Working with data frames

Mark Andrews

July 9, 2018

```
library(dplyr)
library(readr)
library(tidyr)
options(tibble.width = Inf)
```

Introduction

Most of time when we are working with data, we work with *data frames*. Data frames can be seen as similar to spreadsheets, i.e. with multiple rows and multiple columns, and each column representing a variable. In this note, we will deal with data-frame using the `tidyverse` approach. You can do more or less everything shown below in a `base R` way too, but on balance I think the `tidyverse` way is the more efficient way and I think it is more likely to be way we all be doing it in the future anyway.

Read in a data frame from csv file

We'll start by reading in a csv file as a data frame (which could also be done from the RStudio `Import Dataset` menu in `Environment`):

```
(Df <- read_csv('../data/LexicalDecision.csv'))
```

```
## # A tibble: 3,908 x 7
##    subject item    accuracy latency valence length frequency
##      <int> <chr>      <int>   <int>   <dbl>  <int>     <dbl>
##  1       1 alive          1     498    7.25      5     42.5
##  2       1 bandage        1     716    4.54      7      2.53
##  3       1 bright         1     559    7.5       6     55.4
##  4       1 carcass        1     564    3.34      7      1.4
##  5       1 cheer          1     538    8.1       5      7.81
##  6       1 coast          1     463    5.98      5     47.0
##  7       1 detail         1     486    5.55      6     62.2
##  8       1 devil          1     562    2.21      5     17.3
##  9       1 door           1     541    5.13      4    254.
## 10       1 evil           1     507    3.23      4     28.8
## # ... with 3,898 more rows
```

Note that `read_csv`, which is part of the `readr` package, which is loaded above.

Quick look of your data frame

```
glimpse(Df)
```

```
## Observations: 3,908
## Variables: 7
```

```
## $ subject   <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1...
## $ item      <chr> "alive", "bandage", "bright", "carcass", "cheer", "c...
## $ accuracy  <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1...
## $ latency   <int> 498, 716, 559, 564, 538, 463, 486, 562, 541, 507, 52...
## $ valence   <dbl> 7.25, 4.54, 7.50, 3.34, 8.10, 5.98, 5.55, 2.21, 5.13...
## $ length    <int> 5, 7, 6, 7, 5, 5, 6, 5, 4, 4, 4, 3, 4, 5, 7, 6, 4, 6...
## $ frequency <dbl> 42.54, 2.53, 55.40, 1.40, 7.81, 47.01, 62.23, 17.32,...
```

Rename variable names

We can rename as many variables as you like as follows:

```
(Df <- rename(Df,
              word = item,
              reaction_time = latency))
```

```
## # A tibble: 3,908 x 7
##    subject word    accuracy reaction_time valence length frequency
##      <int> <chr>      <int>         <int>   <dbl>  <int>     <dbl>
##  1       1 alive          1           498    7.25      5     42.5
##  2       1 bandage        1           716    4.54      7      2.53
##  3       1 bright         1           559    7.5       6     55.4
##  4       1 carcass        1           564    3.34      7      1.4
##  5       1 cheer          1           538    8.1       5      7.81
##  6       1 coast          1           463    5.98      5     47.0
##  7       1 detail         1           486    5.55      6     62.2
##  8       1 devil          1           562    2.21      5     17.3
##  9       1 door           1           541    5.13      4    254.
## 10       1 evil           1           507    3.23      4     28.8
## # ... with 3,898 more rows
```

The `rename` function takes a data-frame and returns a new data frame. In other words, it does not affect the original data-frame, but produces a copy[1] of the original but with the variables renamed.

Subsetting your data frame

In any data analysis, a lot of time is spent selecting subsets of rows and columns of our data-frame. Doing so efficiently makes everything quicker and easier.

