APPENDIX: {peacesciencer}: An R Package for Quantitative Peace Science Research

Contents

A Section with a Citation to Get Started

Miller (2017)

Vignette: A Discussion of Various Joins in {peacesciencer}

```
library(tidyverse)
library(lubridate)
library(peacesciencer)
```

Users who may wish to improve their own data management skills in R by looking how {peacesciencer} functions are written will see that basic foundation of {peacesciencer}'s functions are so-called "join" functions. The "join" functions themselves come in {dplyr}, a critical dependency for {peacesciencer} and the effective engine of {tidyverse} (which I suggest for a basic workflow tool, and which the user may already be using). Users who have absolutely no idea what these functions are welcome to find more thorough texts about these different types of joins. Their functionality and terminology have a clear basis in SQL, a relational database management system that first appeared in 1974 for data management and data manipulation. My goal here is not to offer a crash course on all these potential "join" functions, though helpful visual primers are available in R and SQL. Instead, I will offer the basic principles these visual primers are communicating as they apply to {peacesciencer}.

Left (Outer) Join

The first type of join is the most important type of join function in {peacesciencer}. Indeed, almost every function in this package that deals with adding variables to a type of data created in {peacesciencer} includes it. This is the "left join" (`left_join()` in {dplyr}), alternatively known as the "outer join" or "left outer join" in the SQL context, and is a type of "mutating join" in the {tidyverse} context. In plain English, the `left_join()` assumes two data objects—a "left" object (x) and a "right" object (y)—and returns all rows from the left object (x) with matching information in the right object (y) by a set of common matching keys (or columns in both x and y).

Here is a simple example of how this works in the {peacesciencer} context. Assume a simple state-year data set of the United States (`ccode: 2`), Canada (`ccode: 20`), and the United Kingdom (`ccode: 200`) for all years from 2016 to 2020. Recreating this simple kind of data is no problem in R and will serve as our "left object" (x) for this simple example.

```
tibble(ccode = c(2, 20, 200)) %>%
  # rowwise() is a great trick for nesting sequences in tibbles
  # This parlor trick, for example, generates state-year data out of raw state
  # data in create_stateyears()
  rowwise() %>%
  # create a sequence as a nested column
  mutate(year = list(seq(2016, 2020))) %>%
  # unnest the column
  unnest(year) -> x
```

```
x
#> # A tibble: 15 x 2
#>    ccode  year
#>    <dbl> <int>
#>  1     2  2016
#>  2     2  2017
#>  3     2  2018
#>  4     2  2019
#>  5     2  2020
#>  6    20  2016
#>  7    20  2017
#>  8    20  2018
#>  9    20  2019
#> 10    20  2020
#> 11   200  2016
#> 12   200  2017
#> 13   200  2018
#> 14   200  2019
#> 15   200  2020
```

Let's assume we're building toward the kind of state-year analysis I describe in the manuscript accompanying this package. For example, the canonical civil conflict analysis by Fearon and Laitin (2003) has an outcome that varies by year, but several independent variables that are time-invariant and serve as variables for making state-to-state comparisons in their model of civil war onset (e.g ethnic fractionalization, religious fractionalization, terrain ruggedness). In a similar manner, we have a basic ranking of the United States, Canada, and the United Kingdom in our case. Minimally, the United States scores "low", Canada scores "medium", and the United Kingdom scores "high" on some metric. There is no variation by time in this simple example.

```
tibble(ccode = c(2, 20, 200),
       ranking = c("low", "medium", "high")) -> y

y
#> # A tibble: 3 x 2
#>   ccode ranking
#>   <dbl> <chr>
#> 1     2 low
#> 2    20 medium
#> 3   200 high
```

This is the "right object" (y) that we want to add to the "left object" that serves as our main data frame. Notice that `x` has no variable for the ranking information we want. It does, however, have matching observations for the state identifiers corresponding with the Correlates of War state codes for the U.S., Canada, and the United Kingdom. The left join (as `left_join()`) merges `y` into `x`, returning all rows of `x` with matching information in `y` based on columns they share in common (here: `ccode`).

```
# alternatively, as I tend to do it: x %>% left_join(., y)
left_join(x, y)
#> # A tibble: 15 x 3
#>    ccode  year ranking
#>    <dbl> <int> <chr>
#>  1     2  2016 low
#>  2     2  2017 low
#>  3     2  2018 low
#>  4     2  2019 low
#>  5     2  2020 low
#>  6    20  2016 medium
#>  7    20  2017 medium
#>  8    20  2018 medium
#>  9    20  2019 medium
#> 10    20  2020 medium
#> 11   200  2016 high
#> 12   200  2017 high
#> 13   200  2018 high
#> 14   200  2019 high
#> 15   200  2020 high
```

