Modifying Data

from Doing LVC with *R*[*]

Matt Hunt Gardner

2022-09-27

Table of contents

If you followed the previous section you now have an object in *R* called `td` in which all character columns are converted to factors. If not, you can load it now with either of the following codes.

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

```r
td <- read.delim("Data/deletiondata.txt")
```

Re-class character columns as factors.

```r
td$Dep.Var <- factor(td$Dep.Var)
td$Stress <- factor(td$Stress)
td$Category <- factor(td$Category)
td$Morph.Type <- factor(td$Morph.Type)
td$Before <- factor(td$Before)
td$After <- factor(td$After)
td$Speaker <- factor(td$Speaker)
td$Sex <- factor(td$Sex)
td$Education <- factor(td$Education)
td$Job <- factor(td$Job)
td$Phoneme.Dep.Var <- factor(td$Phoneme.Dep.Var)
```

Deleting tokens, re-coding tokens, combining factor groups, etc. is, in my opinion, easier to do in *Excel*. But, if you want to do this in *R*, or you only want to perform the modification for a specific analysis or graph, you may find the following functions useful.

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

Removing Rows

The (t, d) deletion data includes tokens in which the previous phoneme is a vowel. Many analyses of (t, d) deletion exclude this context. Here is how to remove these contexts from your data frame.

```
td <- td[td$Before != "Vowel", ]
summary(td)
```

```
     Dep.Var             Stress          Category         Morph.Type
 Deletion:386   Stressed   :1047   Function:  57   Mono     :762
 Realized:803   Unstressed: 142   Lexical :1132   Past     :311
                                                   Semi-Weak:116




             Before              After            Speaker            YOB         Sex
 Liquid         :269   Consonant:215   INGM84  : 57   Min.   :1915   F:659
 Nasal          :209   H        :157   MARM92  : 53   1st Qu.:1952   M:530
 Other Fricative:130   Pause    :558   GARF87  : 52   Median :1984
 S              :332   Vowel    :259   HUNF22  : 32   Mean   :1969
 Stop           :249                   DONF15  : 28   3rd Qu.:1991
 Vowel          :  0                   GARF37  : 28   Max.   :1999
                                       (Other):939
        Education         Job            Phoneme.Dep.Var
 Educated     :496   Blue   :130   t--T          :352
 Not Educated:317   Service:468   t--Deletion  :296
 Student      :376   Student:376   t--Fricative:131
                     White  :215   d--Deletion  : 90
                                   d--T          : 73
                                   d--D          : 46
                                   (Other)       :201
```

The first line of code creates a new object called `td`. The `<-` operator means you're telling *R* that this new `td` is the same as the old `td`, but filtered according to some specific condition. You could also use the `<-` operator to create a new `td` with a different name — e.g., `td.new <- td[td\$Before != "Vowel",]}` — thus giving you two data frames (`td` and `td.new`) to work with.

Throughout *R*, square brackets `[ ]` are used for specifying filtering conditions. The first line of code therefore says make new `td` the same as old `td`, but only where the value in the `Before` column of old `td` (indicated by `td$Before`) does not equal `"Vowel"`. `!=` is the standard operator meaning *does not equal*. The comma after `"Vowel"` is important. If you are filtering a data frame, as you are here, *R* needs to know where to look for the values you want to keep or throw out. It follows this format: `data frame[rows,columns]`. So the comma in the above indicates that the filtering condition relates to values found while searching row by row. It selects any row in which the value in the `Before` column does not equal `"Vowel"`. The quotation marks around `"Vowel"` are also important because the values in the `Before` column are all strings of characters, which (you'll remember from above), are always enclosed in quotation marks.

```
td$Before <- factor(td$Before)
```