Choose a subset of variables (i.e., columns)

Using the `select` function, you will just list out the names of the variables you want to keep:

```
select(Df, subject, word, accuracy, reaction_time)
```

```
## # A tibble: 3,908 x 4
##    subject word    accuracy reaction_time
##      <int> <chr>      <int>         <int>
##  1       1 alive          1           498
##  2       1 bandage        1           716
```

[1] It's not actually a copy of the data but a copy of the pointers to the data. That means that these operations are both fast and memory efficient.

```
##  3       1 bright          1           559
##  4       1 carcass         1           564
##  5       1 cheer           1           538
##  6       1 coast           1           463
##  7       1 detail          1           486
##  8       1 devil           1           562
##  9       1 door            1           541
## 10       1 evil            1           507
## # ... with 3,898 more rows
```

Sometimes, especially when you have many variables, selecting all those you want to keep by explicitly writing down their names as above can be a lot of work. Here are some short-cuts. Let's say you want to keep all but the variables `valence`, you could do:

```
select(Df, -valence) # select all variables except `valence`
```

```
## # A tibble: 3,908 x 6
##    subject word    accuracy reaction_time length frequency
##      <int> <chr>      <int>         <int>  <int>     <dbl>
##  1       1 alive          1           498      5     42.5
##  2       1 bandage        1           716      7      2.53
##  3       1 bright         1           559      6     55.4
##  4       1 carcass        1           564      7      1.4
##  5       1 cheer          1           538      5      7.81
##  6       1 coast          1           463      5     47.0
##  7       1 detail         1           486      6     62.2
##  8       1 devil          1           562      5     17.3
##  9       1 door           1           541      4    254.
## 10       1 evil           1           507      4     28.8
## # ... with 3,898 more rows
```

If you wanted to keep all but `valence` and `frequency`, you can do

```
select(Df, -valence, -frequency)
```

```
## # A tibble: 3,908 x 5
##    subject word    accuracy reaction_time length
##      <int> <chr>      <int>         <int>  <int>
##  1       1 alive          1           498      5
##  2       1 bandage        1           716      7
##  3       1 bright         1           559      6
##  4       1 carcass        1           564      7
##  5       1 cheer          1           538      5
##  6       1 coast          1           463      5
##  7       1 detail         1           486      6
##  8       1 devil          1           562      5
##  9       1 door           1           541      4
## 10       1 evil           1           507      4
## # ... with 3,898 more rows
```

Note that the above code effectively *deletes* the `valence` and `frequency` variables, so we can use it to drop variables from a data frame.

We can also select sequences of variables. For example, we could keep all variables starting with the variables `subject` and ending with `length` as follows:

```
select(Df, subject:length)
```

```
## # A tibble: 3,908 x 6
##    subject word    accuracy reaction_time valence length
##      <int> <chr>      <int>         <int>   <dbl>  <int>
##  1       1 alive          1           498    7.25      5
##  2       1 bandage        1           716    4.54      7
##  3       1 bright         1           559    7.5       6
##  4       1 carcass        1           564    3.34      7
##  5       1 cheer          1           538    8.1       5
##  6       1 coast          1           463    5.98      5
##  7       1 detail         1           486    5.55      6
##  8       1 devil          1           562    2.21      5
##  9       1 door           1           541    5.13      4
## 10       1 evil           1           507    3.23      4
## # ... with 3,898 more rows
```

Although we won't cover them here, there are other more powerful tricks that use *regular expressions*. These are very handy for selecting variables that all begin with the same prefix, e.g. `foo-1`, `foo-2`, `foo-3` ... `foo-78`.

