Piecemeal R

A Tutorial for Data Exploration with R

Kota Minegishi

Last updated: 2017-04-08

Contents

Welcome

Welcome to a tutorial website for data analysis and visualization with R. This site provides a quick overview and topic-based tutorials in a piecemeal fashion.

The site is organized based on two questions;

- How best to quickly introduce R to new audiences and showcase its data analytics tools?
- How best to provide tutorials for topic-based data applications with R?

To answer the first question, Section 1 *Introduction* demonstrates a set of modern data analysis tools in R. Sections 2 describes essential concepts of R. For the second question, Section 3 provides topic-based tutorials. Additional resources are listed in Section 4.

More contents will be added when the author hosts a small workshop *"Data Exploration with R"* at his workplace.

New Contents

- 2017-04-08: *Test upload. VERY Preliminary!*

About

Kota Minegishi is an assistant professor of Dairy Analytics at the University of Minnesota. He is an agricultural economist by training and works in the Department of Animal Science.

Workshop

- dplyr and ggplot2 exercise 3.1: TBA

Chapter 1

Introduction

2017-04-08: *VERY Preliminary!*

A Few Words from the Author

R has come a long way in its evolution. Its download page looks unchanged from many years ago. Don't be fooled by its archaic first look. This may be something to do with how the R developer community honors its history of turning the open-source project into one of the most popular data analytic tools today. The community is extremely supportive, and there are numerous learning resources. Please don't mistake that archaic look as a sign of snobbishness, and I hope you too will appreciate it some day. Welcome to the community.

In below, we assume that you have R and RStudio Desktop (free IDE) installed. It will be handy to have cheat sheets for base R, RStudio IDE, dplyr, and ggplot2 as well.

If you find this introduction too technical, please start with ModernDive open-source textbook (say, up to Chapter 5). The book gave an initial inspiration to start this site. Also, more information on R is available in Section 2, as well as various sources listed in Section 4.

1.1 Materials

The power of R grows with each addition of user-contributed R packages, or a bundle of user-developed programs. Recent developments such as tidy, dplyr, and ggplot2 have greatly streamlined the coding for data manipulation and analysis, which is the starting point for learning R that is chosen for this site. With this new syntax system, you will learn the basic operations of data wrangling and visualization in a very intuitive *data operation language*. Like any language, its grammar and framework provide a particular way of understanding the world. In this case, it will influence your thinking about data.

Following the documentation of dplyr, let's start with a sample dataset of airplane departures and arrivals. The dataset contains information on about 337,000 flights departed from New York City in 2013 (source: Bureau of Transportation Statistics). We load a built-in data frame by command `library(nycflights13)` where `library(package_name)` loads an R package named `package_name` in the current R *session*, or the computing environment. In the R console (i.e., the left bottom pane in RStudio), type `install.packages("nycflights13")` and hit enter.

Generally, R packages are installed in the local computer as an as-needed basis. To install several more packages that we will use, copy the following code and execute it in your R console.

```
# Don't worry about understanding the code here
# "#"" symbole is used to insert comments that are helpful to humans but are ignored by R
```

```
required_pkgs <- c("nycflights13", "dplyr", "ggplot2", "lubridate", "knitr", "tidyr", "broom")
  #  creating a new object "required_pkgs" containing strings "nycflights13", "dplyr",..
  # "c()" concatenates string names here.
  # "<-" operator assigns from the object on the right to left

new_pkgs <- required_pkgs[!(required_pkgs  %in% installed.packages())]
  # checking whether "required_pkgs" are already installed
  # "[]" of required_pkgs[ ] is extraction by logical TRUE or FALSE
  # "%in%" checks whether items on the left are members of the items on the right.
  # ! is a negation

if (length(new_pkgs)) {
  install.packages(new_pkgs, repos = "http://cran.rstudio.com")
}
```

In each R session, we load libraries. Here we load the following;

```
suppressWarnings({
  suppressMessages({
    library(dplyr)  # for data manipulation
    library(ggplot2)  # for figures
    library(lubridate) # for date manipulation
    library(nycflights13)  # sample data of NYC flights
    library(knitr) # for table formatting
    library(tidyr) # for table formatting
    library(broom)  # for table formatting
  })
})
```

Let's see the data.

```
class(flights) # shows the class attribute
dim(flights)   # obtains dimention of rows and columns
```

```
## [1] "tbl_df"     "tbl"        "data.frame"
## [1] 336776     19
```

```
head(flights)  # displays first seveal rows and columns
```

```
## # A tibble: 6 × 19
##    year month   day dep_time sched_dep_time dep_delay arr_time
##   <int> <int> <int>    <int>          <int>     <dbl>    <int>
## 1  2013     1     1      517            515         2      830
## 2  2013     1     1      533            529         4      850
## 3  2013     1     1      542            540         2      923
## 4  2013     1     1      544            545        -1     1004
## 5  2013     1     1      554            600        -6      812
## 6  2013     1     1      554            558        -4      740
## # ... with 12 more variables: sched_arr_time <int>, arr_delay <dbl>,
## #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>,
## #   time_hour <dttm>
```

`dim()` command returns the dimension of a data frame, and `head()` command returns the first several rows and columns. The `flights` dataset contains information on dates, actual departure and arrival times, scheduled departure and arrival times, carriers, origins, destinations, travel times, and distances. These variables are arranged in columns, and each row is an observation of flight.

In R, we refer to a dataset as **data frame**, which is a *class* of R object. The **data frame** class is more general than the **matrix** class in that it can contain variables of more than one mode (numeric, character, factor etc). In case you want an overview of data types right away, here is a summary.

1.2 Arts & Carfts

Crafts

We will focus on six data wrangling functions in the `dplyr` package.

- `filter()`: extracts rows (e.g., observations) of a data frame. We put logical vectors in its arguments.
- `select()`: extracts columns (e.g., variables) of a data frame. We put column names in its arguments.
- `arrange()`: orders rows of a data frame. We put column names in its arguments.
- `summarise()`: collapses a data frame into summary statistics. We put **summary functions** (e.g., statistics functions) using column names in its arguments.
- `mutate()`: creates new variables and adds them to the existing columns. We put **window functions** (e.g., transforming operations) using column names in its arguments.
- `group_by()`: assigns rows into groups within a data frame. We put column names in its arguments.

The very first argument in all these functions is a **data frame**, and by using this we can easily **pipe** a sequence of data wrangling operations through `%>%` operator. The key is to start with a data frame and then formulate a sequence of data wrangling operations in plain English, which we can translate into codes by replacing **then** in the sequence with the `%>%` operator. Say, we want to find the average delays in departures and arrivals from New York to St. Paul-Minneapolis airport (MSP). We can construct the following sequence of instructions; take the flight data frame, apply `filter()` to extract the rows of flights to MSP, and then apply `summarise()` to calculate the mean.

```
flights %>%  # take data frame "flights", then
  filter(dest == "MSP") %>%  # filter rows, then
  summarise(
    # summarise departure and arrival delays for their means
    # and call them mean_dep_delay and mean_arr_delay respectively
    mean_dep_delay = mean(dep_delay, na.rm = TRUE),
    mean_arr_delay = mean(arr_delay, na.rm = TRUE)
    )    # calculate the mean, while removing NA values
```

```
## # A tibble: 1 × 2
##   mean_dep_delay mean_arr_delay
##            <dbl>          <dbl>
## 1       13.32481       7.270169
```

In `summarise()`, one can use **summary functions** that takes a vector as an input and produces a scaler as an output. This includes functions like `mean()`, `sd()` (standard deviation), `quantile()`, `min()`, `max()`, and `n()` (observation count in the `dplyr` package).

Each time we apply `%>%` operator above, we pass a modified data frame from one data operation to another through the first argument. The above code is equivalent to

```
summarise(   # data frame "flights" is inside filter(), which is inside summarise()
    filter(flights, dest == "MSP"),
    mean_dep_delay = mean(dep_delay, na.rm = TRUE),
    mean_arr_delay = mean(arr_delay, na.rm = TRUE)
    )
```

```
## # A tibble: 1 × 2
##   mean_dep_delay mean_arr_delay
##            <dbl>          <dbl>
## 1       13.32481       7.270169
```

You will quickly discover that %>% operator makes the code much easier to read, write, and edit and how that makes you want to play with the data more.

Let's add a few more lines to the previous example. Say, additionally we want to see the average delays by carrier and sort the results by the number of observations (e.g. flights) in descending order.

Okay, what do we do? We make **a sequence of data wrangling operations in plain English** and translate that into **codes** by replacing **then** with %>% operator. For example, we say, "take the data frame `flights`; **then** (%>%) `filter()` to extract the rows of flights to MSP; **then** (%>%) group rows by carrier; **then** (%>%) `summarise()` data for the number of observations and the means; **then** (%>%) `arrange()` the results by the observation count in descending order."

```
flight_stats_MSP <- flights %>%  # assign the results to an object named "flight_stats"
  filter(dest == "MSP") %>%
  group_by(carrier) %>%  #  group rows by carrier
  summarise(
    n_obs = n(),  # count number of rows
    mean_dep_delay = mean(dep_delay, na.rm = TRUE),
    mean_arr_delay = mean(arr_delay, na.rm = TRUE)
  ) %>%
  arrange(desc(n_obs))  # sort by n_obs in descending order

flight_stats_MSP  # show flight_stats object
```

```
## # A tibble: 6 × 4
##   carrier n_obs mean_dep_delay mean_arr_delay
##     <chr> <int>          <dbl>          <dbl>
## 1      DL  2864      10.651392       4.035702
## 2      EV  1773      17.093413      10.527995
## 3      MQ  1293       8.255457       9.559350
## 4      9E  1249      19.658113       8.089776
## 5      OO     4       0.750000      -2.000000
## 6      UA     2      -6.000000      -5.500000
```

Carrier variable is expressed in the International Air Transportation Association (IATA) code, so let's add a column of carrier names by joining another data frame called `airlines`. In RStudio, you can find this data frame under the **Environment** tab (in the upper right corner); switch the display option from *Global Environment* to *package:nycflights13*. To inspect the data frame, type `View(airlines)` in R console. Also, by typing `data()` you can see a list of all datasets that are loaded with libraries.

```
left_join(flight_stats_MSP, airlines, by="carrier") %>%
  # left_join(a,b, by="var") joins two data frames a, b by matching rows of b to a
  # by identifier variable "var".
  kable(digits=2)  # kable() prints a better-looking table here
```

carrier	n_obs	mean_dep_delay	mean_arr_delay	name
DL	2864	10.65	4.04	Delta Air Lines Inc.
EV	1773	17.09	10.53	ExpressJet Airlines Inc.
MQ	1293	8.26	9.56	Envoy Air
9E	1249	19.66	8.09	Endeavor Air Inc.
OO	4	0.75	-2.00	SkyWest Airlines Inc.
UA	2	-6.00	-5.50	United Air Lines Inc.

In the next example, we add new variables to `flights` using `mutate()`.

```
flights %>%
  # keep only columns named "dep_delay" and "arr_delay"
  select(dep_delay, arr_delay) %>%
  mutate(
    gain = arr_delay - dep_delay,
    gain_rank = round(percent_rank(gain), digits = 2)
      # Note: we can immediately use the "gain" variable we just defined.
  )
```

```
## # A tibble: 336,776 × 4
##    dep_delay arr_delay  gain gain_rank
##        <dbl>     <dbl> <dbl>     <dbl>
## 1          2        11     9      0.81
## 2          4        20    16      0.88
## 3          2        33    31      0.94
## 4         -1       -18   -17      0.22
## 5         -6       -25   -19      0.18
## 6         -4        12    16      0.88
## 7         -5        19    24      0.92
## 8         -3       -14   -11      0.37
## 9         -3        -8    -5      0.54
## 10        -2         8    10      0.82
## # ... with 336,766 more rows
```

We extracted specific columns of `flights` by `select()` and added new columns defined in `mutate()`. `mutate()` differs from `summarise()` in that `mutate()` adds new columns to the data frame, while `summarise()` collapses the data frame into a summary table.

There are roughly five types of window functions that are commonly used inside `mutate()`: (1) **summary functions**, which are interpreted as a vector of repeated values (e.g., a column of an identical mean value) : (2) ranking or ordering functions (e.g., `row_number()`, `min_rank()`, `dense_rank()`, `cume_dist()`,`percent_rank()`, and `ntile()`): (3) offset functions, say defining a lagged variable in time series data (`lead()` and `lag()`): (4) cumulative aggregates (e.g., `cumsum()`, `cummin()`, `cummax()`, `cumall()`, `cumany()`, and `cummean()`): (5) fixed-window rolling aggregates such as a windowed mean, median, etc. To find help files for these function, for example, type `?cumsum`.

Before moving to the graphics, let's quickly go over what a **function** is in R and how you can use a custom function inside `summarise()` or `mutate()`. In R, we use `function()` to create a function, which has its name, input arguments separated by comma, and a body (e.g., tasks to perform and what to return as an output).

```
your_function_name <- function(input arguments) {
                        task1
                        task2
                        .
                        .
                        .
                        output_to_return
                      }
```

For a function having only a single expression to execute, we can omit brackets `{ }`.

```
another_function <- function(input args) task_and_output_in_a_single_expression
```

Let's go through a few examples.

```
# generate a sequence from 1 to 10 (by the increment of 1) and name it "vec1".
vec1 <- 1:10
vec1
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
# c() concatenates
vec2 <- c(vec1, NA, NA)
vec2
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 NA NA
```

```
my_mean_1 <- function(x)  mean(x, na.rm = TRUE)
  # Input arguments: x
  # Output: the calculation result of mean(x, na.rm = TRUE).
  # x is required by mean() (and implicitly assumed to be a vector of numeric values).
  # mean() is an existing function. The "na.rm" argument of mean() is set to be TRUE.

my_mean_1(vec1)
```

```
## [1] 5.5
```

```
my_mean_2 <- function(x, na.rm=TRUE)  mean(x, na.rm = na.rm)
  # Input arguments: x and na.rm (optional with the default value of TRUE)
  # Output: the calculation result of mean(x, na.rm = na.rm).
  # The input argument "na.rm" is passed to the input argument "na.rm" of mean()

my_mean_2(vec2)
```

```
## [1] 5.5
```

```
my_mean_2(vec2, na.rm=FALSE)  # not removing NA returns NA for the mean calculation.
```

```
## [1] NA
```

```
my_zscore <- function(x, remove_na=TRUE) {
  (x - my_mean_2(x, na.rm = remove_na))/sd(x, na.rm = remove_na)
}
  # Inputs: x and remove_na (optional: default = TRUE)
  # Output: z-score of vector x
  # my_mean2() and sd() return scalers but are interpreted
  # as a vector of repeated valuses that has the same length as x.

my_zscore(vec1) %>% round(2)
```

```
##  [1] -1.49 -1.16 -0.83 -0.50 -0.17  0.17  0.50  0.83  1.16  1.49
```

Let's apply functions`my_mean_2()` and `my_zscore()` in `summarise()` and `mutate()`.

```
flights %>%
  select(dep_delay) %>%
  summarise(
    mean_dep_delay = my_mean_2(dep_delay),  # using my_mean_2()
    mean_dep_delay_na = my_mean_2(dep_delay, na.rm = FALSE)  # this returns NA
  ) %>%
  kable(digits=2)
```

mean_dep_delay	mean_dep_delay_na
12.64	NA

```
flights_gain <- flights %>%
  select(dep_delay, arr_delay) %>%
  mutate(
    gain = arr_delay - dep_delay,
    gain_z = (gain - my_mean_2(gain))/sd(gain, na.rm=TRUE),  # using my_mean_2()
    gain_z2 = my_zscore(gain_z)  # using my_zscore()
  )

head(flights_gain) %>%  # show the first several rows
  kable(digits=2)
```

dep_delay	arr_delay	gain	gain_z	gain_z2
2	11	9	0.81	0.81
4	20	16	1.20	1.20
2	33	31	2.03	2.03
-1	-18	-17	-0.63	-0.63
-6	-25	-19	-0.74	-0.74
-4	12	16	1.20	1.20

Creating a function spares us from writing similar codes in multiple places. Avoiding such repetitions is important for making reading and editing codes easier and reducing coding errors.

One situation you may consider use of custom function is inside functions like `summarise_each()` and `mutate_each()`. The two functions allow for applying **summary functions** like `mean()` or `sd()` to each column in a data frame. `summarise_each()` and `mutate_each()` work by *calling* a function by its name. They are very easy to use when an operation is to summarize a vector into a statistics without needing to specify additional arguments, say `mean(var1)`. However, providing additional arguments into a function, say `mean(var1, na.rm=TRUE)`, becomes somewhat cumbersome in terms of its syntax.

One approach to get around this problem is to pre-process the data frame before getting to a `summarise_each()` or `mutate_each()` section. For example, if we want to pass the argument `na.rm=TRUE` to `mean()`, we can first filter out rows that contain missing values (`NA`) and then apply `summarise_each()`.

```
flights_gain %>%
  select(dep_delay, arr_delay, gain)  %>%
  filter(!is.na(dep_delay) & !is.na(arr_delay)) %>%
    # filter out rows that have NA values in dep_delay or arr_deplay
  summarise_each("mean") %>%
  kable(digits=2)
```

dep_delay	arr_delay	gain
12.56	6.9	-5.66

The other approach is to use a custom function. For instance, `my_mean_2()` we defined above has default argument `na.rm=TRUE` that gets passed into `mean()`, effectively overwriting the default argument `na.rm=FALSE` of `mean()`. A custom function (as well as any standard summary function) can be called in `summarise_each()` or `mutate_each()` using `funs()`;

```
flights_gain %>%
  select(dep_delay, arr_delay, gain) %>%
  summarise_each(funs("my_mean_2")) %>%
    kable(digits=2)
```

dep_delay	arr_delay	gain
12.64	6.9	-5.66

Being able to use your own functions in `dplyr`-style data wrangling operations will greatly enhance your ability to quickly analyze data in R.

Arts

Now we will cover the basics of data visualization via the ggplot2 package. The `ggplot2` syntax has three essential components for generating graphics: **data**, **aes**, and **geom**. This implements the following philosophy (a quote mentioned in ModernDive);

> A statistical graphic is a mapping of **data** variables to **aesthetic** attributes of **geometric** objects.
> — (Wilkinson, 2005)

While coding complex graphics via `ggplot()` may appear intimidating at first, it boils down to the three primary components:

- **data**: a data frame e.g., the first argument in `ggplot(data, ...)`.
- **aes**: specifications for x-y variables, as well as variables to differentiate **geom** objects by color , shape, or size. e.g., `aes(x = var_x, y = var_y, shape = var_z)`
- **geom**: geometric objects such as points, lines, bars, etc. e.g., `geom_point()`, `geom_line()`, `geom_histogram()`

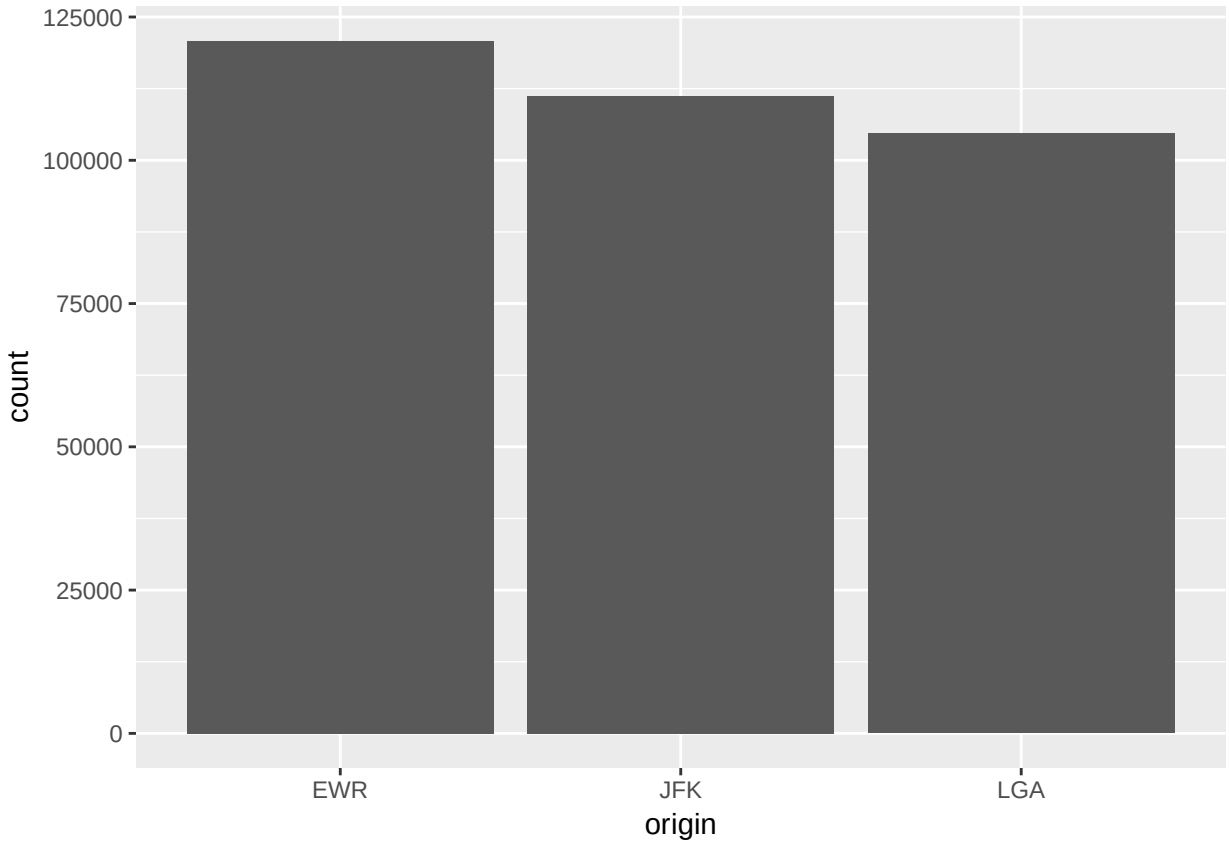
One can refine a plot figure by adding secondary components or characteristics such as:

- stat: data transformation, overlay of statistical inferences etc.
- scales: scaling data points etc.
- coord: Cartesian coordinates, polar coordinates, mapping projections etc.
- facet: laying out multiple plot panels in a grid etc.