An additional line of code is needed for resetting the data structure. The new `td` inherits the data structure from the old `td`. This means *R* thinks the new `td` still has a level in the column `Before` called `"Vowel"` — even though there are zero tokens now in the column with this value. If you run `summary(td)` after the first code above you'll see that *R* still lists `"Vowel"` as a possible level of `Before`, but with zero instances. The additional line of code tells *R* to make a new column `Before` in the data frame `td` (this will replace the old Before column) in which the possible factors (i.e., values) in that column are those that actually exist in the new Before column in td. Run summary() or str() now and you'll see that the empty "Vowel" level is gone.

```
summary(td)
```

```
     Dep.Var            Stress         Category        Morph.Type 
 Deletion:386   Stressed  :1047   Function:  57   Mono     :762  
 Realized:803   Unstressed: 142   Lexical :1132   Past     :311  
                                                  Semi-Weak:116  
                                                                 
                                                                 
                                                                 
                                                                 
            Before          After          Speaker         YOB          Sex    
 Liquid         :269   Consonant:215   INGM84 : 57   Min.   :1915   F:659  
 Nasal          :209   H        :157   MARM92 : 53   1st Qu.:1952   M:530  
 Other Fricative:130   Pause    :558   GARF87 : 52   Median :1984          
 S              :332   Vowel    :259   HUNF22 : 32   Mean   :1969          
 Stop           :249                   DONF15 : 28   3rd Qu.:1991          
                                       GARF37 : 28   Max.   :1999          
                                       (Other):939                         
        Education         Job          Phoneme.Dep.Var
 Educated     :496   Blue    :130   t--T         :352  
 Not Educated:317   Service:468   t--Deletion :296  
 Student      :376   Student:376   t--Fricative:131  
                     White   :215   d--Deletion  : 90  
                                    d--T         : 73  
                                    d--D         : 46  
                                    (Other)      :201  
```

```
str(td)
```

```
'data.frame':   1189 obs. of  12 variables:
 $ Dep.Var        : Factor w/ 2 levels "Deletion","Realized": 2 1 1 1 2 1 1 1 1 1 ...
 $ Stress         : Factor w/ 2 levels "Stressed","Unstressed": 1 1 1 1 1 1 1 1 1 1 ...
 $ Category       : Factor w/ 2 levels "Function","Lexical": 2 2 2 2 2 2 2 2 2 2 ...
 $ Morph.Type     : Factor w/ 3 levels "Mono","Past",..: 1 1 1 1 1 1 1 1 1 1 ...
 $ Before         : Factor w/ 5 levels "Liquid","Nasal",..: 5 5 5 5 5 5 5 5 5 5 ...
 $ After          : Factor w/ 4 levels "Consonant","H",..: 1 1 1 1 1 1 1 1 1 1 ...
 $ Speaker       : Factor w/ 66 levels "ARSM91","BEAM91",..: 3 5 5 6 13 14 15 23 29 33 ...
 $ YOB            : int  1965 1955 1955 1952 1953 1958 1946 1942 1945 1949 ...
 $ Sex            : Factor w/ 2 levels "F","M": 1 1 1 1 2 2 1 2 2 1 ...
 $ Education     : Factor w/ 3 levels "Educated","Not Educated",..: 1 1 1 1 1 2 1 2 2 1 ...
 $ Job            : Factor w/ 4 levels "Blue","Service",..: 4 4 4 2 2 2 2 1 1 2 ...
 $ Phoneme.Dep.Var: Factor w/ 18 levels "d--Affricate",..: 18 12 12 12 18 12 12 12 12 12 ...
```

The following code does exactly the same thing as the previous code, but instead of filtering out "Vowel", it specifies keeping all the other levels. Here == is the standard operator meaning *equals and only equals* and | (called *pipe* or *bar*) is the operator meaning *or*. Again, note the very important comma following the last column condition.

```
td <- td[td$Before == "Liquid" | td$Before == "Nasal" |
    td$Before == "Other Fricative" | td$Before == "S" |
    td$Before == "Stop", ]
td$Before <- factor(td$Before)
```

As you did previously, you must run the second line of code to get rid of the empty `"Vowel"` level.

Re-coding Variables

While it is possible to do *ad-hoc* re-codes in *R*, you must keep in mind that these re-codes will only exist in your *R* data frame, not in your saved tab-delimited-text file. Personally, I think it's both easier and more useful to use Microsoft *Excel* to create new columns in your tab-delimited-text file for every re-code or configuration of factors in a factor group (i.e., levels in a column) that you might need. That being said, there are definitely situations where *ad-hoc* re-codes may be preferable.

If you want to change anything in your data frame you can generate an editable data frame in a popup window with the following function:

```
fix(td)
```

I don't recommend using this method. Like `file.choose()` and `choose.files()`, it introduces non-replicability because the changes you make using `fix()` are not recorded in your script file and therefore cannot be automatically replicated. A better practice for re-coding while maintaining replicability is to specify the cells in a column that include the specific values you want to change, and then to reassign a new value to those cells:

```
td$After[td$After == "H"] <- "Consonant"
td$After <- factor(td$After)
```

The code above uses the `<-` operator to say that any cells in the column `After` that contain the value `"H"` should be changed to `"Consonant"` — another value in the `After` column. Many studies of (t, d) deletion do not distinguish between pre-/h/ and other pre-consonantal contexts. The above re-code might be needed for comparing this (t, d) deletion data to other studies. Notice that there is no comma following `"H"` in the above code. This is because you are filtering a column; see how the filtering brackets `[ ]` come after the column specifier `$`. The comma is only used when filtering whole data frames because data frames can be filtered along two dimensions (e.g., rows and columns) and *R* needs to know which dimensions the filtering conditions apply to. When you filter just a column (or just a row), you don't need the comma because the filtering only occurs along one dimension (in that column, or in that row only). You also need to run the second `factor()` function to get rid of the now empty `"H"` level.

