is because full_join will keep all rows in both tables even when there is no matching ccode, and generate duplicate rows in its attempt to match observations across both dataframes with the same ccode.

Now with the “alsowrong” dataframe, we didn’t specify a “by” variable at all. R then “guesses” which variables to merge on, in this case, all the variables both dataframes have in common: ccode, year, and milex. This is also wrong because it is highly unlikely that we have identical or matching numerical milex estimates for both dataframes, so we end up with multiple observations for matching countries and years. We want to keep the milex estimates from the two dataframes separate.

Now let’s `full_join()` properly!

```r
# We are merging wdi & cow by ccode AND year, because together they uniquely identify
# observations. Before, the lack of the "by" variable "year" meant that R did not have
# enough information to match each unique value.

newdata <- full_join(wdi, cow, by = c("ccode", "year"))%>%
  relocate(ccode, year) #reordering the variables at the end to make it pretty


nrow(wdi) #double checking the number of rows in WDI
nrow(cow) #double checking the number of rows in COW
nrow(newdata) #double checking the number of rows in our new dataset
```

Using the full_join and specifying ‘ccode’ and ‘year’ as our “by” variables is the correct approach in this case because we want to match different data points from different dataframes by country and year. Put differently, we end up with only one row for each unique country-year, where variables for that country-year are listed on the same row, but in different columns.

When working with panel (country-year) data, we can generally assume that we will always want to join on a country identifier (this can be a name, or a version of a code), and the year. However, the approach is still largely dependent on which variables you have from each dataframe to be merged, and which you want to keep.

How many rows do we have here and why?

So the merge worked OK and we have the correct number of rows. However, things aren’t as good as they could be. Both the wdi data and the cow data have military expenditures measures called milex. Now we have a milex.x variable and a milex.y variable. Which is fine, but we have to remember the the .x version came from wdi, which was listed first in our join function and the .y version came from cow, which was listed second in our join. I am going to forget that!

Before we merge datasets together, it is usually better if we name the variables in such a way that we can tell where they came from later. In this case, we should rename the variables to something like milex_wdi and milex_cow. When you prep new data for use in the SPEC lab, we always use _[SUFFIX] to identify what data source each variable comes from.

```r
cow2 <- cow %>%
  rename(milex_cow = milex)
```

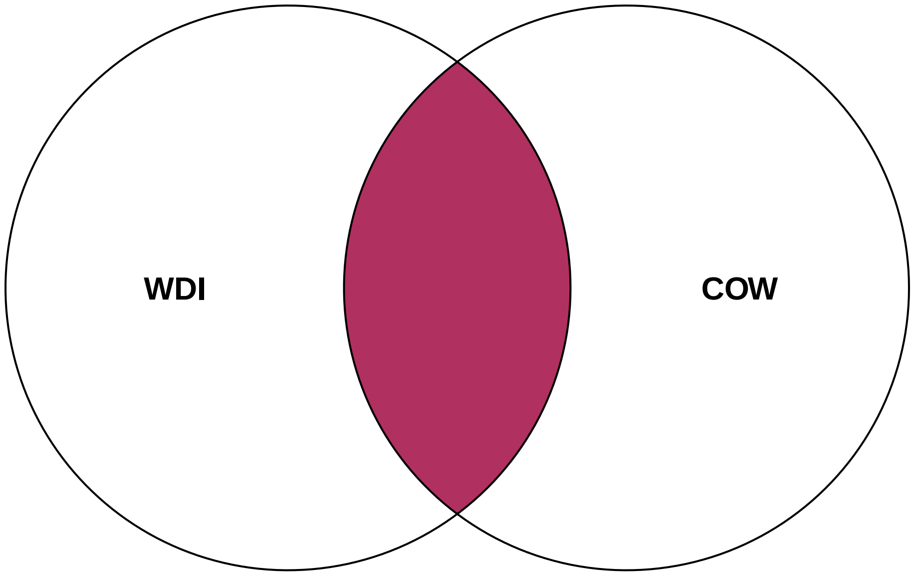
```
wdi2 <- wdi %>%
  rename(milex_wdi = milex)
```

Let's use the function `inner_join()` to only keep the observations that are present in both datasets. You will see this leaves us with fewer observations than 'full_join()'

```
innerData <- inner_join(wdi2, cow2, by = c("ccode", "year")) %>%
  arrange(ccode, year)

head(innerData)
nrow(innerData)
nrow(newdata)
```

This Venn diagram displays which of the information we kept.



Helpful Hint: Remember that if we have a country, for example, that is included in one dataset but not the other it will be dropped using the `inner_join` function. This is because inner_join only includes values that it finds a match to in the other dataframe.

A Conceptual Note: One-to-one vs. Many-to-one vs. One-to-many Merging

As you become more exposed to the range of data out there, you will realize that datasets come in all forms and sizes, and contain a myriad of variables of which we many only want a few. Part of the merging process to produce a dataset that suits the needs of your analysis will inevitably require different forms of merging.

The demonstration above is an example of a one-to-one merge. A one-to-one merge means adding more variables where observations or rows represent the same thing in both datasets, such as the country-year identifiers. Referring directly to the previous example, we match each country-year's GDP data from one dataset, with country-year military expenditure data from another dataset.

There may be situations where data are not neatly identified by country and year. For example, you might encounter data that are coded at the regional level, or data that are coded at the sub-national (e.g., urban unit) level. In the case of adding variables from regional level data, you are coding a variable or variables for which one, broader unit (regions) will match up with many smaller units (countries). The resulting dataframe would have the same number of observations, and new column(s) that have identical values for multiple rows (i.e., where countries are from the same region). Another possibility is where we have only one value for each country that does not tend to change over time, also known as time-invariant values, such as distance between countries, or territorial size. This, along with regional data, are examples of a one-to-many merge.

