STAT 209: Lab 8

Code

Reshaping data to make it “tidy”

Goal

Become comfortable recognizing when reshaping data will make it better suited to the task at hand, and learn how to do so with the `gather()` and `spread()` verbs in the `tidyr` package (part of the all-powerful `tidyverse`).

The Data

Should we try to squeeze some more insight out of the `babynames` data? Let’s try to squeeze some more insight out of the `babynames` data. At least for starters.

Load the packages and data:

Code

In the last lab, we joined the Social Security `babynames` data with the Census `births` data to produce a table that had two records of the total number of births in each year; one from each source.

Here’s the code we used to do it (below is the “full join” version).

Code:

Code

```
## # A tibble: 6 x 4
##    year num_rows births.x births.y
##   <dbl>    <int>    <int>    <int>
## 1  1880     2000   201484       NA
## 2  1881     1935   192696       NA
## 3  1882     2127   221533       NA
## 4  1883     2084   216946       NA
## 5  1884     2297   243462       NA
## 6  1885     2294   240854       NA
```

The `births.x` and `births.y` variables are not very descriptive; also we don't care so much about the `num_rows` variable, so let's do some `select`ion (to remove `num_rows`) and `rename`ing (to replace the uninformative names with informative ones).

Code:

Code

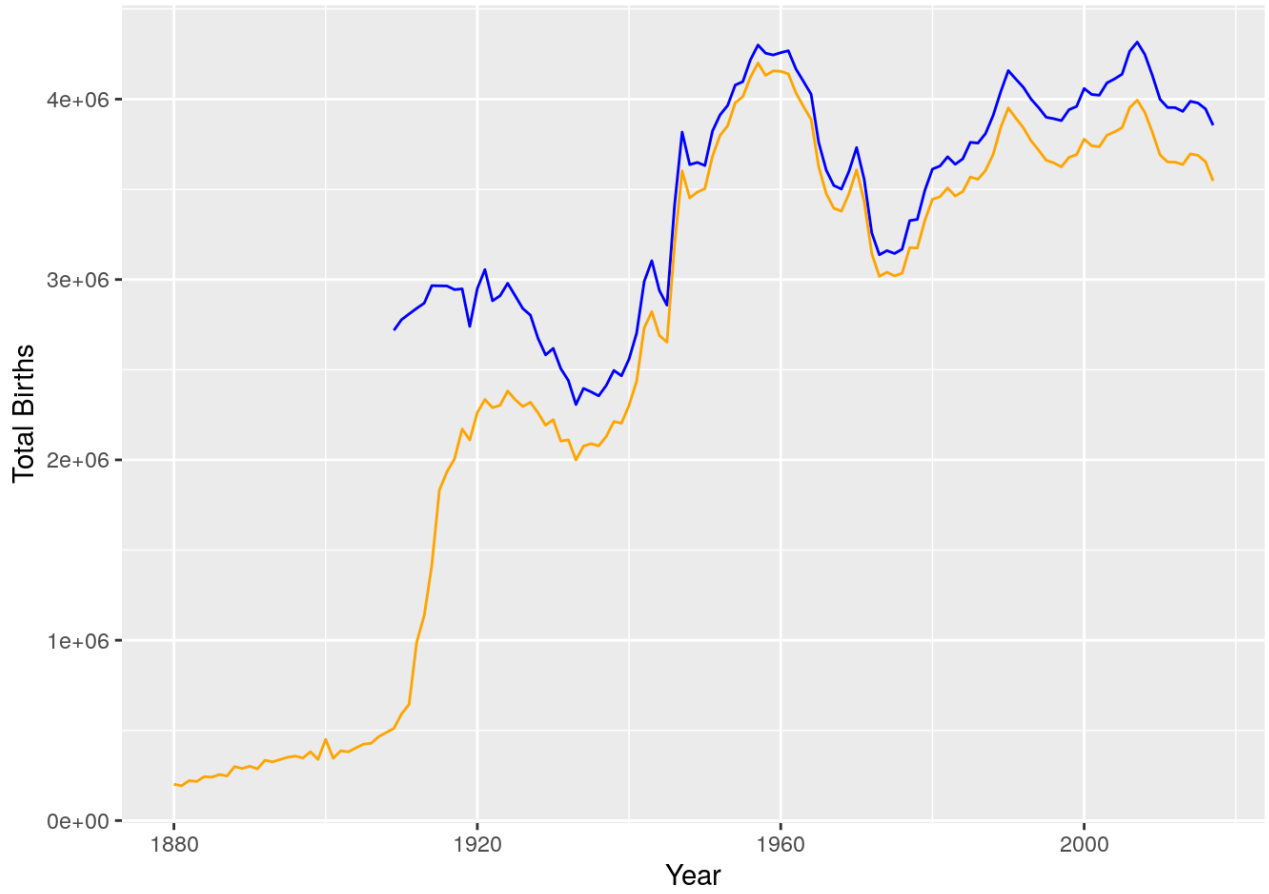
```
## # A tibble: 6 x 3
##    year    ssa census
##   <dbl>  <int>  <int>
## 1  1880 201484     NA
## 2  1881 192696     NA
## 3  1882 221533     NA
## 4  1883 216946     NA
## 5  1884 243462     NA
## 6  1885 240854     NA
```