In below, we will generate five common types of plots: **scatter-plots**, **line-graphs**, **boxplots**, **histograms**, and **barplots**. To provide a context, let's use these plots to investigate what may explain patterns of flight departure delays.

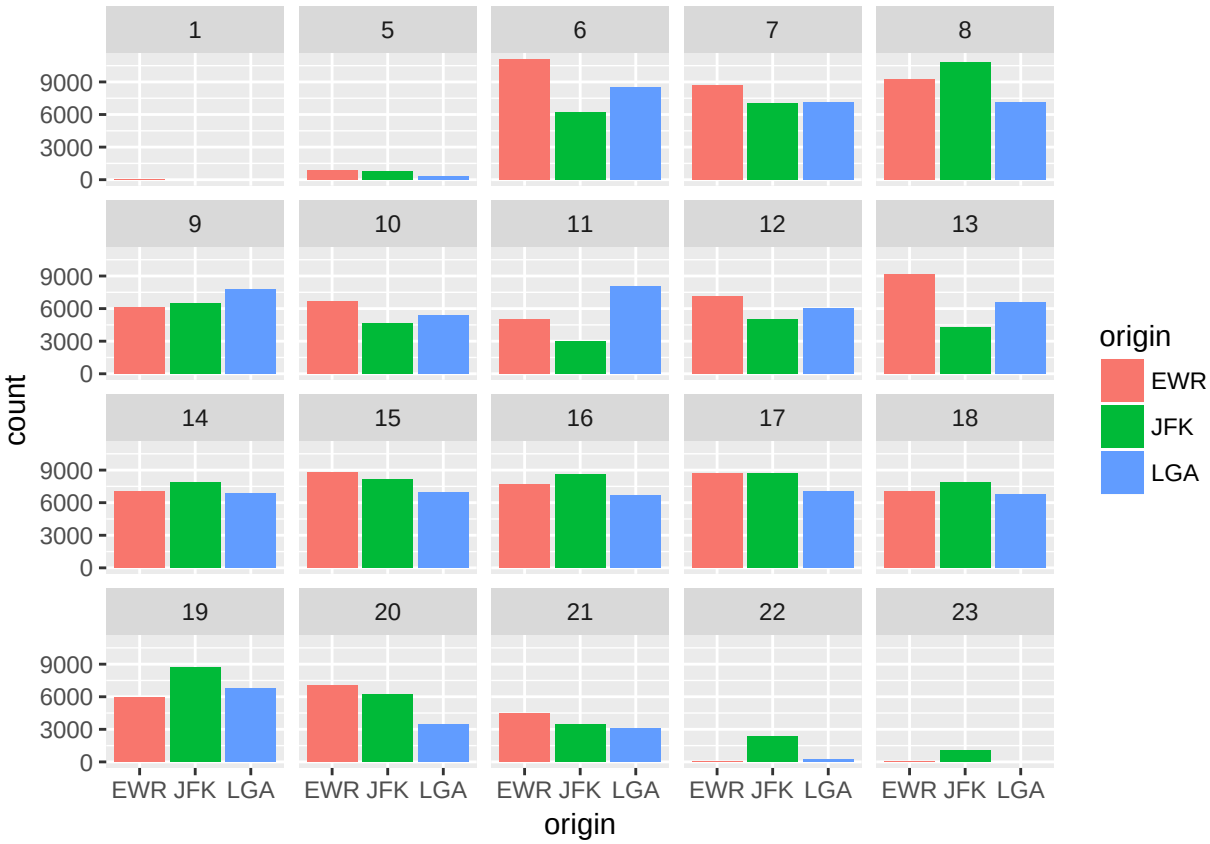
First, let's consider a possibility of congestion at the airport during certain times of the day or certain seasons. We can use **barplots** to see whether there is any obvious pattern in the flight distribution across flight origins (i.e., airports) in New York City. A barplot shows observation counts (e.g., rows) by category.

```
ggplot(data = flights,  # the first argument is the data frame
       mapping = aes(x = origin)) +   # the second argument is "mapping", which is aes()
  geom_bar()  #  after "+" piping operator of ggplot(), we add geom_XXX() elements
```



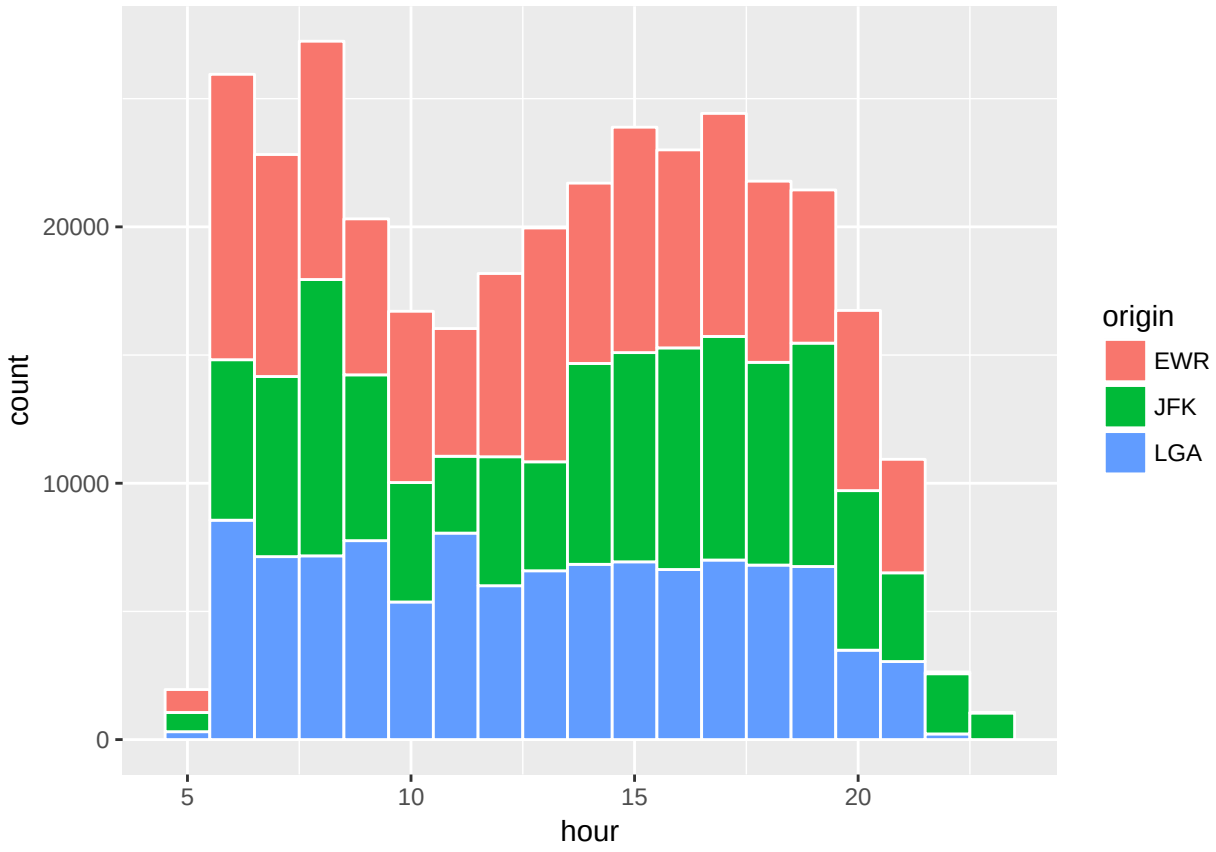
We can make the plot more informative and aesthetic.

```
ggplot(data = flights,
       mapping = aes(x = origin, fill = origin)) +  # here "fill" gives bars distinct colors
  geom_bar() +
  facet_wrap( ~ hour)  #  "facet_wrap( ~ var)" generates a grid of plots by var
```



Another way to see the same information is a **histogram**.

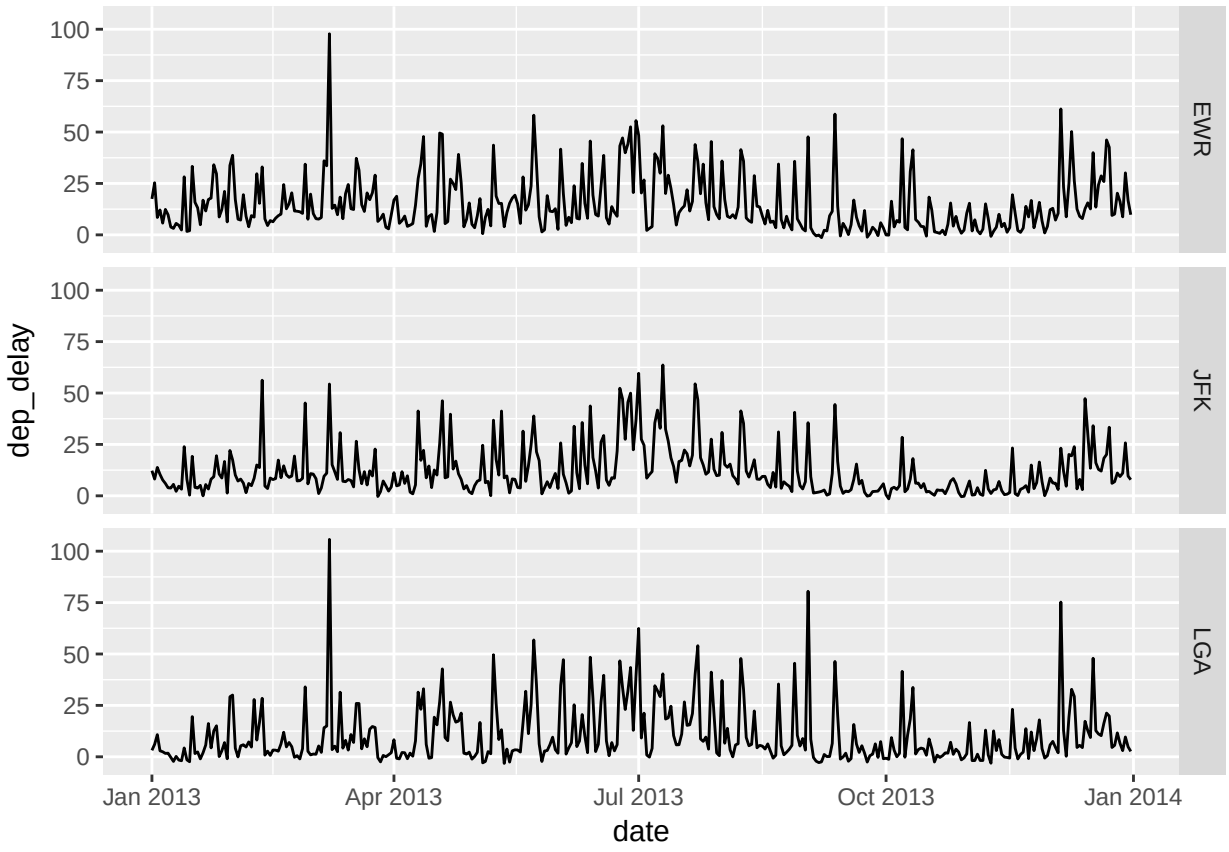
```
flights %>%
  filter(hour >= 5) %>%  # exclude hour earlier than 5 a.m.
  ggplot(aes(x = hour, fill = origin)) + geom_histogram(binwidth = 1, color = "white")
```



While mornings and late afternoons tend to get busy, there is not much difference in the number of flights across airports.

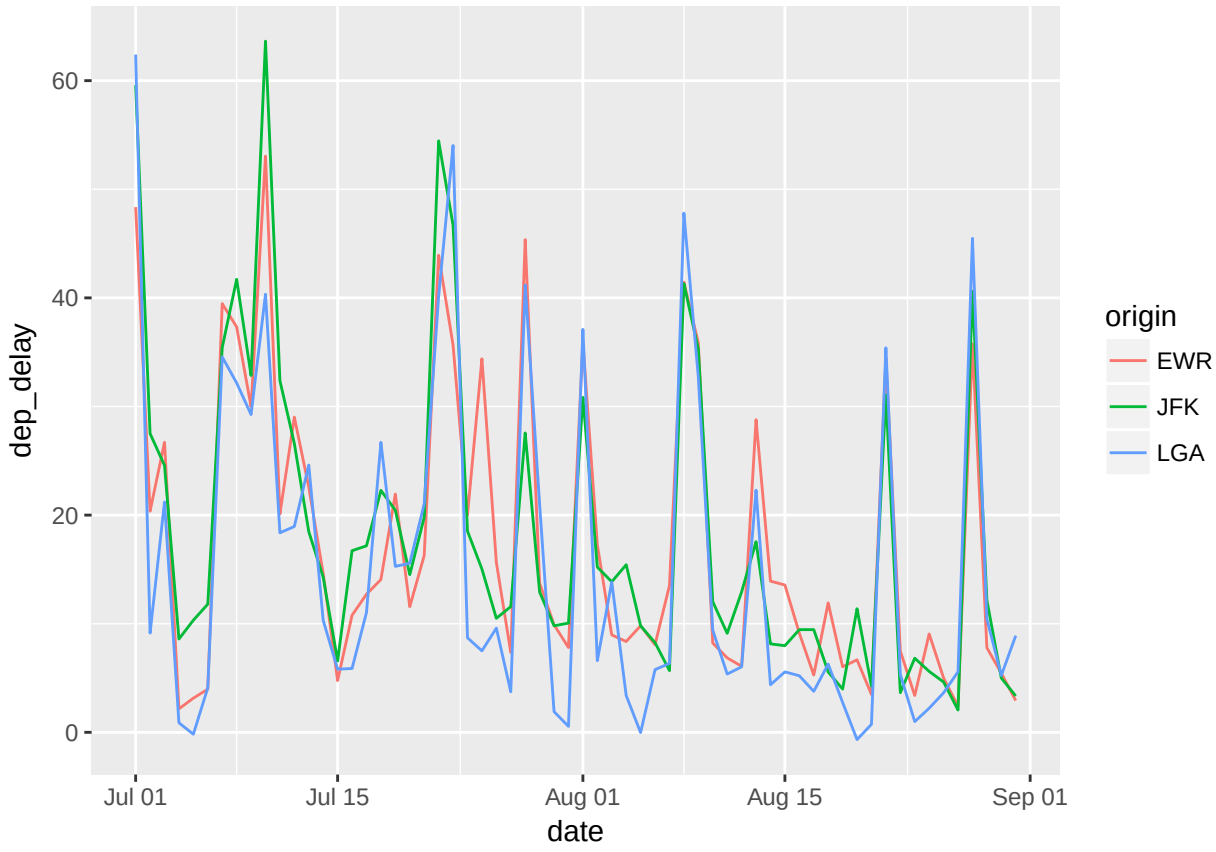
Let's see if there are distinct patters of departure delays over the course of a year. We do this by taking the average of departure delays for each day by flight origin and plot the data as a time series using **line-graphs**.

```
delay_day <- flights %>%
  group_by(origin, year, month, day) %>%
  summarise(dep_delay = mean(dep_delay, na.rm = TRUE))  %>%
  mutate(date = as.Date(paste(year, month, day), format="%Y %m %d")) %>%
 filter(!is.na(dep_delay))   #  exclude rows with dep_delay == NA

delay_day %>%      # "facet_grid( var ~ .)" is similar to "facet_wrap( ~ var)"
  ggplot(aes(x = date, y = dep_delay)) + geom_line() + facet_grid( origin ~ . )
```



The seasonal pattern seems similar across airports, and summer months appear to be busier on average. Let's see how closely these patterns across airports are related to each other by focusing on a few summer months and overlying the line-graphs.

```
delay_day %>%
  filter("2013-07-01" <= date, "2013-08-31" >= date)  %>%
  ggplot(aes(x = date, y = dep_delay, color = origin)) + geom_line()
```



We can see similar patterns of spikes across airports occurring on certain days, indicating a tendency that the three airports get busy on the same days. Would this mean that the three airports tend to be congested at the same time?

In the previous figure, there seems to be some cyclical pattern of delays. A good place to start would be comparing delays by day of the week. Here is a function to calculate day of the week for a given date.

```
my_dow <- function(date) {
  # as.POSIXlt(date)[['wday']] returns integers 0, 1, 2, .. 6, for Sun, Mon, ... Sat.
  # We extract one item from a vector (Sun, Mon, ..., Sat) by position numbered from 1 to 7.
  dow <- as.POSIXlt(date)[['wday']] + 1
  c("Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat")[dow]  # extract "dow"-th element
}
  # Input: date in the format as in "2017-01-23"
  # Output: day of week
Sys.Date()  # Sys.Date() returns the current date
```

```
## [1] "2017-04-08"
```

```
my_dow(Sys.Date())
```

```
## [1] "Sat"
```

Now, let's take a look at the mean delay by day of the week using **boxplots**.

```
delay_day <- flights %>%
  group_by(year, month, day) %>%
  summarise(dep_delay = mean(dep_delay, na.rm = TRUE))  %>%
  mutate(date = as.Date(paste(year, month, day), format="%Y %m %d"),
         # date defined by as.Data() function
```

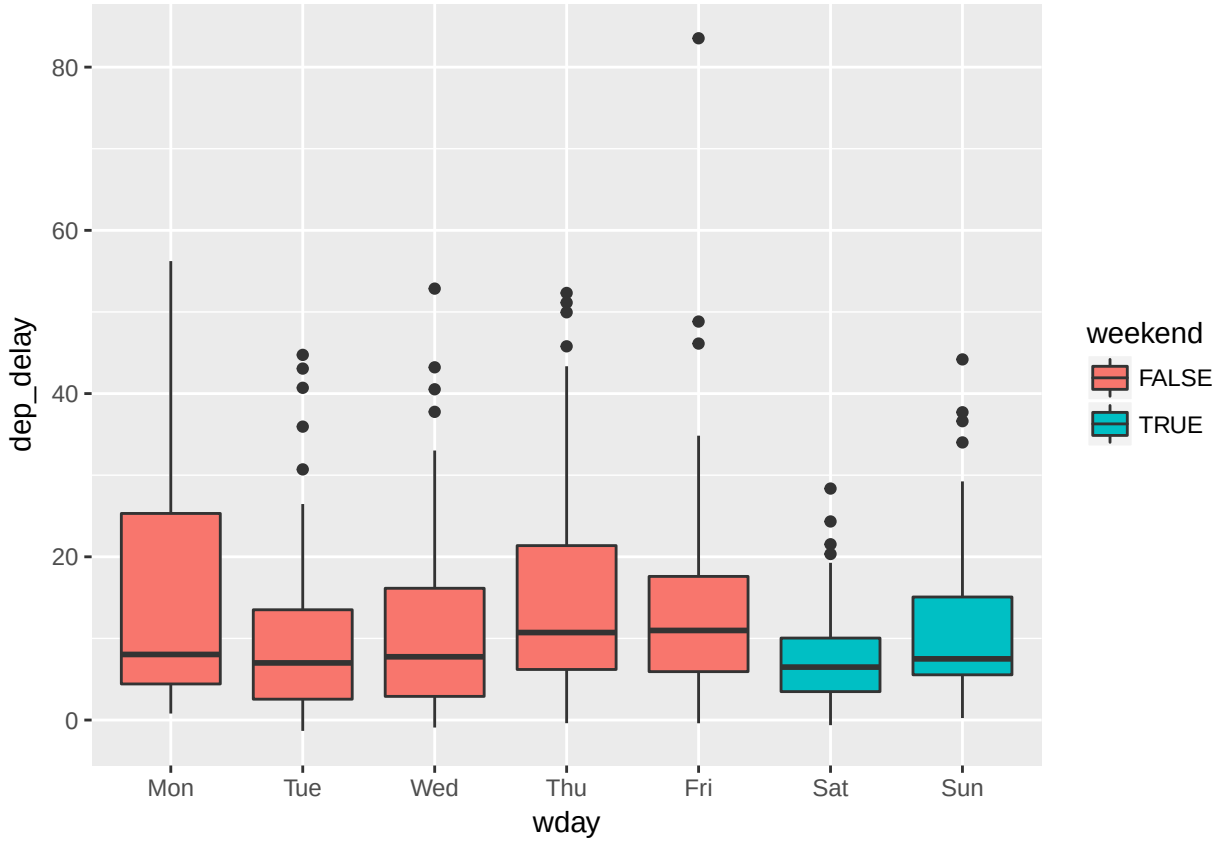
```
          wday = my_dow(date),
          weekend = wday %in% c("Sat", "Sun")
          # %in% operator: A %in% B returns TRUE/FALSE for whether each element of A is in B.
  )

# show the first 10 elements of "wday" variable in "delay_day" data frame
delay_day$wday[1:10]
```

```
##  [1] "Tue" "Wed" "Thu" "Fri" "Sat" "Sun" "Mon" "Tue" "Wed" "Thu"
```

```
delay_day$wday <- ordered(delay_day$wday,
                          levels = c("Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"))
             # adding a sorting order (Mon, Tue, ..., Sun)
delay_day$wday[1:10]
```

```
##  [1] Tue Wed Thu Fri Sat Sun Mon Tue Wed Thu
## Levels: Mon < Tue < Wed < Thu < Fri < Sat < Sun
```

```
delay_day  %>%
  filter(!is.na(dep_delay)) %>%
  ggplot(aes(x = wday, y = dep_delay, fill = weekend)) + geom_boxplot()
```