If you get an error message when trying to re-code using this method and your column contains words (rather than numbers) try first re-classifying (i.e., changing the type of) your original column to a *character* column: `td$After.New <- as.character(td$After.New)`, then proceeding with the above method.

The previous function sequence re-codes all `"H"` cells in the existing `After` column. If instead you wanted to create a new column with your re-code (so both possible coding options were available for later analyses), you could do so by creating a new column with the exact same values as `After` and then re-code that column. If you've been following along in *R*, your `After` column is already re-coded. To go back to the original form of the data as it exists in the tab-delimited-text file, simply reload that text file and assign it to the object `td`. Of course, this resets the deletion of the tokens following a vowel, so you must do that again too. Luckily, you've written all of your functions in a script file (rather than directly into the console) so this is easy to do: just highlight the code and press the execution command.

```
# Start with a fresh import of the (t, d) data
# into R
td <- read.delim("Data/deletiondata.txt")
# Convert each character column into a factor
td$Dep.Var <- factor(td$Dep.Var)
td$Stress <- factor(td$Stress)
td$Category <- factor(td$Category)
td$Morph.Type <- factor(td$Morph.Type)
td$Before <- factor(td$Before)
td$After <- factor(td$After)
td$Speaker <- factor(td$Speaker)
td$Sex <- factor(td$Sex)
td$Education <- factor(td$Education)
td$Job <- factor(td$Job)
td$Phoneme.Dep.Var <- factor(td$Phoneme.Dep.Var)
# Subset data to remove previous 'Vowel' contexts
td <- td[td$Before != "Vowel", ]
td$Before <- factor(td$Before)
# Re-code 'H' to be 'Consonant' in a new column
td$After.New <- td$After
td$After.New[td$After.New == "H"] <- "Consonant"
td$After.New <- factor(td$After.New)
```

The new column you create (After.New) is added at the end of the data frame. You can conceptualized this as the right edge of the data in an *Excel* spreadsheet. You can see this with the summary() or str() functions. The name of the new column you create doesn't really matter, but it cannot include any spaces.

```
summary(td)
```

```
     Dep.Var           Stress         Category        Morph.Type
 Deletion:386   Stressed   :1047   Function:  57   Mono     :762
 Realized:803   Unstressed: 142   Lexical :1132   Past     :311
                                                  Semi-Weak:116



             Before          After        Speaker         YOB       Sex
 Liquid         :269   Consonant:215   INGM84 : 57   Min.   :1915   F:659
 Nasal          :209   H        :157   MARM92 : 53   1st Qu.:1952   M:530
 Other Fricative:130   Pause    :558   GARF87 : 52   Median :1984
 S              :332   Vowel    :259   HUNF22 : 32   Mean   :1969
 Stop           :249                   DONF15 : 28   3rd Qu.:1991
                                       GARF37 : 28   Max.   :1999
                                       (Other):939
        Education        Job          Phoneme.Dep.Var     After.New
 Educated    :496   Blue   :130   t--T        :352    Consonant:372
 Not Educated:317   Service:468   t--Deletion :296    Pause    :558
 Student     :376   Student:376   t--Fricative:131    Vowel    :259
                    White  :215   d--Deletion : 90
                                  d--T        : 73
                                  d--D        : 46
                                  (Other)     :201
```

```
str(td)
```

```
'data.frame':   1189 obs. of  13 variables:
 $ Dep.Var        : Factor w/ 2 levels "Deletion","Realized": 2 1 1 1 2 1 1 1 1 1 ...
 $ Stress         : Factor w/ 2 levels "Stressed","Unstressed": 1 1 1 1 1 1 1 1 1 1 ...
 $ Category       : Factor w/ 2 levels "Function","Lexical": 2 2 2 2 2 2 2 2 2 2 ...
 $ Morph.Type     : Factor w/ 3 levels "Mono","Past",..: 1 1 1 1 1 1 1 1 1 1 ...
 $ Before         : Factor w/ 5 levels "Liquid","Nasal",..: 5 5 5 5 5 5 5 5 5 5 ...
 $ After          : Factor w/ 4 levels "Consonant","H",..: 1 1 1 1 1 1 1 1 1 1 ...
 $ Speaker        : Factor w/ 66 levels "ARSM91","BEAM91",..: 3 5 5 6 13 14 15 23 29 33 ...
 $ YOB            : int  1965 1955 1955 1952 1953 1958 1946 1942 1945 1949 ...
 $ Sex            : Factor w/ 2 levels "F","M": 1 1 1 1 2 2 1 2 2 1 ...
 $ Education      : Factor w/ 3 levels "Educated","Not Educated",..: 1 1 1 1 1 2 1 2 2 1 ...
 $ Job            : Factor w/ 4 levels "Blue","Service",..: 4 4 4 2 2 2 2 1 1 2 ...
 $ Phoneme.Dep.Var: Factor w/ 18 levels "d--Affricate",..: 18 12 12 12 18 12 12 12 12 12 ...
 $ After.New      : Factor w/ 3 levels "Consonant","Pause",..: 1 1 1 1 1 1 1 1 1 1 ...
```

Centering Continuous Variables

Some variables, like year of birth, are not discrete but are continuous. Some statisticians advocate centering continuous variables before including them in certain tests or models. In the Regression Analysis section you'll learn when/if you need to center your variables.