This is obviously a very simple example, but it scales well even if there is some additional complexity. For example, let's assume we added a simple five-year panel of Australia (`ccode: 900`) to the "left object" (x). However, we have no corresponding information about Australia in the "right object" (y). Here is what the left join would produce under these circumstances.

```
tibble(ccode = 900,
       year = c(2016:2020)) %>%
  bind_rows(x, .) -> x

x
#> # A tibble: 20 x 2
#>    ccode  year
#>    <dbl> <int>
#>  1     2  2016
#>  2     2  2017
#>  3     2  2018
#>  4     2  2019
#>  5     2  2020
#>  6    20  2016
#>  7    20  2017
#>  8    20  2018
#>  9    20  2019
#> 10    20  2020
#> 11   200  2016
#> 12   200  2017
#> 13   200  2018
#> 14   200  2019
#> 15   200  2020
#> 16   900  2016
#> 17   900  2017
#> 18   900  2018
#> 19   900  2019
#> 20   900  2020

left_join(x, y)
#> # A tibble: 20 x 3
#>    ccode  year ranking
#>    <dbl> <int> <chr>
#>  1     2  2016 low
#>  2     2  2017 low
#>  3     2  2018 low
#>  4     2  2019 low
#>  5     2  2020 low
#>  6    20  2016 medium
#>  7    20  2017 medium
#>  8    20  2018 medium
#>  9    20  2019 medium
#> 10    20  2020 medium
#> 11   200  2016 high
#> 12   200  2017 high
#> 13   200  2018 high
#> 14   200  2019 high
#> 15   200  2020 high
#> 16   900  2016 <NA>
#> 17   900  2017 <NA>
#> 18   900  2018 <NA>
#> 19   900  2019 <NA>
#> 20   900  2020 <NA>
```

Because have no ranking for Australia in this simple example, the left join returns NAs for Australia. The original dimensions of `x` under these conditions is unaffected.

What would happen if we had an observation in `y` that has no corresponding match in `x`? For example, let's assume our ranking data also included a ranking for Denmark (`ccode: 390`), though Denmark does not appear in `x`. Here is what would happen under these circumstances.

```
tibble(ccode = 390,
       ranking = "high") %>%
  bind_rows(y, .) -> y

y
```

```
#> # A tibble: 4 x 2
#>   ccode ranking
#>   <dbl> <chr>
#> 1     2 low
#> 2    20 medium
#> 3   200 high
#> 4   390 high

left_join(x, y)
#> # A tibble: 20 x 3
#>    ccode  year ranking
#>    <dbl> <int> <chr>
#>  1     2  2016 low
#>  2     2  2017 low
#>  3     2  2018 low
#>  4     2  2019 low
#>  5     2  2020 low
#>  6    20  2016 medium
#>  7    20  2017 medium
#>  8    20  2018 medium
#>  9    20  2019 medium
#> 10    20  2020 medium
#> 11   200  2016 high
#> 12   200  2017 high
#> 13   200  2018 high
#> 14   200  2019 high
#> 15   200  2020 high
#> 16   900  2016 <NA>
#> 17   900  2017 <NA>
#> 18   900  2018 <NA>
#> 19   900  2019 <NA>
#> 20   900  2020 <NA>
```

Notice the output of this left join is identical to the output above. Australia is in x, but not in y. Thus, the rows for Australia are returned but the absence of ranking information for Australia in y means the variable is NA for Australia after the merge. Denmark is in y, but not x. Because the left join returns all rows in x with matching information in y, the absence of observations for Denmark in x means there is nowhere for the ranking information to go in the merge. Thus, Denmark's ranking is effectively ignored.

Why the Left Join, in Particular?

An interested user may ask what's so special about this kind of join that it appears everywhere in {peacesciencer}? One is a matter of taste. I could just as well be doing this explainer piece by reference to the "right join", the mirror join to "left join." The right join in {dplyr}'s right_join(x,y) returns all records from y with matching rows in x by common columns. However, I tend to think "left-handed" when it comes to data management and instruct my students to do the same when I introduce them to data transformation in R. I like the intuition, especially in the pipe-based workflow, to start with a master data object at the top of the pipe and keep it "left" as I add information to it. This approach to data transformation and it recurs in `{peacesciencer}` as a result.