If we want to visualize the number of births over time from two different sources using two overlaid lines, we have to set the `y` aesthetic separately for each line, and if we want different colors, we have to specify them manually, line by line:

Code:

Code

```
## Warning: Removed 29 rows containing missing values (geom_path).
```



We also don't get an automatic legend.

For a graph like this, we'd like to be able to create an aesthetic mapping between the **source** of the data and the color of the line. That mapping could then be used to automatically produce a legend. But `source` isn't a variable in this data; it's distinguished between variables, not between cases.

The `gather()` function

Thinking about what the legend title would be if we created one gives us a clue that we need to wrangle this data into a format conducive to the plot we want.

We need a new variable called something like `source`, and a single variable to map to the y-axis, recording the number of births from the respective source.

We can use `gather()` for this, as follows:

Code:

Code

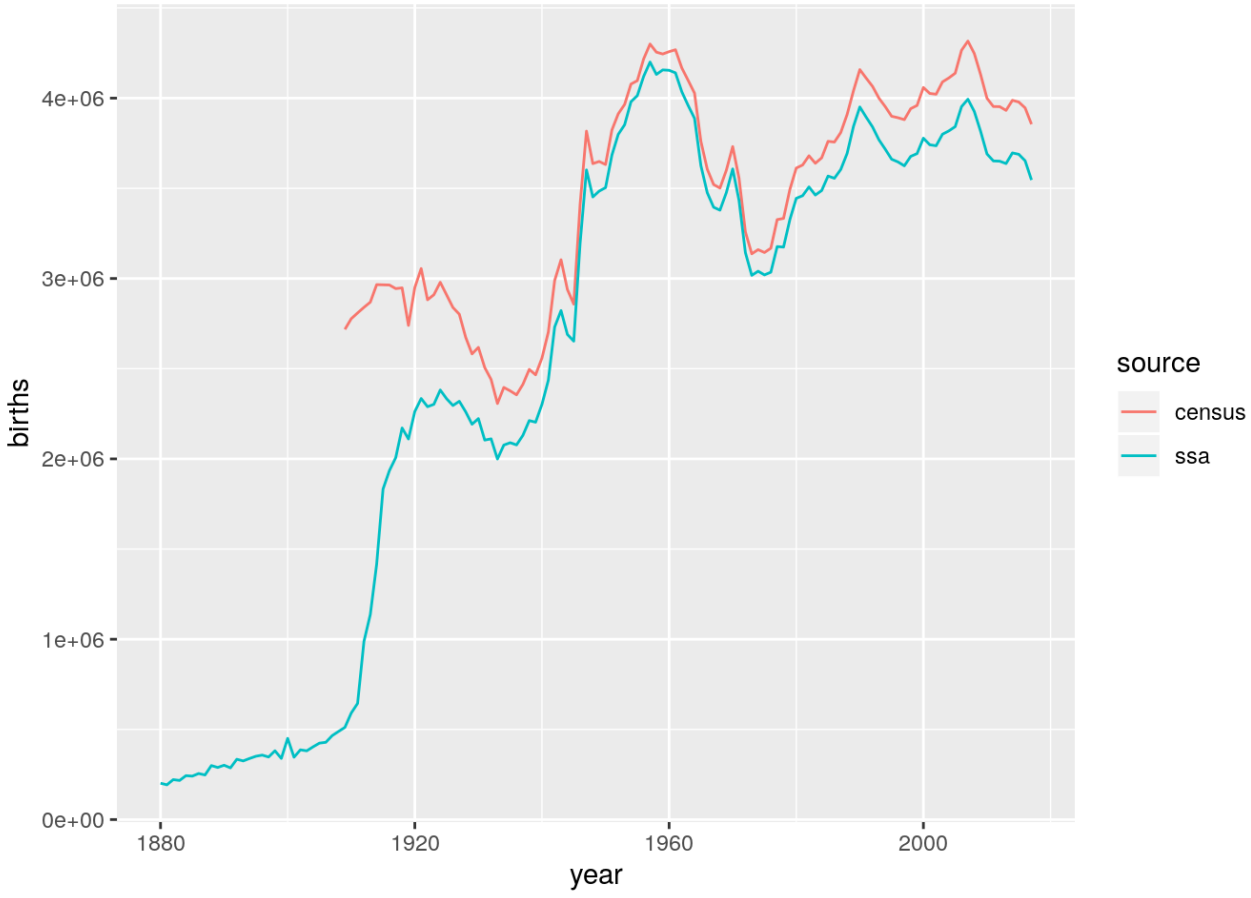
```
## # A tibble: 6 x 3
##    year source births
##   <dbl> <chr>   <int>
## 1  1880 census     NA
## 2  1881 census     NA
## 3  1882 census     NA
## 4  1883 census     NA
## 5  1884 census     NA
## 6  1885 census     NA
```

Having created the `source` variable and having merged all the counts into a single `births` variable, we can now create the line graph we want quite easily (and we get a legend automatically, since the color of the line now comes from a variable in the data table)

Code:

Code

```
## Warning: Removed 29 rows containing missing values (geom_path).
```



The `spread()` function

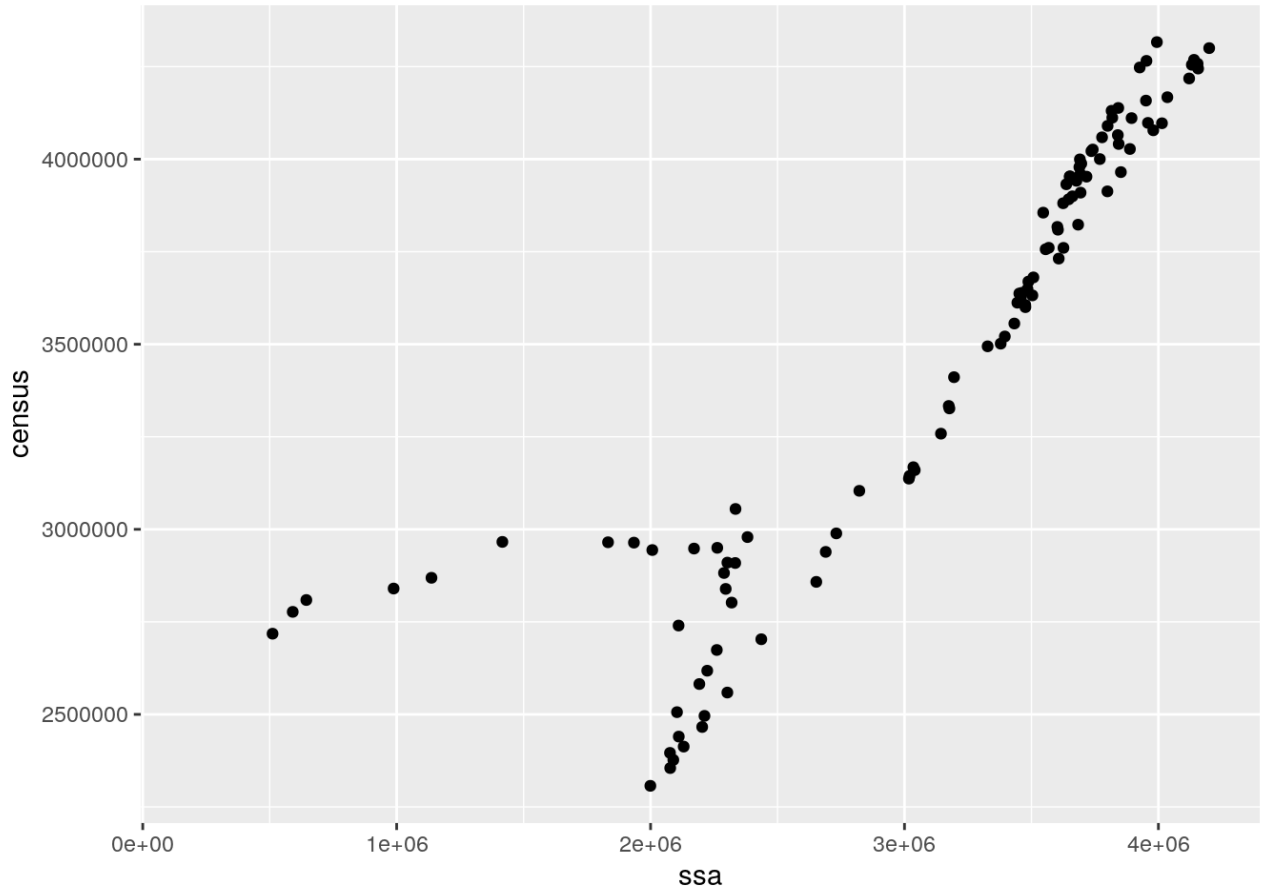
Is the "long" format we've created "better" in an absolute sense? Well, it's better for producing the line graph we wanted, but suppose we wanted to visualize the correlation between the sources with a scatterplot. For a plot like this, we want one axis to be the number of births according to the SSA, and

the other axis to be the number of births according to the Census. This was easy in the original data:

Code:

Code

```
## Warning: Removed 29 rows containing missing values (geom_point).
```



If the data had come to us in the “long” format, however, it would be much less obvious how to create this plot. It’s also not so clear how we’d do something like compute the correlation, or the difference between the estimates in a particular year (I suppose we could use `group_by()` and `summarize()` to do this last one, but it wouldn’t be that straightforward).

There may be times when we want to go the other direction: if we want to compute or plot something that depends on ordered pairs (or ordered tuples more generally), such as computing a correlation, or creating a new variable via `mutate()` that depends on both entries, then it is probably easier if the coordinates of those pairs (or tuples) are stored in separate variables.

The `spread()` function does this:

Code:

Code

Uh oh, we get an error. What’s that about? It seems that the years from 2002 on have duplicate records in the original `births` dataset:

Code:

Code

```
## # A tibble: 10 x 3
##     year source  births
##    <dbl> <chr>    <int>
##  1  2008 census 4247694
##  2  2009 census 4130665
##  3  2010 census 3999386
##  4  2011 census 3953590
##  5  2012 census 3952841
##  6  2013 census 3932181
##  7  2014 census 3988076
##  8  2015 census 3978497
##  9  2016 census 3945875
## 10  2017 census 3855500
```

For `spread()` to work, it needs to know exactly what to put in each new column for each different *key* (each `source` in this case). In order for the "wide" format to be well-defined, there must be a unique mapping from each combination of the other variables (in this case just `year`) to a *value*, for each different *key*. That's violated here, and so `spread()` doesn't know what to do.