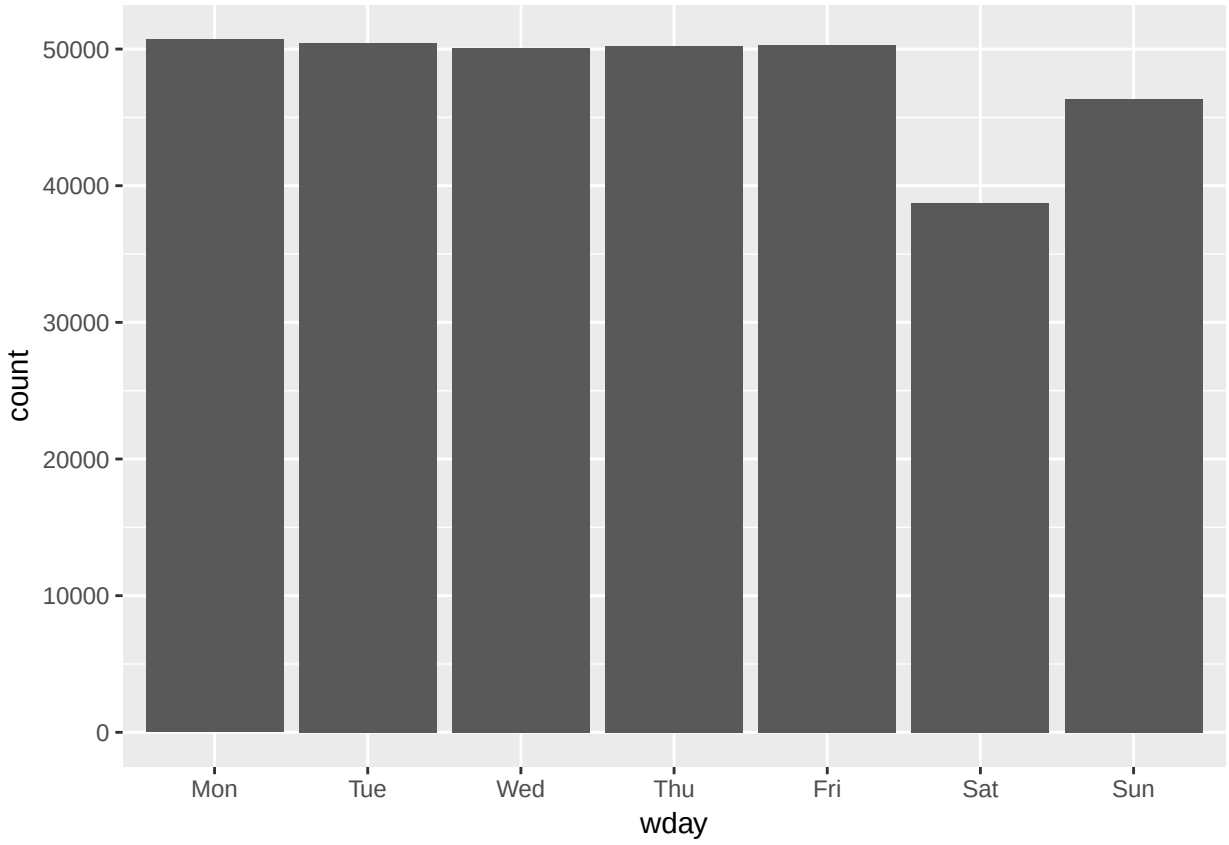
It appears that delays are on average longer on Thursdays and Fridays and shorter on Saturdays. This is plausible if more people are traveling on Thursdays and Fridays before the weekend, and less are traveling on Saturdays to enjoy the weekend. Are Saturdays really less busy? Let's find out.

```
flights_wday <- flights %>%
  mutate(date = as.Date(paste(year, month, day), format="%Y %m %d"),
         wday = ordered(my_dow(date),
```

```
                        levels = c("Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun")),
         weekend = wday %in% c("Sat", "Sun")
  )

flights_wday %>%
  group_by(wday) %>%
  summarise( nobs = n() )
```

```
## # A tibble: 7 × 2
##    wday  nobs
##   <ord> <int>
## 1   Mon 50690
## 2   Tue 50422
## 3   Wed 50060
## 4   Thu 50219
## 5   Fri 50308
## 6   Sat 38720
## 7   Sun 46357
```

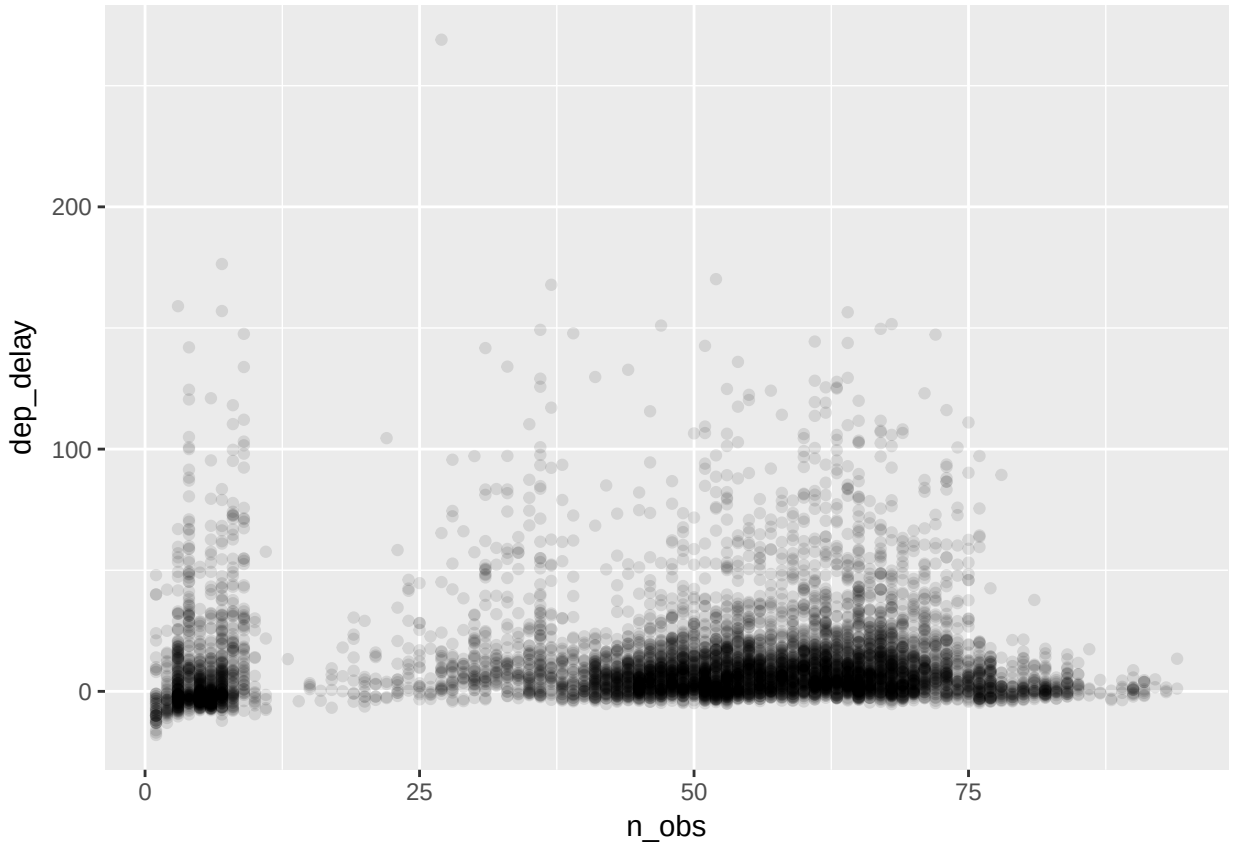
```
flights_wday  %>%
  ggplot(aes(x = wday)) + geom_bar()
```



Yes, Saturdays are less busy for the airports in terms of flight numbers.

Could we generalize this positive relationship between the number of flights and the average delays, which we find across days of the week? To investigate this, we can summarize the data into the average delays by date-hour and see if the busyness of a particular hour of a particular day is correlated with the mean delay. We visualize these data using a **scatter plot**.

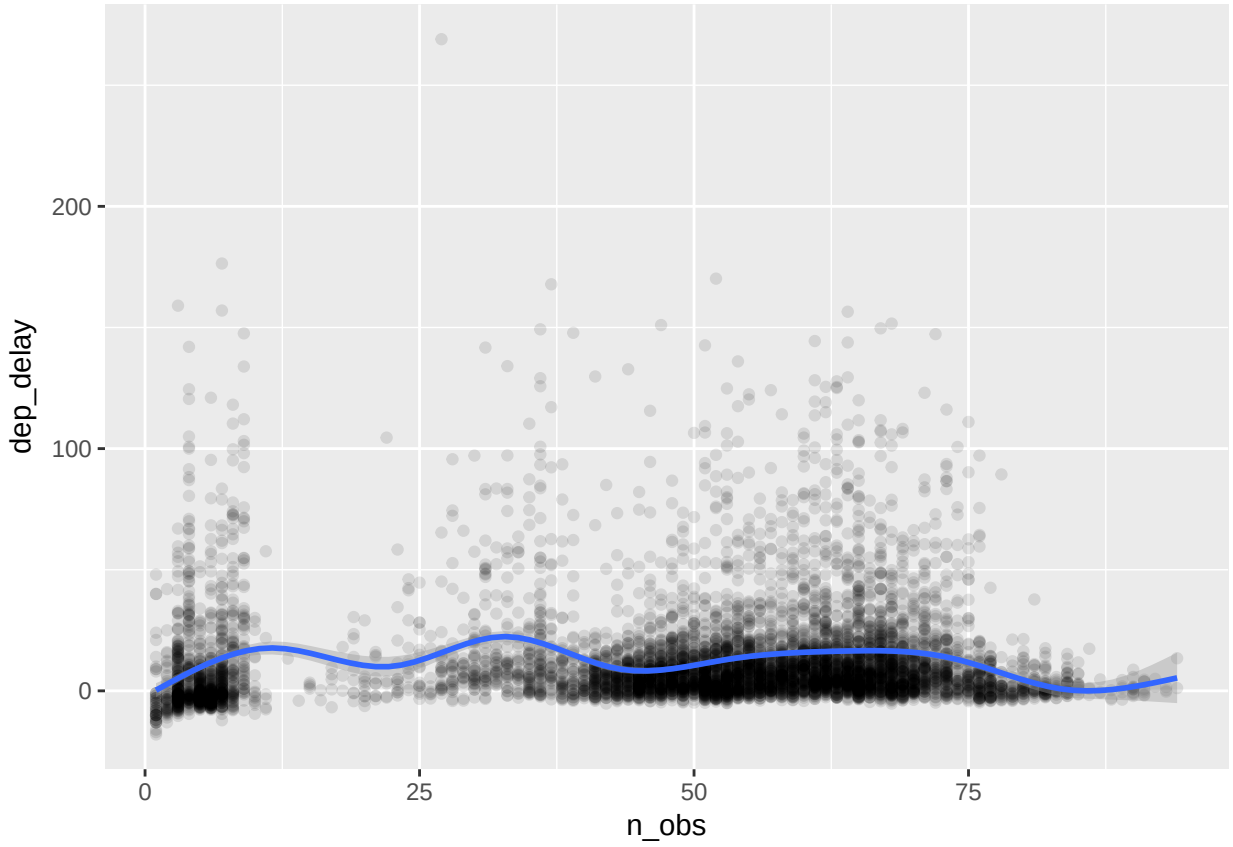
```
delay_day_hr <- flights %>%
  group_by(year, month, day, hour) %>%  # grouping by date-hour
  summarise(
    n_obs = n(),
    dep_delay = mean(dep_delay, na.rm = TRUE)
    )  %>%
  mutate(date = as.Date(paste(year, month, day), format="%Y %m %d"),
         wday = my_dow(date)
  )

plot_delay <- delay_day_hr  %>%
  filter(!is.na(dep_delay)) %>%
  ggplot(aes(x = n_obs, y = dep_delay)) + geom_point(alpha = 0.1)
    # plot of n_obs and the average dep_delay
    # where each point represents an date-hour average
    # "alpha = 0.1"  controls the degree of transparency of points

plot_delay
```



Along the horizontal axis, we can see how the number of flights is distributed across date-hours. Some days are busy, and some hours busier still. It appears that there are two clusters in the number of lights, showing very slow date-hours (e.g., less than 10 flights flying out of New York city per hour) and normal date-hours (e.g., about 50 to 70 flights per hour). We could guess that the delays in the slow hours be caused by bad weather. On the other hand, we may wonder if the excess delays in the normal hours, compared to the slow hours, are caused by congestion at the airports. To see this, let's fit a curve that captures the relationships between `n_obs` and `dep_delay`. Our hypothesis is that the delay would become more likely and longer as the number of flights increases.

```
plot_delay  +
  geom_smooth()    #  geom_smooth() addes a layer of fitted curve(s)
```

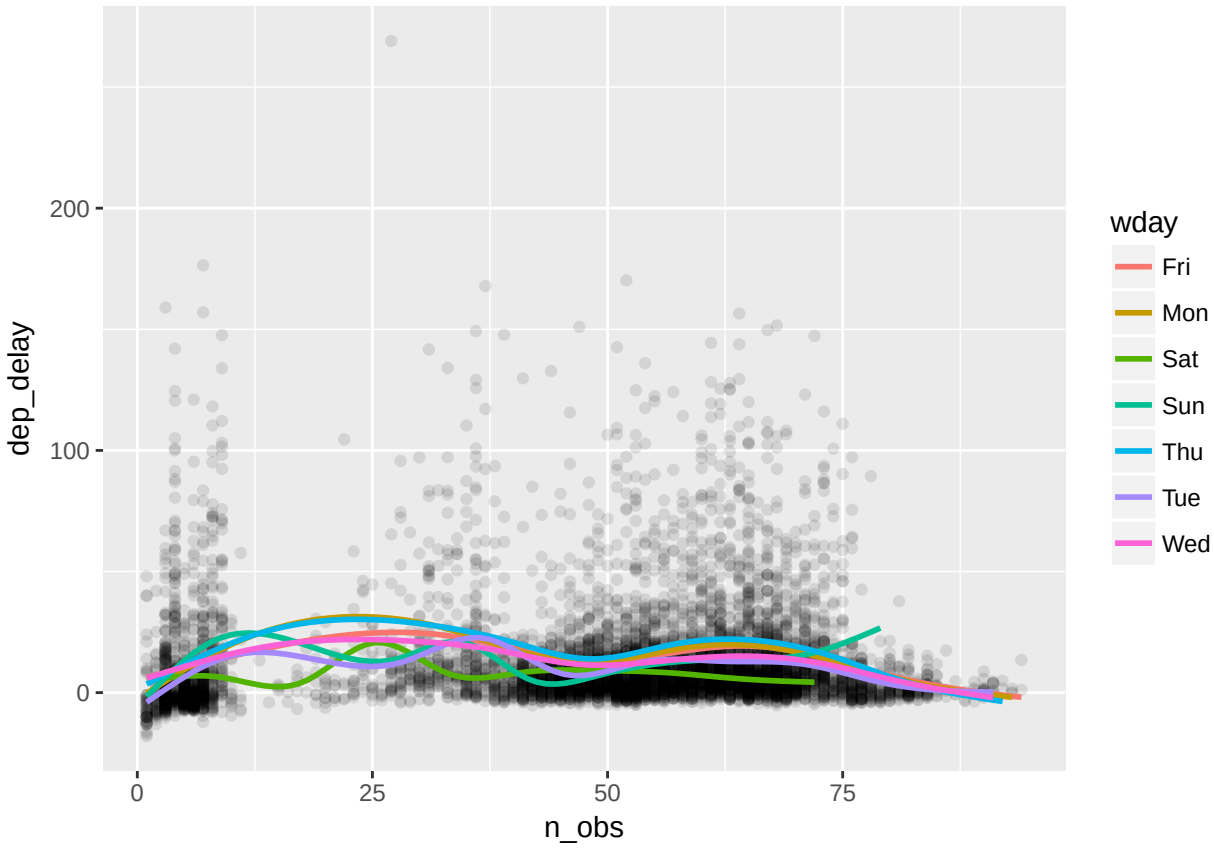
```
## `geom_smooth()` using method = 'gam'
```



We cannot see any clear pattern. How about fitting a curve by day of the week?

```
plot_delay  +
     # additional aes() argument for applying different colors to the day of the week
  geom_smooth(aes(color = wday), se=FALSE)
```

```
## `geom_smooth()` using method = 'gam'
```

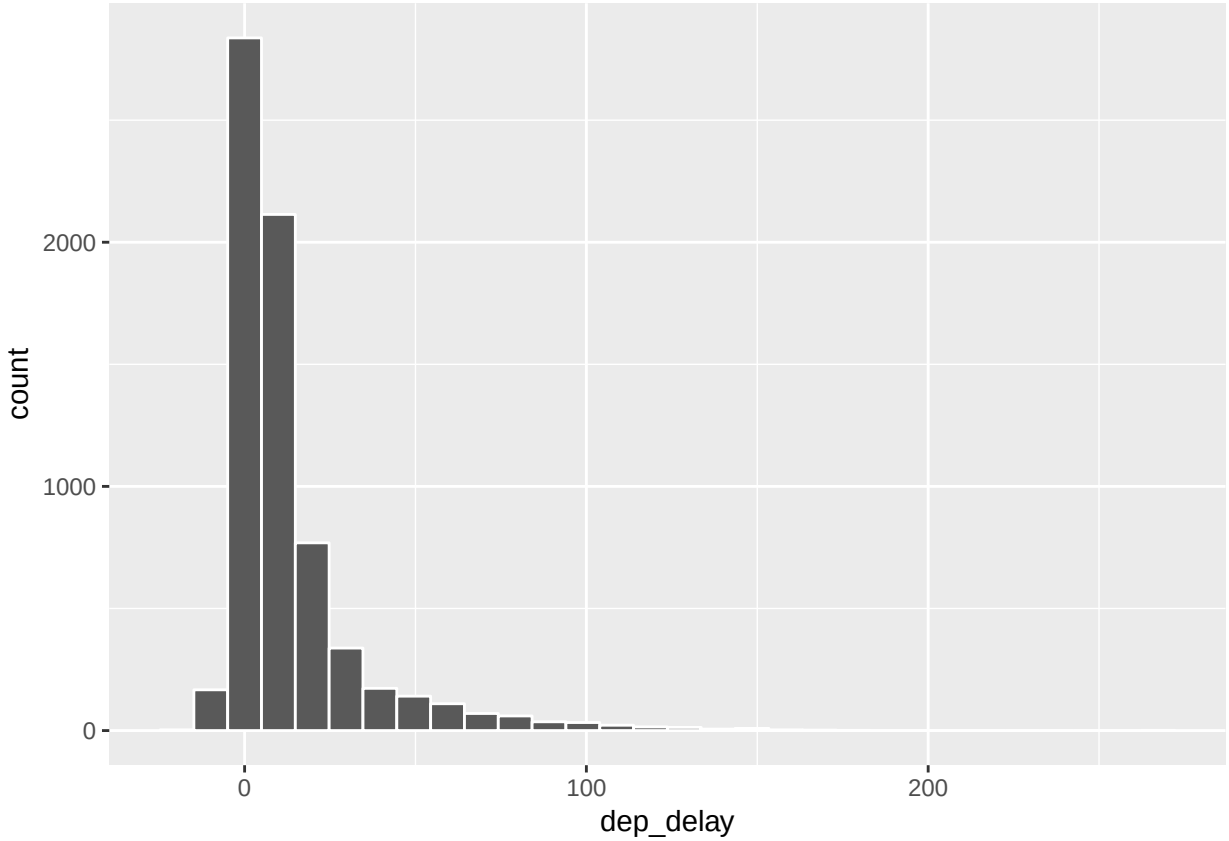


Surprisingly, the delay does not seem to increase with the flights. There are more delays on Thursdays and Fridays and less delays on Saturdays, but we see no evidence of congestion as a cause of delay.