One final handy trick is the `everything` function. Let's say you want to move the variable `frequency` to be the first variable in the data-frame. You could do

```
select(Df, frequency, everything())
```

```
## # A tibble: 3,908 x 7
##    frequency subject word    accuracy reaction_time valence length
##        <dbl>   <int> <chr>      <int>         <int>   <dbl>  <int>
##  1     42.5        1 alive          1           498    7.25      5
##  2      2.53       1 bandage        1           716    4.54      7
##  3     55.4        1 bright         1           559    7.5       6
##  4      1.4        1 carcass        1           564    3.34      7
##  5      7.81       1 cheer          1           538    8.1       5
##  6     47.0        1 coast          1           463    5.98      5
##  7     62.2        1 detail         1           486    5.55      6
##  8     17.3        1 devil          1           562    2.21      5
##  9    254.         1 door           1           541    5.13      4
## 10     28.8        1 evil           1           507    3.23      4
## # ... with 3,898 more rows
```

Choose a subset of the observations (i.e., rows)

If you want to select some rows, you can use a `slice`. In the following, we choose rows 10 to 20:

```
slice(Df, 10:20)
```

```
## # A tibble: 11 x 7
##    subject word    accuracy reaction_time valence length frequency
##      <int> <chr>      <int>         <int>   <dbl>  <int>     <dbl>
##  1       1 evil           1           507    3.23      4     28.8
##  2       1 face           1           524    6.39      4    350.
##  3       1 fat            1           516    2.28      3     46.1
##  4       1 foul           1           554    2.81      4     10.4
##  5       1 glass          1           519    4.75      5     98.6
##  6       1 grenade        1           771    3.6       7      1.94
##  7       1 hatred         1           538    1.98      6     10.5
##  8       1 heal           1           509    7.09      4      5.24
```

```
##  9       1 kettle          1           557    5.22      6      9.25
## 10       1 kick            1           494    4.31      4     23.2
## 11       1 kind            1           569    7.59      4    238.
```

and here we choose rows 10, 20, 30, 40-45.

```
slice(Df, c(10, 20, 30, 40:45)) #
```

```
## # A tibble: 9 x 7
##   subject word    accuracy reaction_time valence length frequency
##     <int> <chr>      <int>         <int>   <dbl>  <int>     <dbl>
## 1       1 evil           1           507    3.23      4     28.8
## 2       1 kind           1           569    7.59      4    238.
## 3       1 safe           1           462    7.07      4     69.7
## 4       1 toy            1           467    7         3      9.77
## 5       1 trust          1           537    6.68      5    102.
## 6       1 useful         1           521    7.14      6    101.
## 7       1 vehicle        1           507    6.27      7     42.2
## 8       1 village        1           517    5.92      7    113.
## 9       1 watch          1           475    5.78      5     95.6
```

and so on.

Filtering observations

Often, slicing is not the easiest ways to select our rows. In fact, it is best to use `slice` only when you know exactly the row indices of the rows you want to keep. For general situtations, it is best to use `filter`. For example, the following will allow us to select only those observations where the reaction times are less than 2000 milliseconds.

```
filter(Df, reaction_time < 2000)
```

```
## # A tibble: 3,885 x 7
##    subject word    accuracy reaction_time valence length frequency
##      <int> <chr>      <int>         <int>   <dbl>  <int>     <dbl>
##  1       1 alive          1           498    7.25      5     42.5
##  2       1 bandage        1           716    4.54      7      2.53
##  3       1 bright         1           559    7.5       6     55.4
##  4       1 carcass        1           564    3.34      7      1.4
##  5       1 cheer          1           538    8.1       5      7.81
##  6       1 coast          1           463    5.98      5     47.0
##  7       1 detail         1           486    5.55      6     62.2
##  8       1 devil          1           562    2.21      5     17.3
##  9       1 door           1           541    5.13      4    254.
## 10       1 evil           1           507    3.23      4     28.8
## # ... with 3,875 more rows
```

