Data Wrangling with R

Claudia A Engel

Last updated: April 17, 2019

Contents

Prerequisites and Preparations

- You should have some **basic knowledge** of R, and be familiar with the topics covered in the Introduction to R.
- Have a recent version of R and RStudio installed.
- Install and load the `tidyverse` package.

```
install.packages("tidyverse")
library(tidyverse)
```

- Create a new RStudio project `R-data-ws` in a new folder `R-data-ws`. Download both CSV files into a subdirectory called `data` like this:
- Download `MS_trafficstops_bw_age.csv`:

```
download.file("http://bit.ly/MS_trafficstops_bw_age",
              "data/MS_trafficstops_bw_age.csv")
```

- Download `MS_acs2015_bw.csv`:

```
download.file("http://bit.ly/MS_acs_2015_bw",
              "data/MS_acs2015_bw.csv")
```

References

Boehmke, Bradley C. (2016) Data Wrangling with R http://link.springer.com/book/10.1007%2F978-3-319-45599-0

Grolemund, G & Wickham, H (2017): R for Data Science http://r4ds.had.co.nz

Wickham, H. (2014): Tidy Data https://www.jstatsoft.org/article/view/v059i10

Acknowledgements

Part of the materials for this tutorial are adapted from http://datacarpentry.org and http://softwarecarpentry.org.

Chapter 1

Data Manipulation using `dplyr`

> Learning Objectives
>
> - Select columns in a data frame with the **`dplyr`** function `select`.
> - Select rows in a data frame according to filtering conditions with the **`dplyr`** function `filter`.
> - Direct the output of one **`dplyr`** function to the input of another function with the 'pipe' operator `%>%`.
> - Add new columns to a data frame that are functions of existing columns with `mutate`.
> - Understand the split-apply-combine concept for data analysis.
> - Use `summarize`, `group_by`, and `tally` to split a data frame into groups of observations, apply a summary statistics for each group, and then combine the results.

We will be working a small subset of the data from the Stanford Open Policing Project. It contains information about traffic stops for blacks and whites in the state of Mississippi during January 2013 to mid-July of 2016.

Let's begin with loading our sample data into a data frame.

```
trafficstops <- read.csv("data/MS_trafficstops_bw_age.csv")
```

Manipulation of dataframes is a common task when you start exploring your data. We might select certain observations (rows) or variables (columns), group the data by a certain variable(s), or calculate summary statistics.

If we were interested in the mean age of the driver in different counties we can do this using the normal base R operations:

```
mean(trafficstops[trafficstops$county_name == "Clay County", "driver_age"], na.rm = TRUE)
```

```
#> [1] 31.8002
```

```
mean(trafficstops[trafficstops$county_name == "Lee County", "driver_age"], na.rm = TRUE)
```

```
#> [1] 34.66915
```

```
mean(trafficstops[trafficstops$county_name == "Yazoo County", "driver_age"], na.rm = TRUE)
```

```
#> [1] 37.05759
```

Bracket subsetting is handy, but it can be cumbersome and difficult to read, especially for complicated operations. Furthermore, there is a fair amount of repetition. Repeating yourself will cost you time, both now and later, and potentially introduce some nasty bugs.

dplyr is a package for making tabular data manipulation easier.

> Brief recap: Packages in R are sets of additional functions that let you do more stuff. Functions like `str()` or `data.frame()`, come built into R; packages give you access to more of them. Before you use a package for the first time you need to install it on your machine, and then you should import it in every subsequent R session when you need it.

If you haven't, please installe the **tidyverse** package.

```
install.packages("tidyverse")
```

tidyverse is an "umbrella-package" that installs a series of packages useful for data analysis which work together well. Some of them are considered **core** packages (among them **tidyr**, **dplyr**, **ggplot2**), because you are likely to use them in almost every analysis. Other packages, like `lubridate` (to work wiht dates) or **haven** (for SPSS, Stata, and SAS data) that you are likely to use not for every analysis are also installed.

If you type the following command, it will load the **core** `tidyverse` packages.

```
library("tidyverse")    ## load the core tidyverse packages, incl. dplyr
```

If you need to use functions from `tidyverse` packages other than the core packages, you will need to load them separately.

1.1 What is dplyr?

dplyr is one part of a larger **tidyverse** that enables you to work with data in tidy data formats. "Tidy datasets are easy to manipulate, model and visualise, and have a specific structure: each variable is a column, each observation is a row, and each type of observational unit is a table." (From Wickham, H. (2014): Tidy Data https://www.jstatsoft.org/article/view/v059i10)

The package **dplyr** provides convenient tools for the most common data manipulation tasks. It is built to work directly with data frames, with many common tasks optimized by being written in a compiled language (C++). An additional feature is the ability to work directly with data stored in an external database. The benefits of doing this are that the data can be managed natively in a relational database, queries can be conducted on that database, and only the results of the query are returned.

This addresses a common problem with R in that all operations are conducted in-memory and thus the amount of data you can work with is limited by available memory. The database connections essentially remove that limitation in that you can have a database of many 100s GB, conduct queries on it directly, and pull back into R only what you need for analysis.

To learn more about **dplyr** after the workshop, you may want to check out the handy data transformation with **dplyr** cheatsheet.

1.2 Subsetting columns and rows

To select columns of a data frame with `dplyr`, use `select()`. The first argument to this function is the data frame (`trafficstops`), and the subsequent arguments are the columns to keep.

```
select(trafficstops, police_department, officer_id, driver_race)
```

```
#>             police_department officer_id driver_race
#> 1 Mississippi Highway Patrol       J042       Black
#> 2 Mississippi Highway Patrol       B026       Black
#> 3 Mississippi Highway Patrol       M009       Black
#> 4 Mississippi Highway Patrol       K035       White
```

```
#> 5 Mississippi Highway Patrol       D028       White
#> 6 Mississippi Highway Patrol       K023       White
```

It is worth knowing that `dplyr` comes with a number of "select helpers", which are functions that allow you to select columns based on their names. For example:

```
select(trafficstops, starts_with("driver"))
```

```
#>   driver_gender driver_birthdate driver_race driver_age
#> 1          male       1950-06-14       Black         63
#> 2          male       1967-04-06       Black         46
#> 3          male       1974-04-15       Black         39
#> 4          male       1981-03-23       White         32
#> 5          male       1992-08-03       White         20
#> 6        female       1960-05-02       White         53
```

To choose rows based on specific criteria, use `filter()`:

```
filter(trafficstops, county_name == "Yazoo County")
```

```
#>              id state  stop_date  county_name county_fips
#> 1 MS-2013-00252    MS 2013-01-02 Yazoo County       28163
#> 2 MS-2013-00253    MS 2013-01-02 Yazoo County       28163
#> 3 MS-2013-00254    MS 2013-01-02 Yazoo County       28163
#> 4 MS-2013-00331    MS 2013-01-02 Yazoo County       28163
#> 5 MS-2013-00350    MS 2013-01-02 Yazoo County       28163
#> 6 MS-2013-00426    MS 2013-01-03 Yazoo County       28163
#>            police_department driver_gender driver_birthdate driver_race
#> 1 Mississippi Highway Patrol          male       1950-05-04       Black
#> 2 Mississippi Highway Patrol        female       1967-05-29       Black
#> 3 Mississippi Highway Patrol          male       1986-12-21       Black
#> 4 Mississippi Highway Patrol        female       1986-02-01       Black
#> 5 Mississippi Highway Patrol          male       1994-11-21       White
#> 6 Mississippi Highway Patrol          male       1994-02-24       White
#>                                                violation_raw officer_id
#> 1 Speeding - Regulated or posted speed limit and actual speed       C037
#> 2 Speeding - Regulated or posted speed limit and actual speed       C011
#> 3 Speeding - Regulated or posted speed limit and actual speed       C011
#> 4 Speeding - Regulated or posted speed limit and actual speed       C037
#> 5 Speeding - Regulated or posted speed limit and actual speed       C037
#> 6 Speeding - Regulated or posted speed limit and actual speed       C014
#>   driver_age violation
#> 1         63  Speeding
#> 2         46  Speeding
#> 3         26  Speeding
#> 4         27  Speeding
#> 5         18  Speeding
#> 6         19  Speeding
```

Here are some other ways to select rows:

- select certain rows by row number: `slice(trafficstops, 1:3) # rows 1-3`
- select random rows:
  - `sample_n(trafficstops, 5) # number of rows to select`
  - `sample_frac(trafficstops, .01) # fraction of rows to select`

To sort rows by variables use the `arrange` function: `arrange(trafficstops, county_name, stop_date)`

```
#>              id state  stop_date  county_name county_fips
#> 1 MS-2013-07659    MS 2013-02-09 Adams County       28001
#> 2 MS-2013-11819    MS 2013-03-02 Adams County       28001
#> 3 MS-2013-14647    MS 2013-03-16 Adams County       28001
#> 4 MS-2013-15430    MS 2013-03-20 Adams County       28001
#> 5 MS-2013-18581    MS 2013-04-06 Adams County       28001
#> 6 MS-2013-20016    MS 2013-04-13 Adams County       28001
#>            police_department driver_gender driver_birthdate driver_race
#> 1 Mississippi Highway Patrol          male       1989-06-12       Black
#> 2 Mississippi Highway Patrol        female       1974-10-16       Black
#> 3 Mississippi Highway Patrol        female       1977-07-15       Black
#> 4 Mississippi Highway Patrol        female       1991-06-15       Black
#> 5 Mississippi Highway Patrol        female       1980-04-18       White
#> 6 Mississippi Highway Patrol        female       1996-01-14       Black
#>                                                violation_raw officer_id
#> 1 Speeding - Regulated or posted speed limit and actual speed       M004
#> 2                              Driving while license suspended       M042
#> 3 Speeding - Regulated or posted speed limit and actual speed       M049
#> 4           Failure to maintain required liability insurance       M049
#> 5           Failure to maintain required liability insurance       M010
#> 6 Speeding - Regulated or posted speed limit and actual speed       M024
#>   driver_age                violation
#> 1         24                 Speeding
#> 2         38 License-Permit-Insurance
#> 3         36                 Speeding
#> 4         22 License-Permit-Insurance
#> 5         33 License-Permit-Insurance
#> 6         17                 Speeding
```

1.3 Pipes

What if you wanted to filter **and** select on the same data? There are three ways to do this: use intermediate steps, nested functions, or pipes.

- Intermediate steps:

With intermediate steps, you essentially create a temporary data frame and use that as input to the next function. This can clutter up your workspace with lots of objects.

```
tmp_df <- filter(trafficstops, driver_age > 85)
select(tmp_df, violation_raw, driver_gender, driver_race)
```

- Nested functions

You can also nest functions (i.e. one function inside of another). This is handy, but can be difficult to read if too many functions are nested as things are evaluated from the inside out.

```
select(filter(trafficstops, driver_age > 85), violation_raw, driver_gender, driver_race)
```

- Pipes!

The last option, pipes, are a fairly recent addition to R. Pipes let you take the output of one function and send it directly to the next, which is useful when you need to do many things to the same dataset. Pipes in R look like `%>%` and are made available via the `magrittr` package, installed automatically with **`dplyr`**. If you use RStudio, you can type the pipe with Ctrl + Shift + M if you have a PC or Cmd + Shift + M if you have a Mac.

```
trafficstops %>%
  filter(driver_age > 85) %>%
  select(violation_raw, driver_gender, driver_race)
```

In the above, we use the pipe to send the `trafficstops` dataset first through `filter()` to keep rows where `driver_race` is Black, then through `select()` to keep only the `officer_id` and `stop_date` columns. Since `%>%` takes the object on its left and passes it as the first argument to the function on its right, we don't need to explicitly include it as an argument to the `filter()` and `select()` functions anymore.

If we wanted to create a new object with this smaller version of the data, we could do so by assigning it a new name:

```
senior_drivers <- trafficstops %>%
  filter(driver_age > 85) %>%
  select(violation_raw, driver_gender, driver_race)

senior_drivers
```

```
#>                                                    violation_raw
#> 1                      Seat belt not used properly as required
#> 2 Speeding - Regulated or posted speed limit and actual speed
#> 3                      Seat belt not used properly as required
#>   driver_gender driver_race
#> 1          male       White
#> 2          male       White
#> 3          male       Black
```

Note that the final data frame is the leftmost part of this expression.

> Challenge
>
> Using pipes, subset the `trafficstops` data to include stops in Tunica County only and retain the columns `stop_date`, `driver_age`, and `violation_raw`. Bonus: sort the table by driver age.

1.4 Add new columns

Frequently you'll want to create new columns based on the values in existing columns. For this we'll use `mutate()`.

