Working with data frames

Mark Andrews

October 9, 2017

```r
library(dplyr)
library(readr)
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 be done from the RStudio `Import Dataset` menu in `Environment`):

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

```
## # A tibble: 3,908 × 7
##    subject    item accuracy latency valence length frequency
##      <int>   <chr>    <int>   <int>   <dbl>  <int>     <dbl>
## 1        1   alive        1     498    7.25      5     42.54
## 2        1 bandage        1     716    4.54      7      2.53
## 3        1  bright        1     559    7.50      6     55.40
## 4        1 carcass        1     564    3.34      7      1.40
## 5        1   cheer        1     538    8.10      5      7.81
## 6        1   coast        1     463    5.98      5     47.01
## 7        1  detail        1     486    5.55      6     62.23
## 8        1   devil        1     562    2.21      5     17.32
## 9        1    door        1     541    5.13      4    253.65
## 10       1    evil        1     507    3.23      4     28.83
## # ... with 3,898 more rows
```

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

Quick summary of your data frame

```r
summary(Df)
```

```
##     subject           item              accuracy         latency
##  Min.   :  1.00   Length:3908        Min.   :0.0000   Min.   :  38.0
##  1st Qu.: 20.00   Class :character   1st Qu.:1.0000   1st Qu.: 458.0
##  Median : 46.50   Mode  :character   Median :1.0000   Median : 519.0
##  Mean   : 49.45                      Mean   :0.9803   Mean   : 575.6
##  3rd Qu.: 77.00                      3rd Qu.:1.0000   3rd Qu.: 609.0
##  Max.   :105.00                      Max.   :1.0000   Max.   :5049.0
##     valence          length        frequency
##  Min.   :1.850   Min.   :3.000   Min.   :  0.33
##  1st Qu.:3.320   1st Qu.:4.000   1st Qu.:  5.83
##  Median :5.220   Median :5.000   Median : 16.00
##  Mean   :5.016   Mean   :5.353   Mean   : 57.31
##  3rd Qu.:6.770   3rd Qu.:6.000   3rd Qu.: 64.90
##  Max.   :8.370   Max.   :9.000   Max.   :590.31
```

Rename variable names

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

```
(Df <- rename(Df,
              word = item,
              reaction.time = latency))
```

```
## # A tibble: 3,908 × 7
##    subject    word accuracy reaction.time valence length frequency
##      <int>   <chr>    <int>         <int>   <dbl>  <int>     <dbl>
## 1        1   alive        1           498    7.25      5     42.54
## 2        1 bandage        1           716    4.54      7      2.53
## 3        1  bright        1           559    7.50      6     55.40
## 4        1 carcass        1           564    3.34      7      1.40
## 5        1   cheer        1           538    8.10      5      7.81
## 6        1   coast        1           463    5.98      5     47.01
## 7        1  detail        1           486    5.55      6     62.23
## 8        1   devil        1           562    2.21      5     17.32
## 9        1    door        1           541    5.13      4    253.65
## 10       1    evil        1           507    3.23      4     28.83
## # ... 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 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 × 4
##    subject    word accuracy reaction.time
##      <int>   <chr>    <int>         <int>
## 1        1   alive        1           498
## 2        1 bandage        1           716
## 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
```

[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.

```
## 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:

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

```
## # A tibble: 3,908 × 6
##    subject    word accuracy reaction.time length frequency
##      <int>   <chr>    <int>         <int>  <int>     <dbl>
## 1        1   alive        1           498      5     42.54
## 2        1 bandage        1           716      7      2.53
## 3        1  bright        1           559      6     55.40
## 4        1 carcass        1           564      7      1.40
## 5        1   cheer        1           538      5      7.81
## 6        1   coast        1           463      5     47.01
## 7        1  detail        1           486      6     62.23
## 8        1   devil        1           562      5     17.32
## 9        1    door        1           541      4    253.65
## 10       1    evil        1           507      4     28.83
## # ... with 3,898 more rows
```

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

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

```
## # A tibble: 3,908 × 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.

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:

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

```
## # A tibble: 3,908 × 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    7.50      6
## 4        1 carcass        1           564    3.34      7
## 5        1   cheer        1           538    8.10      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

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

```
## # A tibble: 3,908 × 7
##    frequency subject    word accuracy reaction.time valence length
##        <dbl>   <int>   <chr>    <int>         <int>   <dbl>  <int>
## 1      42.54       1   alive        1           498    7.25      5
## 2       2.53       1 bandage        1           716    4.54      7
## 3      55.40       1  bright        1           559    7.50      6
## 4       1.40       1 carcass        1           564    3.34      7
## 5       7.81       1   cheer        1           538    8.10      5
## 6      47.01       1   coast        1           463    5.98      5
## 7      62.23       1  detail        1           486    5.55      6
## 8      17.32       1   devil        1           562    2.21      5
## 9     253.65       1    door        1           541    5.13      4
## 10     28.83       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:

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