While this will allow us to select the observations where the reaction times are above 200 and below 2000 milliseconds.

```
filter(Df, reaction_time > 200 & reaction_time < 2000)
```

```
## # A tibble: 3,883 x 7
##    subject word    accuracy reaction_time valence length frequency
##      <int> <chr>      <int>         <int>   <dbl>  <int>     <dbl>
##  1       1 alive          1           498    7.25      5     42.5
##  2       1 bandage        1           716    4.54      7      2.53
```

```
##  3       1 bright          1           559    7.5       6     55.4
##  4       1 carcass         1           564    3.34      7      1.4
##  5       1 cheer           1           538    8.1       5      7.81
##  6       1 coast           1           463    5.98      5     47.0
##  7       1 detail          1           486    5.55      6     62.2
##  8       1 devil           1           562    2.21      5     17.3
##  9       1 door            1           541    5.13      4    254.
## 10       1 evil            1           507    3.23      4     28.8
## # ... with 3,873 more rows
```

We can also filter more than one variable simultaneously. For example, here we'll filter our those observations where the response was accurate (this is denoted by a value of `1`), the reaction time was between 250 and 750, and the length of the word was between 2 and 5.

```
filter(Df,
       accuracy == 1,
       reaction_time > 250 & reaction_time < 750,
       length %in% seq(2, 5))
```

```
## # A tibble: 1,947 x 7
##    subject word  accuracy reaction_time valence length frequency
##      <int> <chr>    <int>         <int>   <dbl>  <int>     <dbl>
##  1       1 alive        1           498    7.25       5     42.5
##  2       1 cheer        1           538    8.1        5      7.81
##  3       1 coast        1           463    5.98       5     47.0
##  4       1 devil        1           562    2.21       5     17.3
##  5       1 door         1           541    5.13       4    254.
##  6       1 evil         1           507    3.23       4     28.8
##  7       1 face         1           524    6.39       4    350.
##  8       1 fat          1           516    2.28       3     46.1
##  9       1 foul         1           554    2.81       4     10.4
## 10       1 glass        1           519    4.75       5     98.6
## # ... with 1,937 more rows
```

Sorting rows

The `arrange` function will sort rows. You just specify which columns to sort by. For example, to sort by `reaction_time`, you'd do:

```
arrange(Df, reaction_time)
```

```
## # A tibble: 3,908 x 7
##    subject word     accuracy reaction_time valence length frequency
##      <int> <chr>       <int>         <int>   <dbl>  <int>     <dbl>
##  1      53 table           0            38    5.22       5    202
##  2      51 shadow          0           157    4.35       6     31.0
##  3       6 neglect         1           268    2.63       7     12.0
##  4      51 safe            1           286    7.07       4     69.7
##  5      17 face            1           300    6.39       4    350.
##  6      84 interest        1           303    6.97       8    276.
##  7      98 idiot           0           310    3.16       5      6.49
##  8      17 kettle          1           313    5.22       6      9.25
##  9      17 table           1           316    5.22       5    202
## 10     100 writer          1           316    5.52       6     37.4
```

```
## # ... with 3,898 more rows
```

To sort by `length` first and then by `reaction_time`, do

```
arrange(Df, length, reaction_time)
```

```
## # A tibble: 3,908 x 7
##    subject word  accuracy reaction_time valence length frequency
##      <int> <chr>    <int>         <int>   <dbl>  <int>     <dbl>
##  1      10 cow          1           327    5.57      3     14.0
##  2     100 fun          1           330    8.37      3     51.2
##  3      13 fat          1           337    2.28      3     46.1
##  4      51 fat          1           338    2.28      3     46.1
##  5      68 hat          1           340    5.46      3     31.4
##  6      17 fat          1           347    2.28      3     46.1
##  7     100 cow          1           363    5.57      3     14.0
##  8      72 hat          1           364    5.46      3     31.4
##  9      10 hat          1           365    5.46      3     31.4
## 10     103 toy          1           366    7         3      9.77
## # ... with 3,898 more rows
```