To create a new column with the year the driver was born we will use the lubridate library, which is installed with `tidyverse`. We use `ymd()` to convert the date column into a date object and then use `year()` to extract the year only.

```
library(lubridate)

trafficstops %>%
  mutate(birth_year = year(ymd(driver_birthdate)))
```

If this runs off your screen and you just want to see the first few rows, you can use a pipe to view the `head()` of the data. (Pipes work with non-**`dplyr`** functions, too, as long as the **`dplyr`** or `magrittr` package is loaded). When piping into a function with no additional arguments, you can call the function with or without parentheses (e.g. `head` or `head()`). (I like to add the parentheses to remind myself that it is a function and not a variable.)

```
trafficstops %>%
  mutate(birth_year = year(ymd(driver_birthdate))) %>%
  head()
```

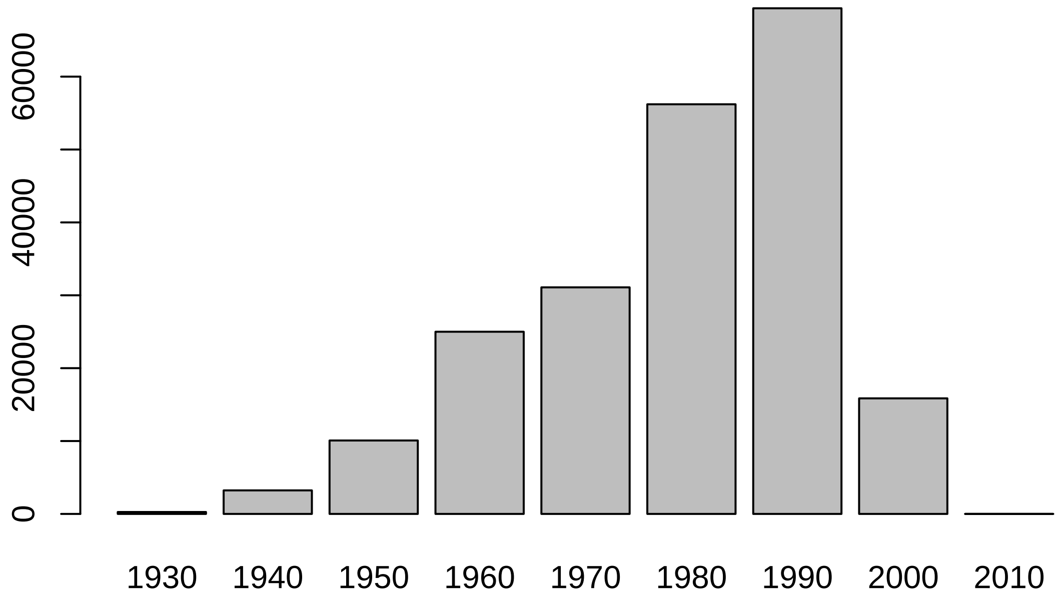


Figure 1.1: Driver Birth Cohorts

You can also create a second new column based on the first new column within the same call of `mutate()`:

```
trafficstops %>%
    mutate(birth_year = year(ymd(driver_birthdate)),
           birth_cohort = round(birth_year/10)*10) %>%
    head()
```

We are beginning to see the power of piping. Here is a slightly expanded example, where we select the column `birth_cohort` that we have created and send it to plot:

```
trafficstops %>%
    mutate(birth_year = year(ymd(driver_birthdate)),
           birth_cohort = round(birth_year/10)*10,
           birth_cohort = factor(birth_cohort)) %>%
    select(birth_cohort) %>%
    plot()
```

> Challenge
>
> Create a new data frame from the `trafficstops` data that meets the following criteria: contains only the `violation_raw` column for female drivers of age 50 that were stopped on a Sunday. For this add a new column to your data frame called `weekday_of_stop` containing the number of the weekday when the stop occurred. Use the `wday()` function from `lubridate` (Sunday = 1).
>
> Think about how the commands should be ordered to produce this data frame!

1.5 What is split-apply-combine?

Many data analysis tasks can be approached using the *split-apply-combine* paradigm: split the data into groups, apply some analysis to each group, and then combine the results.

`dplyr` makes this very easy through the use of the `group_by()` function.

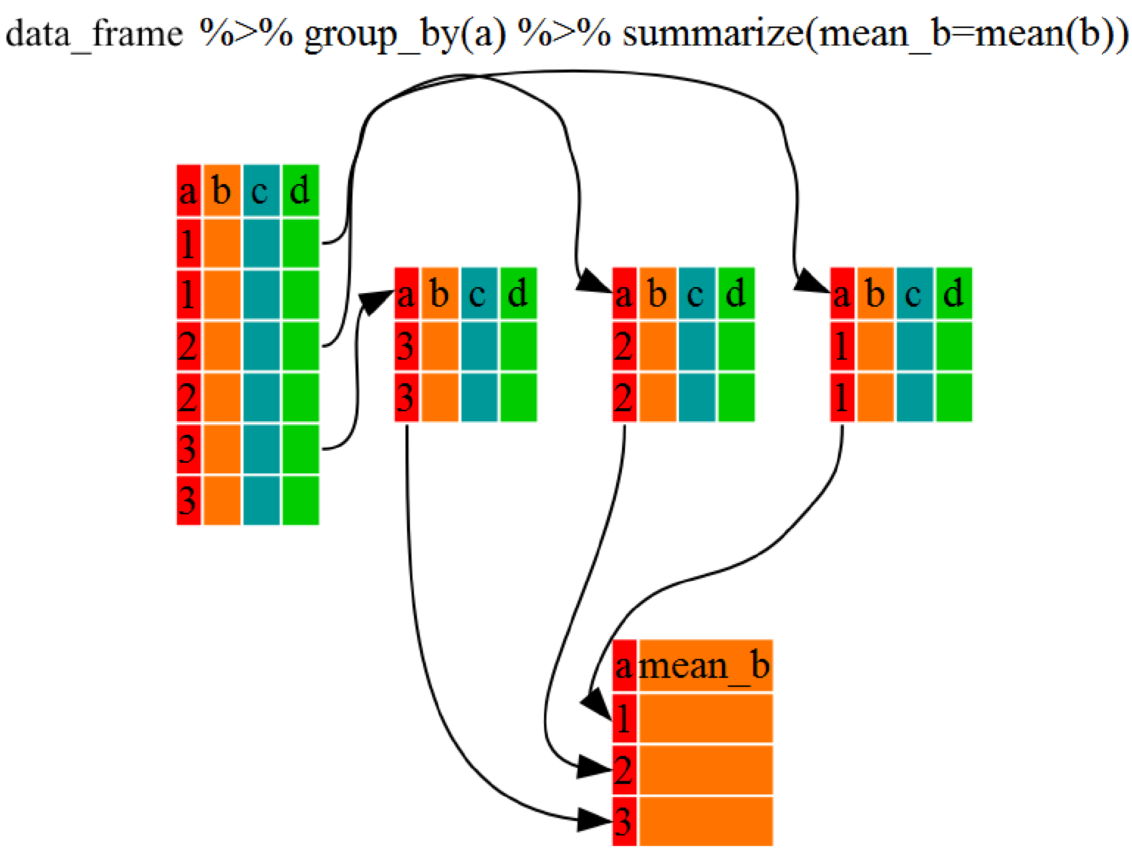


Figure 1.2: Split - Apply - Combine

`group_by()` is often used together with `summarize()`, which collapses each group into a single-row summary of that group. `group_by()` takes as arguments the column names that contain the **categorical** variables for which you want to calculate the summary statistics. So to view the mean age for black and white drivers:

```
trafficstops %>%
  group_by(driver_race) %>%
  summarize(mean_age = mean(driver_age, na.rm=TRUE))
```

```
#> # A tibble: 3 x 2
#>   driver_race mean_age
#>   <fct>          <dbl>
#> 1 Black           34.2
#> 2 White           36.2
#> 3 <NA>            34.5
```

If we wanted to remove the line with `NA` we could insert a `filter()` in the chain:

```
trafficstops %>%
  filter(!is.na(driver_race)) %>%
  group_by(driver_race) %>%
  summarize(mean_age = mean(driver_age, na.rm=TRUE))
```

```
#> # A tibble: 2 x 2
#>   driver_race mean_age
#>   <fct>          <dbl>
#> 1 Black           34.2
#> 2 White           36.2
```

Recall that `is.na()` is a function that determines whether something is an `NA`. The `!` symbol negates the result, so we're asking for everything that is *not* an `NA`.

You may have noticed that the output from these calls looks a little different. That's because **dplyr** has changed our `data.frame` object to an object of class `tbl_df`, also known as a "tibble". Tibble's data structure is very similar to a data frame. For our purposes the only differences are that (1) columns of class `character` are never converted into factors, and (2) in addition to displaying the data type of each column under its name, it only prints the first few rows of data and only as many columns as fit on one screen. If we wanted to print all columns we can use the print command, and set the `width` parameter to `Inf`. To print the first 6 rows for example we would do this: `print(my_tibble, n=6, width=Inf)`.

You can also group by multiple columns:

```
trafficstops %>%
  filter(!is.na(driver_race)) %>%
  group_by(driver_race, driver_gender) %>%
  summarize(mean_age = mean(driver_age, na.rm=TRUE))
```

```
#> # A tibble: 4 x 3
#> # Groups:   driver_race [?]
#>   driver_race driver_gender mean_age
#>   <fct>       <fct>            <dbl>
#> 1 Black       female            33.1
#> 2 Black       male              35.2
#> 3 White       female            35.7
#> 4 White       male              36.5
```

Once the data are grouped, you can also summarize multiple variables at the same time (and not necessarily on the same variable). For instance, we could add a column indicating the minimum age:

```
trafficstops %>%
  filter(!is.na(driver_race)) %>%
  group_by(driver_race, driver_gender) %>%
  summarize(mean_age = mean(driver_age, na.rm=TRUE),
            min_age = min(driver_age, na.rm=TRUE))
```

```
#> # A tibble: 4 x 4
#> # Groups:   driver_race [?]
#>   driver_race driver_gender mean_age min_age
#>   <fct>       <fct>            <dbl>   <dbl>
#> 1 Black       female            33.1      16
#> 2 Black       male              35.2       7
#> 3 White       female            35.7      15
#> 4 White       male              36.5      15
```

1.6 Tallying

When working with data, it is also common to want to know the number of observations found for each factor or combination of factors. For this, **dplyr** provides `tally()`. For example, if we wanted to see how many traffic stops each officer recorded we would do:

```
trafficstops %>%
  group_by(officer_id) %>%
  tally()
```

Here, `tally()` is the action applied to the groups created by `group_by()` and counts the total number of records for each category.

Alternatives:

```
trafficstops %>%
  count(officer_id) # count() calls group_by automatically, then tallies

trafficstops %>%
  group_by(officer_id) %>%
  summarize(n = n()) # n() is useful when count is needed for a calculation
```

We can optionally sort the results in descending order by adding `sort=TRUE`:

```
trafficstops %>%
  group_by(officer_id) %>%
  tally(sort=TRUE)
```

> Challenge
>
> Which 5 counties were the ones with the most stops in 2013? Hint: use the year() function from lubridate.

1.7 Joining two tables

It is not uncommon that we have our data spread out in different tables and need to bring those together for analysis. In this wexample we will combine the numbers of trafficstops for black and white drivers per county together with the numbers of the black and white total population for these counties. Toe population data are the estimated values of the 5 year average from the 2011-2015 American Community Survey (ACS):

```
MS_bw_pop <- read.csv("data/MS_acs2015_bw.csv")
head(MS_bw_pop)
```

```
#>              County  FIPS black_pop white_pop bw_pop
#> 1      Jones County 28067     19711     47154  66865
#> 2 Lauderdale County 28075     33893     43482  77375
#> 3       Pike County 28113     21028     18282  39310
#> 4    Hancock County 28045      4172     39686  43858
#> 5     Holmes County 28051     15498      3105  18603
#> 6    Jackson County 28059     30704    101686 132390
```

In a first step we will use a prevous `dplyr` command to count all the trafficstops per county.

```
trafficstops %>%
  group_by(county_fips) %>%
  summarise(n_stops = n()) %>% head()
```

```
#> # A tibble: 6 x 2
#>   county_fips n_stops
#>         <int>   <int>
#> 1       28001     942
#> 2       28003    3345
#> 3       28005    2921
#> 4       28007    4203
#> 5       28009     214
#> 6       28011    4526
```

We will then pipe this into our next operation.

`dplyr` can help us to bring the two tables together. We will use `left_join`, which returns all rows from the left table, and all columns from the left and the right table. As unique ID, which uniquely identifies the corresponding records in each table we use the County Names.

```
trafficstops %>%
  group_by(county_name) %>%
  summarise(n_stops = n()) %>%
  left_join(MS_bw_pop, by = c("county_name" = "County")) %>%
  head()
```

```
#> # A tibble: 6 x 6
#>   county_name    n_stops  FIPS black_pop white_pop bw_pop
#>   <fct>            <int> <int>     <int>     <int>  <int>
#> 1 Adams County       942 28001     17757     12856  30613
#> 2 Alcorn County     3345 28003      4281     31563  35844
#> 3 Amite County      2921 28005      5416      7395  12811
#> 4 Attala County     4203 28007      8194     10649  18843
#> 5 Benton County      214 28009      3078      5166   8244
#> 6 Bolivar County    4526 28011     21648     11197  32845
```

Now we can, for example calculate the percentage of the population that gets stopped in each county.

> Challenge
>
> Which county has the highest (lowest) percentage of stopped drivers? Use the snippet from above and pipe into the additional operations to do this.

`dplyr` join functions are generally equivalent `merge` from the base command, but there are a few advantages:

- rows are kept in existing order

- much faster
- tells you what keys you're merging by (if you don't supply)
- also work with database tables.

https://groups.google.com/d/msg/manipulatr/OuAPC4VyfIc/Qnt8mDfq0WwJ

See `?dplyr::join` for all the possible joins.

Chapter 2

Data Manipulation using `tidyr`

Learning Objectives

- Understand the concept of a wide and a long table format and for which purpose those formats are useful.
- Understand what key-value pairs are.
- Reshape a data frame from long to wide format and back with the `spread` and `gather` commands from the `tidyr` package.
- Export a data frame to a .csv file.

`dplyr` pairs nicely with `tidyr` which enables you to swiftly convert between different data formats for plotting and analysis.

The package `tidyr` addresses the common problem of wanting to reshape your data for plotting and use by different R functions. Sometimes we want data sets where we have one row per observation. Sometimes we want a data frame where each observation type has its own column, and rows are instead more aggregated groups - like surveys, where each column represents an answer. Moving back and forth between these formats is nontrivial, and `tidyr` gives you tools for this and more sophisticated data manipulation.

To learn more about `tidyr` after the workshop, you may want to check out this cheatsheet about `tidyr`.

2.1 About long and wide table format

The 'long' format is where:

- each column is a variable
- each row is an observation

In the 'long' format, you usually have 1 column for the observed variable and the other columns are ID variables.

For the 'wide' format a row, for example could be a reserach subject for which you have multiple observation variables containing the same type of data, for example responses to a set of survey questions, or repeated observations over time, or a mix of both. Here is an example:

	subject_ID	question_1	question_2	question_3
1	A	4.00	3.00	4.00
2	B	4.00	1.00	5.00
3	C	2.00	5.00	2.00

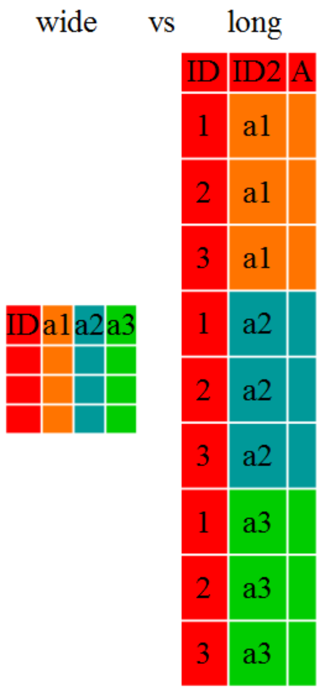


Figure 2.1: Wide vs. Long Table Format

You may find data input may be simpler or some other applications may prefer the 'wide' format. However, many of `R`'s functions have been designed assuming you have 'long' format data. This tutorial will help you efficiently transform your data regardless of original format.