Centering entails expressing each value of a continuous variable as that value's difference from the mean of all values of the variable. For example, the *td* data frame has a column for speaker year of birth: `YOB`. The mean of all the years of birth (after the pre-vowel tokens are removed) is `1969`. Centering this variable simply means expressing years of birth like 1952 and 1989, as 17 ($= 1969 - 1952$) and -20 ($= 1969 - 1989$).

```
# Center YOB
td$Center.Age <- scale(td$YOB, scale = FALSE)
td$Center.Age <- as.numeric(td$Center.Age)
```

```
td$Center.Age <- as.numeric(scale(td$YOB, scale = FALSE))
```

The `scale()` function centers the values in the column `YOB` and assigns those Centered values to a new column called `Center.Age`. The option `scale = FALSE` indicates that the centered values remain in the original units (in this case years). If you change this option to `scale = TRUE`, the overall mean is subtracted from each value and then each values is divided by the overall standard deviation. This is needed if you are including multiple continuous variables in a model that are expressed using different units and vary along differing scales. When the data is centred and scalled the new values are called **z-scores** or **standardized scores** and you can think of each value as describing the raw value in terms of its distance from the mean when measured in standard deviation units.

For example, imagine a study of variable word-initial voice onset time in which you want to look at the effect of following vowel backness and year of birth of the speaker. Each of these variables are continuous, but voice onset time is ± 20 ms around its mean, year of birth is ± 45 years around its mean, and F2 is ± 300 Hz around its mean. By expressing these values as **z-scores**, you can account for both the differing units and differing scales of each. Even if you have one variable, here just `YOB`, it is okay to scale the values, but do *NOT* do this here. Leave `YOB` in years. This will make the interpretation of later statistical estimates a little easier.

After running your `scale()` function you must also run a function to tell *R* how to treat your new column `Center.Age.` *R* knows the column is filled with numbers, but *R* doesn't know if those numbers are continuous, if they are ordered in a specific way, or if they are factors with names expressed as digits. You use the function `as.numeric()` to tell *R* that the values are continuous numbers. This is very similar to what you did above with the function `factor()`, which tells *R* to consider whatever is inside the function to be a factor. Also, with `After.New` above, where the data structure is inherited from `After`, you used `factor()` to remove the remaining `H` level name. You did this by telling `R` to consider whatever was inside the `factor()` function as a factor, and, as part of that, `R` reads all the levels inside that column and chooses them as the factors. Here you are doing the same thing. You're telling `R` to look at the values inside `Center.Age` and consider them as being continuous numbers. You can do this as a secondary step after you create the column `Center.Age`, as shown in the first two rows. Or you can embed the `scale()` function inside the `as.numeric()` function, `as.numeric(scale())`, as shown in the last line.

Dividing a Continuous Variable

There are other ways you can represent age. Instead of a continuous variable, you can categorize speakers as belonging to a "young", "middle aged", or "old" age group. You might also group speakers based on decade of birth or as being born before or after some event — whatever makes sense for your study or community.

How to divide speakers by age is something that should be informed by sociolinguistic theory, demographics, and your own understanding of your data. The birth years 1980 and 1945, used here, represent generational divides in Cape Breton that can be independently justified (Gardner 2013). Grouping speakers like this is very common, and not particularly difficult to do in *R*.

```
# Create a 3-way Age Group
td$Age.Group[td$YOB > 1979] <- "Young"
td$Age.Group[td$YOB > 1944 & td$YOB < 1980] <- "Middle"
td$Age.Group[td$YOB < 1945] <- "Old"
```

First you will need a new column for your age group variable. Here, call it `Age.Group`.You use the assignment operator `<-` to fill all the cells in the new column `Age.Group` based on the values that are in the already existing `YOB` column. The first line says that for any rows in which `YOB` is greater than 1979, fill the empty cell in those same rows in the column `Age.Group` with the value `Young`. The second line does the same thing but includes two conditions: that `YOB` is greater than 1944 and that it is also less than 1980. For these rows, the value `Middle` is inserted in the `Age.Group` column. Even though the two conditions (`YOB > 1944`, `YOB < 1980`) refer to the same column, you need to fully specify each condition with both a column reference and a condition with an operator. So, for example, writing `td$Age.Group[td$YOB >1944 & <1970]` will not work. The third line instructs `R` to put `Old` in the `Age.Group` column for any row where `YOB` is less than 1945. Because this new column includes words, *R* will automatically categorize the column as `character` data. To rectify this, see the next section.

```
class(td$Age.Group)
```

```
[1] "character"
```