You can sort in descending order by using the `desc` function around the variable name. For example, here we sort by reaction time for largest to smallest:

```
arrange(Df, desc(reaction_time))
```

```
## # A tibble: 3,908 x 7
##    subject word    accuracy reaction_time valence length frequency
##      <int> <chr>      <int>         <int>   <dbl>  <int>     <dbl>
##  1      41 heal           0          5049    7.09      4      5.24
##  2       9 carcass        1          4279    3.34      7      1.4
##  3      75 grenade        1          4047    3.6       7      1.94
##  4      12 fun            1          3840    8.37      3     51.2
##  5      10 wasp           1          3815    3.37      4      2.58
##  6      27 carcass        0          3748    3.34      7      1.4
##  7       8 trunk          1          3035    5.09      5      8.16
##  8      55 kind           1          3012    7.59      4    238.
##  9      82 alert          1          2745    6.2       5     16
## 10      88 wife           1          2639    6.33      4    171.
## # ... with 3,898 more rows
```

Adding new variables

The `mutate` function adds new variables. For example, let's say we want to add a new variable that is the logarithm of the frequency of the word. We would do this by

```
mutate(Df, log_frequency = log(frequency))
```

```
## # A tibble: 3,908 x 8
##    subject word    accuracy reaction_time valence length frequency
##      <int> <chr>      <int>         <int>   <dbl>  <int>     <dbl>
##  1       1 alive          1           498    7.25      5     42.5
##  2       1 bandage        1           716    4.54      7      2.53
##  3       1 bright         1           559    7.5       6     55.4
##  4       1 carcass        1           564    3.34      7      1.4
##  5       1 cheer          1           538    8.1       5      7.81
```

```
##  6       1 coast          1           463    5.98      5     47.0
##  7       1 detail         1           486    5.55      6     62.2
##  8       1 devil          1           562    2.21      5     17.3
##  9       1 door           1           541    5.13      4    254.
## 10       1 evil           1           507    3.23      4     28.8
##    log_frequency
##            <dbl>
##  1         3.75
##  2         0.928
##  3         4.01
##  4         0.336
##  5         2.06
##  6         3.85
##  7         4.13
##  8         2.85
##  9         5.54
## 10         3.36
## # ... with 3,898 more rows
```

The previous code appended the new `log_frequency` variable onto the end of the data-frame. If we use the same new for the new variable, we'll replace the old varibale, e.g.

```
mutate(Df, frequency = log(frequency))
```

```
## # A tibble: 3,908 x 7
##    subject word    accuracy reaction_time valence length frequency
##      <int> <chr>      <int>         <int>   <dbl>  <int>     <dbl>
##  1       1 alive          1           498    7.25      5     3.75
##  2       1 bandage        1           716    4.54      7     0.928
##  3       1 bright         1           559    7.5       6     4.01
##  4       1 carcass        1           564    3.34      7     0.336
##  5       1 cheer          1           538    8.1       5     2.06
##  6       1 coast          1           463    5.98      5     3.85
##  7       1 detail         1           486    5.55      6     4.13
##  8       1 devil          1           562    2.21      5     2.85
##  9       1 door           1           541    5.13      4     5.54
## 10       1 evil           1           507    3.23      4     3.36
## # ... with 3,898 more rows
```

If you want to create new variables and only keep the new variables, dropping the old ones, you can use `transmute`. For example, here we create three new variables, keep these and throw away the original variables:

```
transmute(Df,
          fast_rt = if_else(reaction_time < 500, 'fast', 'not.fast'),
          short_word = if_else(length <= 3, 'short', 'not.short'),
          frequency = log(frequency))
```

```
## # A tibble: 3,908 x 3
##    fast_rt  short_word frequency
##    <chr>    <chr>          <dbl>
##  1 fast     not.short      3.75
##  2 not.fast not.short      0.928
##  3 not.fast not.short      4.01
##  4 not.fast not.short      0.336
##  5 not.fast not.short      2.06
##  6 fast     not.short      3.85
```

```
##  7 fast     not.short      4.13
##  8 not.fast not.short      2.85
##  9 not.fast not.short      5.54
## 10 not.fast not.short      3.36
## # ... with 3,898 more rows
```