The choice of data format affects readability. For humans, the wide format is often more intuitive, since we can often see more of the data on the screen due to its shape. However, the long format is more machine readable and is closer to the formatting of databases. The `ID` variables in our dataframes are similar to the fields in a database and observed variables are like the database values.

> Challenge 1
>
> Is trafficstops in a long or wide format?

2.2 Long to Wide with spread

Now let's see this in action. First, using **dplyr**, let's create a data frame with the mean age of each driver by gender and county:

```
trafficstops_ma <- trafficstops %>%
    filter(!is.na(driver_gender)) %>%
    group_by(county_name, driver_gender) %>%
    summarize(mean_age = mean(driver_age, na.rm = TRUE))

head(trafficstops_ma)
```

```
#> # A tibble: 6 x 3
#> # Groups:   county_name [3]
```

```
#>   county_name   driver_gender mean_age
#>   <fct>         <fct>            <dbl>
#> 1 Adams County  female            36.7
#> 2 Adams County  male              38.4
#> 3 Alcorn County female            33.3
#> 4 Alcorn County male              34.1
#> 5 Amite County  female            38.3
#> 6 Amite County  male              40.3
```

Now, to make this long data wide, we use `spread` from `tidyr` to spread out the driver gender into columns. `spread` takes three arguments - the data, the *key* column, or column with identifying information, the *values* column - the one with the numbers. We'll use a pipe so we can ignore the data argument.

```
trafficstops_ma_wide <- trafficstops_ma %>%
  spread(driver_gender, mean_age)

head(trafficstops_ma_wide)
```

```
#> # A tibble: 6 x 3
#> # Groups:   county_name [6]
#>   county_name    female  male
#>   <fct>           <dbl> <dbl>
#> 1 Adams County     36.7  38.4
#> 2 Alcorn County    33.3  34.1
#> 3 Amite County     38.3  40.3
#> 4 Attala County    36.7  38.1
#> 5 Benton County    32.1  34.4
#> 6 Bolivar County   33.2  36.3
```

We can now do things like compare the mean age of men against women drivers. As example we use the age difference to find the counties with the largest and with the smallest number. (A negative number means that female drivers are on average older than male drivers, a positive number means that male drivers are on average older than women drivers.)

```
trafficstops_ma_wide %>%
  mutate(agediff = male - female) %>%
  ungroup() %>%
  filter(agediff %in% range(agediff))
```

```
#> # A tibble: 2 x 4
#>   county_name      female  male agediff
#>   <fct>             <dbl> <dbl>   <dbl>
#> 1 Neshoba County     35.1  31.1   -3.94
#> 2 Yalobusha County   33.4  39.4    5.99
```

Note that `trafficstops_ma_wide` is derived from `trafficstops_ma`, and is a "grouped" data frame, which was created with the `group_by` function above. (Check `class(trafficstops_ma)` and `class(trafficstops_ma_wide)`). That means that any instruction that follows will operate on each group (in this case county) separately. That may be ok for some instances (like `mutate`), but if we are interested in retreiveing the max and the min age difference over all counties we need to `ungroup` the tibble to have the `filter` command operate on the entire dataset.

2.3 Wide to long with gather

What if we had the opposite problem, and wanted to go from a wide to long format? For that, we use `gather` to sweep up a set of columns into one key-value pair. We give it the arguments of a new key and value column name, and then we specify which columns we either want or do not want gathered up. So, to go backwards from `trafficstops_ma_wide`, and exclude `plot_id` from the gathering, we would do the following:

```
trafficstops_ma_long <- trafficstops_ma_wide %>%
  gather(gender, mean_age, -county_name)

head(trafficstops_ma_long)
```

```
#> # A tibble: 6 x 3
#> # Groups:   county_name [6]
#>   county_name    gender mean_age
#>   <fct>          <chr>     <dbl>
#> 1 Adams County   female     36.7
#> 2 Alcorn County  female     33.3
#> 3 Amite County   female     38.3
#> 4 Attala County  female     36.7
#> 5 Benton County  female     32.1
#> 6 Bolivar County female     33.2
```

We could also have used a specification for what columns to include. This can be useful if you have a large number of identifying columns, and it's easier to specify what to gather than what to leave alone. And if the columns are in a row, we don't even need to list them all out – just use the `:` operator!

```
trafficstops_ma_wide %>%
  gather(gender, mean_age, female:male) %>%
  head()
```

```
#> # A tibble: 6 x 3
#> # Groups:   county_name [6]
#>   county_name    gender mean_age
#>   <fct>          <chr>     <dbl>
#> 1 Adams County   female     36.7
#> 2 Alcorn County  female     33.3
#> 3 Amite County   female     38.3
#> 4 Attala County  female     36.7
#> 5 Benton County  female     32.1
#> 6 Bolivar County female     33.2
```

Challenge

1. Make a wide data frame with `year` as columns, `violation_raw` as rows, and the values are the number of traffic stops per each violation. Use year() from the lubridate package. You will need to summarize before reshaping

2. Now take that data frame, and make it long again, so each row is a unique `violation_raw` `year` combination.

Now that you have those commands under your belt, let's go back to our table from before and reshape it so we can easily calculate the percentage of black and white. To clean things up a little, we remove rows where driver race is unknown.

We then make sure that we count our `NA`s as `0`. We know from earlier that in Tunica County all reported stops are for black drivers (Check again with: `trafficstops %>% filter(county_name == "Tunica County")`).

By default spread would set the value for white stops to NA. Sometimes it is fine to leave those as NA. Sometimes we want to fill them as zeros, in which case we would add the argument `fill = 0`. In our case we prefer to count this a 0.

Lastly, we introduce a separator (`sep`) as parameter to the spread command. If `sep` is not `NULL`, the column names will be given by `"<key_name><sep><key_value>"` and make them more explicit, easier for you to interpret, and for anyone who might use your data.

```
trafficstops %>%
  filter(!is.na(driver_race)) %>%
  count(county_name, county_fips, driver_race) %>%
  spread(driver_race, n, fill = 0, sep = "_") %>%
  head()
```

Now we can pipe this into our left join and calculate the percentages:

```
# make sure you this table loaded:
MS_bw_pop <- read.csv("data/MS_acs2015_bw.csv")

trafficstops %>%
  filter(!is.na(driver_race)) %>%
  count(county_name, county_fips, driver_race) %>%
  spread(driver_race, n, fill = 0, sep = "_") %>%
  left_join(MS_bw_pop, by = c("county_fips" = "FIPS")) %>%
  mutate(pct_black_stopped = driver_race_Black/black_pop,
         pct_white_stopped = driver_race_White/white_pop) %>%
  print(n=5, width=Inf)
```

```
#> # A tibble: 82 x 10
#>   county_name   county_fips driver_race_Black driver_race_White
#>   <fct>               <int>             <dbl>             <dbl>
#> 1 Adams County        28001               583               359
#> 2 Alcorn County       28003               468              2877
#> 3 Amite County        28005              1589              1331
#> 4 Attala County       28007              2096              2107
#> 5 Benton County       28009               121                93
#>   County        black_pop white_pop bw_pop pct_black_stopped
#>   <fct>             <int>     <int>  <int>             <dbl>
#> 1 Adams County      17757     12856  30613            0.0328
#> 2 Alcorn County      4281     31563  35844            0.109
#> 3 Amite County       5416      7395  12811            0.293
#> 4 Attala County      8194     10649  18843            0.256
#> 5 Benton County      3078      5166   8244            0.0393
#>   pct_white_stopped
#>               <dbl>
#> 1            0.0279
#> 2            0.0912
#> 3            0.180
#> 4            0.198
#> 5            0.0180
#> # ... with 77 more rows
```

Terrific.

Now let's use some visualization to help us understand our data. Before we do this though, let's save this table out.

2.4 Exporting data

Instead of printing the above output to the screen we will pipe it into another command. Similar to the `read.csv()` function used for reading CSV files into R, there is a `write.csv()` function that generates CSV files from data frames.

Before using `write.csv()`, we are going to create a new folder, `data_output`, in our working directory that will store this generated dataset. We don't want to write generated datasets in the same directory as our raw data. It's good practice to keep them separate. The `data` folder should only contain the raw, unaltered data, and should be left alone to make sure we don't delete or modify it. In contrast, our script will generate the contents of the `data_output` directory, so even if the files it contains are deleted, we can always re-generate them.

We can save the table generated by the join as a CSV file in our `data_output` folder. By default, `write.csv()` includes a column with row names (in our case the names are just the row numbers), so we need to add `row.names = FALSE` so they are not included:

```
trafficstops %>%
  filter(!is.na(driver_race)) %>%
  count(county_name, county_fips, driver_race) %>%
  spread(driver_race, n, fill = 0, sep = "_") %>%
  left_join(MS_bw_pop, by = c("county_fips" = "FIPS")) %>%
  mutate(pct_black_stopped = driver_race_Black/black_pop,
         pct_white_stopped = driver_race_White/white_pop) %>%
  write.csv(file = "data_output/MS_demographic.csv", row.names = FALSE)
```

Chapter 3

Data Visualization with ggplot2

Learning Objectives

- Bind a data frame to a plot
- Select variables to be plotted and variables to define the presentation such as size, shape, color, transparency, etc. by defining aesthetics (**aes**)
- Add a graphical representation of the data in the plot (points, lines, bars) adding "geoms" layers
- Produce scatter plots, barplots, boxplots, and line plots using ggplot.
- Modify the aesthetics for the entire plot as well as for individual "geoms" layers
- Modify plot elements (labels, text, scale, orientation)
- Group observations by a factor variable
- Break up plot into multiple panels (facetting)
- Apply ggplot themes and create and apply customized themes
- Save a plot created by ggplot as an image

We start by loading the required packages. **ggplot2** is included in the **tidyverse** package.

```
library(tidyverse)
```

If not still in the workspace, load the data we saved in the previous lesson.

```
MS_demographic <- read.csv('data_output/MS_demographic.csv')
```

(If you need to, you can also download the data from here: https://github.com/cengel/R-data-wrangling/raw/master/data_output/MS_demographic.csv)

3.1 Plotting with ggplot2

ggplot2 is a plotting package that makes it simple to create complex plots from data in a data frame. It provides a more programmatic interface for specifying what variables to plot, how they are displayed, and general visual properties, so we only need minimal changes if the underlying data change or if we decide to change from a bar plot to a scatterplot. This helps in creating publication quality plots with minimal amounts of adjustments and tweaking.

ggplot generally likes data in the 'long' format: i.e., a column for every dimension, and a row for every observation. Well structured data will save you lots of time when making figures with ggplot.

ggplot graphics are built step by step by adding new elements using the **+** sign.

To build a ggplot we need to:

- bind the plot to a specific data frame using the `data` argument

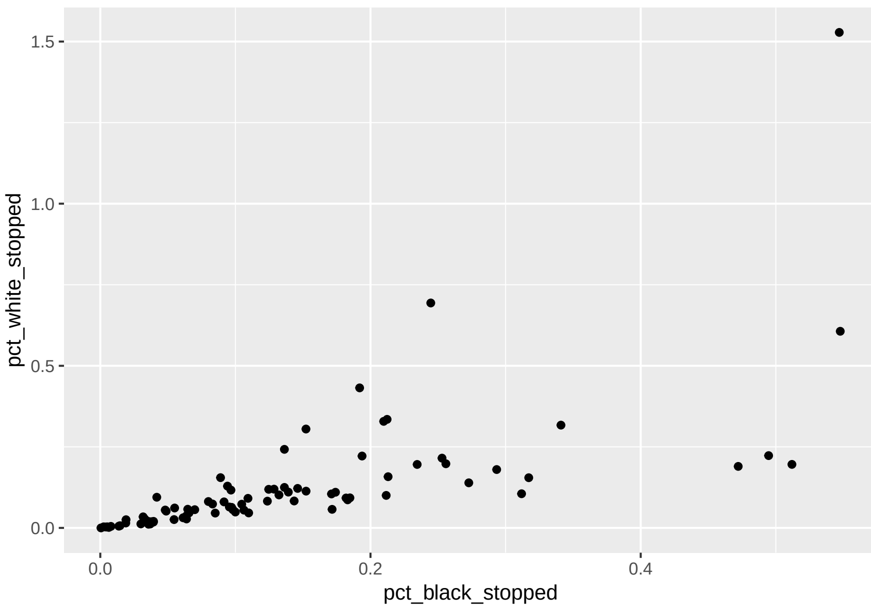
```
ggplot(data = MS_demographic)
```

- define aesthetics (`aes`), by selecting the variables to be plotted and the variables to define the presentation such as plotting size, shape color, etc.

```
ggplot(data = MS_demographic, aes(x = pct_black_stopped, y = pct_white_stopped))
```

- add "geoms" – a graphical representation of the data in the plot (points, lines, bars). To add a geom to the plot use `+` operator

```
ggplot(data = MS_demographic, aes(x = pct_black_stopped, y = pct_white_stopped)) +
  geom_point()
```



The `+` in the **`ggplot2`** package is particularly useful because it allows you to modify existing `ggplot` objects. This means you can easily set up plot "templates" and conveniently explore different types of plots, so the above plot can also be generated with code like this:

```
# Assign plot to a variable
MS_plot <- ggplot(data = MS_demographic, aes(x = pct_black_stopped, y = pct_white_stopped))

# Draw the plot
MS_plot + geom_point()
```

Notes:

- Any parameters you set in the `ggplot()` function can be seen by any geom layers that you add (i.e., these are universal plot settings). This includes the x and y axis you set up in `aes()`.
- Any parameters you set in the `geom_*()` function are treated independently of (and override) the settings defined globally in the `ggplot()` function.
- Geoms are plotted in the order they are added after each `+`, that means geoms last added will display on top of prior geoms.
- The `+` sign used to add layers **must be placed at the end of each line** containing a layer. If, instead, the `+` sign is added in the line before the other layer, **`ggplot2`** will not add the new layer and will return an error message.

```
# this is the correct syntax for adding layers
MS_plot +
  geom_point()

# this will not add the new layer and will return an error message
MS_plot
  + geom_point()
```

To learn more about **ggplot** after the workshop, you may want to check out this cheatsheet about **ggplot**.