Changing the Order of Levels

```
td$Age.Group <- factor(td$Age.Group, levels = c("Young",
    "Middle", "Old"))
```

The code above does two things. First, it tells `R` that you want the new `Age.Group` column to be a column of factors, and second, it also tells `R` how you want those factors to be ordered. When `R` extracts the name of the factors of a column (based on the levels that are in that column) it orders the factors by name alphabetically. For example, the `Sex` column contains two levels: `M` and `F`. The first 500 tokens in your data frame might comes from males but `R` will still list the names of the factors in the column (which are based on those two levels) as `F` and then `M`, because `F` is closer to the start of the alphabet than `M`. If you run `summary(td)` you can see that the factor names listed in all of the factor columns are in alphabetical order. Sometimes this alphabetical ordering doesn't matter. Other times it is makes a big difference.

Any time factors are used in a statistical test or appear in a graph, the order of factor names is very important. For example, which factor is selected as the application value and which factor(s) are the non-application value(s) of a dependent variable is determined by factor name order. As for graphs, all kinds of layout parameters are set by the factor order of a variable. For example, if you left the `Age.Group` factor as it is, it would always list the `Age.Group` levels as `Middle`, `Old`, `Young`. On a graph like a bar graph it would arrange the bars for each age group alphabetically from left to right. This is not desirable. You will always want `Age.Group` ordered as either `Young`, `Middle`, `Old` or `Old`, `Middle`, `Young`. This is what the second part of the `factor()` command does. First it tells `R` to consider the values in the `Age.Group` to be factors, then it specifies that you want the factors (which get their names from the levels) to be ordered in certain way. You use `levels =` and the concatenating `c()` function to specify that you want the factors to get their names from the levels in the following order `"Young", "Middle", "Old"`, and thus also be ordered in that way.

This might seem confusing. Think about coins. Imagine you have a bag of change. Each coin in that bag is like a token in your column in the data frame. The levels of the column are just the different kinds of coins there are. You can root around in the bag and figure out what coins are in it without having to put the coins in any particular order. This is the "levels" of the coins. If you ask *R* what coins are in your bag, it will tell you there are "dimes", "nickels", "pennies", and "quarters". These levels are in alphabetical order, but only for a lack of a better way to tell you them. Telling you the levels doesn't imply any sort of ordering of the coins. Making the coin bag into a factor column is like putting the coins into a change tray, like the change trays in cash registers. Coins inside of a change tray go from being unstructured to being organized based on a structure. This makes them factors. In a cash tray each coin is grouped with other coins just like it. Each slot in the drawer also has a name. *R* just automatically gives the slots the same name as the coins (e.g. levels) that are in it. The order of the slots from right to left is also by default alphabetical based on the name of each slot. So, coin (= level), slot name (= factor name) and slot order (= factor order) are three independent parameters. The function `td$Age.Group <-factor(td$Age.Group, levels = c("Young", "Middle", "Old"))` is basically saying take all the coins out of the cash drawer, then put them back in the drawer in in a specific order, with `Young` going in the first slot. The slots then take the names of the coins that go into them.

Reversing the Order of Levels

To reverse the order of levels you can embed the `rev()` function inside your `factor()` function

```
levels(td$Age.Group)
```

```
[1] "Young"  "Middle" "Old"
```

```
td$Age.Group <- factor(td$Age.Group, levels = rev(levels(td$Age.Group)))
levels(td$Age.Group)
```

```
[1] "Old"    "Middle" "Young"
```

Combining Columns

It is often useful to combine two columns, or factor groups, into one. For example, it might be useful to have a way of grouping tokens not by `Age.Group` or `Sex`, but instead by the combination of `Age.Group` and `Sex`. While it is not difficult to test the potential interaction of these two factor groups in statistical tests in *R*, for specifically generating summary statistics it is much easier to examine the combination of `Age.Group` and `Sex` (or any two columns) by first creating a new column combining them.