Summarizing your variables

You can summarize your variables using `summarize` (or `summarise` if you prefer British-English spellings):

```
summarise(Df,
          mean = mean(reaction_time),
          median = median(reaction_time),
          stdev = sd(reaction_time),
          n = n() # This gives counts
)
```

```
## # A tibble: 1 x 4
##    mean median stdev     n
##   <dbl>  <dbl> <dbl> <int>
## 1  576.    519  257.  3908
```

Often we want to produce summaries of our variables for different groups of observations. In this case, an obvious example is to group our observations according to whether the response for correct or not, and then produce summaries for each subset of data. The way to do this is with the `group_by` function combined with the `summarize` function. In particular, first you group, then you summarize. For example,

```
Df.tmp <- group_by(Df, accuracy) # Create a tmp Df, where the data are grouped
summarise(Df.tmp,
          mean = mean(reaction_time),
          median = median(reaction_time),
          stdev = sd(reaction_time),
          n = n()
)
```

```
## # A tibble: 2 x 5
##   accuracy  mean median stdev     n
##      <int> <dbl>  <int> <dbl> <int>
## 1        0  737.    580  673.    77
## 2        1  572.    518  240.  3831
```

The above code can be done on one line, and without the need for the temporary data-frame, by using a so-called *pipe*. The pipe is given by the command `%>%`. It takes the output from one function and passes it to another function. The above code using the pipe is

```
group_by(Df, accuracy) %>%
  summarise(mean = mean(reaction_time),
            median = median(reaction_time),
            stdev = sd(reaction_time),
            n = n()
  )
```

```
## # A tibble: 2 x 5
##   accuracy  mean median stdev     n
##      <int> <dbl>  <int> <dbl> <int>
## 1        0  737.    580  673.    77
```

```
## 2       1  572.    518  240.  3831
```

Combining operations with %>%

Often, when data wrangling, we want to repeatedly apply functions to our data-frame. The pipe can be very helpful when doing this. As an example, let's say we want to filter out the very fast and the very slow reaction times and the incorrect responses, and then group by subject identity, and calculate the mean reaction time per subject, and then sort by this. To do this, we would do

```
Df %>%
  filter(reaction_time > 250 & reaction_time < 1250,
         accuracy == 1) %>%
  group_by(subject) %>%
  summarise(mean_rt = mean(reaction_time)) %>%
  arrange(mean_rt)
```

```
## # A tibble: 78 x 2
##    subject mean_rt
##      <int>   <dbl>
##  1      17    425.
##  2     100    433.
##  3      44    450.
##  4       4    451.
##  5      68    452.
##  6      84    456.
##  7       2    460
##  8      29    461.
##  9       3    462.
## 10      53    463.
## # ... with 68 more rows
```

Converting wide to long, or long to wide

In a *tidy* data set, every column is a variable and every row is an observation. Often your data needs to be beaten to this shape.

Let's read in a wide format data, which is a commonly used by SPSS users.

```
(Df.wide <- read_csv('../data/widedata.csv'))
```

```
## # A tibble: 7 x 4
##   subject conditionA conditionB conditionC
##     <int>      <int>      <int>      <int>
## 1       1         11         14         15
## 2       2         11         12         11
## 3       3         15         11         11
## 4       4         17         11          8
## 5       5         13         13         19
## 6       6          7         10         14
## 7       7          8          9         10
```

This is fake data, but we'll pretend it gives the memory recall rate of each of 7 subjects in each of three experimental conditions. We can make this into a long, and tidy, format with `gather`. We need to specify

the columns to pull together and then the name, or `key` for the newly gathered variables, and then name of the values of these variables.

```
(Df.long <- gather(Df.wide, conditionA, conditionB, conditionC, key='condition', value='recall'))
```

```
## # A tibble: 21 x 3
##    subject condition  recall
##      <int> <chr>       <int>
##  1       1 conditionA     11
##  2       2 conditionA     11
##  3       3 conditionA     15
##  4       4 conditionA     17
##  5       5 conditionA     13
##  6       6 conditionA      7
##  7       7 conditionA      8
##  8       1 conditionB     14
##  9       2 conditionB     12
## 10       3 conditionB     11
## # ... with 11 more rows
```