Beyond that matter of taste, the left join is everywhere in `{peacesciencer}` because the project endeavors hard to recreate the appropriate universe of cases of interest to the user and allow the user to add stuff to it as they see fit. `create_stateyears()` will create the entire universe of state-years from 1816 to the present for a state-year analysis. `create_dyadyears()` will create the entire universe of dyad-years from 1816 to the present for a dyad-year analysis. The logic here, as it is implemented in `{peacesciencer}`'s multiple functions, is the type of data the user wants to create has been created for them. The user does not want to expand the data any further than that, though the user may want to do something like reduce the full universe of 1816-2020 state-years to just 1946-2010. However, this is a universe discarded, not a universe that has been augmented or expanded.

With that in mind, every function's use of the left join assumes the data object it receives represents the full universe of cases of interest to the researcher. The left join is just adding information to it, based on matching information in one of its many data sets. When done carefully, the left join is a dutiful way of adding information to a data set without changing the number of rows of the original data set. The number of columns will obviously expand, but the number of rows is unaffected.

Potential Problems of the Left Join

"When done carefully" is doing some heavy-lifting in that last sentence. So, let me explain some situations where the left join will produce problems for the researcher (even if the join itself is doing what it is supposed to do from an operational standpoint).

The first is less of a problem, at least as I have implemented in `{peacesciencer}`, but more of a caution. In the above example, our panel consists of just the U.S., Canada, the United Kingdom, and Australia. We happen to have a ranking for Denmark, but Denmark wasn't in our panel of (effectively, exclusively) Anglophone states. Therefore, no row is created for Denmark. If it were that important that the left join create those rows for Denmark, we should've had it in the first place. In this case, the left join is behaving as it should. We should've had Denmark in the panel before trying to match information to it.

`{peacesciencer}` circumvents this issue by creating universal data (e.g. all state-years, all dyad-years, all available leader-years) that the user is free to subset as they see fit. Users should run one of the "create" functions (e.g. `create_stateyears()`, `create_dyadyears()`) at the top of their script before adding information to it because the left join, as implemented everywhere in this package, is building in an assumption that the universe of cases of interest to the user is represented in the "left object" for a left outer join. Basically, do not expect the left join to create new rows in `x` in a situation where there is a state represented in `y` but not in `x`. It will not. This type of join assumes the universe of cases of interest to the researcher already appear in the "left object."

The second situation is a bigger problem. Sometimes, often when bouncing between information denominated in Correlates of War states and Gleditsch-Ward states, there is an unwanted duplicate observation in the data frame to be merged into the primary data of interest to the user. Let's go back to our simple example of `x` and `y` here. Everything here performs nicely, though Australia (in `x`) has no ranking and Denmark (in `y`) is not in our panel of state-years because it wasn't part of the original universe of cases of interest to us.

```
x
#> # A tibble: 20 x 2
#>    ccode  year
#>    <dbl> <int>
#>  1     2  2016
#>  2     2  2017
#>  3     2  2018
#>  4     2  2019
#>  5     2  2020
#>  6    20  2016
#>  7    20  2017
#>  8    20  2018
#>  9    20  2019
#> 10    20  2020
#> 11   200  2016
#> 12   200  2017
#> 13   200  2018
#> 14   200  2019
#> 15   200  2020
#> 16   900  2016
#> 17   900  2017
#> 18   900  2018
#> 19   900  2019
#> 20   900  2020
y
#> # A tibble: 4 x 2
#>   ccode ranking
#>   <dbl> <chr>
#> 1     2 low
#> 2    20 medium
#> 3   200 high
#> 4   390 high
```

Let's assume, however, we mistakenly entered the United Kingdom twice into y. We know these data are supposed to be simple state-level rankings. Each state is supposed to be in there just once. The United Kingdom appears in there twice.

```
tibble(ccode = 200,
       ranking = "high") %>%
  bind_rows(y, .) -> y2
```