3.2 Building your plots iteratively

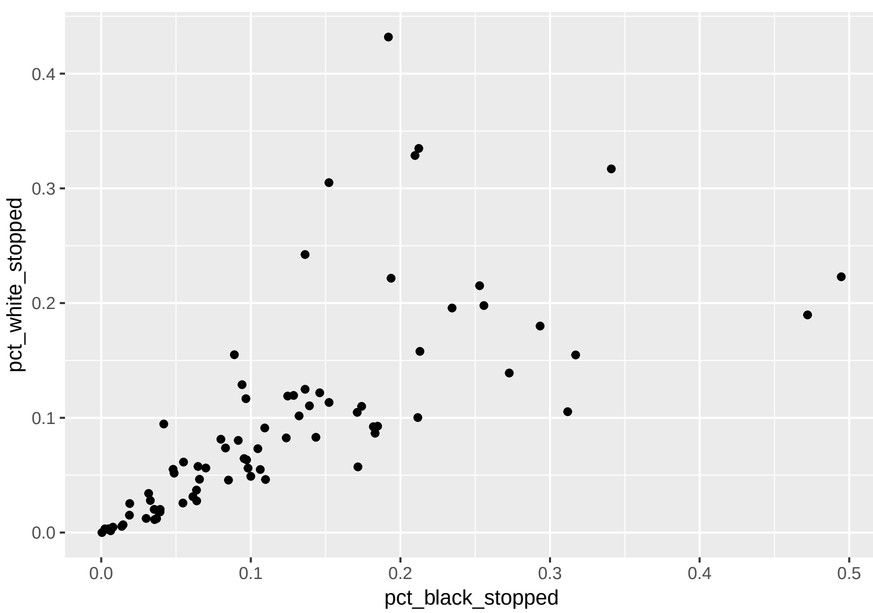
Building plots with ggplot can be of great help when you engage in exploratory data analysis. It is typically an iterative process, where you go back and forth between your data and their graphical representation, which helps you in the process of getting to know your data better.

Conveniently, `ggplot` works with pipes. The code below does the same thing as above:

```
MS_demographic %>%
  ggplot(aes(x = pct_black_stopped, y = pct_white_stopped)) +
  geom_point()
```

We pipe the content of the table into `ggplot()`, so we can omit the first (`data =` argument). Now let's use this to clean up a few odd outliers in our data before we pass them to ggplot.

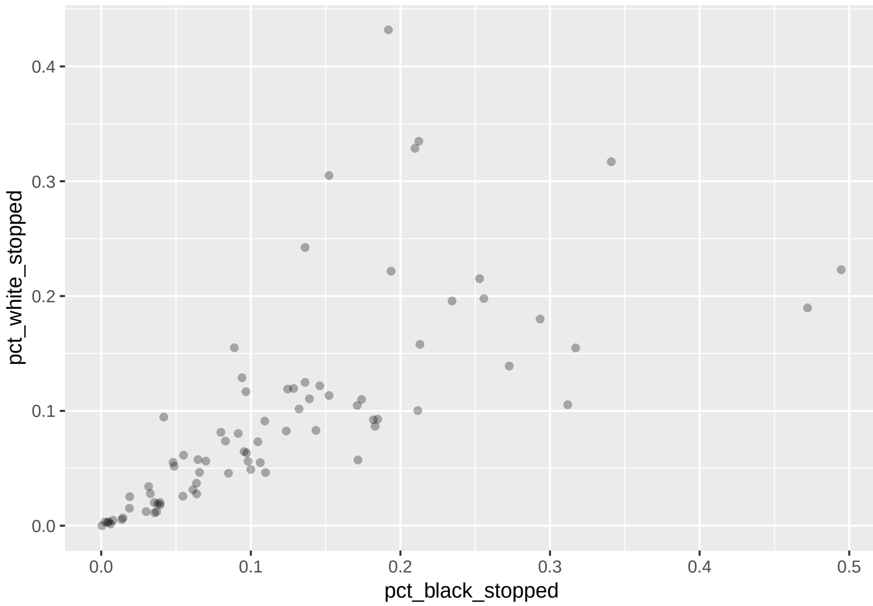
```
MS_demographic %>%
  filter(pct_white_stopped < 0.5 & pct_black_stopped < 0.5) %>%
  ggplot(aes(x = pct_black_stopped, y = pct_white_stopped)) +
  geom_point()
```



Then we can start modifying this plot to extract more information from it. For instance, we can add transparency (`alpha`) to avoid overplotting:

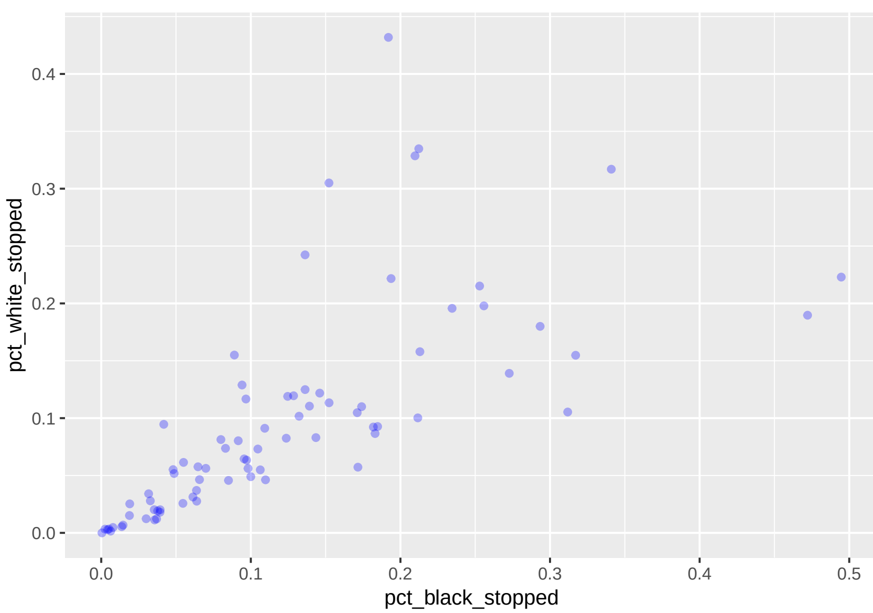
```
MS_demographic %>%
  filter(pct_white_stopped < 0.5 & pct_black_stopped < 0.5) %>%
```

```
  ggplot(aes(x = pct_black_stopped, y = pct_white_stopped)) +
  geom_point(alpha = 0.3)
```



We can also add a color for all the points:

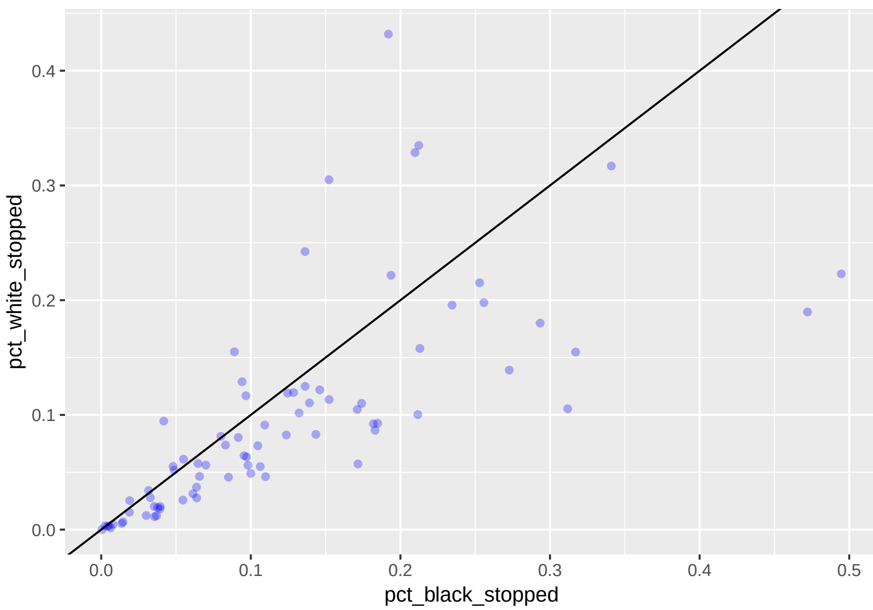
```
MS_demographic %>%
  filter(pct_white_stopped < 0.5 & pct_black_stopped < 0.5) %>%
  ggplot(aes(x = pct_black_stopped, y = pct_white_stopped)) +
  geom_point(alpha = 0.3, color= "blue")
```



We can add another layer to the plot with +:

```
MS_demographic %>%
  filter(pct_white_stopped < 0.5 & pct_black_stopped < 0.5) %>%
  ggplot(aes(x = pct_black_stopped, y = pct_white_stopped)) +
  geom_point(alpha = 0.3, color= "blue") +
```

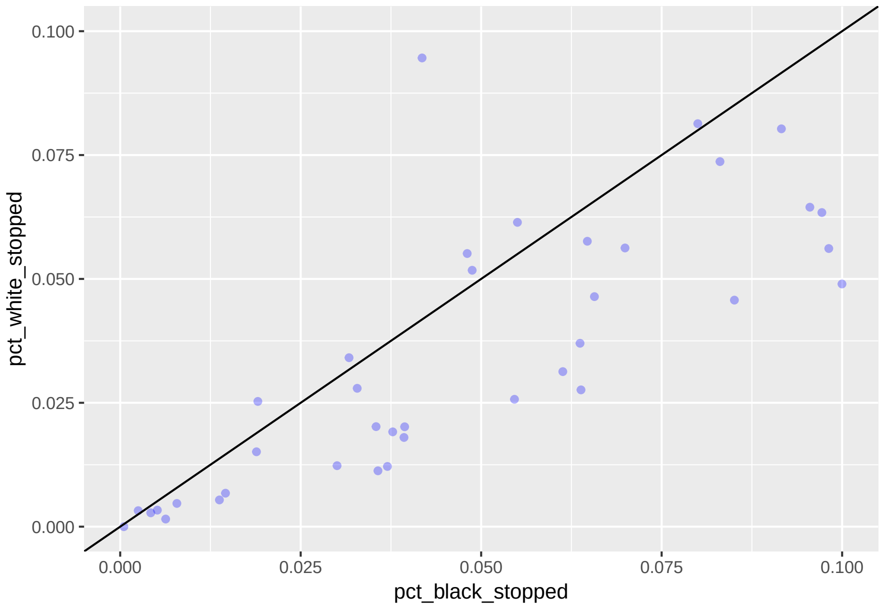
```
  geom_abline(intercept = 0)
```



If we wanted to "zoom" into the plot, we could filter to a smaller range of values before passing them to ggplot, but we can also tell ggplot to only plot the x and y values for certain ranges. For this we use `scale_x_continuous` and `scale_y_continuous`. You will receive a message from ggplot telling you how many rows it has removed from the plot.

```
MS_demographic %>%
  filter(pct_white_stopped < 0.5 & pct_black_stopped < 0.5) %>%
  ggplot(aes(x = pct_black_stopped, y = pct_white_stopped)) +
  geom_point(alpha = 0.3, color= "blue") +
  geom_abline(intercept = 0) +
  scale_x_continuous(limits = c(0, 0.1)) +
  scale_y_continuous(limits = c(0, 0.1))
```

```
#> Warning: Removed 40 rows containing missing values (geom_point).
```



Challenge

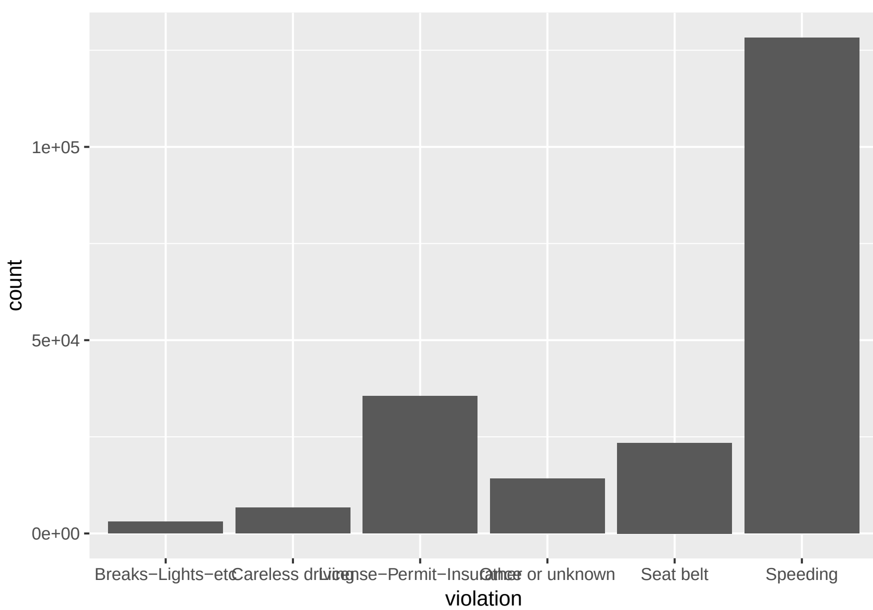
Modify the plot above to display different color for both points and abline, and show a different range of data. How might you change the size of the dots?

3.3 Barplot

There are two types of bar charts in ggplot, `geom_bar` and `geom_col`. `geom_bar` makes the height of the bar proportional to the number of cases in each group and counts the number of cases at each x position.

