How to manipulate data with dplyr in R - Storybench

ARCHIE'S MECHANICS
KNOW YOUR PLIERS
MULTI-PURPOSE PLIER
IDEAL FOR HOME HANDYMAN WITH LIMITED TOOLS, SERVES AS PLIER, WIRE CUTTER, SMALL WRENCH, AND SCREW DRIVER.
LONG-NOSE PLIER
HANDY FOR SMALL WORK WHERE SLIM JAWS ARE NEEDED. THE PLIER HAS EITHER CENTER OR SIDE CUTTERS FOR WIRE.

In the last tutorial we introduced the concept of tidy data, which is characterized by having one observation per row, and one variable per column. We also went over how to change to shape of our data set with tidyr using data sets from the fivethirtyeight package.

For newcomers to R, check out my introductory tutorial for *Storybench* here.

Data manipulation

In this tutorial we will dive a little deeper into data manipulation to focus on processing and creating variables. Most of the time you'll need to do some manipulation in order to get your data into a structure/arrangement to suit your needs; whether you're building models, creating visualizations, or just passing a processed dataset onto another analyst.

The clearest example of this fact is the paper survey or data collection form, which has to be re-entered into a separate software environment or spreadsheet for analysis. Web-based collection tools like Survey Monkey and Qualtrics has made the process easier, but not perfect. There might be occasions when the data management is arranged in such a way that allows for a seamless transition between data collection and analysis, but I think these cases are rare.

A quick note on terminology

The preparation work for a dataset before analysis or modeling has many names: *munging*, *wrangling*, *cleaning*, etc. As you can see by the general theme of these terms, this process is often viewed as the "grunt-work" of any analysis/modeling project. I prefer the term "Data Carpentry" from David Mimno. I suggest not thinking of any data set as "dirty." This implies there is an immaculate version underneath the grime and that isn't necessarily true.

[perfectpullquote align="right" cite="" link="" color="" class="" size=""]"Looking at data carpentry as a fundamental data skill transforms these burdensome, monotonous tasks into a set of bedrock competencies"[/perfectpullquote]

For example, say you have a dataset with a variable containing date information from the `year` down to the `millisecond`, but all you need is the `month`. The other information is not 'dirt' – in fact, you might later decide you need `month` and `year`.

Every bit in your data set is telling you *something*, but the information you need might not be accessible in it's current state. The carpentry skills take the data from its current form into a something you can use to gain insight.

The process doesn't have to be painful, either! I've found looking at data carpentry as a fundamental data skill transforms these burdensome, monotonous tasks into a set of bedrock competencies to take pride in. As the famous coach John Wooden wrote, "These seemingly trivial matters, taken together and added to many, many other so-called trivial matters build into something very big: namely, your success." Data analysis isn't different than college basketball in this sense–mastering the fundamentals is essential to success.

Thinking in verbs

The tidyverse package for manipulating data is dplyr (pronounced "d-plier"" where "plier" is pronounced just like the tool). The `dplyr` package comes with an entire grammar for data manipulation, which uses a small set of verbs to accomplish an array of data processing tasks. There is a different verb – or "plier" – for each different data manipulation task. Read about all the different types of pliers here

When you combine `dplyr` with `magrittr`, you'll be able to create data carpentry pipelines that are efficient and easy to read.

```
library(tidyverse)
library(dplyr)
library(magrittr)
```

```
library(fivethirtyeight)
# data(package = "fivethirtyeight")
```

The first data set we will be using from the `fivethirtyeight` package is from the article titled, "Every mention of the 2016 primary candidates in hip-hop songs".

To view the status of each data set in the `fivethiryeight` package, you can look at the Google sheet available here.

First we will use the `tbl_df` function from the tibble package to put the data set into the working environment.

Quick tip: 1) to use a particular function within a package you can use the syntax `package::function`

```
hiphop_cand_lyrics <- fivethirtyeight::hiphop_cand_lyrics %>% tbl_df
hiphop_cand_lyrics
```

The first verb we will use to explore this data set is `select()`.

`select()`

`select()` works on variables (columns). For example, we can use it to pick out a single column, like the candidate's name (`candidate`):

```
hiphop_cand_lyrics %>%
    dplyr::select(candidate) %>%
    glimpse()
```

Quick tip: The `glimpse()` function is also from the `tibble` package and is very handy for viewing data frames (or tibbles). The result prints out the variables transposed as rows, the variable class (`<chr>` in this case), and as much of the data as that will fit on the screen.

Let's assume for our current purpose, we don't need the `url`. There are a few ways we can remove this variable.

First, we can use the `select()` function to name the variables we want explicitly. We can also use this opportunity to reorder the variables in our data set and rename the `album_release_date` to something a little easier to write, like `date`.

```
hiphop_cand_lyrics %>%
    dplyr::select(
        candidate,
        date = album_release_date,
        song,
        artist,
        sentiment,
        theme,
        line) %>% glimpse()
```

But if our data set has a ton of variables, we can always just select the variables we want to remove using a `-` symbol.

```
hiphop_cand_lyrics %>%
    dplyr::select(
        -(url),
        date = album_release_date) %>%
    glimpse()
```

We also need to explicitly assign these variables to a new data frame with the variables we've selected.

```
(hiphop <- hiphop_cand_lyrics %>%
    dplyr::select(
        candidate,
        date = album_release_date,
        song,
        artist,
        sentiment,
        theme,
        line))
```

Quick Tip: In RStudio, you can get the command to print in a notebook or R Markdown file by enclosing the call in parentheses `()`.

