Doing it all again, but tidy

from Doing LVC with *R*[*]

Matt Hunt Gardner

2022-09-27

Table of contents

Doing It All Again, But tidy

The package dplyr is part of a larger "universe" of *R* packages called tidyverse. This collection of packages is specifically focused on data science and offers some shortcuts that are useful to learn. The packages that make up the tidyverse are dplyr, ggplot2, purr, tibble, tidyr, stingr, readr, and forcats, among others. Throughout this guide I try to use the most basic *R* syntax for accomplishing a task. This way you learn how *R* works. I will also show how to complete the same task using packages from the tidyverse. Using the tidyverse methods is usually optional — though once you get the hang of it, you might always use the tidyverse methods.

```r
# Install the tidyverse package
install.packages("tidyverse")
```

```r
# Load the tidyverse package
library(tidyverse)
```

```r
# List the packages loaded by the tidyverse
# package
tidyverse_packages()
```

```
 [1] "broom"         "cli"           "crayon"        "dbplyr"
 [5] "dplyr"         "dtplyr"        "forcats"       "ggplot2"
 [9] "googledrive"   "googlesheets4" "haven"         "hms"
[13] "httr"          "jsonlite"      "lubridate"     "magrittr"
[17] "modelr"        "pillar"        "purrr"         "readr"
[21] "readxl"        "reprex"        "rlang"         "rstudioapi"
[25] "rvest"         "stringr"       "tibble"        "tidyr"
[29] "xml2"          "tidyverse"
```

[*] https://lingmethodshub.github.io/content/R/lvc_r/

Before we get started with the tidyverse, there are two important new things to learn about. The first is the pipe operator %>% and the second is the the alternative to a *data frame* called a *tibble*.

The Pipe %>%

The pipe operator %>%[1] is introduced by the magrittr package[2] and it is extremely useful. The pipe operator passes the output of a function to the first argument of the next function, which mean you can chain several steps together.

For example, lets find the mean year of birth in our data. We already know that when the pre-vowel contexts are removed, the mean year of birth is 1969.

Get the data first

If you don't have the td data loaded in *R*, go back to Getting Your Data into *R*[a] and run the code.

[a]https://lingmethodshub.github.io/content/R/lvc_r/020_lvcr.html

```
# Find mean YOB using mean() function
mean(td$YOB)
```

```
[1] 1969.447
```

```
# Find the mean YOB by piping the td data to the
# mean() function
td$YOB %>%
    mean()
```

```
[1] 1969.447
```

The functionality of %>% might seem trivial at this point; however, when you need to perform multiple tasks sequentially, it saves a lot of time and space when writing your code.

Tibbles

A *tibble* is an updated version of a *data frame*. *Tibbles* keep the features that have stood the test of time, and drop the features that used to be convenient but are now frustrating (i.e. converting character vectors to factors). For our purposes, the difference between the two is negligible, but you should be aware that *tibbles* look a bit different from *data frames*. Run these two commands and compare.

```
as.data.frame(td)
```

```
as_tibble(td)
```

Notice that the *tibble* lists the dimensions of the tibble at the top, as well as the class of each of the columns. It also only displays the first 10 rows. You'll also notice that the row numbers have reset when we converted td to a *tibble*. If we want to view the entire tibble, we can use the print() function and specify the n= plus the number of rows we want to see, including all rows (n=Inf). You can see below how the pipe operator makes doing this pretty easy.

[1]Not to be confused with the operator |, which means "or" and whose symbol is also called "pipe".
[2]Loading dplyr will also let you use it.

```r
# Embedding functions
print(as_tibble(td), n = 20)
```