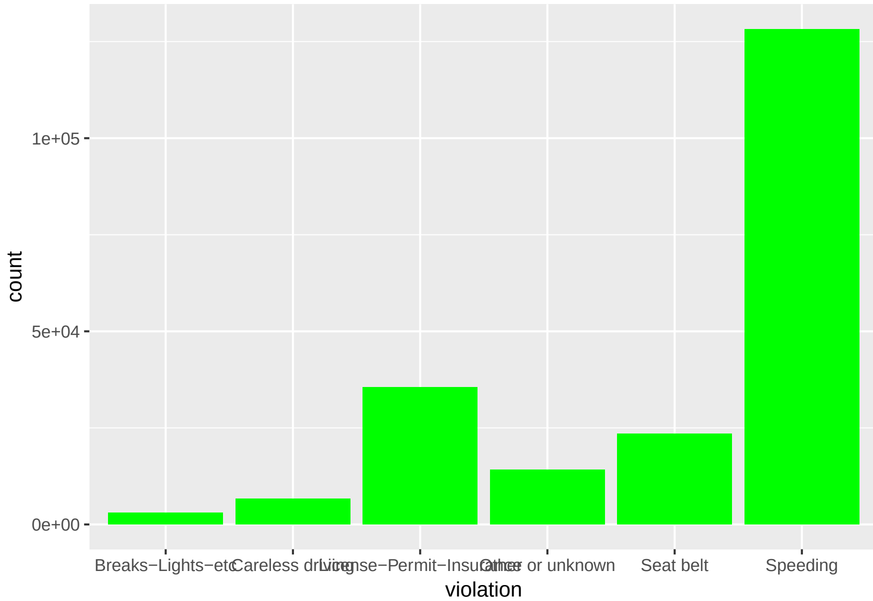
If we wanted to see how many violations we have of each type could say:

```
ggplot(trafficstops, aes(violation)) +
  geom_bar()
```



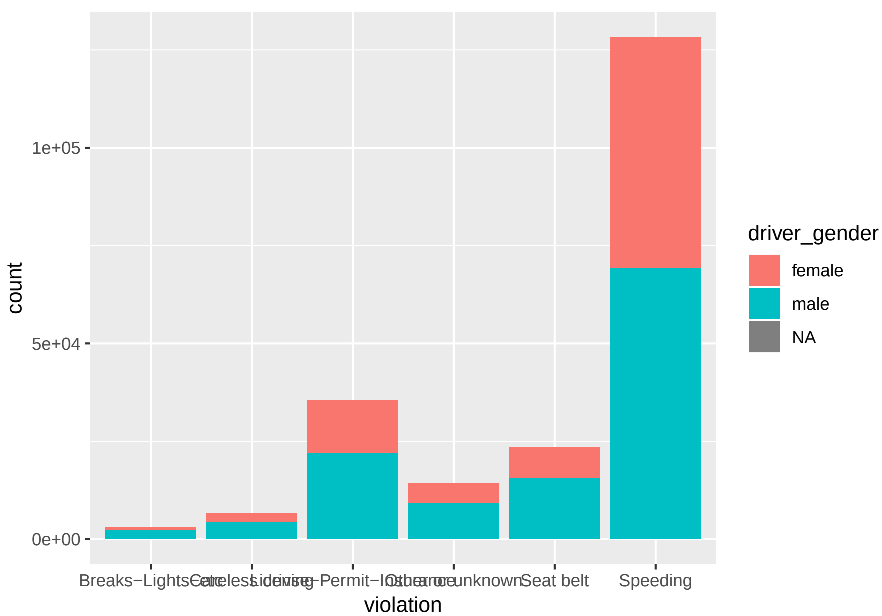
As we have seen we could color the bars, but instead of `color` we use `fill`. (What happens when you use `color`?)

```
ggplot(trafficstops, aes(violation)) +
  geom_bar(fill = "green")
```



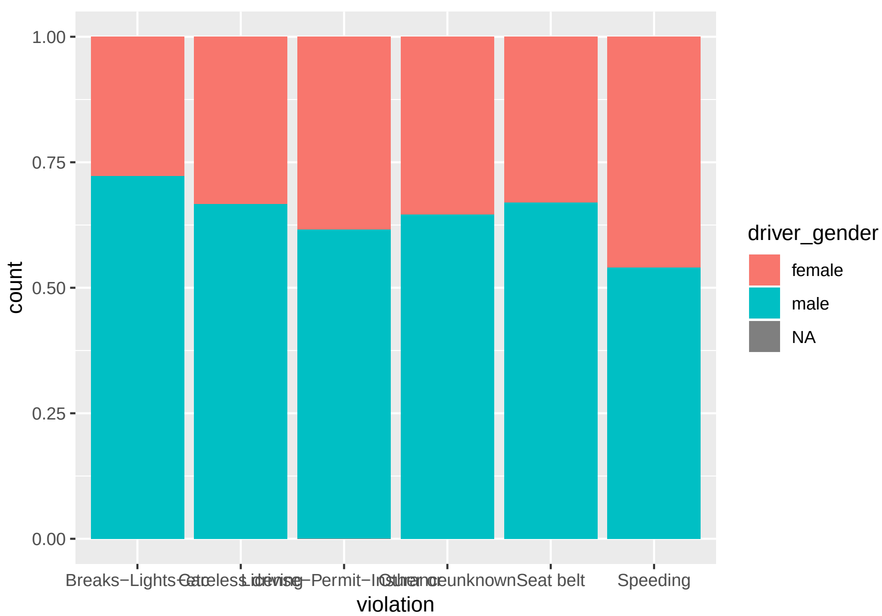
Instead of coloring everything the same we could also color by another category, say gender. For this we have to set the parameter within the `aes()` function, which takes care of mapping the values to different colors:

```
ggplot(trafficstops, aes(violation)) +
  geom_bar(aes(fill = driver_gender))
```



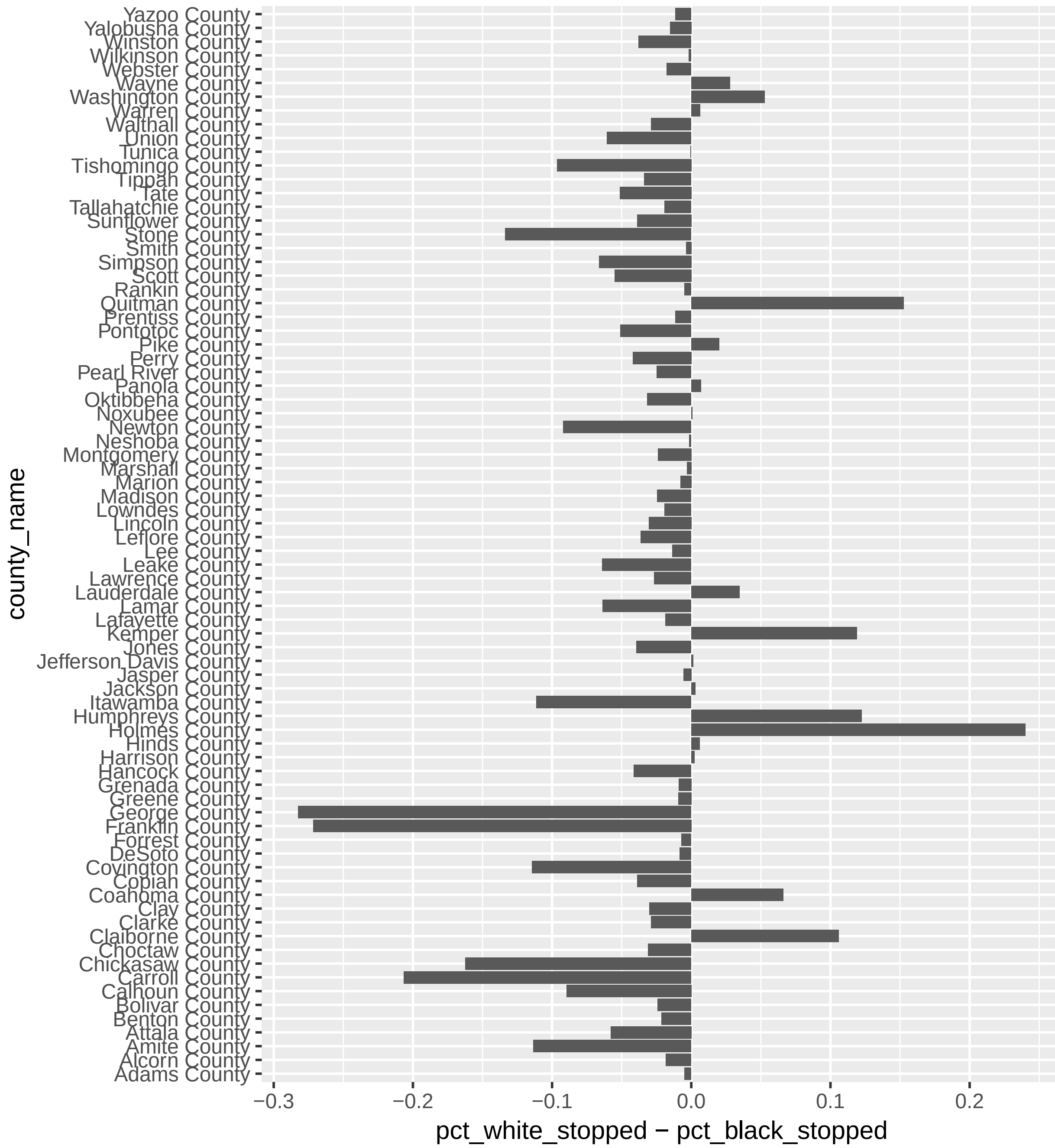
If we wanted to see the proportions within each category we can tell ggplot to stretch the bars between 0 and 1, we can set the position parameter to 'fill':

```r
ggplot(trafficstops, aes(violation)) +
  geom_bar(aes(fill = driver_gender), position = "fill")
```



The other type of barchart, `geom_col`, is used if you want the heights of the bars to represent values in the data. It leaves the data as is. For example, we can use `geom_col` for a different way of visualizing the data shown in the scatterplot above. For readability I have also flipped the coordinates:

```r
MS_demographic %>%
  filter(pct_white_stopped < 0.5 & pct_black_stopped < 0.5) %>%
  ggplot(aes(x = county_name, y = pct_white_stopped - pct_black_stopped)) +
    geom_col() +
    coord_flip()
```



Challenge

Make a barplot that shows for each race the proportion of stops for male and female drivers. How could you get rid of the NAs?

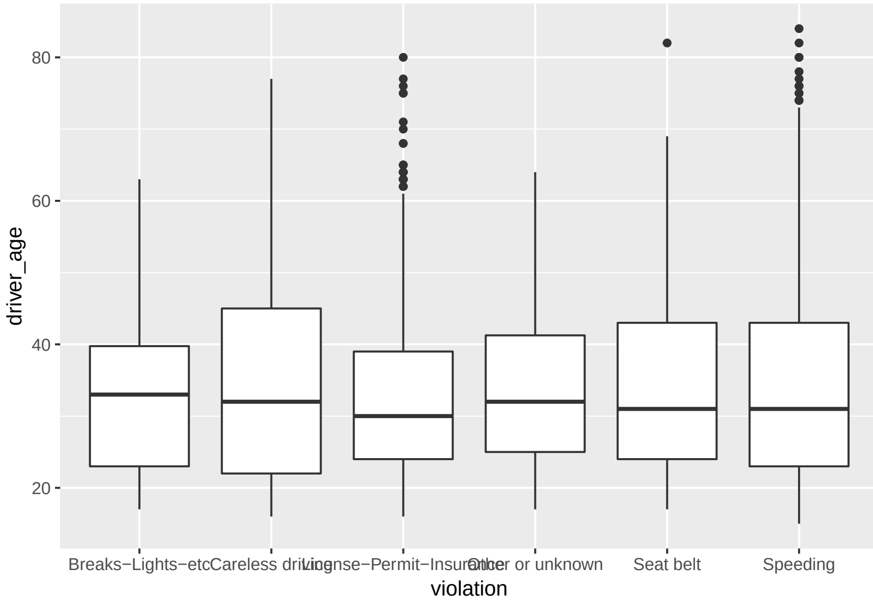
3.4 Boxplot

For this segment let's extract and work with the stops for Chickasaw County only.

```
Chickasaw_stops <- filter(trafficstops, county_name == "Chickasaw County")
```

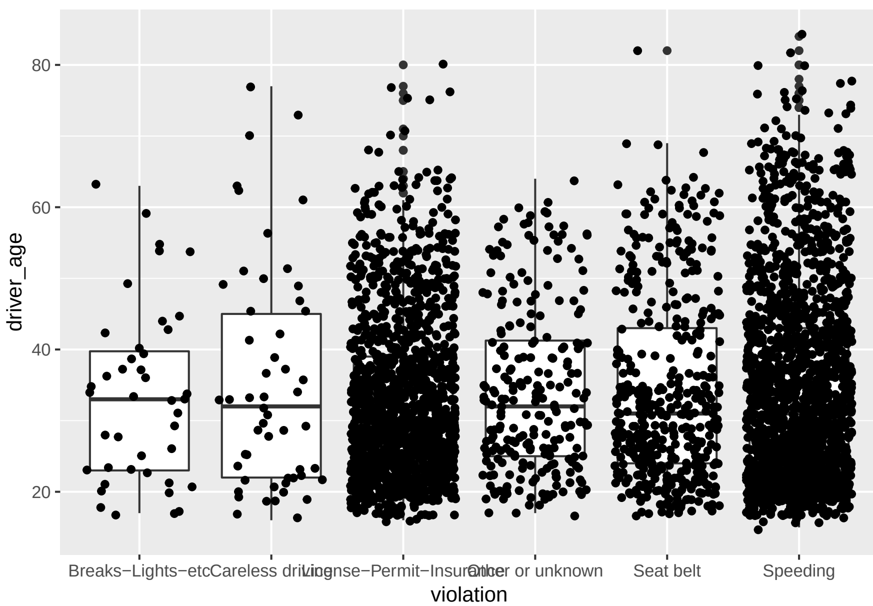
We can use boxplots to visualize the distribution of driver age within each violation:

```
ggplot(data = Chickasaw_stops, aes(x = violation, y = driver_age)) +
    geom_boxplot()
```