Let's take a closer look at the distribution of the delays. If it is not normally distributed, we may want to apply a transformation.

```
delay_day_hr %>%  filter(!is.na(dep_delay)) %>%
  ggplot(aes(x = dep_delay)) + geom_histogram(color = "white")
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



The distribution of the average delays are greatly skewed.
In applying a logarithmic transformation, here we have to shift the variable so that its minimum is greater than zero.

```
# define new column called "dep_delay_shifted"
delay_day_hr$dep_delay_shifted <- delay_day_hr %>%
  with(dep_delay - min(dep_delay, na.rm = TRUE) + 1)
    # with() function takes a data frame in the first argument and allows for
    # referencing its variable names.

delay_day_hr %>%
  ungroup() %>%   # removing group_by() attribute
  select(dep_delay, dep_delay_shifted) %>%
  with(
    apply(., 2, summary)
    # apply(data, num, fun)  applies function "fun" for each item
    # in dimension "num" (1 = cows, 2= columns) of the data frame
    # Data referenced by "." means all variables of the dataset inside with().
    ) %>% t() #  transpose rows and columns
```

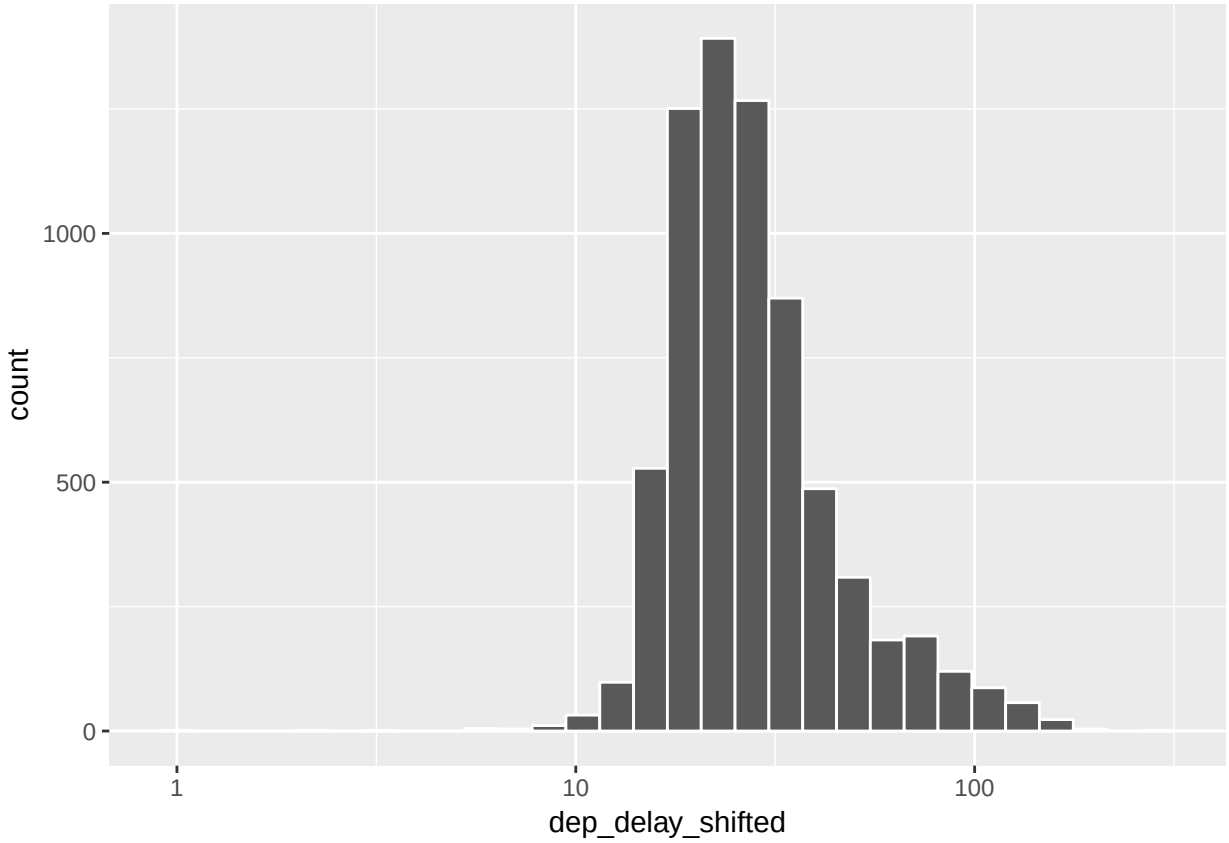
```
##                   Min. 1st Qu. Median  Mean 3rd Qu. Max. NA's
## dep_delay          -18   1.054  6.571 12.99   15.44  269   13
## dep_delay_shifted    1  20.050 25.570 31.99   34.44  288   13
```

Now the transformed distribution;

```
# Under the log of 10 transformation, the distribution looks closer to a normal distribution.
delay_day_hr %>% filter(!is.na(dep_delay_shifted)) %>%
  ggplot(aes(x = dep_delay_shifted))  +
```

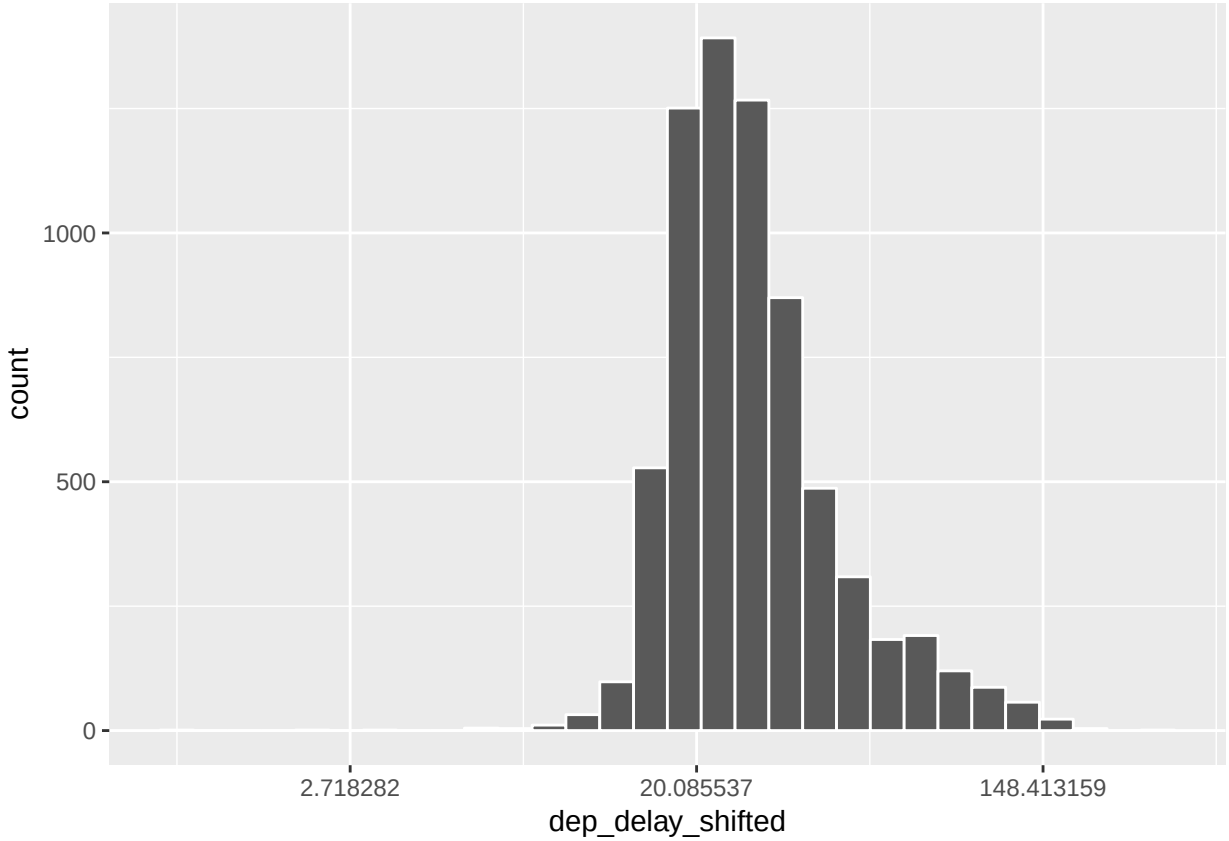
```
  scale_x_log10() +
  geom_histogram(color = "white")
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```




```
# Alternatively, one can apply the natural logarithm to transform a variable. Histogram shows no differ
delay_day_hr %>% filter(!is.na(dep_delay_shifted)) %>%
  ggplot(aes(x = dep_delay_shifted)) +
  scale_x_continuous(trans = "log") +
  geom_histogram(color = "white")
```

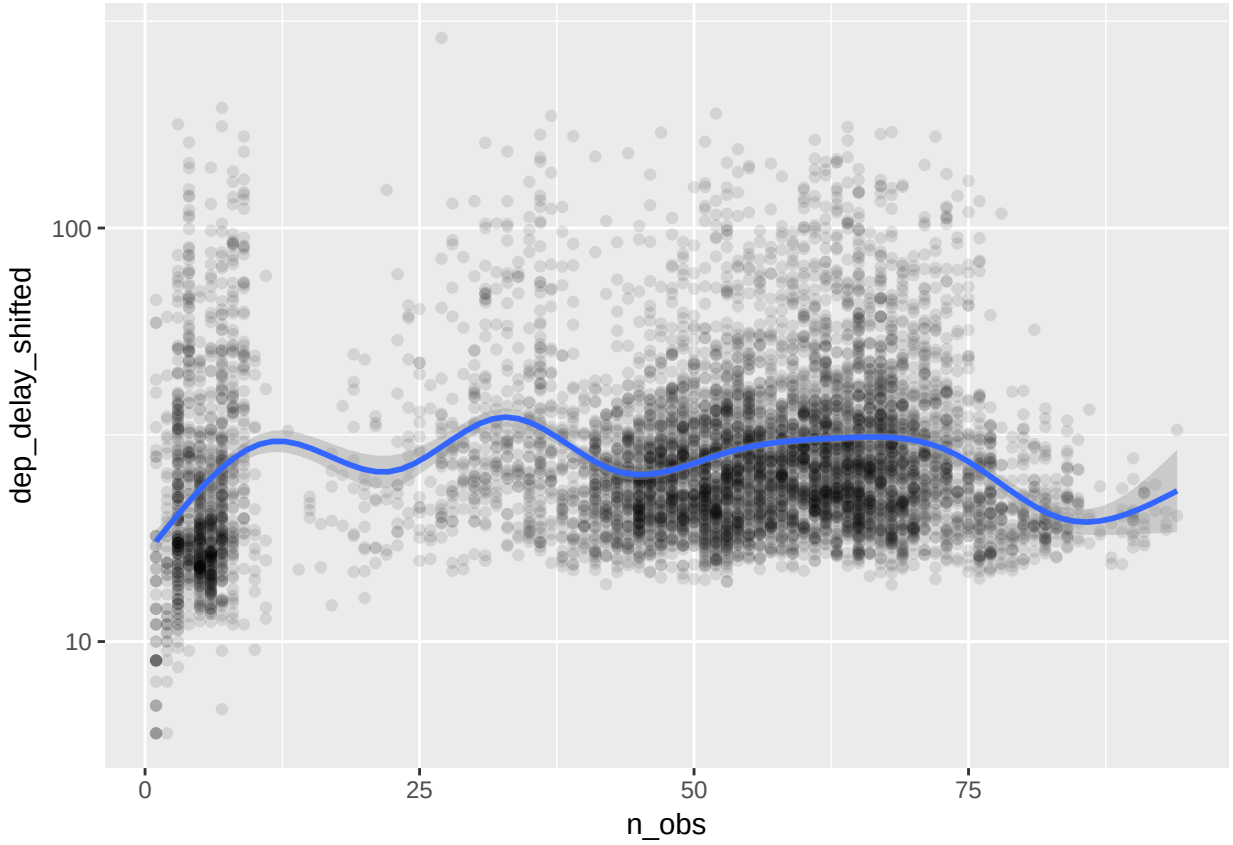
```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



The transformed distribution is much less skewed than the original. Now, let's plot the relationship between delays and flights again.

```
delay_day_hr  %>% filter(!is.na(dep_delay_shifted), dep_delay_shifted > 5) %>%
  ggplot(aes(x = n_obs, y = dep_delay_shifted)) +
  scale_y_log10() +     # using transformation scale_y_log10()
  geom_point(alpha = 0.1)  +
  geom_smooth()
```

```
## `geom_smooth()` using method = 'gam'
```



We still do not see a pattern that busier hours have more delays. This seems to suggest that the airports in New York City manage the fluctuating number of flights without causing congestion.

1.3 Huning down numbers

This section is optional but contains more examples of `dplyr` and `ggplot2` functions.

Previously, we find that the congestion at the airports is unlikely the cause of delays. Then, what else may explain the patterns of delays? Are the airlines partly responsible? Recall that earlier we observe that some airlines have longer delays than others for NYC-MSP flights. Let's take a look at the overall average delays by carrier.

```
stat_carrier <- flights %>%
  group_by(carrier) %>%
  summarise(n_obs = n(),
            dep_delay = mean(dep_delay, na.rm = TRUE),
            arr_delay = mean(arr_delay, na.rm = TRUE)
            ) %>%
  left_join(airlines, by="carrier") %>%
  arrange(desc(n_obs))

stat_carrier %>% kable(digit=2)
```

carrier	n_obs	dep_delay	arr_delay	name
UA	58665	12.11	3.56	United Air Lines Inc.
B6	54635	13.02	9.46	JetBlue Airways
EV	54173	19.96	15.80	ExpressJet Airlines Inc.
DL	48110	9.26	1.64	Delta Air Lines Inc.
AA	32729	8.59	0.36	American Airlines Inc.
MQ	26397	10.55	10.77	Envoy Air
US	20536	3.78	2.13	US Airways Inc.
9E	18460	16.73	7.38	Endeavor Air Inc.
WN	12275	17.71	9.65	Southwest Airlines Co.
VX	5162	12.87	1.76	Virgin America
FL	3260	18.73	20.12	AirTran Airways Corporation
AS	714	5.80	-9.93	Alaska Airlines Inc.
F9	685	20.22	21.92	Frontier Airlines Inc.
YV	601	19.00	15.56	Mesa Airlines Inc.
HA	342	4.90	-6.92	Hawaiian Airlines Inc.
OO	32	12.59	11.93	SkyWest Airlines Inc.

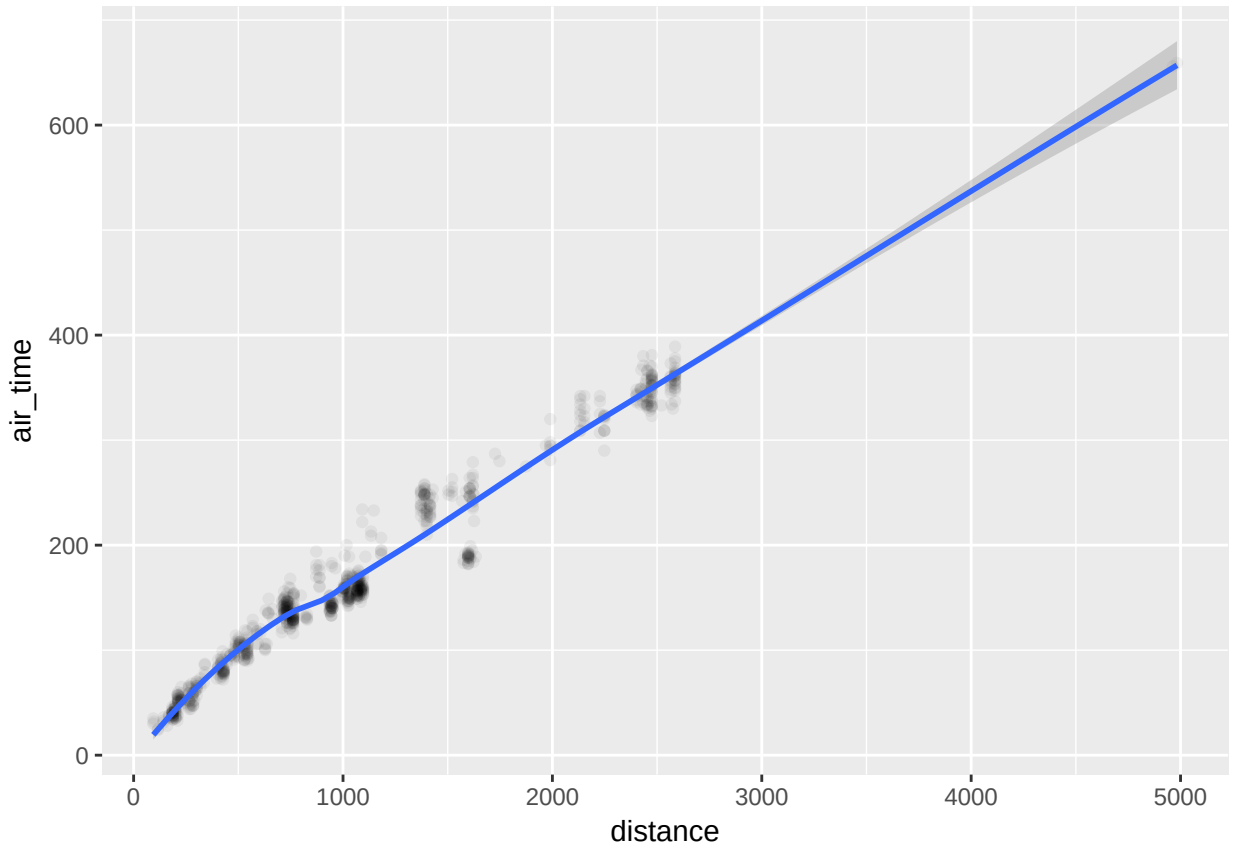
There could be some differences across carriers. However, the simple average of delays across various routes, days, and hours of flights may not be a good measure to compare the carriers. For example, some carriers may serve the routes and hours that tend to have more delays. Also, given that our dataset covers only the flights from New York City, the comparison may not be nationally representative since carriers use different airports around the country for their regional hubs.

For our purposes, let's compare the average air time among carriers, while accounting for flight's destination and timing. The differences in air time are not the same as the differences in delays, but they may indicate some efficiency difference among carriers.

Let's first check how air time relates to flight distance.

```
flights %>%
  filter (month == 1, day == 1, !is.na(air_time)) %>%
  ggplot(aes(x = distance, y = air_time)) +
  geom_point(alpha = 0.05)  +
  geom_smooth()
```

```
## `geom_smooth()` using method = 'loess'
```



`air_time` and `distance` shows a general linear relationship. We can better account for this relationship if we calculate the average air time for each flight destination from New York City.

First, we will consider a simple approach to control for such average air time for each destination and compare the variation in air time among carriers. We can do this by fitting a linear regression model with fixed destination effects and comparing the residuals. This resembles the ANOVA for comparing the mean air times among carriers, but the fixed destination effects here difference out the average air time for each destination from the total variation.