Since in this case the problem is caused by duplicate entries, we can just remove the duplicates before `spread()`ing the data. There are multiple ways to do that; here's an easy one:

Code

```
## # A tibble: 6 x 3
##    year source  births
##   <dbl> <chr>    <int>
## 1  2012 census 3952841
## 2  2013 census 3932181
## 3  2014 census 3988076
## 4  2015 census 3978497
## 5  2016 census 3945875
## 6  2017 census 3855500
```

OK, let's try `spread()`ing the data again.

Code

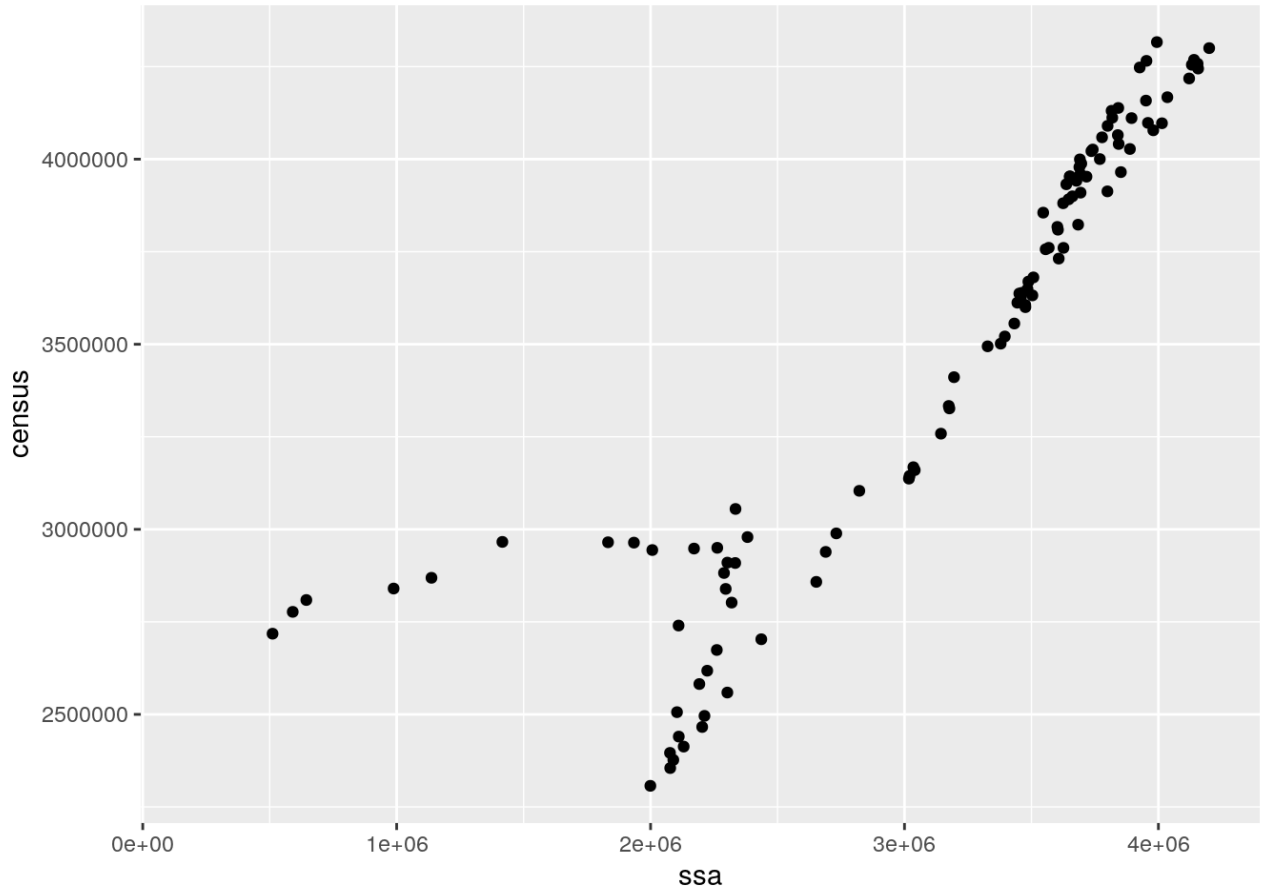
```
## # A tibble: 6 x 3
##    year census    ssa
##   <dbl>  <int>  <int>
## 1  1880     NA 201484
## 2  1881     NA 192696
## 3  1882     NA 221533
## 4  1883     NA 216946
## 5  1884     NA 243462
## 6  1885     NA 240854
```