If we were to left join y2 into x, we get an unwelcome result. The United Kingdom is duplicated for all yearly observations.

```
left_join(x, y2) %>% data.frame
#>    ccode year ranking
#> 1       2 2016      low
#> 2       2 2017      low
#> 3       2 2018      low
#> 4       2 2019      low
#> 5       2 2020      low
#> 6      20 2016   medium
#> 7      20 2017   medium
#> 8      20 2018   medium
#> 9      20 2019   medium
#> 10     20 2020   medium
#> 11    200 2016     high
#> 12    200 2016     high
#> 13    200 2017     high
#> 14    200 2017     high
#> 15    200 2018     high
#> 16    200 2018     high
#> 17    200 2019     high
#> 18    200 2019     high
#> 19    200 2020     high
#> 20    200 2020     high
#> 21    900 2016     <NA>
#> 22    900 2017     <NA>
#> 23    900 2018     <NA>
#> 24    900 2019     <NA>
#> 25    900 2020     <NA>
```

It doesn't matter that the duplicate ranking in `y2` for the UK was the same. It would be messier, sure, if the ranking was different for the duplicate observation, but it matters more here that it was duplicated. In a panel like this, a user who is not careful will have the effect of overweighting those observations that duplicate. In a simple example like this, subsetting to just complete cases (i.e. Australia has no ranking), the UK is 50% of all observations despite the fact it should just be a third of observations. That's not ideal for a researcher.

`{peacesciencer}` goes above and beyond to make sure this doesn't happen in the data it creates. Functions are aggressively tested to make sure nothing duplicates, and various parlor tricks (prominently group-by slices) are used internally to cull those duplicate observations. The release of a function that makes prominent use of the left join is done with the assurance it doesn't create a duplicate. No matter, this is the biggest peril of the left join for a researcher who may want to duplicate what `{peacesciencer}` does on their own. Always inspect the data you merge, and the output.

Semi-Join

The "semi-join" (`semi_join()` in `{dplyr}`) returns all rows from the left object (x) that have matching values in the right object (y). It is a type of "filtering join", which affects the observations and not the variables. It appears just twice in `{peacesciencer}`, serving as a final join in `create_leaderdays()` and `create_leaderyears()`. In both cases, it serves as a means of standardizing leader data (denominated in the Gleditsch-Ward system) to the Correlates of War system.

Here is a basic example of what a semi-join is doing in this context, with an illustration of the kind of difficulties that manifest in standardizing Archigos' leader data to the Correlates of War state system. Assume this simple state system that has just two states—“A” and “B”—over a two-week period at the start of 1975 (Jan. 1, 1975 to Jan. 14, 1975). In this simple system, “A” is a state for the full two week period (Jan. 1-Jan.14) whereas “B” is a state for just the first seven days (Jan. 1-Jan. 7) because, let's say, A occupied B and ended its statehood. We also happened to have some leader data for these two states. Over this two week period, our leader data suggests A had just one continuous leader—“Archie”—whereas B had three. “Brian” was the leader of B before he retired from office and was replaced by “Cornelius.” However, he was deposed when A invaded B and was replaced by a puppet head of state, “Pete.” Our data look like this.

```
tibble(code = c("A", "B"),
       stdate = make_date(1975, 01, 01),
       enddate = c(make_date(1975, 01, 14),
                   make_date(1975, 01, 07))) -> state_system

state_system
#> # A tibble: 2 x 3
#>   code  stdate     enddate
#>   <chr> <date>     <date>
#> 1 A     1975-01-01 1975-01-14
#> 2 B     1975-01-01 1975-01-07

tibble(code = c("A", "B", "B", "B"),
       leader = c("Archie", "Brian", "Cornelius", "Pete"),
       stdate = c(make_date(1975, 01, 01), make_date(1975, 01, 01),
                  make_date(1975, 01, 04), make_date(1975, 01, 08)),
       enddate = c(make_date(1975, 01, 14), make_date(1975, 01, 04),
                   make_date(1975, 01, 08), make_date(1975, 01, 14))) -> leaders

leaders
#> # A tibble: 4 x 4
#>   code  leader    stdate     enddate
#>   <chr> <chr>     <date>     <date>
#> 1 A     Archie    1975-01-01 1975-01-14
#> 2 B     Brian     1975-01-01 1975-01-04
#> 3 B     Cornelius 1975-01-04 1975-01-08
#> 4 B     Pete      1975-01-08 1975-01-14
```