```
# a copy of flights data
flights2 <- flights

# TRUE/FALSE vector showing whther air_time is not NA.
idx0 <- flights %>% with(!is.na(air_time))

flights2$res <- NA # prepare a column of residuals to be defined below

flights2$res[idx0] <- flights2 %>%  # replace rows with idx0 = TRUE
  filter(!is.na(air_time)) %>%
  with(
   lm( air_time ~ as.factor(dest))
        # lm() estimates a linear model.
        # "y ~ x"" is the formula for regressing y on x.
        # as.factor() converts "dest" to a factor (categorical) class
        # which is used as a set of dummy variables in the regression.
   ) %>%
  residuals()  # obtains residuals of the lm() object
```

```
stat_res <- flights2 %>%
  group_by(carrier) %>%
  summarise(
    mean_res = mean(res, na.rm = TRUE), # mean residual by carrier
    sd_res = sd(res, na.rm = TRUE)
    )

left_join(stat_carrier, stat_res, by="carrier") %>% kable(digit=2)
```

carrier	n_obs	dep_delay	arr_delay	name	mean_res	sd_res
UA	58665	12.11	3.56	United Air Lines Inc.	-0.87	14.59
B6	54635	13.02	9.46	JetBlue Airways	0.28	11.55
EV	54173	19.96	15.80	ExpressJet Airlines Inc.	-0.37	8.94
DL	48110	9.26	1.64	Delta Air Lines Inc.	-0.20	12.32
AA	32729	8.59	0.36	American Airlines Inc.	0.68	13.86
MQ	26397	10.55	10.77	Envoy Air	0.45	8.87
US	20536	3.78	2.13	US Airways Inc.	-0.42	9.43
9E	18460	16.73	7.38	Endeavor Air Inc.	0.84	8.76
WN	12275	17.71	9.65	Southwest Airlines Co.	0.16	12.55
VX	5162	12.87	1.76	Virgin America	3.26	17.58
FL	3260	18.73	20.12	AirTran Airways Corporation	1.16	8.75
AS	714	5.80	-9.93	Alaska Airlines Inc.	-2.13	16.17
F9	685	20.22	21.92	Frontier Airlines Inc.	3.12	15.16
YV	601	19.00	15.56	Mesa Airlines Inc.	-0.05	7.06
HA	342	4.90	-6.92	Hawaiian Airlines Inc.	5.64	20.69
OO	32	12.59	11.93	SkyWest Airlines Inc.	1.02	7.26

The differences in air time across carriers (“mean_res”) somewhat differ from the patterns of differences in the simple averages of delays (“dep_delay” and “arr_delay”). The patterns are different between “dep_delay” and “arr_delay” for that matter.

To some extent, it appears to make sense that the average air time is longer for low-cost carriers such as Virgin America, Frontier Airlines, and Hawaiian Airlines. The differences across other carriers, on the other hand, are small, compared to the standard deviations. To get a sense of whether these differences have any statistical significance, let’s use t-test to compare the mean residual between United Airlines and American Airlines.

```
# t-test comparing UA vs AA for the mean air time
flights2 %>%
  with({
    idx_UA <- carrier == "UA"
    idx_AA <- carrier == "AA"
    t.test(res[idx_UA], res[idx_AA])
    })
```

```
##
##  Welch Two Sample t-test
##
## data:  res[idx_UA] and res[idx_AA]
## t = -15.722, df = 68826, p-value < 2.2e-16
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -1.741133 -1.355142
## sample estimates:
##  mean of x  mean of y
```

```
## -0.8689523  0.6791852
```

With a large number of observations, a seemingly-small difference in the means often turn out to be a statistically significant difference. Nonetheless, statistical significance is not sufficient for being an empirically significant difference that matters in the real world. The average difference of about 1.5 minute air time per flight appears very small.

In fact, we can do this sort of pair-wise comparisons all at once using a regression. Using carrier fixed effects in addition to destination fixed effects, we can directly compare the mean effects across carriers. We will set United Airlines to be a reference of the carrier fixed effects, so that the fixed effect for United Airlines is set to zero (i.e., omitted category), from which the fixed effects of all other airlines are estimated.

```
flights2$carrier <- relevel(factor(flights2$carrier), ref="UA")
# reference level is United Airlines
flights2$carrier %>% table()
```

```
## .
##    UA    9E    AA    AS    B6    DL    EV    F9    FL    HA    MQ    OO
## 58665 18460 32729   714 54635 48110 54173   685  3260   342 26397    32
##    US    VX    WN    YV
## 20536  5162 12275   601
```

```
flights2 %>%
  with({
    n_carrier <- unique(carrier) %>% length()
    n_dest <- unique(dest) %>% length()
    print(paste('There are', n_carrier, 'distinct carriers and',
                n_dest,'distinct destinations in the data.' ))
  })
```

```
## [1] "There are 16 distinct carriers and 105 distinct destinations in the data."
```

With 16 carriers and 105 destinations minus 2 reference levels for carriers and destinations, the total of 119 coefficients will be estimated for the fixed effects.

```
f1 <- flights2 %>%
 with(
   lm( air_time  ~  as.factor(carrier) + as.factor(dest) )
    # fixed effects for carriers and destinations
 )

tidy(f1)[1:20,] # show the first 20 coefficients
```

```
##                    term     estimate  std.error    statistic       p.value
## 1           (Intercept)  247.9884874 0.75069658  330.3445016  0.000000e+00
## 2  as.factor(carrier)9E    1.8015498 0.12723996   14.1586788  1.702649e-45
## 3  as.factor(carrier)AA    1.9326712 0.09731105   19.8607572  1.002388e-87
## 4  as.factor(carrier)AS   -1.9071536 0.49596319   -3.8453531  1.204017e-04
## 5  as.factor(carrier)B6    1.1808039 0.08495098   13.8998267  6.535025e-44
## 6  as.factor(carrier)DL    0.7531812 0.08722600    8.6348244  5.907432e-18
## 7  as.factor(carrier)EV    0.4174574 0.11044837    3.7796605  1.570702e-04
## 8  as.factor(carrier)F9    3.8891981 0.48090201    8.0872985  6.120836e-16
## 9  as.factor(carrier)FL    2.6434074 0.27600661    9.5773336  1.002386e-21
## 10 as.factor(carrier)HA   11.0125104 0.89821710   12.2604106  1.503557e-34
## 11 as.factor(carrier)MQ    1.4592669 0.11892133   12.2708590  1.321669e-34
## 12 as.factor(carrier)OO    1.8091432 2.21222472    0.8177936  4.134757e-01
## 13 as.factor(carrier)US    0.1319337 0.13826299    0.9542230  3.399715e-01
## 14 as.factor(carrier)VX    4.5298528 0.18441295   24.5636378 4.086448e-133
```

```
## 15 as.factor(carrier)WN    1.2226161 0.17520980    6.9780125  2.999500e-12
## 16 as.factor(carrier)YV    0.5167461 0.52737831    0.9798395  3.271661e-01
## 17   as.factor(dest)ACK -207.1011095 1.04478912 -198.2228803  0.000000e+00
## 18   as.factor(dest)ALB -216.6188634 0.95130943 -227.7059972  0.000000e+00
## 19   as.factor(dest)ANC  165.1365126 4.26930687   38.6799351  0.000000e+00
## 20   as.factor(dest)ATL -136.1282095 0.75641976 -179.9638460  0.000000e+00
```

```
# a function to clean up the coefficient table above
clean_lm_rlt <- function(f) {
  # keep only rows for which column "term" contains "carrier"  e.g., rows 2 to 16 above
  rlt <- tidy(f) %>% filter(grepl("carrier",term))

  # create column named carrier
  rlt <- rlt %>% mutate(carrier = gsub('as.factor\\(carrier\\)','', term))

  # drop column term
  rlt <- rlt %>% select(-term)

  # add columns of carrier, name, and n_obs from the stat_carrier data frame
  stat_carrier %>%
    select(carrier, name, n_obs) %>%
    left_join(rlt, by="carrier")
}

lm_rlt1 <- clean_lm_rlt(f1)
lm_rlt1 %>% kable(digit=2)
```

carrier	name	n_obs	estimate	std.error	statistic	p.value
UA	United Air Lines Inc.	58665	NA	NA	NA	NA
B6	JetBlue Airways	54635	1.18	0.08	13.90	0.00
EV	ExpressJet Airlines Inc.	54173	0.42	0.11	3.78	0.00
DL	Delta Air Lines Inc.	48110	0.75	0.09	8.63	0.00
AA	American Airlines Inc.	32729	1.93	0.10	19.86	0.00
MQ	Envoy Air	26397	1.46	0.12	12.27	0.00
US	US Airways Inc.	20536	0.13	0.14	0.95	0.34
9E	Endeavor Air Inc.	18460	1.80	0.13	14.16	0.00
WN	Southwest Airlines Co.	12275	1.22	0.18	6.98	0.00
VX	Virgin America	5162	4.53	0.18	24.56	0.00
FL	AirTran Airways Corporation	3260	2.64	0.28	9.58	0.00
AS	Alaska Airlines Inc.	714	-1.91	0.50	-3.85	0.00
F9	Frontier Airlines Inc.	685	3.89	0.48	8.09	0.00
YV	Mesa Airlines Inc.	601	0.52	0.53	0.98	0.33
HA	Hawaiian Airlines Inc.	342	11.01	0.90	12.26	0.00
OO	SkyWest Airlines Inc.	32	1.81	2.21	0.82	0.41

The "estimate" column shows the mean difference in air time with United Airlines, accounting for flight destination. The estimate tends to be more precise (i.e., smaller standard errors) for carriers with a larger number of observations. This time, we find that Virgin America, Air Tran, Frontier Airlines, and Hawaiian Airlines tend to show particularly longer air times than United Airlines.

Next, let's take a step further to account for flight timing as well. We can do this by adding fixed effects for flight dates and hours.

```
flights2 <- flights2 %>%
  mutate( date_id = month*100 + day )
```

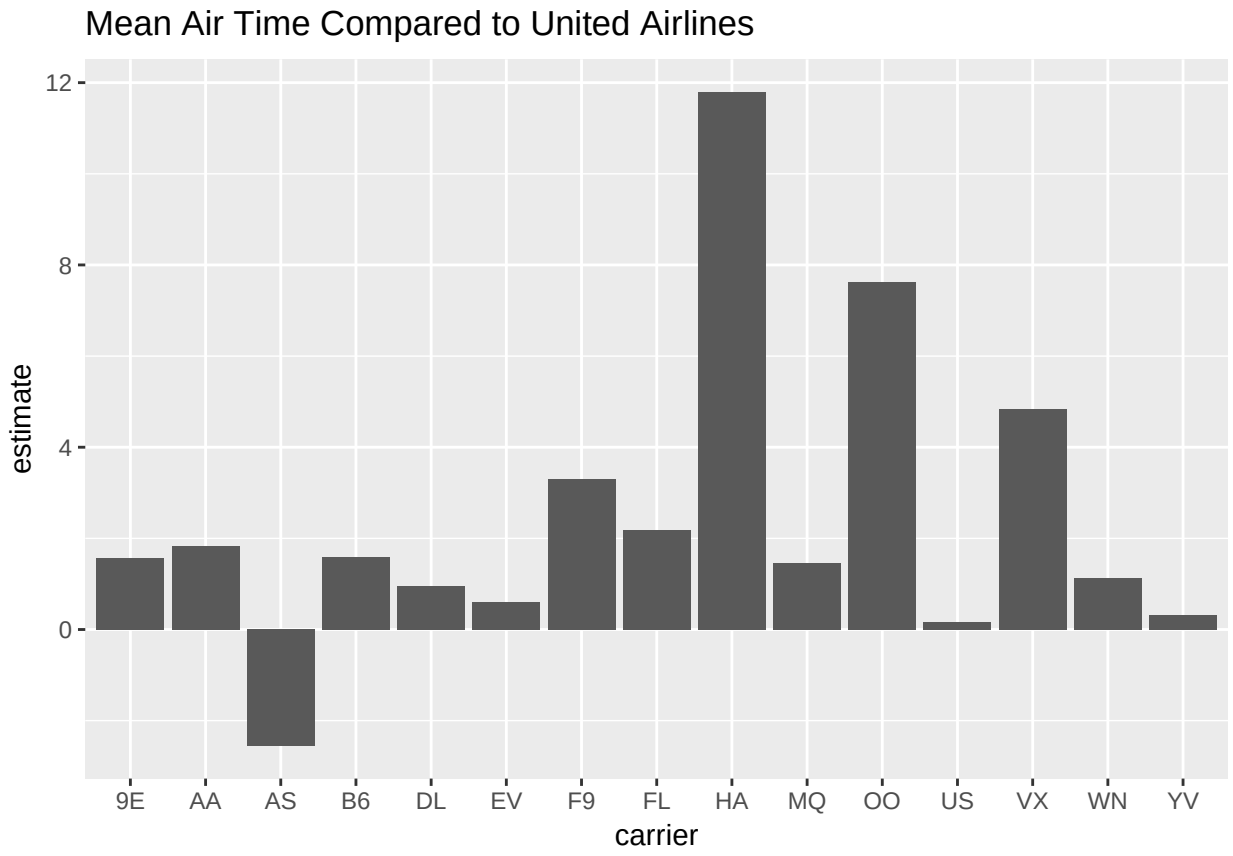
```
flights2$date_id %>% unique() %>% length()
```

```
## [1] 365
```

```
f2 <- flights2 %>%
 with(
   lm( air_time ~  as.factor(carrier) + as.factor(dest) +
        + as.factor(date_id) + as.factor(hour) )
 )

lm_rlt2 <- clean_lm_rlt(f2)
lm_rlt2 %>% kable(digit=2)
```

carrier	name	n_obs	estimate	std.error	statistic	p.value
UA	United Air Lines Inc.	58665	NA	NA	NA	NA
B6	JetBlue Airways	54635	1.60	0.07	22.50	0.00
EV	ExpressJet Airlines Inc.	54173	0.61	0.09	6.67	0.00
DL	Delta Air Lines Inc.	48110	0.95	0.07	13.03	0.00
AA	American Airlines Inc.	32729	1.84	0.08	22.81	0.00
MQ	Envoy Air	26397	1.45	0.10	14.70	0.00
US	US Airways Inc.	20536	0.17	0.11	1.51	0.13
9E	Endeavor Air Inc.	18460	1.57	0.11	14.72	0.00
WN	Southwest Airlines Co.	12275	1.14	0.15	7.82	0.00
VX	Virgin America	5162	4.85	0.15	31.57	0.00
FL	AirTran Airways Corporation	3260	2.19	0.23	9.58	0.00
AS	Alaska Airlines Inc.	714	-2.55	0.41	-6.21	0.00
F9	Frontier Airlines Inc.	685	3.31	0.40	8.29	0.00
YV	Mesa Airlines Inc.	601	0.32	0.44	0.73	0.46
HA	Hawaiian Airlines Inc.	342	11.79	0.75	15.80	0.00
OO	SkyWest Airlines Inc.	32	7.63	1.83	4.17	0.00

```
lm_rlt2 %>% filter(carrier!='UA') %>%
  ggplot(aes(x = carrier, y = estimate)) + geom_col() +
  labs(title = "Mean Air Time Compared to United Airlines")
```



The results are similar to the previous linear mode except that this time SkyWest Airlines shows much longer air time.

Before wrapping up, our final model is a check for the robustness of the above results. We would like to replace the date and hour fixed effects in the previous model with date-hour fixed effects (i.e., the interaction between date and hour). We could add such fixed effects using `time_hour` variable defined above. However, that would mean adding nearly 7,000 dummy variables to our linear regression, which is computationally too intensive.

To work around this issue, we approximate this estimation by pre-processing the dependent variable. Specifically, we calculate the average air time for each combination of `time_hour` and `dest` and define a new dependent variable by subtracting this average value from the original air time variable (i.e., the new variable is centered at zero-mean for each combination of `time_hour` and `dest`). Then, we estimate a linear model with carrier and destination fixed effects.

```
## Adding time_hour fixed effects is too computationally intensive
# f1 <- flights %>%
#  with(
#    lm( air_time  ~  as.factor(carrier) + as.factor(dest) + as.factor(time_hour))
#  )

unique(flights2$time_hour) %>% length()  # 6,936 unique time_hour
```

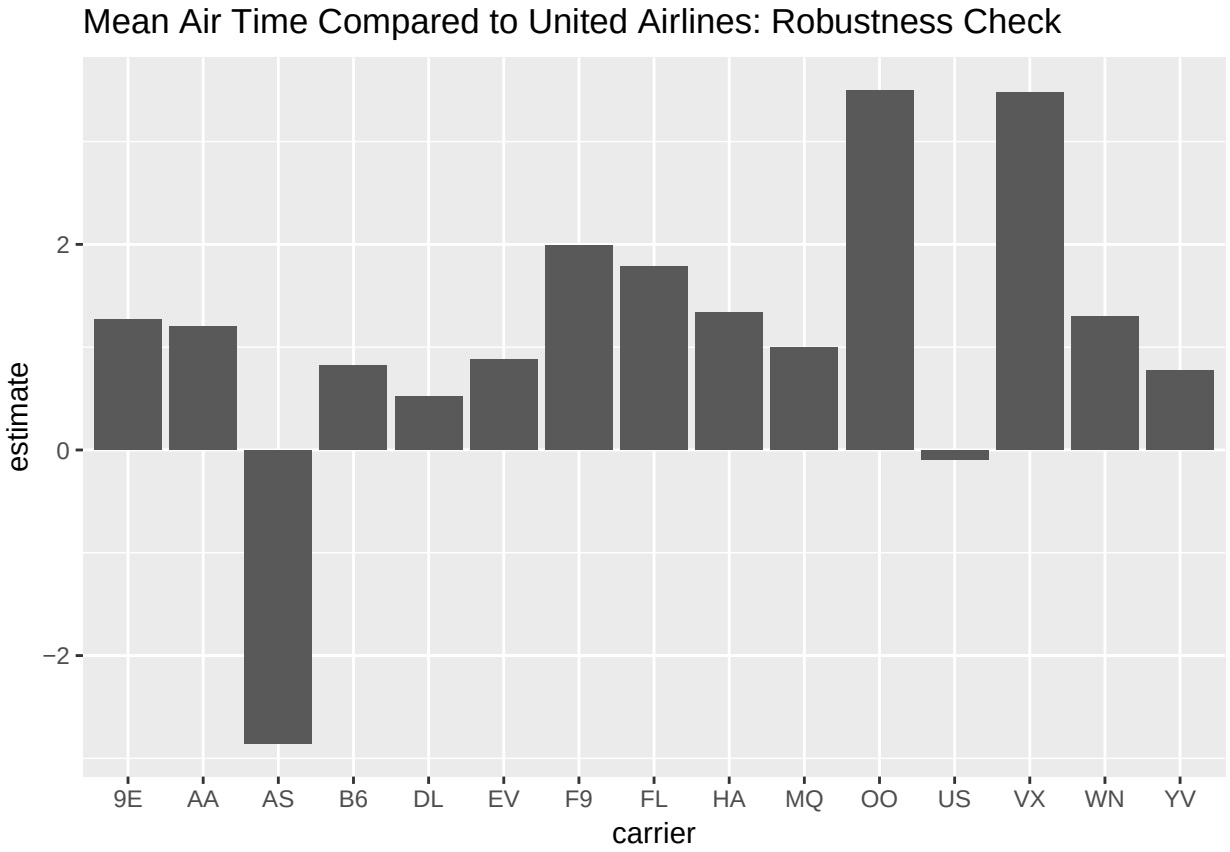
```
## [1] 6936
```

```
flights2 <- flights2 %>%
  group_by(dest, time_hour) %>%
  mutate(
    air_time_centered = air_time - mean(air_time, na.rm=TRUE)
```

```
  )

f3 <- flights2 %>%
 with(
   lm( air_time_centered  ~  as.factor(carrier) + as.factor(dest) )
 )

lm_rlt3 <- clean_lm_rlt(f3)
lm_rlt3 %>% kable(digit=2) # Note: standard errors, t-stat, and p-val are incorrect
```

carrier	name	n_obs	estimate	std.error	statistic	p.value
UA	United Air Lines Inc.	58665	NA	NA	NA	NA
B6	JetBlue Airways	54635	0.82	0.03	32.24	0.00
EV	ExpressJet Airlines Inc.	54173	0.88	0.03	26.50	0.00
DL	Delta Air Lines Inc.	48110	0.52	0.03	19.85	0.00
AA	American Airlines Inc.	32729	1.20	0.03	41.06	0.00
MQ	Envoy Air	26397	1.00	0.04	27.84	0.00
US	US Airways Inc.	20536	-0.09	0.04	-2.21	0.03
9E	Endeavor Air Inc.	18460	1.27	0.04	33.07	0.00
WN	Southwest Airlines Co.	12275	1.30	0.05	24.70	0.00
VX	Virgin America	5162	3.47	0.06	62.59	0.00
FL	AirTran Airways Corporation	3260	1.78	0.08	21.48	0.00
AS	Alaska Airlines Inc.	714	-2.86	0.15	-19.15	0.00
F9	Frontier Airlines Inc.	685	1.99	0.14	13.73	0.00
YV	Mesa Airlines Inc.	601	0.78	0.16	4.89	0.00
HA	Hawaiian Airlines Inc.	342	1.34	0.27	4.96	0.00
OO	SkyWest Airlines Inc.	32	3.50	0.67	5.26	0.00

```
lm_rlt3 %>% filter(carrier!='UA') %>%
  ggplot(aes(x = carrier, y = estimate)) + geom_col() +
  labs(title = "Mean Air Time Compared to United Airlines: Robustness Check")
```



The point estimates should be *approximately* close to what we would obtain if we regress `air_time` on the fixed effects of `carrier`, `dest`, and `time_hour`. However, the standard errors are not correctly displayed in the table because the centered variable has less total variation compared to the original `air_time` variable. (Correct standard errors can be obtained, for example, through a bootstrapping technique.)

Overall, we see again a tendency that low-cost carriers like Sky West Airlines, Virgin America, Frontier Airlines, and Air Tran show particularly longer air time than United Airlines. Jet Blue Airways, another low-cost carrier, shows a less obvious difference from United Airlines, possibly suggesting that their operation focused on the East Cost is efficient for the flights departing from New York City. Hawaiian Airlines and Alaskan Airlines appear to be somewhat different from other carriers perhaps because they are more specialized in particular flight times and destinations compared to their rivals. In particular, the flights to Hawaii may have distinct delay patterns that are concentrated on certain date-hours of the peak vacation seasons.

1.4 Reflections

In this introduction, we have reviewed the tools of `deplyr` and `ggplot2` packages as a starting point for data analyses and visualization in R. This new generation of tools is a data exploration language as much as a set of functions to shortcut traditional data manipulation methods in R. This language provides a very intuitive system of translating our inquiries to the data analysis in R.

Using the flight dataset, we have also investigated flight delay patterns. We find that airport congestion is unlikely a major cause of delay in New York City. There are small differences in the air time (e.g. less than 5 minutes) across carriers for a given destination although it remains unclear how this relates to the delay patterns.

In fact, the concept of "delay" is complicated because it is defined in reference to the scheduled departure

and arrival times, which may differ by carrier. A delay would not include the time sitting in the airplane before taking off or after landing as long as it is within the schedule. It might be more interesting to compare scheduled flight duration instead of delays or air time. (Such an analysis would involve somewhat complicated manipulations of date and time with our flight data.)

This leads us to the final point of this exercise; an interesting data analysis requires **knowledge on the real-world process that generated the data** and **the ability to ask interesting questions**. `deplyr` and `ggplot2` packages can let you employ a variety of data analytics tools with ease, but the ultimate power of the analysis will always rest on your knowledge and creativity.

Chapter 2

Essentials

2017-04-08: *VERY Preliminary!*

This section provides an overview of the essential concepts for manipulating data and programming in R.

2.1 Cheatsheets

Cheatsheets are useful for glancing at various functions.

- Base R
- RStudio IDE
- dplyr
- ggplot2

2.2 Data types

2.2.1 Atomic

In most cases, each atomic element has a type (mode) of;

- `numeric`: number
- `logical`: TRUE or FALSE (T or F for shortcuts)
- `character`: character string
- `factor`: a level of categorical variable

Other types include date and nonexistent `NULL`. The factor is also a *class* of its own, meaning that many R functions apply operations that are specific to the *factor class*.