Looks just like the data we started with (except for the order of the columns, and the removal of the duplicate rows)!

Now we can produce a scatterplot… **Code:**

Code

```
## Warning: Removed 29 rows containing missing values (geom_point).
```



(which we could do with the original data, but not with the “long” format data; here we’re just undoing what we did, but in real applications we will sometimes have data come to us in “long” format and need to convert it to “wide”)

We can also compute the correlation: **Code:**

Code

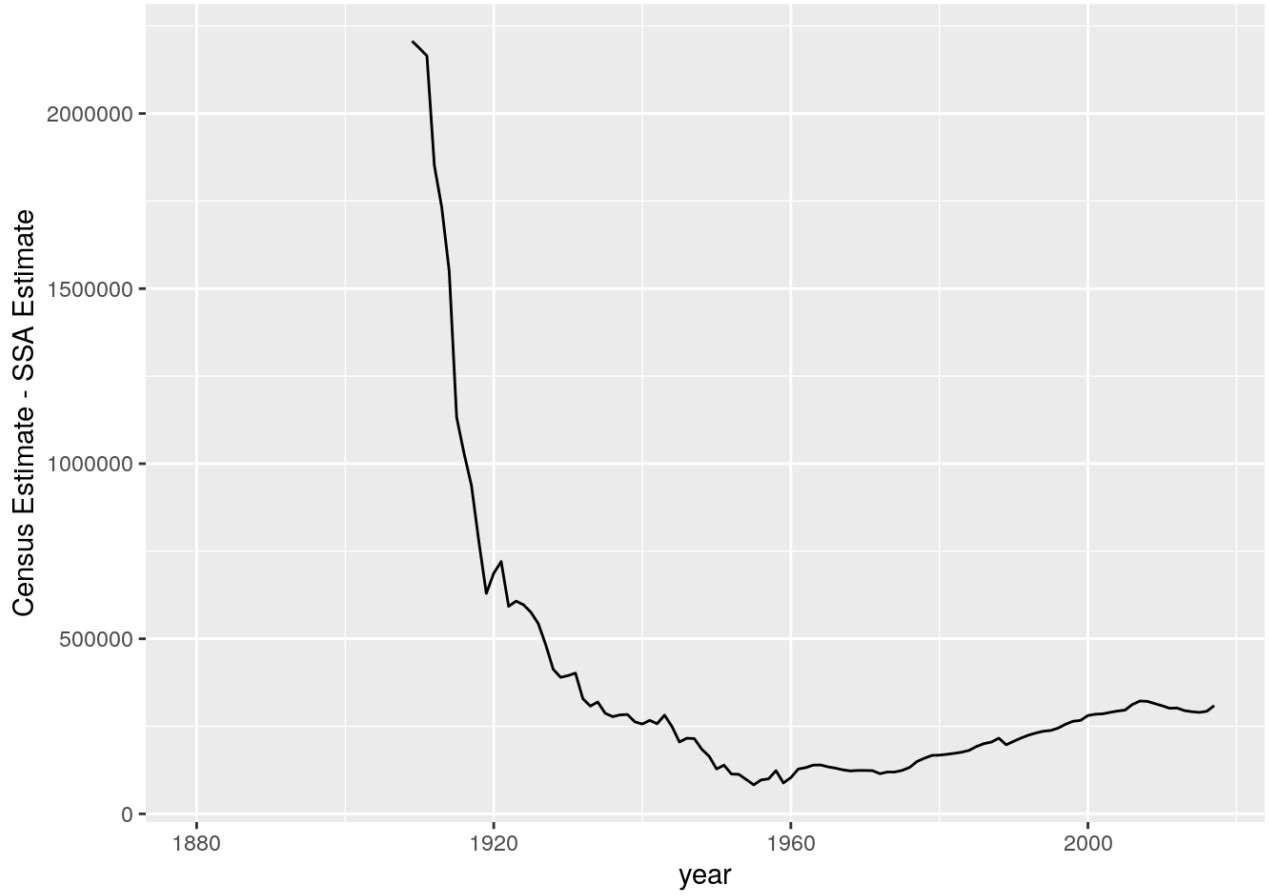
```
## Registered S3 method overwritten by 'mosaic':
##   method                           from
##   fortify.SpatialPolygonsDataFrame ggplot2
```

```
## [1] 0.8945154
```