```
## # A tibble: 11 × 7
##    subject    word accuracy reaction.time valence length frequency
##      <int>   <chr>    <int>         <int>   <dbl>  <int>     <dbl>
## 1        1    evil        1           507    3.23      4     28.83
## 2        1    face        1           524    6.39      4    349.78
## 3        1     fat        1           516    2.28      3     46.11
## 4        1    foul        1           554    2.81      4     10.43
## 5        1   glass        1           519    4.75      5     98.56
## 6        1 grenade        1           771    3.60      7      1.94
## 7        1  hatred        1           538    1.98      6     10.52
## 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     23.24
## 11       1    kind        1           569    7.59      4    237.97
```

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

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

```
## # A tibble: 9 × 7
##   subject    word accuracy reaction.time valence length frequency
##     <int>   <chr>    <int>         <int>   <dbl>  <int>     <dbl>
## 1       1    evil        1           507    3.23      4     28.83
## 2       1    kind        1           569    7.59      4    237.97
## 3       1    safe        1           462    7.07      4     69.73
## 4       1     toy        1           467    7.00      3      9.77
## 5       1   trust        1           537    6.68      5    101.98
## 6       1  useful        1           521    7.14      6    100.71
## 7       1 vehicle        1           507    6.27      7     42.23
## 8       1 village        1           517    5.92      7    113.40
## 9       1   watch        1           475    5.78      5     95.57
```

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.

```r
filter(Df, reaction.time < 2000)
```

```
## # A tibble: 3,885 × 7
##    subject    word accuracy reaction.time valence length frequency
##      <int>   <chr>    <int>         <int>   <dbl>  <int>     <dbl>
## 1        1   alive        1           498    7.25      5     42.54
## 2        1 bandage        1           716    4.54      7      2.53
## 3        1  bright        1           559    7.50      6     55.40
## 4        1 carcass        1           564    3.34      7      1.40
## 5        1   cheer        1           538    8.10      5      7.81
## 6        1   coast        1           463    5.98      5     47.01
## 7        1  detail        1           486    5.55      6     62.23
## 8        1   devil        1           562    2.21      5     17.32
## 9        1    door        1           541    5.13      4    253.65
## 10       1    evil        1           507    3.23      4     28.83
## # ... 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.

```r
filter(Df, reaction.time > 200 & reaction.time < 2000)
```

```
## # A tibble: 3,883 × 7
##    subject    word accuracy reaction.time valence length frequency
##      <int>   <chr>    <int>         <int>   <dbl>  <int>     <dbl>
## 1        1   alive        1           498    7.25      5     42.54
## 2        1 bandage        1           716    4.54      7      2.53
## 3        1  bright        1           559    7.50      6     55.40
## 4        1 carcass        1           564    3.34      7      1.40
## 5        1   cheer        1           538    8.10      5      7.81
## 6        1   coast        1           463    5.98      5     47.01
## 7        1  detail        1           486    5.55      6     62.23
## 8        1   devil        1           562    2.21      5     17.32
## 9        1    door        1           541    5.13      4    253.65
## 10       1    evil        1           507    3.23      4     28.83
## # ... 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.

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

```
## # A tibble: 1,947 × 7
##    subject  word accuracy reaction.time valence length frequency
##      <int> <chr>    <int>         <int>   <dbl>  <int>     <dbl>
## 1        1 alive        1           498    7.25      5     42.54
## 2        1 cheer        1           538    8.10      5      7.81
## 3        1 coast        1           463    5.98      5     47.01
## 4        1 devil        1           562    2.21      5     17.32
## 5        1  door        1           541    5.13      4    253.65
## 6        1  evil        1           507    3.23      4     28.83
## 7        1  face        1           524    6.39      4    349.78
## 8        1   fat        1           516    2.28      3     46.11
## 9        1  foul        1           554    2.81      4     10.43
## 10       1 glass        1           519    4.75      5     98.56
## # ... 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:

```r
arrange(Df, reaction.time)
```

```
## # A tibble: 3,908 × 7
##    subject     word accuracy reaction.time valence length frequency
##      <int>    <chr>    <int>         <int>   <dbl>  <int>     <dbl>
## 1       53    table        0            38    5.22      5    202.00
## 2       51   shadow        0           157    4.35      6     30.95
## 3        6  neglect        1           268    2.63      7     12.05
## 4       51     safe        1           286    7.07      4     69.73
## 5       17     face        1           300    6.39      4    349.78
## 6       84 interest        1           303    6.97      8    276.11
## 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    202.00
## 10     100   writer        1           316    5.52      6     37.42
## # ... with 3,898 more rows
```