```
# assess objects 123, "abc", and TRUE for their types
str(123)  # str() returns the structure
```

```
##  num 123
```

```
str("abc")
```

```
##  chr "abc"
```

```
str(TRUE)
```

```
##  logi TRUE
```

```
c(is.numeric(123), is.numeric("abc"), is.numeric(TRUE))
```

```
## [1]  TRUE FALSE FALSE
```

```
c(is.logical(123), is.logical("abc"), is.logical(TRUE))
```

```
## [1] FALSE FALSE  TRUE
```

```
c(is.character(123), is.character("abc"), is.character(TRUE))
```

```
## [1] FALSE  TRUE FALSE
```

```
# "<-" means an assignment from right to left
factor1 <- as.factor(c(1,2,3)) # Looks like numeric but not
factor1
```

```
## [1] 1 2 3
## Levels: 1 2 3
```

```
factor2 <- as.factor(c("a","b","c"))  # Looks like characters but not
factor2
```

```
## [1] a b c
## Levels: a b c
```

```
factor3 <- as.factor(c(TRUE,FALSE,T)) # Looks like logicals but not
factor3
```

```
## [1] TRUE  FALSE TRUE
## Levels: FALSE TRUE
```

```
c(is.factor(factor1[1]), is.factor(factor2[1]), is.factor(factor3[1]))
```

```
## [1] TRUE TRUE TRUE
```

```
# Extract the first element (factor1[1] etc.)
factor1[1]
```

```
## [1] 1
## Levels: 1 2 3
```

```
factor2[2]
```

```
## [1] b
## Levels: a b c
```

```
factor3[3]
```

```
## [1] TRUE
## Levels: FALSE TRUE
```

`NULL` has zero-length. Also, empty numeric, logical, and character objects have zero-length.

```
length(NULL)
```

```
## [1] 0
```

```
length(numeric(0))  # numeric(N) returns a vector of N zeros
```

```
## [1] 0
```

```
length(logical(0))  # logical(N) returns a vector of N FALSE objects
```

```
## [1] 0
```

```
length(character(0)) # character(N) returns a vector of N "" objects
```

```
## [1] 0
```

Each **vector** has a type of `numeric`, `logical`, `character`, or `factor`. Each **matrix** has a type of `numeric`, `logical`, or `character`. A **data frame** can contain mixed types across columns where each column (e.g., a variable) has a type of `numeric`, `logical`, `character` or `factor`.

```
vector1 <- c(1, NA, 2, 3) # read as numeric
vector1
```

```
## [1]  1 NA  2  3
```

```
vector2 <- c(TRUE, FALSE, T, F) # read as logical
vector2
```

```
## [1]  TRUE FALSE  TRUE FALSE
```

```
vector3 <- c(1, NA, "abc", TRUE, "TRUE") # read as character
vector3
```

```
## [1] "1"    NA     "abc"  "TRUE" "TRUE"
```

```
vector4 <- as.factor(c(1, NA, "abc", TRUE, "TRUE")) # read as factor
vector4
```

```
## [1] 1    <NA> abc  TRUE TRUE
## Levels: 1 abc TRUE
```

```
matrix1 <- matrix(c(1:6), nrow = 3) # read as numeric
matrix1
```

```
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

```
matrix2 <- matrix(c(TRUE,FALSE,rep(T,3),F), nrow = 3)  # read as logical
matrix2
```

```
##       [,1]  [,2]
## [1,]  TRUE  TRUE
## [2,] FALSE  TRUE
## [3,]  TRUE FALSE
```

```
matrix3 <- matrix(c(1,2,3,"a","b","abc"), nrow = 3) # read as character
matrix3
```

```
##      [,1] [,2]
## [1,] "1"  "a"
## [2,] "2"  "b"
## [3,] "3"  "abc"
```

```
df1 <- data.frame(
        num  = c(1,2,3),           # read as numeric
        fac1 = c("a","b","abc"),   # read as factor
        logi = c(TRUE, FALSE, T),  # read as logical
        fac2  = c(1,"a",TRUE)      # read as factor
```

```
       )
df1
```

```
##   num fac1  logi fac2
## 1   1    a  TRUE    1
## 2   2    b FALSE    a
## 3   3  abc  TRUE TRUE
```

```
df1$num   # "$" symbol is used to extract a column
```

```
## [1] 1 2 3
```

```
df1$fac1  # character type is converted into a factor
```

```
## [1] a   b   abc
## Levels: a abc b
```

```
df1$logi
```

```
## [1]  TRUE FALSE  TRUE
```

```
df1$fac2  # mixed types within a column is converted into a factor
```

```
## [1] 1    a    TRUE
## Levels: 1 a TRUE
```

```
# additional argument "stringsAsFactors = FALSE" preserves character types.
df2 <- data.frame(
        num  = c(1,2,3),           # read as numeric
        char = c("a","b","abc"),   # read as character
        logi = c(TRUE, FALSE, T),  # read as logical
        fac2  = as.factor(c(1,"a",TRUE)),      # read as factor
        stringsAsFactors = FALSE
       )
df2
```

```
##   num char  logi fac2
## 1   1    a  TRUE    1
## 2   2    b FALSE    a
## 3   3  abc  TRUE TRUE
```

```
df2$num
```

```
## [1] 1 2 3
```

```
df2$char
```

```
## [1] "a"   "b"   "abc"
```

```
df2$logi
```

```
## [1]  TRUE FALSE  TRUE
```

```
df2$fac2
```

```
## [1] 1    a    TRUE
## Levels: 1 a TRUE
```

2.2.2 Factor

A factor object is defined with a set of categorical levels, which may be labeled. The levels are either **ordered** (defined by `ordered()`) or **unordered** (defined by `factor()`). Ordered factor objects are treated in the specific order by certain statistical and graphical procedures.

```
# We will convert the columns of df into factors
df <- data.frame(
      fac1 = c(0,1,1,4,4,2,2,3),
      fac2 = c(1,2,3,1,1,2,2,3),
      fac3 = c(4,2,3,4,4,2,2,3)
      )

# convert fac1 to ordered factors
df$fac1 <- ordered(df$fac1,
                   levels = c(0,4,3,2,1) # defines the order
                   )
df$fac1
```

```
## [1] 0 1 1 4 4 2 2 3
## Levels: 0 < 4 < 3 < 2 < 1
```

```
summary(df$fac1)  # gives the table of counts for each level
```

```
## 0 4 3 2 1
## 1 2 1 2 2
```

```
# convert fac2 to unordered factors with labels
df$fac2 <- factor(df$fac2,
                   levels = c(1,2,3), #  no particular order
                   # attach labels to factors: 1=red, 2=blue, 3=green
                   labels = c("red", "blue", "green")
                   )
df$fac2
```

```
## [1] red   blue  green red   red   blue  blue  green
## Levels: red blue green
```

```
summary(df$fac2)
```

```
##   red  blue green
##     3     3     2
```

```
# convert fac3 to ordered factors with labels
df$fac3 <- ordered(df$fac3,
                   levels = c(2,3,4),
                   # attach labels to factors: 2=Low, 3=Medium, 4=High
                   labels = c("Low", "Medium", "High")
                   )
df$fac3
```

```
## [1] High   Low    Medium High   High   Low    Low    Medium
## Levels: Low < Medium < High
```

```
summary(df$fac3)
```

```
##    Low Medium   High
##      3      2      3
```

2.2.3 Matrix

`matrix()` defines a matrix from a vector. The default is to arrange the vector by column (`byrow = FALSE`).

```
# byrow = FALSE  (the default)
matrix(data = c(1:6), nrow = 2, ncol = 3, byrow = FALSE, dimnames = NULL)
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

```
# byrow = TRUE
matrix(data = c(1:6), nrow = 2, ncol = 3, byrow = TRUE, dimnames = NULL)
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
```

```
# give row and column names to a matrix
mat1 <- matrix(data = c(1:6), nrow = 2, ncol = 3, byrow = FALSE,
       dimnames = list(c("r1","r2"), c("c1","c2","c3")))
mat1
```

```
##    c1 c2 c3
## r1  1  3  5
## r2  2  4  6
```

```
dim(mat1)  # dimension: row by column
```

```
## [1] 2 3
```

```
colnames(mat1)
```

```
## [1] "c1" "c2" "c3"
```

```
rownames(mat1)
```

```
## [1] "r1" "r2"
```

```
colnames(mat1) <- c("v1","v2","v3")  # change column names by assignment "<-"
mat1
```

```
##    v1 v2 v3
## r1  1  3  5
## r2  2  4  6
```

```
# R makes a guess when only nrow or ncol is supplied
matrix(data = c(1:6), nrow = 2)
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

```
matrix(data = c(1:6), ncol = 3)
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

```
# combine matrices by column via "cbind()" or by row via "rbind()"
cbind(mat1,mat1)
```

```
##    v1 v2 v3 v1 v2 v3
```

```
## r1  1  3  5  1  3  5
## r2  2  4  6  2  4  6
```

```
rbind(mat1,mat1)
```

```
##    v1 v2 v3
## r1  1  3  5
## r2  2  4  6
## r1  1  3  5
## r2  2  4  6
```

There are recycling rules in R.

```
# the vector shorter than the length of all elements of a matrix
matrix(data = c(1:4), nrow = 2, ncol= 3)
```

```
## Warning in matrix(data = c(1:4), nrow = 2, ncol = 3): data length [4] is
## not a sub-multiple or multiple of the number of columns [3]
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    1
## [2,]    2    4    2
```

```
# R treats a scaler as a vector of length that conforms cbind() or rbind()
cbind(mat1, colA = 1)
```

```
##    v1 v2 v3 colA
## r1  1  3  5    1
## r2  2  4  6    1
```

```
rbind(mat1, rowA= 1, rowB= 2, rowC= 3)
```

```
##      v1 v2 v3
## r1    1  3  5
## r2    2  4  6
## rowA  1  1  1
## rowB  2  2  2
## rowC  3  3  3
```

To replace elements of a matrix, we can use assignment operator `<-`.

```
mat1[1,1] <- 10
mat1
```

```
##    v1 v2 v3
## r1 10  3  5
## r2  2  4  6
```

```
mat1[,2] <- c(7,8)
mat1
```

```
##    v1 v2 v3
## r1 10  7  5
## r2  2  8  6
```

```
mat1[,1] <- 0  # recycling rule
mat1
```

```
##    v1 v2 v3
## r1  0  7  5
## r2  0  8  6
```

Matrix allows for easy extraction for rows and columns separated by comma.

```
mat1
```

```
##    v1 v2 v3
## r1  0  7  5
## r2  0  8  6
```

```
mat1[1, ]  # row = 1 and all columns
```

```
## v1 v2 v3
##  0  7  5
```

```
mat1[, 1]  # all rows and col = 1
```

```
## r1 r2
##  0  0
```

```
mat1[c(TRUE,FALSE),]  # by a logical vector
```

```
## v1 v2 v3
##  0  7  5
```

```
mat1[, c(TRUE,FALSE)]
```

```
##    v1 v3
## r1  0  5
## r2  0  6
```

```
mat1[2,3]  # row = 2 and col = 3
```

```
## [1] 6
```

```
mat1[1:2, 2:3]  # row = 1:2 and col = 2:3
```

```
##    v2 v3
## r1  7  5
## r2  8  6
```

```
mat1[1:2, 2:3][2,2]  # subset of a subset
```

```
## [1] 6
```

```
mat1[, 1][2]  # vector extraction is done with one-dimensional index
```

```
## r2
##  0
```

Important: when a single row or column is extracted, it gets converted to a vector with no dimension.

```
mat1[1,]
```

```
## v1 v2 v3
##  0  7  5
```

```
is.matrix(mat1[1, ])
```

```
## [1] FALSE
```

```
dim(mat1[1,])
```

```
## NULL
```

```
length(mat1[1, ])
```

```
## [1] 3
```

```
# to keep a row or column vector structure, use drop = FALSE
mat1[1,, drop = FALSE]
```

```
##    v1 v2 v3
## r1  0  7  5
```

```
is.matrix(mat1[1,,drop = FALSE])
```

```
## [1] TRUE
```

```
dim(mat1[1,,drop = FALSE])
```

```
## [1] 1 3
```

```
length(mat1[1,,drop = FALSE])
```

```
## [1] 3
```

```
mat1[,1, drop = FALSE]
```

```
##    v1
## r1  0
## r2  0
```

```
is.matrix(mat1[,1,drop = FALSE])
```

```
## [1] TRUE
```

```
dim(mat1[,1,drop = FALSE])
```

```
## [1] 2 1
```

```
length(mat1[,1,drop = FALSE])
```

```
## [1] 2
```

Another way of extraction from a matrix is to use row or column names.

```
mat1[,'v1']
```

```
## r1 r2
##  0  0
```

```
mat1[,c('v1','v3')]
```

```
##    v1 v3
## r1  0  5
## r2  0  6
```

```
mat1['r2',,drop= FALSE]
```

```
##    v1 v2 v3
## r2  0  8  6
```

`apply()` applies a function for a specified margin (dimension index number) of the matrix.

```
mat1
```

```
##    v1 v2 v3
## r1  0  7  5
## r2  0  8  6
```

```
apply(mat1,1,mean)  # dimension 1 (across rows)
```

```
##       r1       r2
```

```
## 4.000000 4.666667
```

```
apply(mat1,2,mean)  # dimension 2 (across columns)
```

```
##  v1  v2  v3
## 0.0 7.5 5.5
```

```
# one can write a custom function inside apply(). (called annonymous function)
# Its argument corresponds to the row or column vector passed by apply().
apply(mat1,2, function(x) sum(x)/length(x) )  # x is the internal vector name
```

```
##  v1  v2  v3
## 0.0 7.5 5.5
```

```
ans1 <- apply(mat1,2, function(x) {
                          avg = mean(x)
                          sd = sd(x)
                           # return the results as a list
                          list(Avg = avg, Sd = sd)
                        }
        )

unlist(ans1[[2]])  # results for the second column
```

```
##       Avg        Sd
## 7.5000000 0.7071068
```

```
unlist(ans1[[3]])  # results for the third column
```

```
##       Avg        Sd
## 5.5000000 0.7071068
```

Arrays are a generalization of matrices and can be more than 2 dimension.

```
array(c(1:18), c(2,3,3))  # dimension 2 by 2 by 3
```

```
## , , 1
##
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,]    7    9   11
## [2,]    8   10   12
##
## , , 3
##
##      [,1] [,2] [,3]
## [1,]   13   15   17
## [2,]   14   16   18
```

```
array(c(1:9), c(2,3,3))  # R recycles the vector
```

```
## , , 1
##
##      [,1] [,2] [,3]
```

```
## [1,]    1    3    5
## [2,]    2    4    6
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,]    7    9    2
## [2,]    8    1    3
##
## , , 3
##
##      [,1] [,2] [,3]
## [1,]    4    6    8
## [2,]    5    7    9
```

2.2.4 Data Frame

A data frame is similar to a matrix, but it accepts multiple types (modes) of variables across columns (e.g., a dataset in typical data analysis programs like SAS, SPSS, Stata etc.). In some cases matrices and data frames may be treated interchangeably, but generally they need to be distinguished. Data manipulation functions are often written for data frames, while some base R functions are written for matrices.

```
mymat1 <- matrix(data = c(1:6), nrow = 2, ncol = 3,
       dimnames = list(c("r1","r2"), c("c1","c2","c3")))
mymat1
```

```
##    c1 c2 c3
## r1  1  3  5
## r2  2  4  6
```

```
class(mymat1)
```

```
## [1] "matrix"
```

```
colnames(mymat1)
```

```
## [1] "c1" "c2" "c3"
```

```
names(mymat1)
```

```
## NULL
```

```
mydf1 <- data.frame(
                mymat1,
                num  = c(1,2),
                fac1 = c("a","abc"),
                logi = c(TRUE, FALSE),
                fac2  = c(1,"a")
              )
mydf1
```

```
##    c1 c2 c3 num fac1  logi fac2
## r1  1  3  5   1    a  TRUE    1
## r2  2  4  6   2  abc FALSE    a
```

```
class(mydf1)
```

```
## [1] "data.frame"
```

```
colnames(mydf1)
```

```
## [1] "c1"   "c2"   "c3"   "num"  "fac1" "logi" "fac2"
```

```
names(mydf1)   # colnames and names are the same
```

```
## [1] "c1"   "c2"   "c3"   "num"  "fac1" "logi" "fac2"
```

Extracting elements from a data frame is similar to extracting from a matrix, but there are a few additional methods.

```
mydf1[1,]   # row = 1 and all columns
```

```
##    c1 c2 c3 num fac1 logi fac2
## r1  1  3  5   1    a TRUE    1
```

```
mydf1[,1]   # all rows and col = 1
```

```
## [1] 1 2
```

```
# data frame preserves dimension while extracting a row but not a column
dim(mydf1[1,])
```

```
## [1] 1 7
```

```
dim(mydf1[,1])
```

```
## NULL
```

```
dim(mydf1[,1,drop=FALSE])   # use drop = FALSE to keep a column vector
```

```
## [1] 2 1
```

```
mydf1[,1,drop=FALSE]
```

```
##    c1
## r1  1
## r2  2
```

```
mydf1[, c('c1','num','logi')]
```

```
##    c1 num  logi
## r1  1   1  TRUE
## r2  2   2 FALSE
```

```
class(mydf1[, c('c1','num','logi')])
```

```
## [1] "data.frame"
```

```
# extraction by column name with "$" symbol:  df$varname
mydf1$c1
```

```
## [1] 1 2
```

```
dim(mydf1$c1)
```

```
## NULL
```

```
# one can use quote ' ' or " " as well
mydf1$'c1'
```

```
## [1] 1 2
```

```
# similarly, extraction by column name with [[ ]]: df[['varname']]
mydf1[['c1']]
```

```
## [1] 1 2
```

```
dim(mydf1[['c1']])
```

```
## NULL
```

```
# or by index
mydf1[[1]]
```

```
## [1] 1 2
```

```
# [[ ]] method is useful when passing a variable name as a string
set_to_na <- function(df, var) {
               df[[var]] <- NA
               df
             }
mydf1
```

```
##    c1 c2 c3 num fac1  logi fac2
## r1  1  3  5   1    a  TRUE    1
## r2  2  4  6   2  abc FALSE    a
```

```
mydf2 <- set_to_na(mydf1, "c2")
mydf2
```

```
##    c1 c2 c3 num fac1  logi fac2
## r1  1 NA  5   1    a  TRUE    1
## r2  2 NA  6   2  abc FALSE    a
```

```
# add a variable
mydf1$newvar <- c(4, 4)
mydf1
```

```
##    c1 c2 c3 num fac1  logi fac2 newvar
## r1  1  3  5   1    a  TRUE    1      4
## r2  2  4  6   2  abc FALSE    a      4
```

```
mydf1$newvar2 <- mydf1$c2 + mydf1$c3
mydf1
```

```
##    c1 c2 c3 num fac1  logi fac2 newvar newvar2
## r1  1  3  5   1    a  TRUE    1      4       8
## r2  2  4  6   2  abc FALSE    a      4      10
```

`apply()` may not work well with data frames since data frames are not exactly matrices. We can use simplified apply `sapply()` or list apply `lapply()` instead.