or compute and plot the discrepancy: **Code:**

Code

```
## Warning: Removed 29 rows containing missing values (geom_path).
```



Even the original data has some features to its organization that might require `spread()` ing in order to easily compute or visualize certain things.

For example, recall that the rows in the `babynames` dataset correspond to unique combinations of year, name, and sex. If we wanted to find the total number of births associated with a particular name in a particular year irrespective of sex, we have previously needed to use `group_by()` and `summarize()` like so:

Code:

Code

```
## # A tibble: 820,608 x 3
## # Groups:   year [51]
##     year name    num_births
##    <dbl> <chr>        <int>
##  1  1950 Aaron          805
##  2  1950 Abagail          5
##  3  1950 Abbe            10
##  4  1950 Abbey            7
##  5  1950 Abbie           68
##  6  1950 Abbott           9
##  7  1950 Abby            73
##  8  1950 Abdul            8
##  9  1950 Abe             46
## 10  1950 Abel           170
## # … with 820,598 more rows
```

which is fine, but suppose we wanted both the total *and* the number associated with the two recorded sexes in a single table. Previously this involved the somewhat awkward step of using `ifelse()` to tally up the number of births for a subset of the data:

Code:

Code

```
## # A tibble: 820,608 x 5
## # Groups:   year [51]
##     year name    num_males num_females total_births
##    <dbl> <chr>       <dbl>       <dbl>        <int>
##  1  1950 Aaron           1           1          805
##  2  1950 Abagail         0           1            5
##  3  1950 Abbe            0           1           10
##  4  1950 Abbey           0           1            7
##  5  1950 Abbie           0           1           68
##  6  1950 Abbott          1           0            9
##  7  1950 Abby            1           1           73
##  8  1950 Abdul           1           0            8
##  9  1950 Abe             1           0           46
## 10  1950 Abel            1           0          170
## # … with 820,598 more rows
```

A more elegant solution to achieve this same thing would be to `spread()` the values in the original `n` column into two columns: one for births tagged `"M"`, and one for births tagged `"F"`.