Combining columns is done using the function `paste()`, which you embed inside of the `factor()` function so that the resulting column will be a column of factors. Inside `paste()` you list the two columns you want to combine (you could list more) and then tell *R* how to separate the values from each of the columns. Here the code tells *R* to put an underscore *_* between the values for `Age.Group` and `Sex`, resulting in new values like `Old_M` and `Middle_F`. The new column will be called `Age_Sex`. If we want to order these values in any particular way, we can do that with the `level=` option in `factor()`

```
# Combine Columns
td$Age_Sex <- factor(paste(td$Age.Group, td$Sex, sep = "_"),
    levels = c("Young_F", "Middle_F", "Old_F", "Young_M",
        "Middle_M", "Old_M"))
levels(td$Age_Sex)
```

```
[1] "Young_F"  "Middle_F" "Old_F"    "Young_M"  "Middle_M" "Old_M"
```

```
# Reorder factor levels
td$Age_Sex <- factor(td$Age_Sex, levels = c("Young_F",
    "Young_M", "Middle_F", "Middle_M", "Old_F", "Old_M"))
levels(td$Age_Sex)
```

```
[1] "Young_F"  "Young_M"  "Middle_F" "Middle_M" "Old_F"    "Old_M"
```

Splitting Columns

You may have noticed that the data frame `td` has one column that is actually two variables. The column `Phoneme.Dep.Var` combines both the underlying phoneme of the token, either `t` (t) or `d` (d), with a more descriptive coding of the dependent variable. In the dialect where this data comes from (t) and (d) can be realized in up to nine different ways, only one of which is `Deletion`. In your analysis you might want to consider if deletion is more likely if the underlying phoneme is (t) or (d). In order to do this you must break `Phoneme` away from the more elaborate `Dep.Var` coding. To do this, you will use a function that is beyond *R*'s base functionality and part of the `dplyr` package. First you load the `dplyr` package from your library using `library()`, then you use the function `mutate()` to tell *R* how to break up the column. The `dplyr` package is very powerful. We will look at additional uses as this guide progresses.

To be honest, splitting columns is an operation that is much simpler and usually faster to do using *Excel*, or another spreadsheet program. I never use *R* to split columns because, while the `mutate()` function itself is not tricky, figuring out the exact sequence of `regular expressions` I need to split my columns is challenging. If you absolutely must split columns in *R*, below are instructions. Keep in mind that this method will only work for columns with predictable structures, like `Phoneme.Dep.Var`.

The values in the column `Phoneme.Dep.Var` are predictable, in other words, they all follow the same pattern. You can exploit this regularity to tell *R* exactly where to break the `Phoneme.Dep.Var` values into two. If you execute the command `levels(td$Phoneme.Dep.Var)` you can very easily see the pattern.

```
levels(td$Phoneme.Dep.Var)
```

```
 [1] "d--Affricate"     "d--D"               "d--Deletion"
 [4] "d--Ejective"      "d--Flap"            "d--Fricative"
 [7] "d--Glottal Stop"  "d--Other Weakening" "d--T"
[10] "t--Affricate"     "t--D"               "t--Deletion"
[13] "t--Ejective"      "t--Flap"            "t--Fricative"
[16] "t--Glottal Stop"  "t--Other Weakening" "t--T"
```

The first part of every value is either `t` or `d`, then there are two hyphens, then a word describing the realization of (t) or (d). If you split `Phoneme.Dep.Var` at the hyphens you'll be left with two values: one that is either `t` or `d`, and another that describes the realization of (t) or (d). So, for the new `Phoneme` column you want the one character before the hyphens from `Phoneme.Dep.Var`, and for the new `Dep.Var.Full` column, you want all the characters — however many there are – that come after the two hyphens.

```r
# Break Phoneme.Dep.Var into two columns
library(dplyr)
td <- mutate(td, Phoneme = sub("^(.)(--.*)$", "\\1",
    Phoneme.Dep.Var), Dep.Var.Full = sub("^(.--)(.*)$",
    "\\2", Phoneme.Dep.Var), Phoneme.Dep.Var = NULL)
td$Dep.Var.Full <- factor(td$Dep.Var.Full)
td$Phoneme <- factor(td$Phoneme)
```

> **Note**
>
> The procedure here is somewhat complicated. I always have to double-check how these complicated procedures work. Either by looking up the functions or checking my old script files. You can look up how a function like `mutate()` works by placing a question mark before the function in the console window, e.g. `?mutate()`.

Above, inside the `mutate()` function, first you specify the data frame object you want to mutate (here `td`), and then how to mutate it. The mutations include creating a new column called `Phoneme` and then telling *R* what do put in it, creating a new column called `Dep.Var.Full` and then telling *R* what to put in it, and then taking the `Phoneme.Dep.Var` column and making it equal `NULL` — in other words, deleting it. For the `Phoneme` and `Dep.Var.Full` columns you specify what values to insert in each cell using the `sub()` function, which returns a "sub" or divided section of a value within some cell. The `sub()` function's first argument is a pattern of *regular expressions* to search for, the second is a character string to replace the pattern with, and the third is the column in which to search for the pattern.