The opposite of a `gather` is a `spread`. This converts a long to a wide format. To illustrate, we'll just go backwards from `D.long` to `Df.wide`. Here, we need only state the variable to "spread" and which variable's values to use as the values of the newly spread variables.

```
spread(Df.long, key=condition, value=recall)
```

```
## # A tibble: 7 x 4
##   subject conditionA conditionB conditionC
##     <int>      <int>      <int>      <int>
## 1       1         11         14         15
## 2       2         11         12         11
## 3       3         15         11         11
## 4       4         17         11          8
## 5       5         13         13         19
## 6       6          7         10         14
## 7       7          8          9         10
```

Combining and merging data frames

For these examples, we'll first read in some new data sets:

```
lexicon_A <- read_csv('../data/lexiconA.csv')
lexicon_B <- read_csv('../data/lexiconB.csv')
lexicon_C <- read_csv('../data/lexiconC.csv')
behav_data <- read_csv('../data/data.csv')
```

The data frames `lexicon_A` and `lexicon_C` have the same column names and so we can stack them on top of each other:

```
bind_rows(lexicon_A, lexicon_C) #
```

```
## # A tibble: 7 x 3
##   word    length pos
##   <chr>    <int> <chr>
## 1 dog          3 noun
## 2 walk         4 verb
```

```
## 3 happy        5 adj
## 4 quickly      7 adv
## 5 dragon       6 noun
## 6 cat          3 noun
## 7 mouse        5 noun
```

The data frames `lexicon_A` and `behav_data` have the same number of rows, so we can stack them side by side:

```
bind_cols(lexicon_A, behav_data)
```

```
## # A tibble: 5 x 5
##   word    length pos   reaction.time accuracy
##   <chr>    <int> <chr>         <int>    <int>
## 1 dog          3 noun            200        1
## 2 walk         4 verb            300        0
## 3 happy        5 adj             450        1
## 4 quickly      7 adv             500        0
## 5 dragon       6 noun            345        1
```

A more interesting case is where we want to merge values from two data frames according to common variables. The data frames `lexicon_A` and `lexicon_B` have a common variable, i.e. `word`. We can merge them by this common variable:

```
inner_join(lexicon_A, lexicon_B, by='word')
```

```
## # A tibble: 5 x 4
##   word    length pos   valence
##   <chr>    <int> <chr>   <int>
## 1 dog          3 noun        3
## 2 walk         4 verb        3
## 3 happy        5 adj         7
## 4 quickly      7 adv         4
## 5 dragon       6 noun        1
```

Note that this will drop all rows of `lexicon_A` that do not have matching `word` in `lexicon_B`, and vice versa. As it happens, all rows in `lexicon_A` do have a matching `word` in `lexicon_B`, but all rows in `lexicon_B` do not have a matching `word` in `lexicon_A`. If we want to include all rows in `lexicon_B` regardless, we could do:

```
right_join(lexicon_A, lexicon_B, by='word')
```

```
## # A tibble: 8 x 4
##   word    length pos   valence
##   <chr>    <int> <chr>   <int>
## 1 dog          3 noun        3
## 2 dragon       6 noun        1
## 3 money       NA <NA>        7
## 4 walk         4 verb        3
## 5 happy        5 adj         7
## 6 quickly      7 adv         4
## 7 baby        NA <NA>        1
## 8 kitten      NA <NA>        7
```

Note how we include missing values for the `length` and `pos` of those words in `lexicon_B` that are not in `lexicon_A`.