We can use some basic `rowwise()` transformation to recast these data as daily, resulting in state-day data and leader-day data.

```
state_system %>%
  rowwise() %>%
  mutate(date = list(seq(stdate, enddate, by = '1 day'))) %>%
  unnest(date) %>%
  select(code, date) -> state_days

state_days %>% data.frame
#>    code       date
#> 1     A 1975-01-01
#> 2     A 1975-01-02
#> 3     A 1975-01-03
#> 4     A 1975-01-04
#> 5     A 1975-01-05
#> 6     A 1975-01-06
#> 7     A 1975-01-07
#> 8     A 1975-01-08
#> 9     A 1975-01-09
#> 10    A 1975-01-10
#> 11    A 1975-01-11
#> 12    A 1975-01-12
#> 13    A 1975-01-13
#> 14    A 1975-01-14
#> 15    B 1975-01-01
#> 16    B 1975-01-02
#> 17    B 1975-01-03
#> 18    B 1975-01-04
#> 19    B 1975-01-05
#> 20    B 1975-01-06
#> 21    B 1975-01-07

leaders %>%
  rowwise() %>%
  mutate(date = list(seq(stdate, enddate, by = '1 day'))) %>%
  unnest(date) %>%
  select(code, leader, date) -> leader_days

leader_days %>% data.frame
#>    code   leader       date
#> 1     A   Archie 1975-01-01
#> 2     A   Archie 1975-01-02
#> 3     A   Archie 1975-01-03
#> 4     A   Archie 1975-01-04
#> 5     A   Archie 1975-01-05
#> 6     A   Archie 1975-01-06
#> 7     A   Archie 1975-01-07
#> 8     A   Archie 1975-01-08
#> 9     A   Archie 1975-01-09
#> 10    A   Archie 1975-01-10
#> 11    A   Archie 1975-01-11
#> 12    A    Archie 1975-01-12
#> 13    A    Archie 1975-01-13
#> 14    A    Archie 1975-01-14
#> 15    B     Brian 1975-01-01
#> 16    B     Brian 1975-01-02
#> 17    B     Brian 1975-01-03
#> 18    B     Brian 1975-01-04
#> 19    B Cornelius 1975-01-04
#> 20    B Cornelius 1975-01-05
#> 21    B Cornelius 1975-01-06
#> 22    B Cornelius 1975-01-07
#> 23    B Cornelius 1975-01-08
#> 24    B      Pete 1975-01-08
#> 25    B      Pete 1975-01-09
#> 26    B      Pete 1975-01-10
#> 27    B      Pete 1975-01-11
#> 28    B      Pete 1975-01-12
#> 29    B      Pete 1975-01-13
#> 30    B      Pete 1975-01-14
```

If we wanted to standardize these leader-day data to the state system data, we would semi-join the leader-day data (the left object) with the state-day object (the right object), returning just the leader-day data with valid days in the state system data.

```
leader_days %>%
  semi_join(., state_days) %>%
  data.frame
#>    code    leader       date
#> 1     A    Archie 1975-01-01
#> 2     A    Archie 1975-01-02
#> 3     A    Archie 1975-01-03
#> 4     A    Archie 1975-01-04
#> 5     A    Archie 1975-01-05
#> 6     A    Archie 1975-01-06
#> 7     A    Archie 1975-01-07
#> 8     A    Archie 1975-01-08
#> 9     A    Archie 1975-01-09
#> 10    A    Archie 1975-01-10
#> 11    A    Archie 1975-01-11
#> 12    A    Archie 1975-01-12
#> 13    A    Archie 1975-01-13
#> 14    A    Archie 1975-01-14
#> 15    B     Brian 1975-01-01
#> 16    B     Brian 1975-01-02
#> 17    B     Brian 1975-01-03
#> 18    B     Brian 1975-01-04
#> 19    B Cornelius 1975-01-04
#> 20    B Cornelius 1975-01-05
#> 21    B Cornelius 1975-01-06
#> 22    B Cornelius 1975-01-07
```

Notice that Pete drops from these data because, in this simple example, Pete was a puppet head of state installed by Archie when state A invaded and occupied B. The semi-join here is simply standardizing the leader data to the state system data, which is effectively what's happening with the semi-joins in `create_leaderdays()` (and its aggregation function: `create_leaderyears()`).

Anti-Join

The anti-join is another type of filtering join, returning all rows from the left object (`x`) with*out* a match in the right object (`y`). This type of join appears just once in `{peacesciencer}`. Prominently, `{peacesciencer}` prepares and presents two data sets in this package—`false_cow_dyads` and `false_gw_dyads`—that represent directed dyad-years in the Correlates of War and Gleditsch-Ward systems that were active in the same year, but never at the same time on the same year.