> **Note**
>
> I don't have *R*'s regular expressions memorized; I always have to look them up here[a].
>
> [a] https://stat.ethz.ch/R-manual/R-devel/library/base/html/regex.html

For the `sub()` functions meant to create values for `Phoneme` and for `Dep.Var.Full` you tell *R* to search `Phoneme.Dep.Var` for the pattern `"^.--.*$"`. This series of regular expressions describes the following pattern: a text string beginning with one single character, followed by two hyphens, and then any number of characters. The quotation marks `"  "` indicate a text string. The `^` and `$` indicate the beginning and end of a text string, respectively.[1] The period `.` indicates a single character and the asterisk`*` indicates zero or more of whatever comes before that asterisk. So `.*` means any one or more characters. The two hyphens `--` are just two hyphens. The parentheses in the regular expressions relate to the second substitution element. In the `sub()` function responsible for creating values for the new `Phoneme` column, you break the values that match the regular expression pattern into two parts: the one character before the two hyphens `(.)` and then everything after it `(--.*)`; then you tell *R* to substitute (or rather place) the first of those two elements `\\1` as text `"  "` in the new column. In the `sub()` function responsible for creating the values for the new `Dep.Var.Full` column, you break the values that match the regular expression pattern into two parts: the two hyphens and the one character that comes before them `(.--)` and then the one or more characters that come after the hyphens `(.*)`; then tell *R* to to substitute (or rather place) the second of those elements `\\2` as text `"  "` in the new column.

The last two lines simply make these two new columns into factor columns.

[1] You use `$` in other places for specifying columns within data frames, but here it's inside quotation marks and serving a different function.

Partitioning Data Frames

Many times in my own work I have only wanted to work in *R* with a subset of my full dataset. For example, I frequently want to run separate regression analyses on data from old speakers, middle age speakers, and young speakers. Other times I've had large datasets that combine multiple corpora and have wanted to run tests on data from just one corpus. Being able to partition (subset) my data frame has therefore been very useful.

There are two main ways to partition your data. One method involves using the filtering functionality of *R* detailed above. For example, the first line code below represents the `td` data from `young` speakers only. The second line of codes assigns that subset of `td` to a completely new data frame.

```
# Subset of td where Age.Group only equals Young
td[td$Age.Group == "Young", ]
```

```
# Create td.young which equals subset of td where
# Age. Group only equals Young
td.young <- td[td$Age.Group == "Young", ]
```

In your tests you can use the filtered version of `td` or you can use the new data frame `td.young`. I recommend using the new data frame if you are using any centered continuous variables. This is because centered continuous variables are centered around the mean of the values in the column that is centered. If you partition the data, that mean will be different because the range of values within the the centered column are now different. So, if you partition your data, especially by age, you should re-center your continuous variables.

The second way to partition your data is using the `subset()` function.

```
# Create three partitions based on Age.Group
td.young <- subset(td, Age.Group == "Young")
td.middle <- subset(td, Age.Group == "Middle")
td.old <- subset(td, Age.Group == "Old")
# Re-Center Center.Age
td.young$Center.Age <- scale(td.young$YOB, scale = FALSE)
td.middle$Center.Age <- scale(td.middle$YOB, scale = FALSE)
td.old$Center.Age <- scale(td.old$YOB, scale = FALSE)
```

The usefulness of `subset()` is when you want to partition your data frame based on several factors. For example, compare the filtering versus `subset()` methods when partitioning `td` for just middle-age, blue-collar men. Using `subset()` saves you from having to type `td$` multiple times, and you don't need the final `,`. Below `&` is the standard operator meaning "and".

```
# Subset of td where Age.Group is Middle, Sex is
# M, and Job is Blue
td.midmenblue <- td[td$Age.Group == "Middle" & td$Sex ==
    "M" & td$Job == "Blue", ]
# Subset of td where Age.Group is Middle, Sex is
# M, and Job is Blue
td.midmenblue <- subset(td, Age.Group == "Middle" &
    Sex == "M" & Job == "Blue")
```