`select()` also comes with a lot of handy pattern matching abilities on column names. For example, we can use the `matches` function to look for variables that contain a specific character.

Say we only wanted variables that had the letter `t` in them.

```
hiphop %>%
    dplyr::select(matches("t")) %>%
    glimpse()
```

You can also use periods as placeholders (as in regular expressions) to select variables that only have a `t` in the third position or after, but not in the first or second (i.e. omitting the `theme` variable).

```
hiphop %>%
    dplyr::select(matches(".t")) %>%
    #                      candidate
    #                      artist
    #                      date
    #                      sentiment
    #                      but not 'theme'
    glimpse()
```

There are different ways to pick columns based on the column names:

`filter()`

The `filter()` command works on observations the same way `select()` works on variables. Lets say we are looking for observations that contain song lyrics that were only referring to "Jeb Bush". We can do this using the double equal symbol `==`:

```
hiphop %>% filter(candidate == "Jeb Bush") %>% glimpse()
```

We can also `filter()` with multiple criteria. We can use the `%in%` to identify lyrics referring to "Jeb Bush" or "Chris Christie".

```
hiphop %>%
    filter(candidate %in% c("Jeb Bush",
                            "Chris Christie")) %>%
    glimpse()
```

We can also use logical operators (`>`, `>=`, `<`, `<=`, `!=`, `==`) to filter any numerical variables.

```
hiphop %>%
    filter(date > 2005 & date <= 2014) %>% # all observations after 2005 and
before or on 2014
    glimpse()
```

`filter()` also comes with three scoped variations. For example, we can use `filter_all` to return all variables and observations that are greater than `2005`.

```
hiphop %>% filter_all(any_vars(. > 2005)) %>% glimpse()
```

Because `date` is the only variable with numeric values, we know this new data frame only contains observations after `2005`.

`mutate()`

Often times you'll want to create a new variable based on existing variables. We will create a new variable called (`new_2010`) and base it on songs that were released before or after `2010`. A song will be considered new if it was released on or after 2010, and not new if it was before 2010. In order to do this we will need to introduce another handy tool for creating new variables, `if_else`.

The components to this function are pretty straightforward.

```
if_else(condition, true, false, missing = NULL)
```

It takes a condition, evaluates it, then returns a value based on it being true or false.

```
hiphop %>%
    mutate(new_2010 =
       if_else(date >= 2010, "new song", "not new")) %>%
    glimpse()
```

NOTE: Avoid using periods (`.`) in the names of variables (or anything in R, really) because it can mess with R's programming model. Either use `snake_case` or `CamelCase`, but try to pick one and stick with it so its easier for people to follow.

This is helpful, but most of these data are text. The article categorized the mentions of candidates by their `sentiment` ("negative", "neutral", "positive"), but what if we also wanted to see if the lyrics contained obscenities (specifically, the word "fuck.").

This isn't just out of a juvenile interest in vulgar and lewd language. The use of obscene language has been used as a reason for certain albums to be censored or receive a "Parental Advisory: Explicit Content" label (see 2 Live Crew).

I want to know how the use of the word "fuck" relates to the three categories of `sentiment`. I can do this by creating a new variable called `lewd_lang` that has the values "fucks" and "zero fucks".

We will use the `mutate()` function, and also introduce the `grepl` function for dealing with text variables. Read about `grepl`.

```
grepl(pattern, x, ignore.case = FALSE, perl = FALSE,
      fixed = FALSE, useBytes = FALSE)
```

"search for matches to argument `pattern` within each element of a character vector"

We will use the `grepl` command with `if_else`, but also use the `filter` function to return the observations with lewd language.

```
hiphop %>%
    mutate(lewd_lang =
              if_else(grepl("fuck", line),
                          "fucks", "zero fucks")) %>%
               filter(lewd_lang == "fucks") %>% glimpse()
```

Now, the final command we will use can give us a cross-tabulation of how many songs contained lewd language by their sentiment (i.e "negative", "neutral", or "positive").

`count()`

The `count()` function is one you'll use constantly if you deal with categorical or nominal variables (i.e. non-numeric values). `count()` actually combines three functions, `group_by()`, `tally()`, and `ungroup()`.

When we use `count()` with one variable, we get the relative distribution in each value (or level) of the variable. We can test this new function out on the `theme` variable.

`theme` = "Theme of lyric"

```
hiphop %>%
    count(theme)
```

Jenny Bryan was nice enough to compile a list of methods from a tweet for creating cross tabulations. Feel free to experiment with all of them (found here)

For example, we can get a cross tabulation using a function from the `tidyr` function (`spread()`) we covered in the last tutorial.

```
hiphop %>%
    mutate(lewd_lang =
    if_else(grepl("fuck", line),
                  "fucks", "zero fucks")) %>%
    count(sentiment, lewd_lang) %>%
    spread(sentiment, n)
```

We can extend this further and use `mutate()` with the `prop.table` function to get relative the percentages in each category.

```
hiphop %>%
    mutate(lewd_lang =
    if_else(grepl("fuck", line),
                  "fucks", "zero fucks")) %>%
    count(sentiment, lewd_lang) %>%
    mutate(prop = prop.table(n)) %>%
    spread(sentiment, n)
```