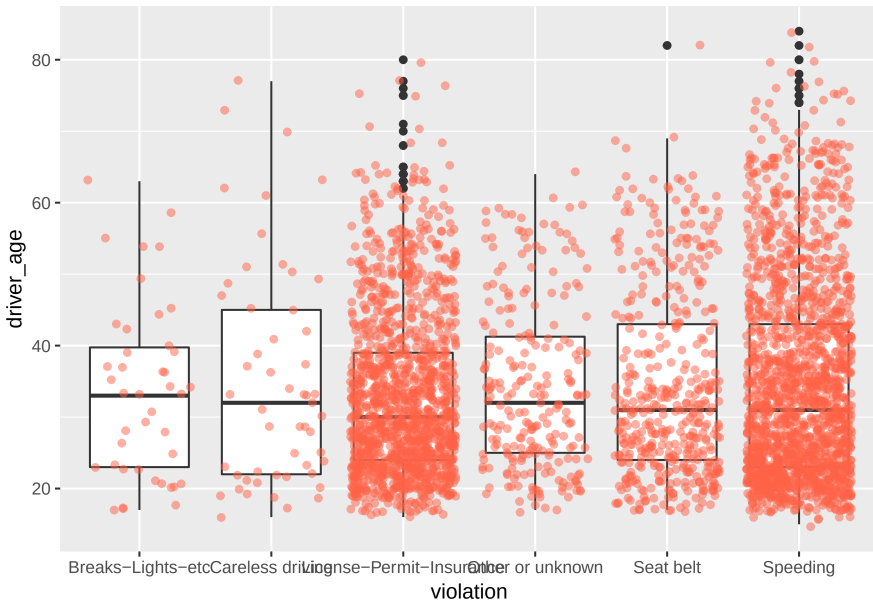
By adding points to boxplot, we can have a better idea of the number of measurements and of their distribution.

```
ggplot(data = Chickasaw_stops, aes(x = violation, y = driver_age)) +
    geom_boxplot() +
    geom_jitter()
```



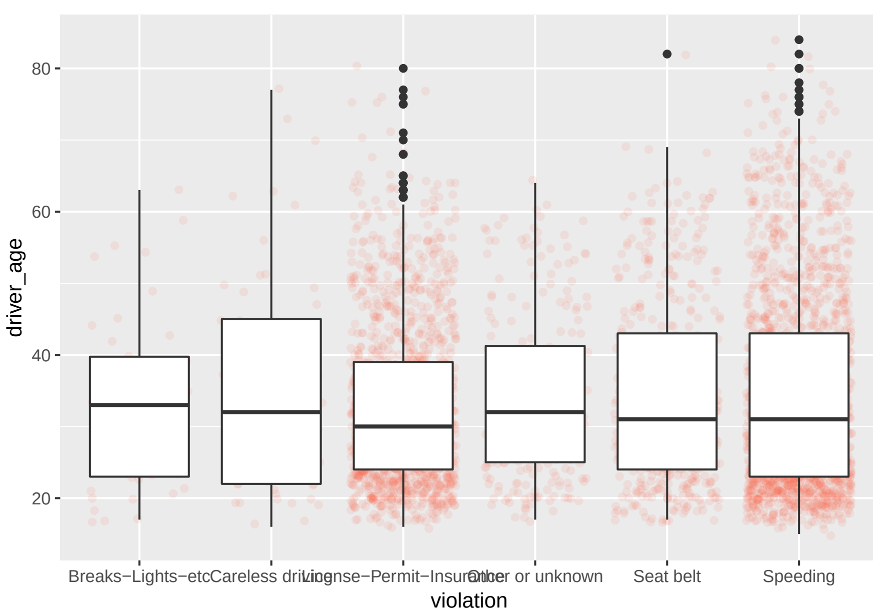
That looks quite messy. Let's clean it up by using the `alpha` parameter to make the dots more transparent and also change their color:

```
ggplot(data = Chickasaw_stops, aes(x = violation, y = driver_age)) +
    geom_boxplot() +
    geom_jitter(alpha = 0.5, color = "tomato")
```



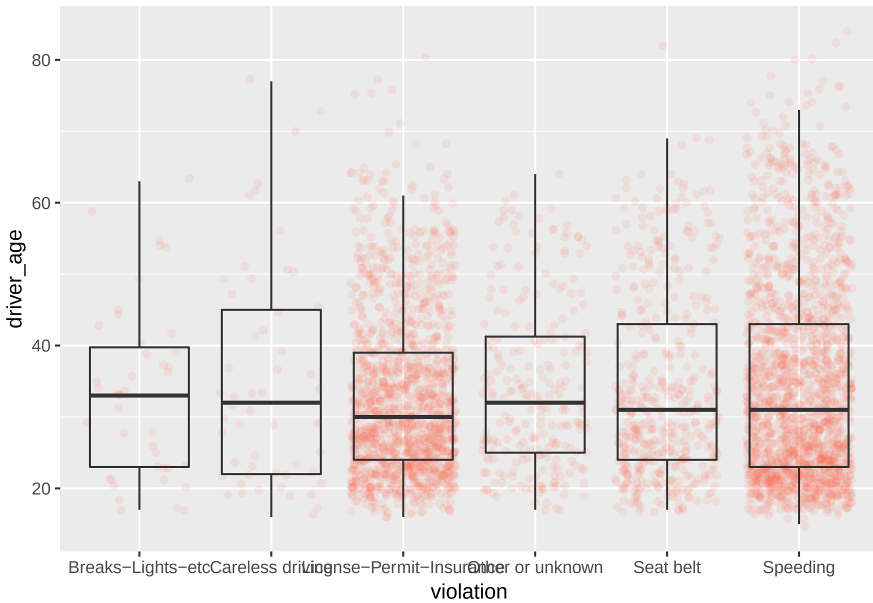
Notice how the boxplot layer is behind the jitter layer. We will change the plotting order to keep the boxplot visible.

```
ggplot(data = Chickasaw_stops, aes(x = violation, y = driver_age)) +
    geom_jitter(alpha = 0.1, color = "tomato") +
    geom_boxplot()
```



And finally we will change the transparency of the box plot so it does not cover the points:

```
ggplot(data = Chickasaw_stops, aes(x = violation, y = driver_age)) +
    geom_jitter(alpha = 0.1, color = "tomato") +
    geom_boxplot(alpha = 0)
```



Challenge

Boxplots are useful summaries, but hide the *shape* of the distribution. For example, if there is a bimodal distribution, it would not be observed with a boxplot. An alternative to the boxplot is the violin plot (sometimes known as a beanplot), where the shape (of the density of points) is drawn.

- Replace the box plot with a violin plot; see `geom_violin()`.

So far, we've looked at the distribution of age within violations Try making a new plot to explore the distribution of age for another variable:

- Create the age box plot for `driver_race`. Overlay the boxplot layer on a jitter layer to show actual measurements.

3.5 Plotting time series data

To make things a little easer we first convert the date column we plan to use to Date format.

```
library(lubridate)
class(trafficstops$stop_date)
trafficstops$stop_date <- ymd(trafficstops$stop_date)
class(trafficstops$stop_date)
```

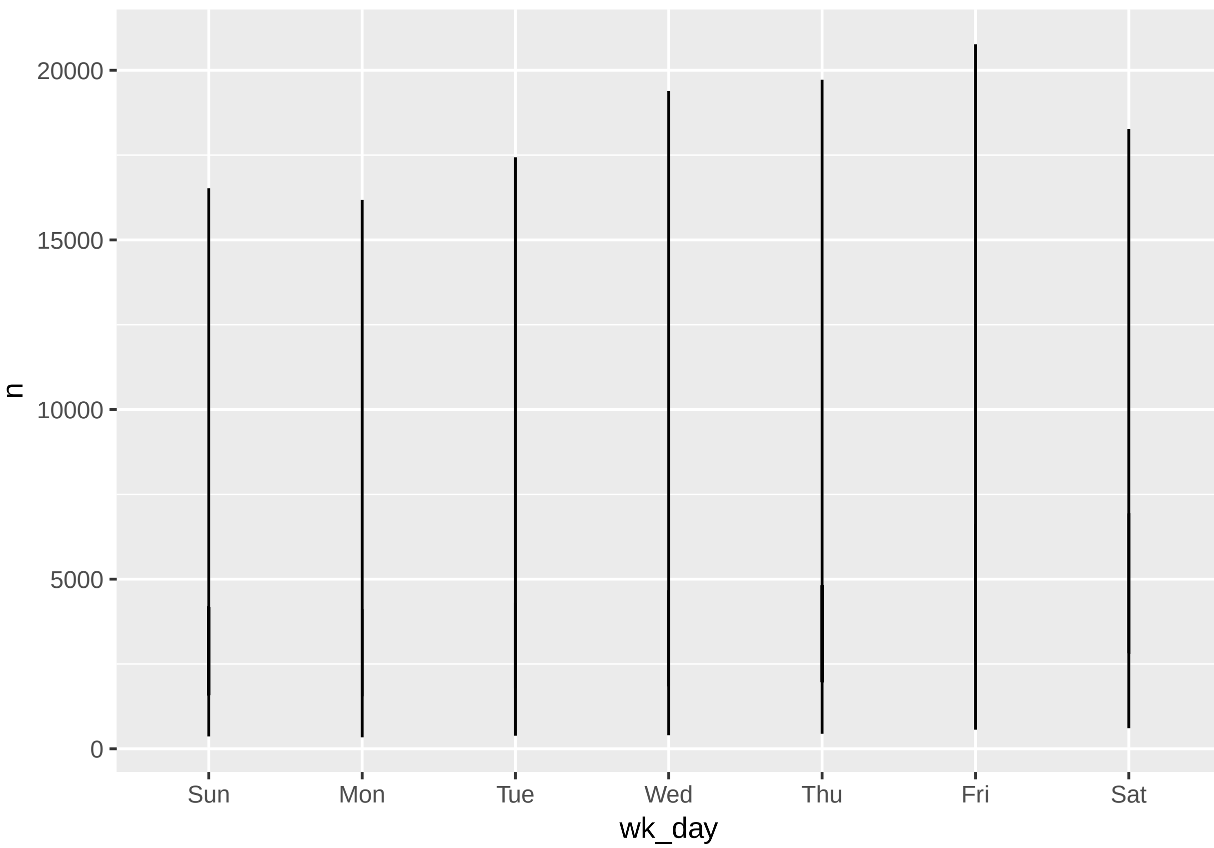
Let's calculate number of violation per weekday. For better understanding we will label the weekdays. First we need to group the data and count records within each group:

```
trafficstops %>%
  mutate(wk_day = wday(stop_date, label = TRUE)) %>%
  group_by(wk_day, violation) %>%
  tally
```

Timelapse data can be visualized as a line plot (with – you guessed it – `geom_line()`) mapping the days to the x axis and counts to the y axis. So we pipe the output from above into ggplot like this:

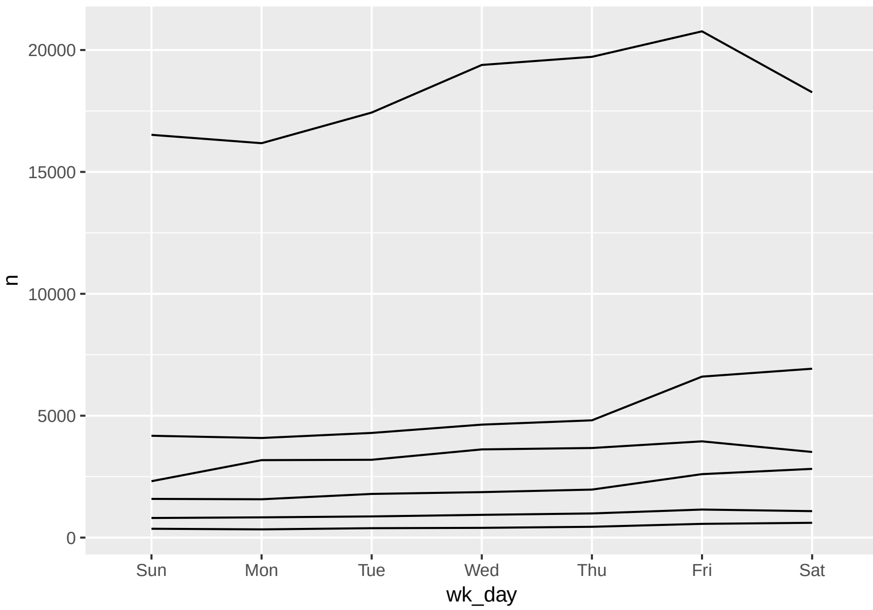
```
trafficstops %>%
  mutate(wk_day = wday(stop_date, label = TRUE)) %>%
```

```
  group_by(wk_day, violation) %>%
  tally %>%
  ggplot(aes(x = wk_day, y = n)) +
    geom_line()
```



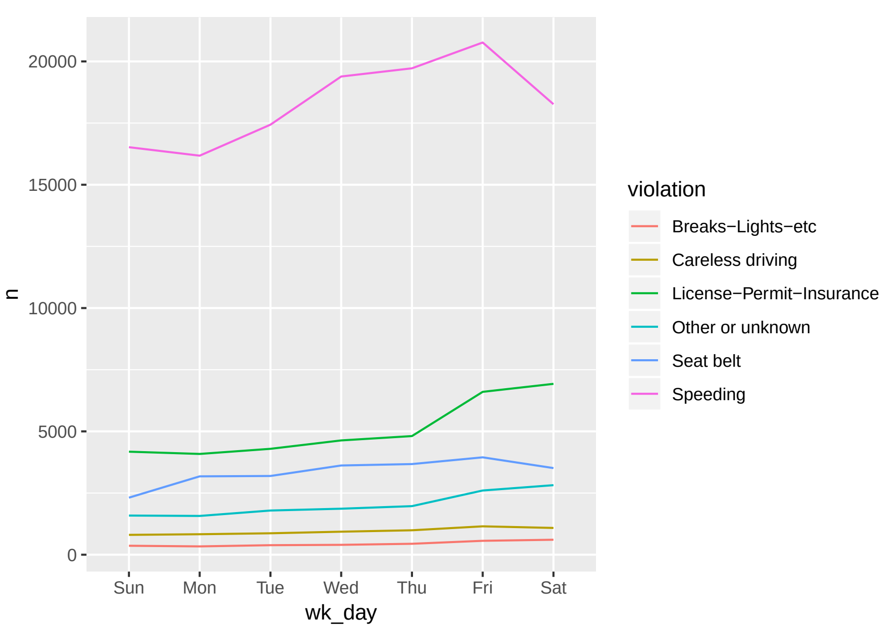
Unfortunately, this does not work because we plotted data for all the violations together. So what ggplot displays is the range of all values for each year in a vertial line. We need to tell ggplot to draw a line for each violation by modifying the aesthetic function to include `group = violation`:

```
trafficstops %>%
  mutate(wk_day = wday(stop_date, label = TRUE)) %>%
  group_by(wk_day, violation) %>%
  tally %>%
  ggplot(aes(x = wk_day, y = n, group = violation)) +
    geom_line()
```



We will be able to distinguish violations in the plot if we add colors. (Colors groups automatically if the variable is numeric).