When you subset your data frame, using either method, and assign it to a new data frame, your new data frame inherits the structure of your old data frame. This means that there will be lots of columns that list empty levels. If you execute `summary(td.midmenblue)}` you'll see see that `Job` still lists `Service`, `Student`, and White as factors with 0 tokens each. To remove these empty levels you could use the factor() function on each column with empty levels (as you did above), or you can use the function droplevels() to tell *R* to drop any empty levels from each column in the data frame. You need to make sure to also assign the dropped levels data frame to the original data frame, as shown below.

```
summary(td.midmenblue)
```

```
    Dep.Var             Stress         Category       Morph.Type
 Deletion:30   Stressed  :49   Function: 3   Mono      :38
 Realized:27   Unstressed: 8   Lexical :54   Past      :15
                                             Semi-Weak: 4



             Before           After          Speaker         YOB          Sex
 Liquid          :13   Consonant:16   GREM45 :18   Min.   :1945   F: 0
 Nasal           : 3   H        : 9   HOLM52 :16   1st Qu.:1945   M:57
 Other Fricative: 6   Pause    :20   SMIM61 :16   Median :1952
 S               :18   Vowel    :12   CLAM73 : 4   Mean   :1954
 Stop            :17                  HANM57 : 3   3rd Qu.:1961
                                      ARSM91 : 0   Max.   :1973
                                      (Other): 0
        Education          Job          After.New      Center.Age        Age.Group
 Educated     : 0   Blue    :57   Consonant:25   Min.   :-24.447   Old   : 0
 Not Educated:57   Service: 0   Pause    :20   1st Qu.:-24.447   Middle:57
 Student      : 0   Student: 0   Vowel    :12   Median :-17.447   Young : 0
                    White  : 0                  Mean   :-15.394
                                                3rd Qu.: -8.447
                                                Max.   :  3.553

    Age_Sex    Phoneme    Dep.Var.Full
 Young_F : 0   d:11    Deletion :30
 Young_M : 0   t:46    T        :19
 Middle_F: 0           Fricative: 6
 Middle_M:57           D        : 1
 Old_F   : 0           Flap     : 1
 Old_M   : 0           Affricate: 0
                       (Other)  : 0
```

```
# Drop empty levels across dataset
td.midmenblue <- droplevels(td.midmenblue)
summary(td.midmenblue)
```

```
    Dep.Var             Stress         Category       Morph.Type
 Deletion:30   Stressed  :49   Function: 3   Mono      :38
 Realized:27   Unstressed: 8   Lexical :54   Past      :15
                                             Semi-Weak: 4



             Before           After         Speaker         YOB          Sex
 Liquid          :13   Consonant:16   CLAM73: 4   Min.   :1945   M:57
 Nasal           : 3   H        : 9   GREM45:18   1st Qu.:1945
```

```
Other Fricative: 6   Pause    :20   HANM57: 3   Median :1952
S              :18   Vowel    :12   HOLM52:16   Mean   :1954
Stop           :17                  SMIM61:16   3rd Qu.:1961
                                                Max.   :1973
       Education     Job         After.New     Center.Age        Age.Group
Not Educated:57   Blue:57   Consonant:25   Min.   :-24.447   Middle:57
                            Pause    :20   1st Qu.:-24.447
                            Vowel    :12   Median :-17.447
                                           Mean   :-15.394
                                           3rd Qu.: -8.447
                                           Max.   :  3.553
    Age_Sex    Phoneme    Dep.Var.Full
Middle_M:57   d:11    D        : 1
              t:46    Deletion :30
                      Flap     : 1
                      Fricative: 6
                      T        :19
```

Interim Summary

Below is the full code for loading the data file and then re-coding it. As you progress through the next sections, if for any reason you have a problem, it may be useful to recreate the *R* object `td` from scratch. Going forward in these instructions, the `td` object will be the object as it exists at the end of this code.

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

Or…

```
td <- read.delim("Data/deletiondata.txt")
```

Then…

```
# Subset data to remove previous 'Vowel' contexts
td <- td[td$Before != "Vowel", ]
td$Before <- factor(td$Before)
# Re-code 'H' to be 'Consonant' in a new column
td$After.New <- td$After
td$After.New[td$After.New == "H"] <- "Consonant"
td$After.New <- factor(td$After.New)
# Center Year of Birth
td$Center.Age <- as.numeric(scale(td$YOB, scale = FALSE))
# Create Age.Group
td$Age.Group[td$YOB > 1979] <- "Young"
td$Age.Group[td$YOB > 1944 & td$YOB < 1980] <- "Middle"
td$Age.Group[td$YOB < 1945] <- "Old"
td$Age.Group <- factor(td$Age.Group, levels = c("Young",
    "Middle", "Old"))
# Combine Age and Sex
td$Age_Sex <- factor(paste(td$Age.Group, td$Sex, sep = "_"))
# Break Phoneme.Dep.Var into two columns
library(dplyr)
td <- mutate(td, Phoneme = sub("^(.)(--.*)$", "\\1",
    Phoneme.Dep.Var), Dep.Var.Full = sub("^(.--)(.*)$",
    "\\2", Phoneme.Dep.Var), Phoneme.Dep.Var = NULL)
td$Phoneme <- factor(td$Phoneme)
td$Dep.Var.Full <- factor(td$Dep.Var.Full)
# Create three partitions based on Age.Group
td.young <- droplevels(subset(td, Age.Group == "Young"))
td.middle <- droplevels(subset(td, Age.Group == "Middle"))
td.old <- droplevels(subset(td, Age.Group == "Old"))
# Re-center Center.Age
td.young$Center.Age <- scale(td.young$YOB, scale = FALSE)
td.middle$Center.Age <- scale(td.middle$YOB, scale = FALSE)
td.old$Center.Age <- scale(td.old$YOB, scale = FALSE)
```