Here are those dyads for context.

```
false_cow_dyads
#> # A tibble: 60 x 4
#>    ccode1 ccode2  year in_ps
#>     <dbl>  <dbl> <int> <dbl>
#>  1    115    817  1975     1
#>  2    210    255  1945     1
#>  3    211    255  1945     1
#>  4    223    678  1990     1
#>  5    223    680  1990     1
#>  6    255    210  1945     1
#>  7    255    211  1945     1
#>  8    255    260  1990     1
#>  9    255    265  1990     1
#> 10    255    290  1945     1
#> # ... with 50 more rows

false_gw_dyads
#> # A tibble: 38 x 4
#>    gwcode1 gwcode2  year in_ps
#>      <dbl>   <dbl> <int> <dbl>
#>  1      99     100  1830     1
#>  2      99     211  1830     1
#>  3     100      99  1830     1
#>  4     100     615  1830     1
#>  5     115     817  1975     1
#>  6     211      99  1830     1
#>   7      211      615   1830      1
#>   8      255      850   1945      1
#>   9      300      305   1918      1
#>  10      300      345   1918      1
#> # ... with 28 more rows
```

These were created by two scripts that, for each year in the respective state system data, creates every possible *daily* dyadic pairing and truncates the dyads to just those that had at least one day of overlap. This is a computationally demanding procedure compared to what `{peacesciencer}` does (which creates every possible dyadic pair in a given year, given the state system data supplied to it). However, it creates the possibility of same false dyads in a given year that showed no overlap.

Consider the case of Suriname (115) and the Republic of Vietnam (817) in 1975 as illustrative here.

```r
check_both <- function(x) {

  gw_states %>%
    mutate(data = "G-W") %>%
    filter(gwcode %in% x) -> gwrows

  cow_states %>%
    mutate(startdate = ymd(paste0(styear,"/",stmonth, "/", stday)),
           enddate = ymd(paste0(endyear,"/",endmonth,"/",endday))) %>%
    select(stateabb:statenme, startdate, enddate) %>%
    mutate(data = "CoW") %>%
    rename(statename = statenme) %>%
    filter(ccode %in% x) -> cowrows

  dat <- bind_rows(gwrows, cowrows) %>%
    select(gwcode, ccode, stateabb, everything())

  return(dat)
}

check_both(c(115, 817))
#> # A tibble: 4 x 7
#>   gwcode ccode stateabb statename             startdate  enddate    data
#>    <dbl> <dbl> <chr>    <chr>                 <date>     <date>     <chr>
#> 1    115    NA SUR      Surinam               1975-11-25 2017-12-31 G-W
#> 2    817    NA RVN      Vietnam, Republic of  1954-05-01 1975-04-30 G-W
#> 3     NA   115 SUR      Suriname              1975-11-25 2016-12-31 CoW
#> 4     NA   817 RVN      Republic of Vietnam   1954-06-04 1975-04-30 CoW
```

Notice both Suriname and Republic of Vietnam were both active in 1975. Suriname appears on Nov. 25, 1975 whereas the Republic of Vietnam exits on April 30, 1975. However, there is no daily overlap between the two because they did not exist at any point on the same day in 1975. These are false dyads. `anti_join()` is used in the `create_dyadyears()` function to remove these observations before presenting them to the user.

Here is a simple example of what an anti-join is doing with these examples in mind.

```
valid_dyads <- tibble(ccode1 = c(2, 20, 200),
                      ccode2 = c(20, 200, 900),
                      year = c(2016, 2017, 2018))

valid_dyads %>%
  bind_rows(., false_cow_dyads %>% select(ccode1:year)) -> valid_and_invalid

valid_and_invalid
#> # A tibble: 63 x 3
#>    ccode1 ccode2  year
#>     <dbl>  <dbl> <dbl>
#>  1      2     20  2016
#>  2     20    200  2017
#>  3    200    900  2018
#>  4    115    817  1975
#>  5    210    255  1945
#>  6    211    255  1945
#>  7    223    678  1990
#>  8    223    680  1990
#>  9    255    210  1945
#> 10    255    211  1945
#> # ... with 53 more rows

valid_and_invalid %>%
  # remove those invalid dyads-years
  anti_join(., false_cow_dyads)
#> # A tibble: 3 x 3
#>   ccode1 ccode2  year
#>    <dbl>  <dbl> <dbl>
#> 1      2     20  2016
#> 2     20    200  2017
#> 3    200    900  2018
```