The above produces the same as the following:

```r
# Using %>% to pass the results from the first
# function to the second function
as_tibble(td) %>%
    print(n = 20)
```

```
# A tibble: 1,189 x 17
  Dep.Var  Stress   Category Morph.Type Before After     Speaker  YOB Sex  Education   Job     After.New Cent
  <chr>    <chr>    <chr>    <chr>      <fct>  <chr>     <chr>  <int> <chr> <chr>      <chr>   <fct>         <dbl>
 1 Realized Stressed Lexical  Mono       Stop   Consonant BOUF65  1965 F    Educated    White   Consonant      -
 2 Deletion Stressed Lexical  Mono       Stop   Consonant CHIF55  1955 F    Educated    White   Consonant     -1
 3 Deletion Stressed Lexical  Mono       Stop   Consonant CHIF55  1955 F    Educated    White   Consonant     -1
 4 Deletion Stressed Lexical  Mono       Stop   Consonant CLAF52  1952 F    Educated    Service Consonant      -
 5 Realized Stressed Lexical  Mono       Stop   Consonant DONM53  1953 M    Educated    Service Consonant      -
 6 Deletion Stressed Lexical  Mono       Stop   Consonant DONM58  1958 M    Not Educated Service Consonant
 7 Deletion Stressed Lexical  Mono       Stop   Consonant DOUF46  1946 F    Educated    Service Consonant      -
 8 Deletion Stressed Lexical  Mono       Stop   Consonant GARM42  1942 M    Not Educated Blue    Consonant      -
 9 Deletion Stressed Lexical  Mono       Stop   Consonant GREM45  1945 M    Not Educated Blue    Consonant      -
10 Deletion Stressed Lexical  Mono       Stop   Consonant HOLF49  1949 F    Educated    Service Consonant      -
11 Deletion Stressed Lexical  Mono       Stop   Consonant HOLM52  1952 M    Not Educated Blue    Consonant      -
12 Deletion Stressed Lexical  Mono       Stop   Consonant INGM84  1984 M    Educated    Service Consonant
13 Deletion Stressed Lexical  Mono       Stop   Consonant INGM87  1987 M    Educated    Service Consonant
14 Deletion Stressed Lexical  Mono       Stop   Consonant KAYF29  1929 F    Not Educated Service Consonant
15 Deletion Stressed Lexical  Mono       Stop   Consonant KAYM29  1929 M    Not Educated Blue    Consonant      -
16 Realized Stressed Lexical  Mono       Stop   Consonant LATF53  1953 F    Educated    Service Consonant      -
17 Realized Stressed Lexical  Mono       Stop   Consonant LEOF66  1966 F    Educated    White   Consonant      -
18 Deletion Stressed Lexical  Mono       Stop   Consonant MOFM55  1955 M    Educated    White   Consonant     -1
19 Deletion Stressed Lexical  Mono       Stop   Consonant NATF84  1984 F    Educated    Service Consonant
20 Deletion Stressed Lexical  Mono       Stop   Consonant NEIF49  1949 F    Educated    Service Consonant      -
# ... with 1,169 more rows
# i Use `print(n = ...)` to see more rows
```

Getting a glimpse()

Another useful addition to data exploration is the glimpse() function from the pilllar package and re-exported by dplyr. The glipmpse() function is like a cross between print() (which shows the data) and str() (which shows the structure of the data). I use glimpse() almost as frequently as I use summary(). In fact, if you have very wide data, i.e., with lots of columns, glimpse() may prove more useful than summary() for getting a quick snapshot of your data. glimpse() shows the number of rows, the number of columns, the name of each column, its class, and however many values in each column as will fit horizontally in the console.

```r
glimpse(td)
```

```
Rows: 1,189
Columns: 17
$ Dep.Var      <chr> "Realized", "Deletion", "Deletion", "Deletion", "Realized~
$ Stress       <chr> "Stressed", "Stressed", "Stressed", "Stressed", "Stressed~
$ Category     <chr> "Lexical", "Lexical", "Lexical", "Lexical", "Lexical", "L~
$ Morph.Type   <chr> "Mono", "Mono", "Mono", "Mono", "Mono", "Mono", "Mono", "~
$ Before       <fct> Stop, Stop, Stop, Stop, Stop, Stop, Stop, Stop, Stop, Sto~
$ After        <chr> "Consonant", "Consonant", "Consonant", "Consonant", "Cons~
$ Speaker      <chr> "BOUF65", "CHIF55", "CHIF55", "CLAF52", "DONM53", "DONM58~
$ YOB          <int> 1965, 1955, 1955, 1952, 1953, 1958, 1946, 1942, 1945, 194~
$ Sex          <chr> "F", "F", "F", "F", "M", "M", "F", "M", "M", "F", "M", "M~
$ Education    <chr> "Educated", "Educated", "Educated", "Educated", "Educated~
$ Job          <chr> "White", "White", "White", "Service", "Service", "Service~
$ After.New    <fct> Consonant, Consonant, Consonant, Consonant, Consonant, Co~
$ Center.Age   <dbl> -4.446594, -14.446594, -14.446594, -17.446594, -16.446594~
$ Age.Group    <fct> Middle, Middle, Middle, Middle, Middle, Middle, Middle, O~
$ Age_Sex      <fct> Middle_F, Middle_F, Middle_F, Middle_F, Middle_M, Middle_~
$ Phoneme      <fct> t, t, t, t, t, t, t, t, t, t, t, t, t, t, t, t, t, t, t, ~
$ Dep.Var.Full <fct> T, Deletion, Deletion, Deletion, T, Deletion, Deletion, D~
```

Manipulating data with dplyr

The dplyr package is great for manipulating data in a data frame/tibble. Some common things that diplyr can do include:

Function	Description
mutate()	add new variables or modify existing ones
select()	select variables
filter()	filter
summarize()	summarize/reduce
arrange()	sort
group_by()	group
rename()	rename columns

Lets redo all our data manipulation of td but with dplyr and its pipe %>% operator

```r
# Read in token file
td <- read.delim("Data/deletiondata.txt")
```

or…

```r
# Read in token file
td <- read.delim("https://www.dropbox.com/s/jxlfuogea3lx2pu/deletiondata.txt?dl=1")
```

then…

```r
# Subset data to remove previous 'Vowel'
# contexts: filter td to include everything that
# is not 'Vowel' in the column Before
td <- td %>%
    filter(Before != "Vowel")

# Re-code 'H' to be 'Consonant' in a new column:
# create a new column called After.New that
# equals a re-code of After in which H is
# re-coded as Consonant
td <- td %>%
    mutate(After.New = recode(After, H = "Consonant"))

# Center Year of Birth: create a new column
# called Center.Age equal to the YOB column but
# scaled
td <- td %>%
    mutate(Center.Age = as.numeric(scale(YOB, scale = FALSE)))

# Create Age.Group: cut YOB into discrete
# categories.
td <- td %>%
    mutate(Age.Group = cut(YOB, breaks = c(-Inf, 1944,
        1979, Inf), labels = c("Old", "Middle", "Young")))
```

Before we continue, a note about the `cut()` function. The `breaks=` option is a concatenated list of boundaries. It should start and end with `-Inf` and `Inf` (negative and positive infinity) as these will be the lower and upper bounds. The other values are the boundaries or cut-off points. By default `cut()` has the setting `right=TRUE`, which means the boundary values are considered the last value in a group (e.g., rightmost value). Above, this means `1944` will be the highest value in the `Old` category and `1979` will the the highest value in the `Middle` category. To reverse this you can add the option `right=FALSE` in which case 1944 would be the lowest value in the `Middle` category (e.g. leftmost value) and 1979 would be the lowest value in the `Young` category.

Let's continue.

```
# Combine Age and Sex: use the unite() function
# from the tidyr package, if remove=TRUE the
# original Age.Group and Sex columns will be
# deleted
td <- td %>%
    unite("Age_Sex", c(Age.Group, Sex), sep = "_",
        remove = FALSE)

# Break Phoneme.Dep.Var into two columns: same as
# before, but with td passed to mutate() by the
# %>% operator
td <- td %>%
    mutate(Phoneme = sub("^(.)(--.*)$", "\\1", Phoneme.Dep.Var),
        Dep.Var.Full = sub("^(.--)(.*)$", "\\2", Phoneme.Dep.Var),
        Phoneme.Dep.Var = NULL)
```

At this point we have done everything except partition the data and re-center YOB in the partitioned data frames. You may ask, "How is this better?". Well, the answer is that because all these modifications feed into one another, we can actually include them all together in one serialized operation. Behold!

All of the above code can be simplified as follows:

or…

```
# Read in token file
td <- read.delim("https://www.dropbox.com/s/jxlfuogea3lx2pu/deletiondata.txt?dl=1")
```

then…

```
# Subset data to remove previous 'Vowel' contexts,
# then modify several columns with mutate,
# then convert any character column to a factor column
td <- td %>%
      filter(Before != "Vowel")%>%
      mutate(
        After.New = recode(After, "H" = "Consonant"),
        Center.Age = as.numeric(scale(YOB, scale = FALSE)),
        Age.Group = cut(YOB, breaks = c(-Inf, 1944, 1979, Inf),
                        labels = c("Old", "Middle", "Young")),
        Phoneme = sub("^(.)(--.*)$", "\\1", Phoneme.Dep.Var),
        Dep.Var.Full = sub("^(.--)(.*)$", "\\2", Phoneme.Dep.Var),
        Phoneme.Dep.Var = NULL
        )%>%
      mutate_if(is.character, as.factor)
```

Now, doesn't the above look so much cleaner and easier to follow? You'll notice that after some lines there is a #. This an optional way to signal the end of a line of code when your code is broken over more than one line. Above, the mutate() function could have been written in one single continuous line, but breaking it up over multiple lines makes seeing each mutation much easier.

To partition the data we still need separate functions. Also, remember to re-centre any continuous variables after partioning.

```
td.young <- td %>%
    filter(Age.Group == "Young") %>%
    mutate(Center.Age = as.numeric(scale(YOB, scale = FALSE)))

td.middle <- td %>%
    filter(Age.Group == "Middle") %>%
    mutate(Center.Age = as.numeric(scale(YOB, scale = FALSE)))

td.old <- td %>%
    filter(Age.Group == "Old") %>%
    mutate(Center.Age = as.numeric(scale(YOB, scale = FALSE)))
```