```
mydf1
```

```
##    c1 c2 c3 num fac1  logi fac2 newvar newvar2
## r1  1  3  5   1    a  TRUE    1      4       8
## r2  2  4  6   2  abc FALSE    a      4      10
```

```
# sapply()
idx_num <- sapply(mydf1, is.numeric)
idx_num
```

```
##      c1      c2      c3     num    fac1    logi    fac2  newvar newvar2
##    TRUE    TRUE    TRUE    TRUE   FALSE   FALSE   FALSE    TRUE    TRUE
```

```
apply(mydf1[,idx_num], 2, mean)
```

```
##      c1      c2      c3     num  newvar newvar2
##     1.5     3.5     5.5     1.5     4.0     9.0
```

```
sapply(mydf1[,idx_num], mean)
```

```
##      c1      c2      c3     num  newvar newvar2
##     1.5     3.5     5.5     1.5     4.0     9.0
```

```
# lapply()
idx_num2 <- unlist(lapply(mydf1, is.numeric))
idx_num2
```

```
##      c1      c2      c3     num    fac1    logi    fac2  newvar newvar2
##    TRUE    TRUE    TRUE    TRUE   FALSE   FALSE   FALSE    TRUE    TRUE
```

```
unlist(lapply(mydf1[,idx_num2], mean))
```

```
##      c1      c2      c3     num  newvar newvar2
##     1.5     3.5     5.5     1.5     4.0     9.0
```

2.2.5 List

A list is an ordered collection of (possibly unrelated) objects. The objects in a list are referenced by [[1]], [[2]], ..., or [['var1']], [['var2']], ... etc.

```
mylist1 <- list(v1 = c(1,2,3),
             v2 = c("a","b"),
             v3 = factor(c("blue","red","orange","yellow")),
             v4 = data.frame( u1 = c(1:3), u2 = c("p","q","r"))
             )
mylist1
```

```
## $v1
## [1] 1 2 3
##
## $v2
## [1] "a" "b"
##
## $v3
## [1] blue   red    orange yellow
## Levels: blue orange red yellow
##
## $v4
##   u1 u2
## 1  1  p
## 2  2  q
## 3  3  r
```

```
# extraction
mylist1[[1]]
```

```
## [1] 1 2 3
```

```
mylist1[["v2"]]
```

```
## [1] "a" "b"
```

```
mylist1$v3
```

```
## [1] blue   red    orange yellow
## Levels: blue orange red yellow
```

```
mylist1$v4$u2
```

```
## [1] p q r
## Levels: p q r
```

```
# assignment
mylist1$v5 <- c("a",NA)
mylist1$v5
```

```
## [1] "a" NA
```

```
# a list can be nested
mylist1$v6 <- list(y1 = c(2,9), y2 = c(0,0,0,1))
mylist1$v6
```

```
## $y1
## [1] 2 9
##
## $y2
## [1] 0 0 0 1
```

lapply() is very versatile since the items in a list can be completely unrelated.

```
unlist(lapply(mylist1, class))
```

```
##           v1           v2           v3           v4           v5
##    "numeric"  "character"     "factor" "data.frame"  "character"
##           v6
##       "list"
```

```
unlist(lapply(mylist1, attributes))  # some variables have attributes
```

```
##    v3.levels1    v3.levels2    v3.levels3    v3.levels4      v3.class
##        "blue"      "orange"         "red"      "yellow"      "factor"
##     v4.names1     v4.names2 v4.row.names1 v4.row.names2 v4.row.names3
##          "u1"          "u2"           "1"           "2"           "3"
##      v4.class     v6.names1     v6.names2
##  "data.frame"          "y1"          "y2"
```

```
lapply(mylist1, function(x) {
                  if (is.numeric(x)) return(summary(x))
                  if (is.character(x)) return(x)
                  if (is.factor(x)) return(table(x))
                  if (is.data.frame(x)) return(head(x))
                  if (is.list(x)) return(unlist(lapply(x,class)))
                }
)
```

```
## $v1
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##     1.0     1.5     2.0     2.0     2.5     3.0
##
## $v2
## [1] "a" "b"
```

Table 2.1: Basic R Operators

Operation	Description
x + y	Addition
x - y	Subtraction
x * y	Multiplication
x / y	Division
x ^ y	Exponentiation
x %% y	Modular arithmatic
x %/% y	Integer division
x == y	Test for equality
x <= y	Test for less than or equal to
x >= y	Test for greater than or equal to
x && y	Boolean AND for scalars
x || y	Boolean OR for scalers
x & y	Boolean AND for vectors
x | y	Boolean OR for vectors
!x	Boolean negation

```
##
## $v3
## x
##   blue orange    red yellow
##      1      1      1      1
##
## $v4
##   u1 u2
## 1  1  p
## 2  2  q
## 3  3  r
##
## $v5
## [1] "a" NA
##
## $v6
##        y1        y2
## "numeric" "numeric"
```

2.3 Programming

2.3.1 Operator

source: (Matloff, 2011)

2.3.2 If else

```
if (1 > 0) {
  print("result: if")
} else {
```

```
  print("result: else")
}
```

```
## [1] "result: if"
```

```
# {} brackets can be used to combine multiple expressions
# They can be skipped for a single-expression if-else statement.
if (1 > 2) print("result: if")  else print("result: else")
```

```
## [1] "result: else"
```

```
ifelse(c(1,2,3) > 2, 1, -1)  # return 1 if TRUE and -1 if else
```

```
## [1] -1 -1  1
```

```
Sys.time()
```

```
## [1] "2017-04-08 15:58:55 CDT"
```

```
time <- Sys.time()
hour <- as.integer(substr(time, 12,13))
# sequential if-else statements
if (hour > 8 & hour < 12) {
    print("morning")
} else if (hour < 18) {
    print("afternoon")
} else {
    print("private time")
}
```

```
## [1] "afternoon"
```

2.3.3 Loop

```
for (i in 1:3) {
  print(i)
}
```

```
## [1] 1
## [1] 2
## [1] 3
```

```
for (i in c(1,3,5)) print(i)
```

```
## [1] 1
## [1] 3
## [1] 5
```

```
i <- 1
while (i < 5) {
  print(i)
  i <- i + 1
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
```

2.3.4 Function

We can avoid repeating ourselves with writing similar lines of codes if we turn them into a function. Functions contain a series of tasks that can be applied to varying objects such as different vectors, matrices, characters, data frames, lists, and functions.

A function consists of input arguments, tasks (R expressions), and output as an object (e.g. a vector, matrix, character, data frame, list, or function etc.). It can be named or remain anonymous (typically used inside a function like `lapply()`).

```
##  name_to_be_assigned  <- function(input args) {
##                           tasks
##                         }
```

The output of the function, aside from those that are printed, saved, or exported, is the very last task (expression). If variable `result` is created inside the function, having `result` at the very end will return this item as an output. When multiple objects are created, it is often convenient to return those as a *list*. Use `return()` to return a specific item in the middle of the function and skip the rest of the evaluations. For checking errors and halting evaluations, use `stop()` or `stopifnot()`.

```
myfun1 <- function() print("Hello world")  # just returning "Hello world"
myfun1()
```

```
## [1] "Hello world"
```

```
myfun2 <- function(var) var^2
myfun2(var = 3)
```

```
## [1] 9
```

```
myfun3 <- function(var = 1)  ifelse(var>0, log(var), var)
myfun3()   # default argument is var = 1
```

```
## [1] 0
```

```
myfun3(2)
```

```
## [1] 0.6931472
```

```
myfun4 <- function(x1, x2) {
            if (!is.numeric(x1) | !is.numeric(x2)) stop('demo of error: numeric args needed')
            x1*x2
          }
# try(myfun4(1, "a"))
myfun4(4, 3)
```

```
## [1] 12
```

2.3.5 Environment

A function, formally known as a *closure*, consists of its arguments (called *formals*), a body, and an *environment*. An *environment* is a collection of existing objects at the time when the function is created. Functions created at the *top level* have `.GlobalEnv` as their environments (R may refer to it as `R_GlobalEnv` as well).

```
environment()  # .GlobalEnv  (or R_GlobalEnv) is the top-level environment
```

```
## <environment: R_GlobalEnv>
```

```
f1 <- function(arg1) environment()
formals(f1)  # arguments of f1()
```

```
## $arg1
```

```
body(f1)     # body of f1()
```

```
## environment()
```

```
environment(f1)  # environment of f1(), which is .GlobalEnv
```

```
## <environment: R_GlobalEnv>
```

```
f1()  # inside f1 has its own enviornment
```

```
## <environment: 0x7fd1cbb13298>
```

A function can access to the objects in its environment (i.e., *global* to the function) and those defined inside (i.e., *local* to the function) and generally cannot overwrite the global objects. It allows for using common names such as "x1", "var1" etc. defined inside functions, but those objects are only accessible within the function.

```
a <- NULL  # object named "a" in .GlobalEnv
f2 <- function() {
        a <- 1  # object named "a" in an environment inside f2
        print(a)
        environment()
      }
f2()  #  one instance creating an environment
```

```
## [1] 1
```

```
## <environment: 0x7fd1cb9f41b8>
```

```
f2()  #  another instance creating another environment
```

```
## [1] 1
```

```
## <environment: 0x7fd1cc23eb88>
```

```
a  # stays NULL
```

```
## NULL
```

```
ls()  # ls() shows all objects of an environment (here .GlobablEnv)
```

```
##  [1] "a"           "ans1"        "df"          "df1"
##  [5] "df2"         "f1"          "f2"          "factor1"
##  [9] "factor2"     "factor3"     "hour"        "i"
## [13] "idx_num"     "idx_num2"    "mat1"        "matrix1"
## [17] "matrix2"     "matrix3"     "mydf1"       "mydf2"
## [21] "myfun1"      "myfun2"      "myfun3"      "myfun4"
## [25] "mylist1"     "mymat1"      "set_to_na"   "tbl_operator"
## [29] "time"        "vector1"     "vector2"     "vector3"
## [33] "vector4"
```

```
rm(list = ls())  # rm() removes items of an  environment (here .GlobablEnv)
ls()  # all gone in GlobalEnv
```

```
## character(0)
```

Using global assignment `<<-` operator, one can bend this general rule of not affecting global objects. This can be useful when it is desirable to make certain objects accessible across multiple functions without explicitly passing them through arguments.

```
a <- NULL
b <- NULL
f1 <- function() {
        a <<- 1  # global assignment
        # another way to assign to GlobalEnv
        assign("b", 2, envir = .GlobalEnv)
      }
f1()
a
```

```
## [1] 1
```

```
b
```

```
## [1] 2
```

```
a <- 2
f2 <- function() {
        # Since there is no "a" local to f2, R looks for "a"
        # in a parent environment, or .GlobalEnv
        print(a)

        # g() assigns a number to "a" in g()'s environment
        g <- function() a <<- 5

        a <- 0  #  object called "a" local to f2
        print(a)

        # g() updates only the local "a" to f2(), but not "a" in GlobalEnv
        # R's scope hierarchy starts from local to its environment
        g()
        print(a)
      }
a <- 3

# the first "a" is in .GlobalEnv when f2() is called
# the second "a" is local to an instace of f2()
# the third "a" is the updated version of the local "a" by g()
f2()
```

```
## [1] 3
## [1] 0
## [1] 5
```

```
a   # object "a" in GlobalEnv: unchanged by g()
```

```
## [1] 3
```

It is convenient to use `<<-` if you are sure about which object to overwrite. Otherwise, the use of `<<-` should be avoided.

2.3.6 Debugging

`browser()` and `traceback()` are common debugging tools. A debugging session starts where `browser()` is inserted and allows for a line by line execution onward. Putting `browser()` inside a loop or function is useful because it allows for accessing the objects at a particular moment of execution in its environment. After

Table 2.2: Common R Statistical Distribution Functions

Distribution	Density_pmf	cdf	Quantiles	Random_draw
Normal	dnorm( )	pnorm( )	qnorm( )	rnorm( )
Chi square	dchisq( )	pchisq( )	qchisq( )	rchisq( )
Binomial	dbinom( )	pbinom( )	qbinom( )	rbinom()

an error, executing `traceback()` shows at which process the error occurred. Other tools include `debug()`, `debugger()`, and `stopifnot()`.

2.3.7 Stat func.

source: (Matloff, 2011)

2.3.8 String func.

R has built-in string manipulation functions. They are commonly used for;

- detecting a certain pattern in a vector (`grep()` returning a location index vector, `grepl()` returning a logical vector)
- replacing a certain pattern with another (`gsub()`)
- counting the length of a string (`nchar()`)
- concatenating characters and numbers as a string (`paste()`, `paste0()`, `sprintf()`)
- extracting a segment of a string by character position range (`substr()`)
- splitting a string with a particular pattern (`strsplit()`)
- finding a character position of a pattern in a string (`regexpr()`)

```
oasis <- c("Liam Gallagher", "Noel Gallagher", "Paul Arthurs", "Paul McGuigan", "Tony McCarroll")
grep(pattern = "Paul", oasis)
```

```
## [1] 3 4
```

```
grepl(pattern = "Gall", oasis)
```

```
## [1]  TRUE  TRUE FALSE FALSE FALSE
```

```
gsub("Gallagher", "Gallag.", oasis)
```

```
## [1] "Liam Gallag."   "Noel Gallag."   "Paul Arthurs"   "Paul McGuigan"
## [5] "Tony McCarroll"
```

```
nchar(oasis)
```

```
## [1] 14 14 12 13 14
```

```
paste(oasis)
```

```
## [1] "Liam Gallagher" "Noel Gallagher" "Paul Arthurs"   "Paul McGuigan"
## [5] "Tony McCarroll"
```

```
paste(oasis, collapse=", ")
```

```
## [1] "Liam Gallagher, Noel Gallagher, Paul Arthurs, Paul McGuigan, Tony McCarroll"
```

```
sprintf("%s from %d to %d", "Oasis", 1991, 2009)
```

```
## [1] "Oasis from 1991 to 2009"
```

```
substr(oasis, 1, 6)
```

```
## [1] "Liam G" "Noel G" "Paul A" "Paul M" "Tony M"
```

```
strsplit(oasis, split=" ")  # split by a blank space
```

```
## [[1]]
## [1] "Liam"      "Gallagher"
##
## [[2]]
## [1] "Noel"      "Gallagher"
##
## [[3]]
## [1] "Paul"    "Arthurs"
##
## [[4]]
## [1] "Paul"     "McGuigan"
##
## [[5]]
## [1] "Tony"      "McCarroll"
```

```
regexpr("ll", oasis[1])[1]
```

```
## [1] 8
```

Common regular expressions used in R include;

- `"[char]"` (any string containing either “c”, “h”, “a”, or “r”)
- `"a.c"` (any string containing “a” followed by any letter followed by “c”)
- `"\\."` (any string containing symbol “.”).

```
grepl("[is]", oasis)
```

```
## [1]  TRUE FALSE  TRUE  TRUE FALSE
```

```
grepl("P..l", oasis)
```

```
## [1] FALSE FALSE  TRUE  TRUE FALSE
```

```
grepl("\\.", c("Liam", "Noel", "Paul A.", "Paul M.", "Tony"))
```

```
## [1] FALSE FALSE  TRUE  TRUE FALSE
```

2.3.9 Set func.

The functions for common set operations include `union()`, `intersect()`, `setdiff()`, and `setequal()`. The most commonly used function is `%in%` operator; `X %in% Y` returns a logical vector indicating whether an each element of `X` is a member of `Y`.

```
c(1,2,3,4,5) %in% c(3,2,5)
c("a","b","t","s") %in% c("t","a","a")
```

2.4 Housekeeping

2.4.1 Working directory

`getwd()` returns the current working directly. `setwd(new_directory)` sets a specified working directory.

2.4.2 R session

`sessionInfo()` shows the current session information. In RStudio, `.rs.restartR()` restarts a session.

2.4.3 Save & load

R objects can be saved and loaded by `save(object1, object2, ..., file="file_name.RData")` and `load(file="file_name.RData")`. A `ggplot` object can be save by `ggsave("file_name.png")`.

2.4.4 Input & Output

A common method to read and write data files is `read.csv("file_name.csv")` and `write.csv(data_frame, file = "file_name.csv")`. `scan()` is a more general function to read data files and interact with user keyboard inputs. `file()` is also a general functions to read data through *connections*, which refer to R's mechanism for various I/O operations. `dir()` returns the file names in your working directory.

A useful function is `cat()`, which can print a cleaner output to the screen, compared to `print()`.

```
print("example")
```

```
## [1] "example"
```

```
cat("example\n")  # end with \n
```

```
## example
```

```
cat("some string", c(1:4), "more string\n")
```

```
## some string 1 2 3 4 more string
```

```
cat("some string", c(1:4), "more string\n", sep="_")
```

```
## some string_1_2_3_4_more string
```

2.4.5 Updating

R needs regular updates for R distribution, individual R packages, and RStudio. Generally, updating once or twice a year would suffice. For updating RStudio, go to `Help` and then `Check for Updates`. Also, RStudio also makes it easy to update packages; go to `Tools` and the `Check for Package Updates`. Do these updates when you have time or you know that you need to update a particular package; updating R and R packages can be trickier than it seems.

```
# check R version
getRversion()
```

```
## [1] '3.3.3'
```

```
version
```

```
##                _
## platform       x86_64-apple-darwin13.4.0
## arch           x86_64
## os             darwin13.4.0
## system         x86_64, darwin13.4.0
## status
## major          3
## minor          3.3
## year           2017
## month          03
## day            06
## svn rev        72310
## language       R
## version.string R version 3.3.3 (2017-03-06)
## nickname       Another Canoe
```

```
# check installed packages
## installed.packages()

# list all packages where an update is available
## old.packages()

# update all available packages of installed packages
## update.packages()

# update, without prompt
## update.packages(ask = FALSE)
```

For windows users, one can automate the process using `installr` package.

```
## --- execute the following ---
## install.packages("installr") # install
## setInternet2(TRUE) # only for R versions older than 3.3.0
## installr::updateR() # updating R.
```

Sometimes, you can accidentally corrupt sample datasets that come with packages. To restore the original datasets, you have to remove the package by `remove.packages()` and then install it again. Use `class()`, `attributes()`, and `str()` to check for any unrecognized attributes attached to the dataset. Also, if you suspect that you have accidentally corrupted R itself, you should re-install the R distribution.

Chapter 3

Piecemeal Topics

Workshop materials will be added here.

3.1 Unusual Deaths in Mexico

2017-04-08: *VERY Preliminary!*

Materials

This is a practice session of dplyr and ggplot2 using a case study related to tidyr package.

The case is about investigating the causes of death in Mexico that have unusual temporal patterns within a day. The data on mortalities in 2008 has the following pattern by hour;

Do you find anything unusual or unexpected? The figure shows several peaks within a day, indicating some increased risks of deaths in certain times of the day. What would be generating these patterns?

Wickham, the author, finds;

> The causes of [unusual] death fall into three main groups: murder, drowning, and transportation related. Murder is more common at night, drowning in the afternoon, and transportation related deaths during commute times (Wickham, 2014).

We will use two datasets: `deaths` containing the timing and coded causes of deaths and `codes` containing the lookup table for the coded causes.

The dataset `deaths` has over 53,000 records (rows), so we use `head()` to look at the first several rows.