Vignette: {peacesciencer} Data Versions

```
library(tidyverse)
library(peacesciencer)
library(kableExtra)
```

These are the data versions available in {peacesciencer}. Do note the user can find specific data versions with more targeted use of the ps_version() function in this package. Without an argument, ps_version() produces a data frame of all the data versions in this package.

```
ps_version() %>%
  kbl(., caption = "Data Versions in `{peacesciencer}`",
      align =c("c", "l", "c", "c"))  %>%
  kable_styling(position = "center", full_width = F, bootstrap_options = "striped")
```

Every data set that is used by a function in this package is included in this table (and with the `ps_version()` function). Users can see a category of the type of data (which can be used for more careful searches of the underlying data), a description of the data set, a version number associated with that data set in `{peacesciencer}`, along with a BibTeX key. Users should interpret data versions as years as instances where the data are not formally versioned, per se, and the year corresponds with a year of a last update or a year of publication. For example, Anders et al. (2020) released their state-year simulations of population, surplus domestic product, and gross domestic product in 2020. The data are not formally versioned and the year corresponds with, in this case, the publication.

Users can use the BibTeX key (`bibtexkey`) to search the dataframe of citations in this package. For example, we can use `ps_cite()` to get a full citation for Carter and Smith's (2020) estimates of leader willingness to use force.

```
ps_cite("cartersmith2020fml", column = "bibtexkey")
#> @Article{cartersmith2020fml,
#>   Author = {Jeff Carter and Charles E. Smith},
#>   Journal = {American Political Science Review},
#>   Number = {4},
#>   Pages = {1352--1358},
#>   Title = {A Framework for Measuring Leaders' Willingness to Use Force},
#>   Volume = {114},
#>   Year = {2020},
#>   Keywords = {lwuf, add_lwuf()}
#> }
```

Table A.1: Data Versions in ‘peacesciencer‘

category	data	version	bibtexkey
states	Correlates of War State System Membership	2016	cowstates2016
leaders	LEAD	2015	ellisetal2015lead
leaders	Archigos	4.1	goemansetal2009ia
alliance	ATOP	5	leedsetal2002atop
democracy	Polity	2017	marshalletal2017p
democracy	{QuickUDS}	0.2.3	marquez2016qme
democracy	V-Dem	10	coppedgeetal2020vde
capitals	{peacesciencer}	2020	peacesciencer-packag
contiguity	Correlates of War Direct Contiguity	3.2	stinnettetal2002cow
igo	Correlates of War International Governmental Organizations (IGOs)	3	pevehouseetal2020to
majors	Correlates of War	2016	cowstates2016
conflict_interstate	Correlates of War Militarized Interstate Disputes	5	palmeretal2021mid5
distance	{Cshapes}	2	schvitz2021mis
capabilities	Correlates of War National Material Capabilities	5	singer1987rcwd
gdp	SDP	2020	andersetal2020bbgb
sdp	SDP	2020	andersetal2020bbgb
population	SDP	2020	andersetal2020bbgb
trade	Correlates of War Trade	4	barbierietal2009td
conflict_intrastate	Correlates of War Intra-State War	4.1	dixonsarkees2016giv
conflict_interstate	Correlates of War Inter-State War	4	sarkeeswayman2010r
fractionalization	CREG	2012	nardulli2012creg
polarization	CREG	2012	nardulli2012creg
conflict_interstate	Gibler-Miller-Little (GML)	2.2.1	gibleretal2016amid
states	Gleditsch-Ward	2017	gledtischward1999rl
leaders	Leader Willingness to Use Force	2020	cartersmith2020fml
terrain	Ruggedness	2012	nunnpuga2012r
terrain	% Mountainous	2014	giblermiller2014etts
rivalries	Thompson and Dreyer	2012	thompsondreyer2012
conflict_intrastate	UCDP Armed Conflicts	20.1	gleditschetal2002ac
conflict_intrastate	UCDP Onsets	19.1	pettersson2019ov
dyadic_similarity	FPSIM	2	haege2011cc

References

Miller, Steven V (2017) The effect of terrorism on judicial confidence. *Political Research Quarterly* 70(4): 790–802.