To sort by `length` first and then by `reaction.time`, do

```r
arrange(Df, length, reaction.time)
```

```
## # A tibble: 3,908 × 7
##    subject  word accuracy reaction.time valence length frequency
##      <int> <chr>    <int>         <int>   <dbl>  <int>     <dbl>
## 1       10   cow        1           327    5.57      3     13.96
## 2      100   fun        1           330    8.37      3     51.21
## 3       13   fat        1           337    2.28      3     46.11
## 4       51   fat        1           338    2.28      3     46.11
## 5       68   hat        1           340    5.46      3     31.37
## 6       17   fat        1           347    2.28      3     46.11
## 7      100   cow        1           363    5.57      3     13.96
## 8       72   hat        1           364    5.46      3     31.37
## 9       10   hat        1           365    5.46      3     31.37
## 10     103   toy        1           366    7.00      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:

```r
arrange(Df, desc(reaction.time))
```

```
## # A tibble: 3,908 × 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      1.40
## 3       75 grenade        1          4047    3.60      7      1.94
## 4       12     fun        1          3840    8.37      3     51.21
## 5       10    wasp        1          3815    3.37      4      2.58
## 6       27 carcass        0          3748    3.34      7      1.40
## 7        8   trunk        1          3035    5.09      5      8.16
## 8       55    kind        1          3012    7.59      4    237.97
## 9       82   alert        1          2745    6.20      5     16.00
## 10      88    wife        1          2639    6.33      4    171.06
## # ... 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 × 8
##    subject    word accuracy reaction.time valence length frequency
##      <int>   <chr>    <int>         <int>   <dbl>  <int>     <dbl>
## 1        1   alive        1           498    7.25      5     42.54
## 2        1 bandage        1           716    4.54      7      2.53
## 3        1  bright        1           559    7.50      6     55.40
## 4        1 carcass        1           564    3.34      7      1.40
## 5        1   cheer        1           538    8.10      5      7.81
## 6        1   coast        1           463    5.98      5     47.01
## 7        1  detail        1           486    5.55      6     62.23
## 8        1   devil        1           562    2.21      5     17.32
## 9        1    door        1           541    5.13      4    253.65
## 10       1    evil        1           507    3.23      4     28.83
##    log.frequency
##            <dbl>
## 1      3.7504448
## 2      0.9282193
## 3      4.0145796
## 4      0.3364722
## 5      2.0554050
## 6      3.8503603
## 7      4.1308372
## 8      2.8518619
## 9      5.5359554
## 10     3.3614165
## # ... 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 × 7
##    subject    word accuracy reaction.time valence length frequency
##      <int>   <chr>    <int>         <int>   <dbl>  <int>     <dbl>
## 1        1   alive        1           498    7.25      5 3.7504448
## 2        1 bandage        1           716    4.54      7 0.9282193
## 3        1  bright        1           559    7.50      6 4.0145796
## 4        1 carcass        1           564    3.34      7 0.3364722
## 5        1   cheer        1           538    8.10      5 2.0554050
## 6        1   coast        1           463    5.98      5 3.8503603
## 7        1  detail        1           486    5.55      6 4.1308372
## 8        1   devil        1           562    2.21      5 2.8518619
## 9        1    door        1           541    5.13      4 5.5359554
## 10       1    evil        1           507    3.23      4 3.3614165
## # ... 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 × 3
##     fast.rt short.word frequency
##       <chr>      <chr>     <dbl>
## 1      fast  not.short 3.7504448
## 2  not.fast  not.short 0.9282193
## 3  not.fast  not.short 4.0145796
## 4  not.fast  not.short 0.3364722
## 5  not.fast  not.short 2.0554050
## 6      fast  not.short 3.8503603
## 7      fast  not.short 4.1308372
## 8  not.fast  not.short 2.8518619
## 9  not.fast  not.short 5.5359554
## 10 not.fast  not.short 3.3614165
## # ... 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 × 4
##       mean median    stdev     n
##      <dbl>  <dbl>    <dbl> <int>
## 1 575.5983    519 256.5554  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
summarize(Df.tmp,
          mean = mean(reaction.time),
          median = median(reaction.time),
          stdev = sd(reaction.time),
          n = n()
)
```

```
## # A tibble: 2 × 5
##   accuracy     mean median    stdev     n
##      <int>    <dbl>  <int>    <dbl> <int>
## 1        0 737.1688    580 673.0910    77
## 2        1 572.3508    518 240.0386  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) %>%
  summarize(mean = mean(reaction.time),
            median = median(reaction.time),
            stdev = sd(reaction.time),
            n = n()
  )
```

```
## # A tibble: 2 × 5
##   accuracy     mean median    stdev     n
##      <int>    <dbl>  <int>    <dbl> <int>
## 1        0 737.1688    580 673.0910    77
## 2        1 572.3508    518 240.0386  3831
```

Combining operations with %>%

Often, when data wrangling, we want to combine 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 × 2
##    subject  mean.rt
##      <int>    <dbl>
## 1       17 424.5556
## 2      100 432.5660
## 3       44 450.3333
## 4        4 451.4815
## 5       68 451.5370
## 6       84 455.7170
## 7        2 460.0000
## 8       29 461.4348
## 9        3 462.4444
## 10      53 463.4444
## # ... with 68 more rows
```