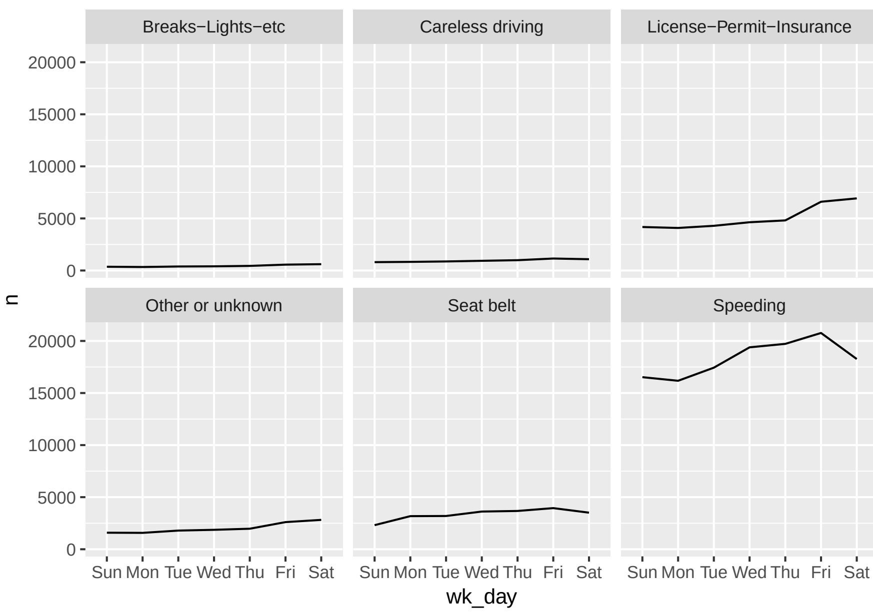
```
trafficstops %>%
  mutate(wk_day = wday(stop_date, label = TRUE)) %>%
  group_by(wk_day, violation) %>%
  tally %>%
  ggplot(aes(x = wk_day, y = n, group = violation, color = violation)) +
     geom_line()
```



3.6 Faceting

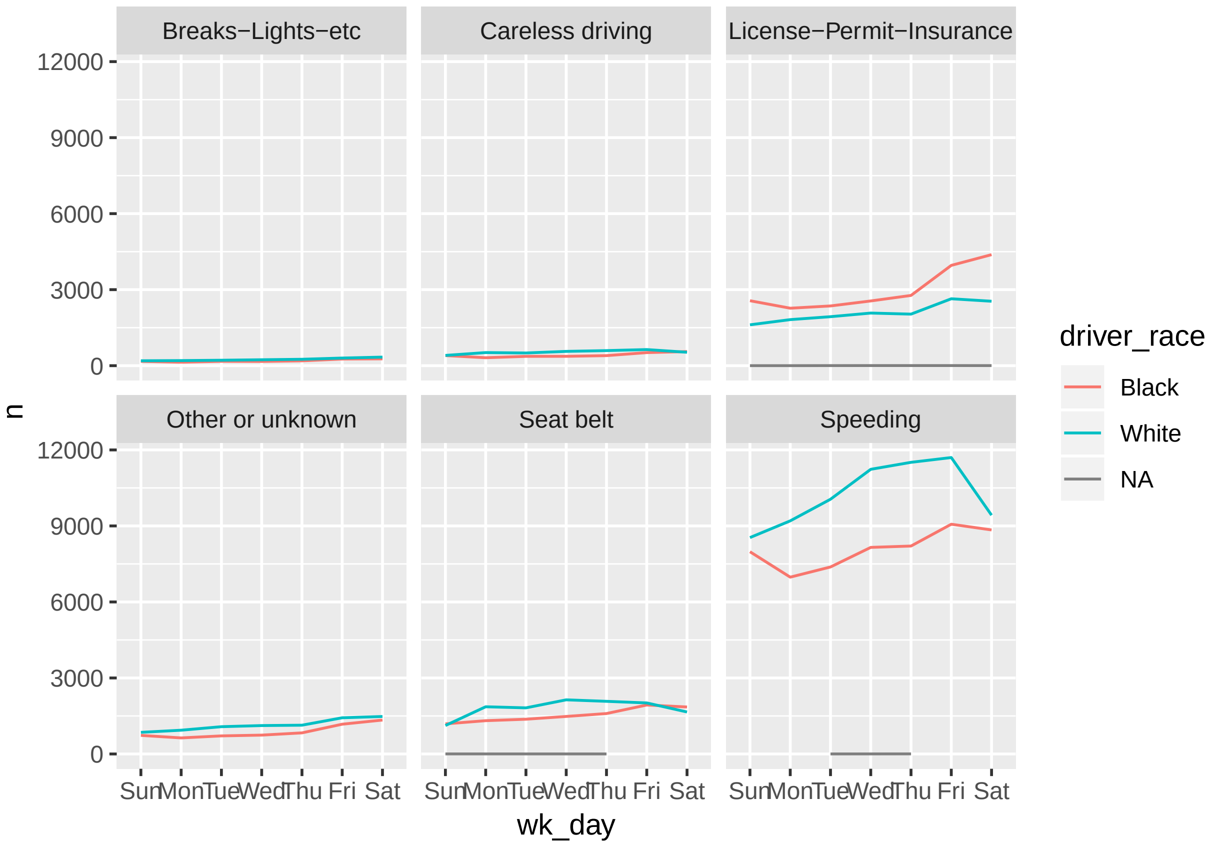
ggplot has a special technique called *faceting* that allows to split one plot into multiple plots based on a factor included in the dataset. We will use it to make a time series plot for each violation:

```
trafficstops %>%
  mutate(wk_day = wday(stop_date, label = TRUE)) %>%
  group_by(wk_day, violation) %>%
  tally %>%
  ggplot(aes(x = wk_day, y = n, group = violation)) +
     geom_line() +
     facet_wrap(~ violation)
```



Now we would like to split the line in each plot by the race of the driver. To do that we need to make counts in the data frame grouped by `day`, `violation`, and `driver_race`. We then make the faceted plot by splitting further by race using `color` and `group` (within a single plot):

```
trafficstops %>%
  mutate(wk_day = wday(stop_date, label=TRUE)) %>%
  group_by(wk_day, violation, driver_race) %>%
  tally %>%
  ggplot(aes(x = wk_day, y = n, color = driver_race, group = driver_race)) +
  geom_line() +
  facet_wrap(~ violation)
```



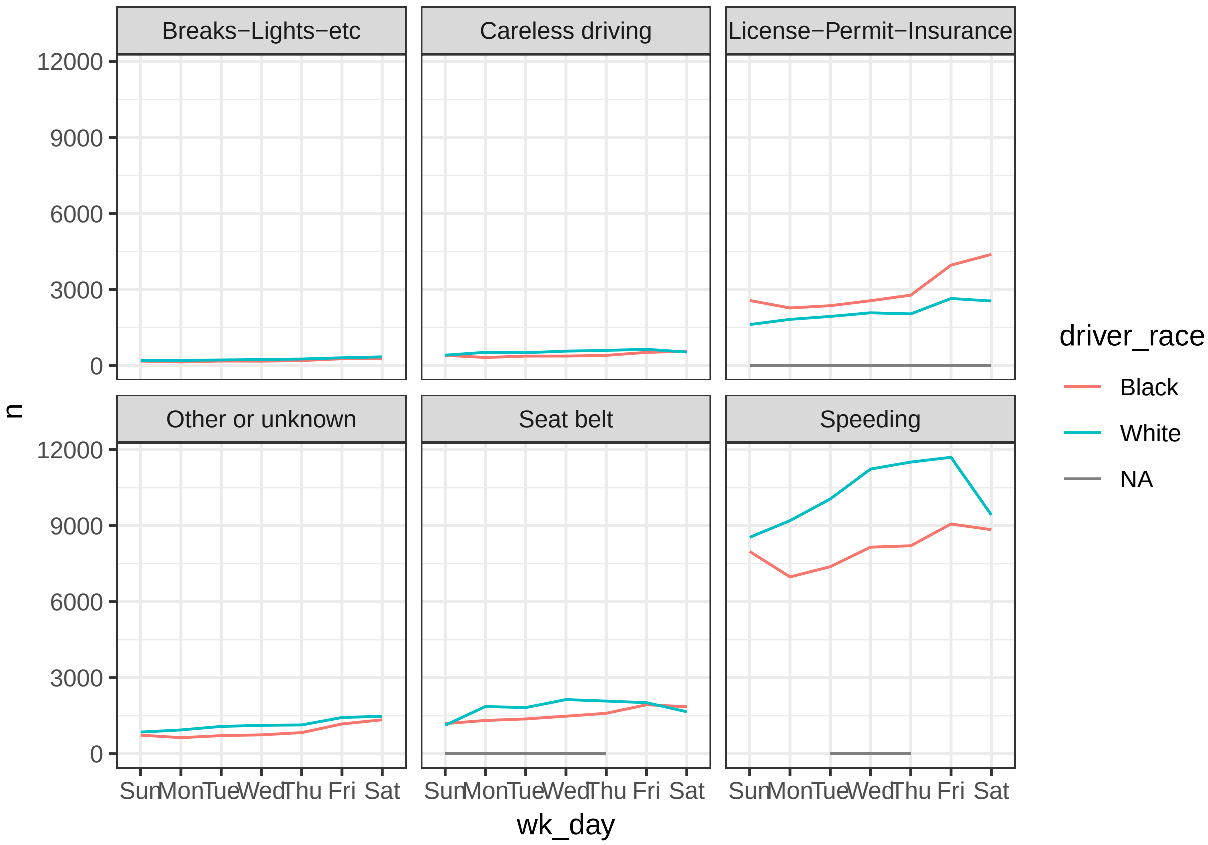
Note that there is an alternative, the `facet_grid` geometry, which allows you to explicitly specify how you want your plots to be arranged via formula notation (`rows ~ columns`; a `.` can be used as a placeholder that indicates only one row or column).

> Challenge
>
> Use what you just learned to create a plot that depicts how the average age of each driver for the two recorded ethnicities changes through the week. Hint: make sure you remove the records with driver_age under 16. How would you go about visualizing both lines and points on the plot? How would you split your plot into one per each violation type?

3.7 ggplot2 themes

`ggplot2` comes with several other themes which can be useful to quickly change the look of your visualization, for example `theme_bw()` changes the plot background to white:

```
trafficstops %>%
  mutate(wk_day = wday(stop_date, label=TRUE, abbr=TRUE)) %>%
  group_by(wk_day, violation, driver_race) %>%
  tally %>%
  ggplot(aes(x = wk_day, y = n, color = driver_race, group = driver_race)) +
  geom_line() +
  facet_wrap(~ violation) +
  theme_bw()
```



The complete list of themes is available at http://docs.ggplot2.org/current/ggtheme.html. `theme_minimal()` and `theme_light()` are popular, and `theme_void()` can be useful as a starting point to create a new hand-crafted theme.

The ggthemes package provides a wide variety of options (including an Excel 2003 theme). The **ggplot2** extensions website provides a list of packages that extend the capabilities of **ggplot2**, including additional themes.

3.8 Customization

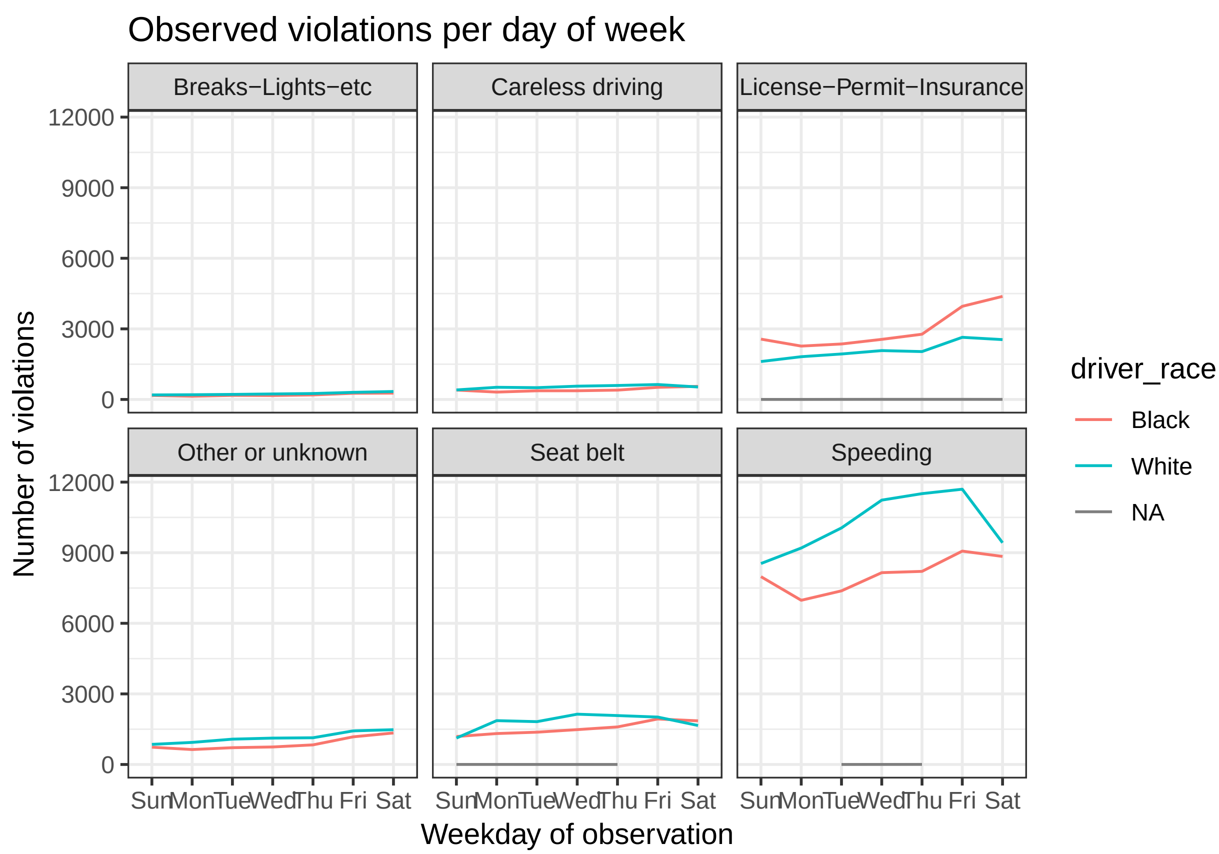
There are endless possibilities to customize your plot, particularly when you are ready for publication or presentation. Let's look into just a few examples. Before we do that we will assign our plot above to a variable.

```
stops_facet_plot <- trafficstops %>%
  mutate(wk_day = wday(stop_date, label=TRUE, abbr=TRUE)) %>%
  group_by(wk_day, violation, driver_race) %>%
  tally %>%
  ggplot(aes(x = wk_day, y = n, color = driver_race, group = driver_race)) +
  geom_line() +
  facet_wrap(~ violation)
```