Code

```
## # A tibble: 6 x 4
##    year name        F     M
##   <dbl> <chr>   <dbl> <dbl>
## 1  1950 Aaron       7   798
## 2  1950 Abagail     5     0
## 3  1950 Abbe       10     0
## 4  1950 Abbey       7     0
## 5  1950 Abbie      68     0
## 6  1950 Abbott      0     9
```

Note that because the original data also had a `prop` column, showing the proportion of births in a particular year,sex combination that had a specific name, using `spread()` without first removing this column would not do what we want. This is because `spread()` needs to be able to group rows together that have identical values for all variables *except* the ones specified as the `key=` and the `value=`. And `prop` is different for each `sex`; so it would cause the output to have two rows for each name again: one with a value for `F` and missing data for `M`, and one with the reverse.

By removing `prop` before we spread, we ensure that all the non-`key` non-`value` columns will be identical for the group of rows that is to be consolidated into a single row.

Now that we have `F` and `M` as separate columns, we can simply use `mutate()` to find the total number of births with each name in each year.

Code:

Code

By the way, now we can quite easily produce a measure of the “unisex” quality of a name.

(Make sure you understand what’s going on at each step below: this is all involving verbs you’ve worked with before) **Code:**

Code

```
## # A tibble: 1,533 x 6
##    name        M     F total prop_male asymmetry
##    <chr>   <dbl> <dbl> <dbl>     <dbl>     <dbl>
##  1 Unknown  6949  6967 13916     0.499  0.000647
##  2 Kris    12369 12742 25111     0.493  0.00743
##  3 Carey   12231 11708 23939     0.511  0.0109
##  4 Peyton   9576 10006 19582     0.489  0.0110
##  5 Kerry   43412 46000 89412     0.486  0.0145
##  6 Jaime   50004 47051 97055     0.515  0.0152
##  7 Ashton  12052 13180 25232     0.478  0.0224
##  8 Blair   10351  9215 19566     0.529  0.0290
##  9 Kendall 23721 20757 44478     0.533  0.0333
## 10 Justice  6100  5312 11412     0.535  0.0345
## # … with 1,523 more rows
```

Exercises

Exercise 1 Find an interesting dataset from the Gapminder repository here (http://www.gapminder.org/data/) and download it first as an Excel spreadsheet (an `.xlsx` file). Open it in Excel or a similar program, and export it as a `.csv`.

Exercise 2 Upload the `.csv` file to RStudio (assuming you are working on the server), and read it in using `read_csv()` (the function with an underscore is part of the tidyverse, and tends to work better than the built-in one with a period). You will need to supply as the argument to `read_csv()` the path to the file *relative* to the directory where your `.Rmd` is.

Exercise 3 The cases in the Gapminder datasets are countries. Use `rename()` to change the first variable name to `country` (since `rename()` expects a variable name *without* quotes, you may need to surround the oriiginal variable name with backticks (the same syntax you use to get a code font in Markdown) if it has spaces or special characters. Pro-tip: never use spaces or special characters (other than underscores) in variable names.

Example:

Code

Exercise 4 Use `dim()` to find out how many rows and columns are in your data.

Exercise 5 We will convert our data to a format with exactly three columns: `country`, `year`, and `value` (whatever value is for your chosen dataset). *Before you write any code*, sketch *on paper* what the "tidified" data will look like. Be sure to indicate how many rows it will have.

Exercise 6 Use `gather()` to convert your data into this format. Use `head()` and `dim()` to verify that it worked as expected.

Exercise 7 The `year` variable may be stored as text instead of as a number, which will make mapping it to a spatial dimension challenging. Fix this using `mutate()`, with the help of the `parse_number()` function (supplied by the `readr` package, which is *also* part of the tidyverse). Type `?parse_number` at the console to see how to use it if it's not clear.

Exercise 8 Plot your variable as a time series line graph, using color to distinguish countries.

Getting credit

1. Post your graph from Exercise 8 to the #lab8 channel, along with the Honor Pledge
2. Answer the following via DM to me and to Chris Ikeokwu: If you got stuck somewhere along the way doing this lab, where was it? What did you do to get unstuck? If you didn't get stuck, what part of this process did you find the most difficult?