In the case of adding data coded at the urban level, you are coding a variable or variables for which many, smaller units (cities or metropolitan areas) will match up with a single, broader unit (countries). The resulting dataframe would have a greater number of observations, as each country-year will end up with separate rows for each city coded from the new dataset. For existing variables in the primary dataset that are coded at the country and year level, R automatically populates with identical values across rows for each given country-year unit.

It is important to note that both one-to-many and many-to-one merges operate in the exact same way as a one-to-one merge. You will still need to have variables in common across datasets in order to properly merge them, and to make sure that you can identify the appropriate "by" variables. In this case, both the regional and the urban datasets will ideally have both country and year variables. Data for time invariant variables must have at least the geographical areas specified.

Under what conditions would we want to use inner_join()?

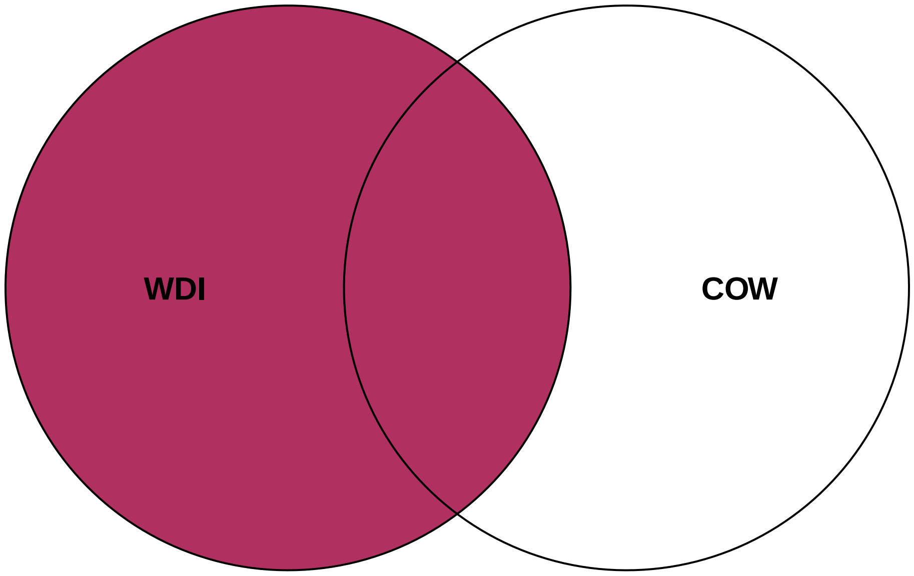
We would want to use inner_join() when we only want to see the observations that overlap between the two. In these data, this will only keep observations that have "ccode" and "year" values in both the wdi and cow data.

Now, we'll use the function `left_join()` to keep only the observations in the first dataset listed in the join command, in this case, wdi.

```r
leftData <- left_join(wdi2, cow2, by = c("ccode", "year"))

nrow(wdi) == nrow(leftData)
```

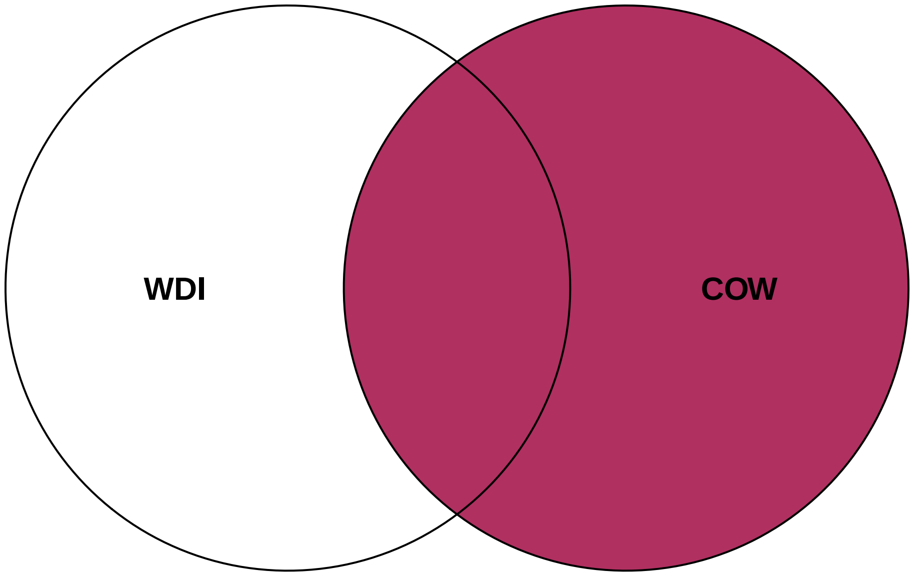
This Venn diagram displays which of the information we kept.



Under what conditions would we want to use left_join()?

We would want to use left_join() in the event that you only want the entire universe of values in the left dataset that are also represented in the right dataset. In these data, that means that R searched for all observations with the same "ccode" and "year" value in the wdi data that were also present in the cow data.

We could perform the same analysis on the cow data by using `right_join`. See Venn diagram below.



Final Thoughts

The reason we were able to merge the cow and wdi datasets in their current form is that they were both already tidy, and they both already had the same unique identifiers for their observations. In each dataset, each observation is a country-year, and those unique observations are identified by the same variables, *cccode* and *year*.

But data don't always come to us tidy. There is a LOT of work to clean these data to prepare them to be merged together. In fact, Data Management IIA is all about how we do that in the lab, working with Gleditsch-Ward numbers (gwno) as our numeric country codes.

One key step in making tidy data is to make sure that each row in our dataframe represents a single observation and each column represents a variable. To do that, we often need to reshape datasets. To learn more about reshaping datasets, refer back to Part 1 of this walkthrough.