Now, let's change names of axes to something more informative than 'wk_day' and 'n' and add a title to the figure:

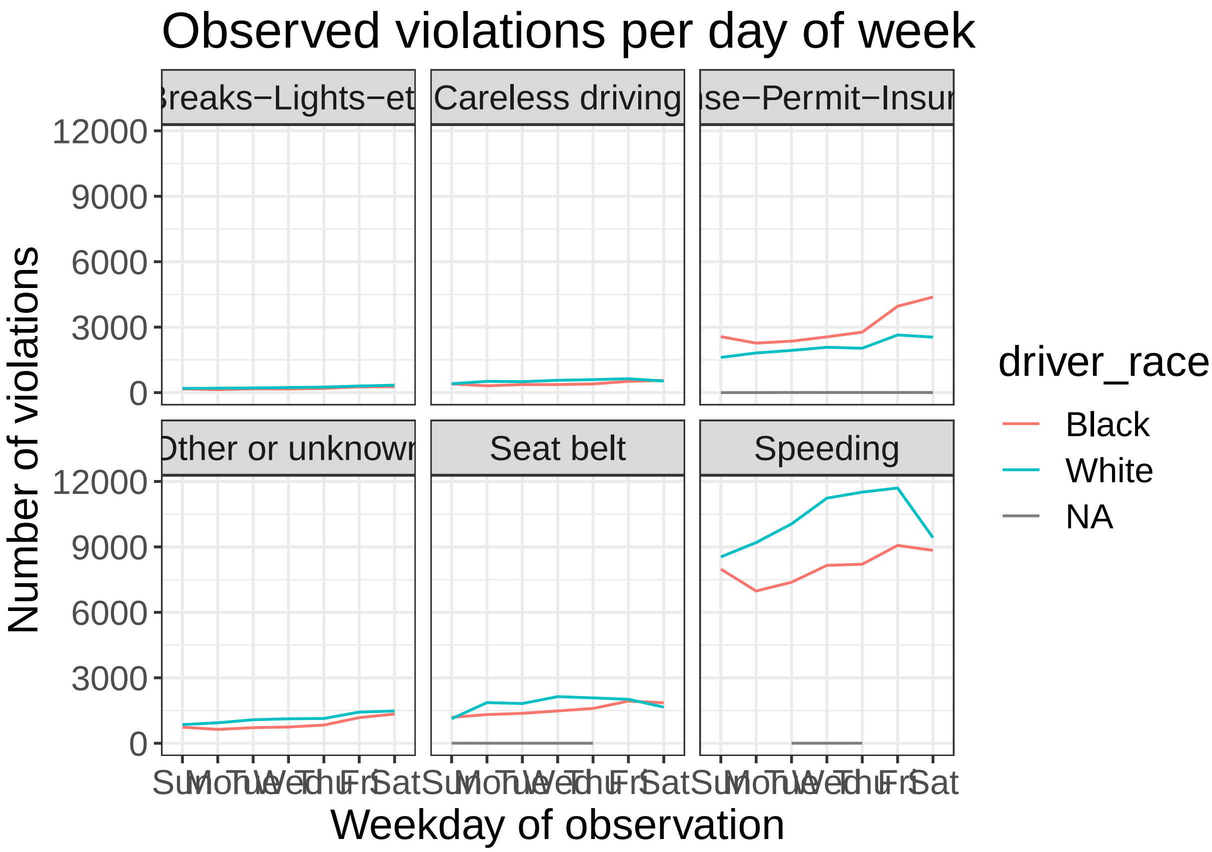
```
stops_facet_plot +
  labs(title = 'Observed violations per day of week',
```

```
         x = 'Weekday of observation',
         y = 'Number of violations') +
  theme_bw()
```



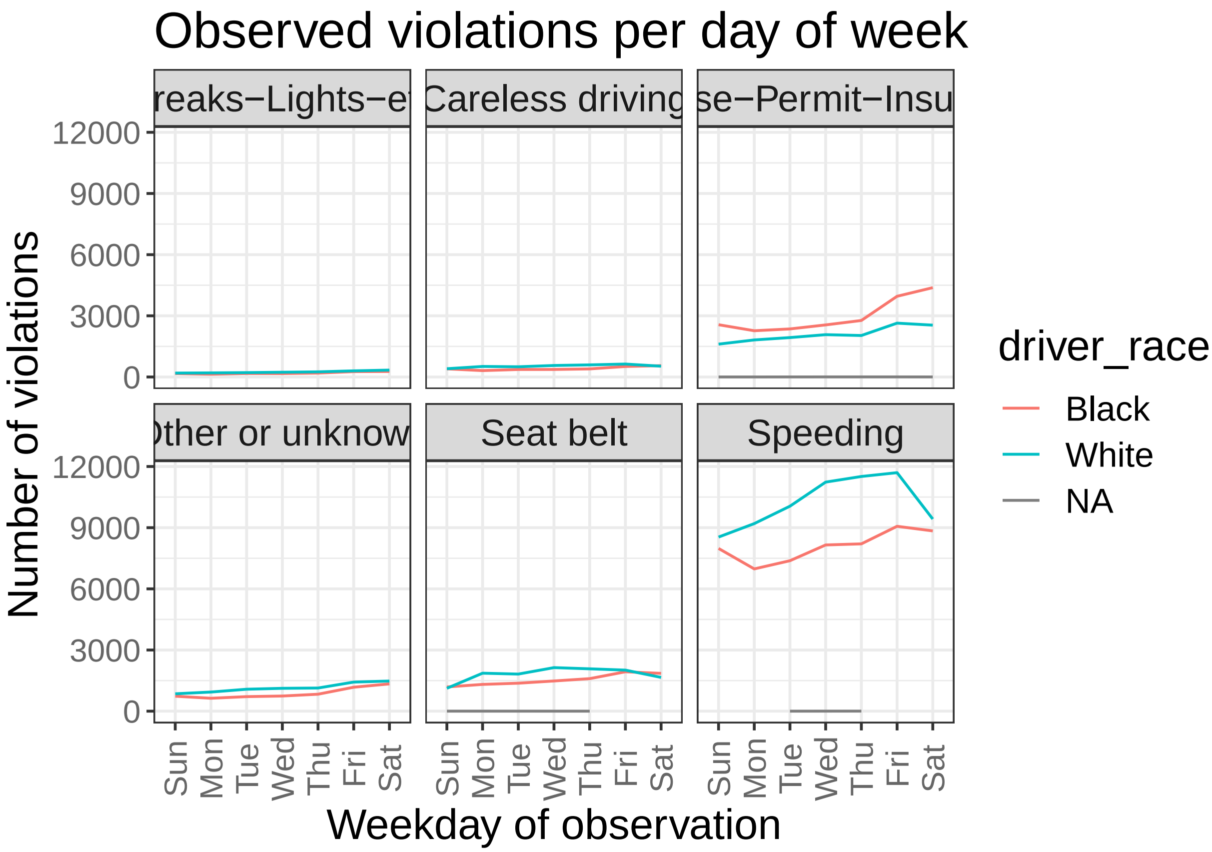
The axes have more informative names, but their readability can be improved by increasing the font size:

```
stops_facet_plot +
  labs(title = 'Observed violations per day of week',
         x = 'Weekday of observation',
         y = 'Number of violations') +
  theme_bw() +
  theme(text = element_text(size=16))
```



After our manipulations, you may notice that the values on the x-axis are still not properly readable. Let's change the orientation of the labels and adjust them vertically and horizontally so they don't overlap. You can use a 90 degree angle, or experiment to find the appropriate angle for diagonally oriented labels:

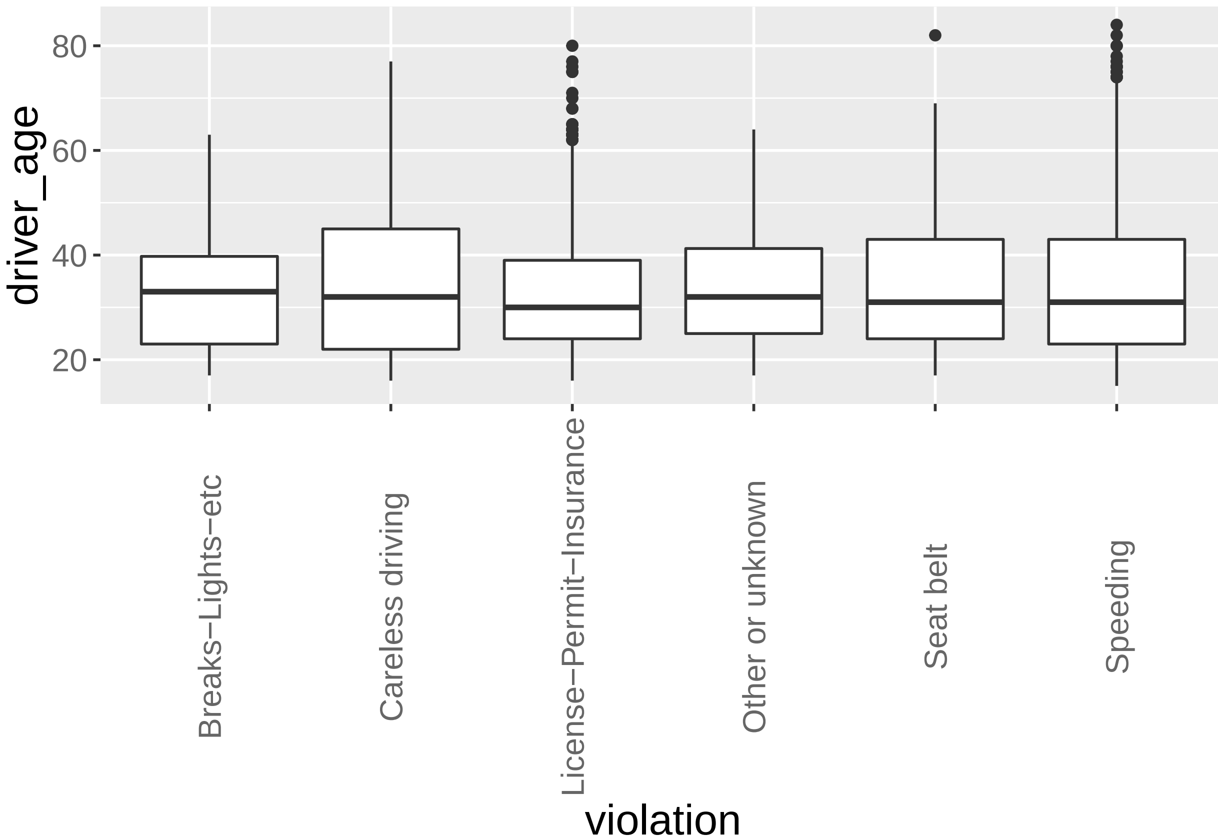
```
stops_facet_plot +
  labs(title = 'Observed violations per day of week',
         x = 'Weekday of observation',
         y = 'Number of violations') +
  theme_bw() +
  theme(axis.text.x = element_text(colour="grey40", size=12, angle=90, hjust=.5, vjust=.5),
        axis.text.y = element_text(colour="grey40", size=12),
        strip.text = element_text(size=14),
        text = element_text(size=16))
```



If you like the changes you created better than the default theme, you can save them as an object to be able to easily apply them to other plots you may create:

```
grey_theme <- theme(axis.text.x = element_text(colour="grey40", size=12, angle=90, hjust=.5, vjust=.5),
                    axis.text.y = element_text(colour="grey40", size=12), text=element_text(size=16))

ggplot(data = Chickasaw_stops, aes(x = violation, y = driver_age)) +
  geom_boxplot() +
  grey_theme
```



Note that it is also possible to change the fonts of your plots. If you are on Windows, you may have to install the **extrafont** package, and follow the instructions included in the README for this package.

> Challenge
>
> With all of this information in hand, please take another five minutes to either improve one of the plots generated in this exercise or create a beautiful graph of your own. Use the RStudio **`ggplot2`** cheat sheet for inspiration.
>
> Here are some ideas:
>
> - See if you can change the thickness of the lines.
> - Can you find a way to change the name of the legend? What about its labels?
> - Try using a different color palette (see http://www.cookbook-r.com/Graphs/Colors_(ggplot2)/).

After creating your plot, you can save it out to a file in your prefered format. You can change the dimension (and resolution) of your plot by adjusting the appropriate arguments (`width`, `height` and `dpi`):

```
my_plot <- stops_facet_plot +
  labs(title = 'Observed violations per day of week',
         x = 'Weekday of observation',
         y = 'Number of violations') +
  theme_bw() +
  theme(axis.text.x = element_text(colour="grey40", size=12, angle=90, hjust=.5, vjust=.5),
        axis.text.y = element_text(colour="grey40", size=12),
        strip.text = element_text(size=14),
        text = element_text(size=16))

ggsave("name_of_file.png", my_plot, width=15, height=10)
```

Note: The parameters `width` and `height` also determine the font size in the saved plot.