```
# "deaths08b" is a renamed dataset with easier-to-read column names
head(deaths08b)
```

```
##   Year of Death (yod) Month of Death (mod) Day of Death (dod)
## 1                2008                    1                  1
## 2                2008                    1                  1
## 3                2008                    1                  1
## 4                2008                    1                  1
## 5                2008                    1                  1
## 6                2008                    1                  1
##   Hour of Death (hod) Cause of Death (cod)
## 1                   1                 B20
## 2                   1                 B22
```

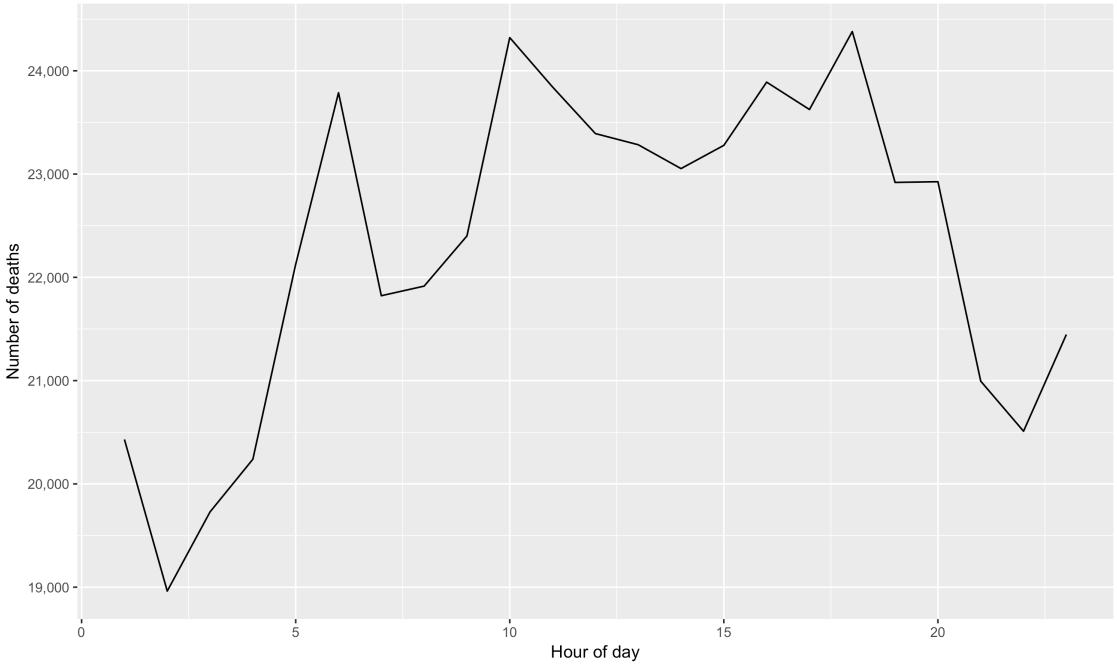


Figure 3.1: Temporal pattern of all causes of death

```
## 3                1                C18
## 4                1                C34
## 5                1                C50
## 6                1                C50
```

The dataset `codes` has 1851 records.

Show 10 entries Search:

	Cause of Death (cod)	Disease
1	A00	Cholera
2	A01	Typhoid and paratyphoid fevers
3	A02	Other salmonella infections
4	A03	Shigellosis
5	A04	Other bacterial intestinal infections
6	A05	Other bacterial foodborne intoxications, not elsewhere classified
7	A06	Amebiasis
8	A07	Other protozoal intestinal diseases
9	A08	Viral and other specified intestinal infections
10	A09	Diarrhea and gastroenteritis of infectious origin

Showing 1 to 10 of 1,851 entries Previous 1 2 3 4 5 … 186 Next

This table is generated by DT and webshot packages. In the search box, you can type in key words like "bacteria", "nutrition", and "fever", as well as "assault" and "exposure" to see what items are in the data.

We will reproduce this case study and practice using functions of `dplyr` and `ggplot2`.

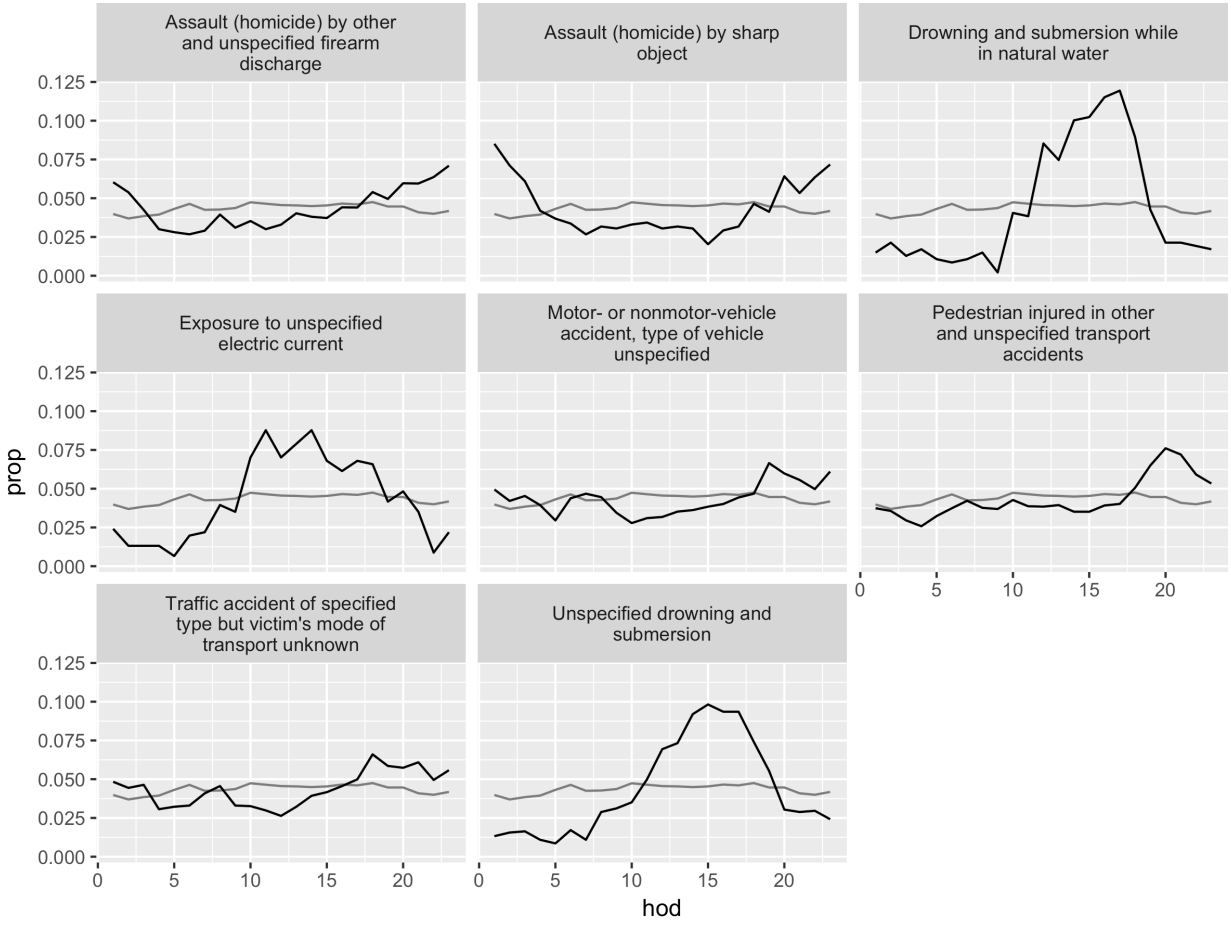


Figure 3.2: Causes of death with unusual temporal courses. Overall hourly death rate shown in grey. Causes of death with more than 350 deaths over a year.

Arts & Crafts

Let's recap key ingredients of dplyr and ggplot2 from the introduction in Section 1.

The six important functions in `dplyr` are:

- `filter()`: extracts rows (e.g., observations) of a data frame. We put logical vectors in its arguments.
- `select()`: extracts columns (e.g., variables) of a data frame. We put column names in its arguments.
- `arrange()`: orders rows of a data frame. We put column names in its arguments.
- `summarise()`: collapses a data frame into summary statistics. We put **summary functions** (e.g., statistics functions) using column names in its arguments.
- `mutate()`: creates new variables and adds them to the existing columns. We put **window functions** (e.g., transforming operations) using column names in its arguments.
- `group_by()`: assigns rows into groups within a data frame. We put column names in its arguments.

We use piping operator `%>%` (read as *then*) to translate a sentence of sequential instructions. For example, take dataset `deaths08`, `%>%` (*then*) group the data by month of death, and `%>%` (*then*) summarise the grouped data for the number of observations.

```
deaths08 %>%
  group_by(mod) %>%   # mod: month of death
  summarise( nobs = n() )  # n(): a dplyr funciton to count rows
```

```
## # A tibble: 12 × 2
##      mod  nobs
##    <int> <int>
## 1      1 49002
## 2      2 41685
## 3      3 44433
## 4      4 39845
## 5      5 41710
## 6      6 38592
## 7      7 40198
## 8      8 40297
## 9      9 39481
## 10    10 41671
## 11    11 43341
## 12    12 42265
```

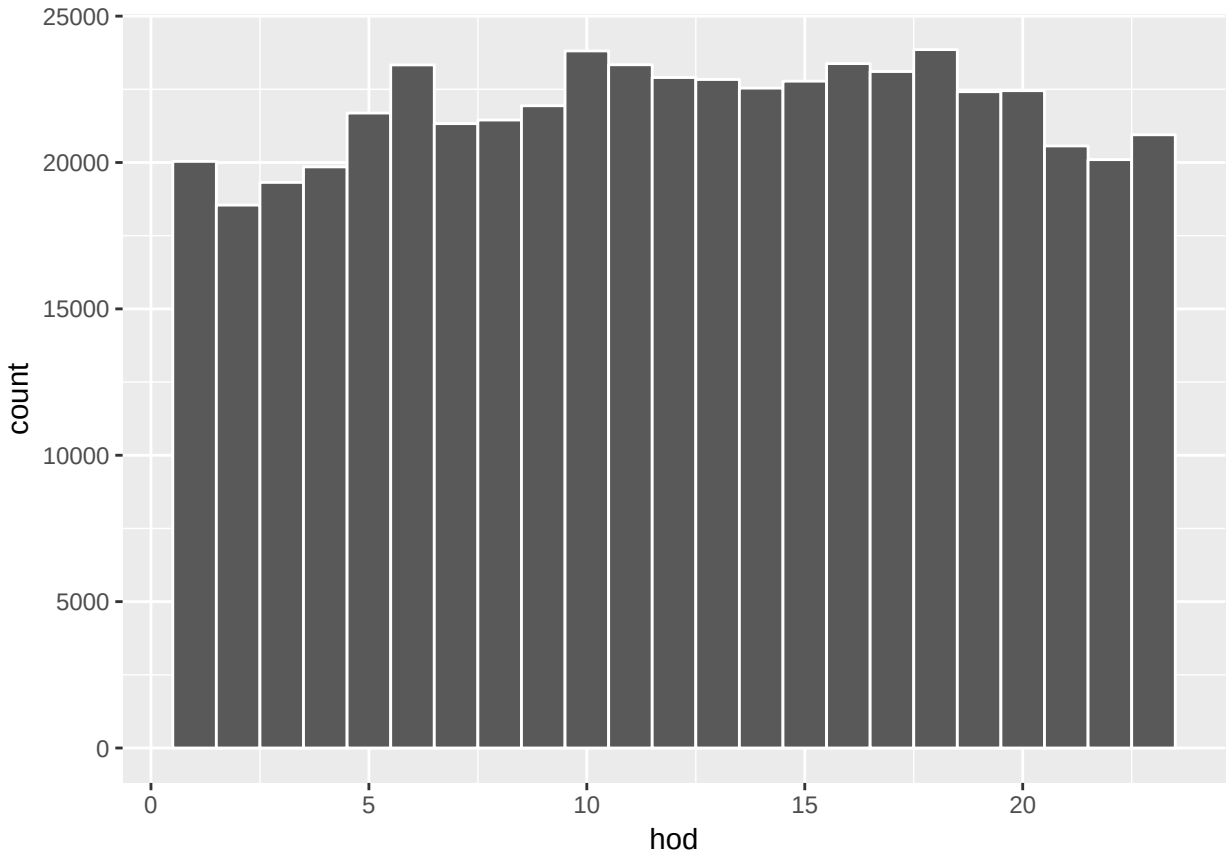
The graphics with `ggplot2` consist of three components:

- **data**: a data frame e.g., the first argument in `ggplot(data, ...)`.
- **aes**: specifications for x-y variables, as well as variables to differentiate **geom** objects by color , shape, or size. e.g., `aes(x = var_x, y = var_y, shape = var_z)`
- **geom**: geometric objects such as points, lines, bars, etc. e.g., `geom_point()`, `geom_line()`, `geom_histogram()`

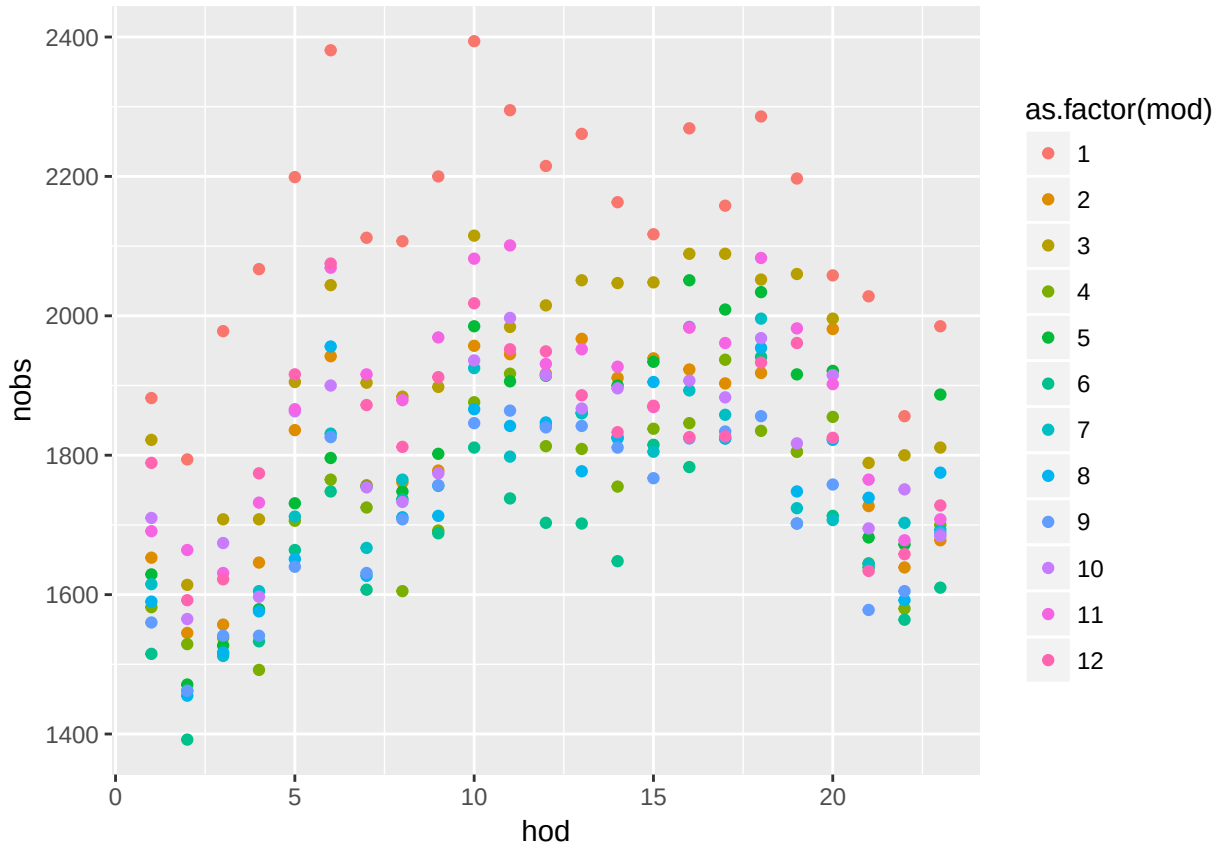
We specify **data** and **aes** in `ggplot()` and then add **geom** objects followed by `+` symbol (read as *add a layer of*); e.g., `ggplot(data = dataset, mapping = aes(x = ...)) + geom_point()`. The order of layers added by `+` symbol is generally interchangeable.

Combined with `%>%` operator, we can think of the code as a sentence. For example, take dataset `deaths08`, `%>%` (*then*) plot with `gglpot()` with `aes()` featuring hour of day on the x-axis, `+` (*and add a player of*) geom object `geom_histogram()`.

```
#  a histogram version of the line-graph for the total number of deaths above
deaths08 %>%
  ggplot(aes(x = hod)) + geom_histogram(binwidth = 1, color = "white")
```




```
# a summary by month of day and hour of day.
# e.g, Jan-1am, ..,Jan-12pm, Feb-1am,..., Feb-12pm, ...
n_month_hour <- deaths08 %>%
  group_by(mod, hod) %>%
  summarise( nobs = n() )

n_month_hour %>%
  ggplot(aes(x = hod, y = nobs, color = as.factor(mod))) +
  geom_point()
```




```
# "last_plot() + " allows for adding more layers to the previous plot
last_plot() + geom_line()
```

Exercise

Now it is your turn. The exercise is to reproduce the above results for the unusual causes of deaths.

1. Download materials: case study paper and case study data
2. Set working directly: `setwd(your_directory)`
3. Load libraries: `library(dplyr)`, `library(ggplot2)`, `library(MASS)` (used for fitting data by robust regression)

- Note 1: There is a minor error in the case study that the author accidentally kept several records of data from years other than 2008. This virtually has no effect on the results, and we would do the same to reproduce the exact results.
- Note 2: You could look at the codes in the paper for hints. However, the code is written with the functions of plyr package, or the precesssor of `dplyr`. Do not load both `plyr` and `dplyr` libraries in the same R session; they do not seem to have good compatibility. Restart R if you accidentally loaded both.

Part A. Display overall hourly deaths

We will reproduce:

Hints:

- Filter `NA` in the hour of day (hod) variable
- Use `group_by()`, `summarise()`, `n()` to obtain death counts by group
- Use `ggplot() + geom_line()` to produce plot

- Use `+ labs( x = "x lable", y = "y label")` for axis labels
- see help file of `scale_y_continous()` for comma (use `?function_name` for help)

Part B. Count deaths per hour, per disease

We will reproduce:

hod	cod	freq	disease	prop	freq_all	prop_all
8	B16	4	Acute hepatitis B	0.04	21915	0.04
8	E84	3	Cystic fibrosis	0.03	21915	0.04
8	I21	2205	Acute myocardial infarction	0.05	21915	0.04
8	N18	315	Chronic renal failure	0.04	21915	0.04
9	B16	7	Acute hepatitis B	0.07	22401	0.04
9	E84	1	Cystic fibrosis	0.01	22401	0.04
9	I21	2209	Acute myocardial infarction	0.05	22401	0.04
9	N18	333	Chronic renal failure	0.04	22401	0.04
10	B16	10	Acute hepatitis B	0.10	24321	0.05
10	E84	7	Cystic fibrosis	0.07	24321	0.05
10	I21	2434	Acute myocardial infarction	0.05	24321	0.05
10	N18	343	Chronic renal failure	0.04	24321	0.05
11	B16	6	Acute hepatitis B	0.06	23843	0.05
11	E84	3	Cystic fibrosis	0.03	23843	0.05
11	I21	2128	Acute myocardial infarction	0.05	23843	0.05
(a)			(b)	(c)	(d)	

Table 16: A sample of four diseases and four hours from `hod2` data frame.

Panel (a) of the table contains the frequency (i.e. the number of rows) for each combination of hour of day (hod) and cause of death (cod), supplemented by the disease description in panel (b). Panel (c) shows the proportion (prop) of each hod-cod combination in the total deaths by the cod. Panel (d) contains the frequency and proportion (freq_all and prop_all) of the deaths by hour of day.

That is, panel (a) is the raw counts (e.g., frequency) of observations by **each pair** of hour of day (hod) and cause of death (cod), and panel (b) makes it easy to see the cause of death (cod). Panel (c) converts this frequency of *hod-cod pair* into the relative frequency within the total frequency of **cod**, so that we see at which **hour** a disproportinately large number of deaths occurs for **cod**. Panel (d), on the other hand, presents the overall hourly death rates; if every hour has the same probablity of death, we would see prop_all ≈ 0.042 (i.e., 1/24). Here, we see the author's idea of identifying "unusual deaths" by comparing "prop" of each **hod-cod pairs** for the deviation from to "prop_all" (see figure 3.2).

Hints for creating panels (a)

- Use more than one variable in `group_by()`
- Use `summarise()` with `n()` to obtain death counts by group

Hints for creating panels (b)

- Use `left_join()` to add the information from dataset `codes`

Hints for creating panel (c)

- Use `mutate()` with `sum()` on the joined dataset

Hints for creating panels (d)

- Create a new data frame by using `summarise()` on the joined and mutated data frame. (`summarise()` will reduce the dimension of the data frame to its summmary, which is the basis of panel (d). Once the desired summary is created, merge it to the data frame of panels (a)-(c).)

- Before using `summarise()` above, use `group_by()` to specify new grouping

- First create `freq_all` variable via `summarise()` with `n()` and then create `prop_all` variable via `mutate()` with `sum()` (call this data frame `overall_freq`, which will be used again at the very end)

- Use `left_join()` to join panels (a)-(c) and panel (d) (`overall_freq`), which we refer to as `master_hod` data frame.

Hints for extracting the same rows as in the Table 16 above

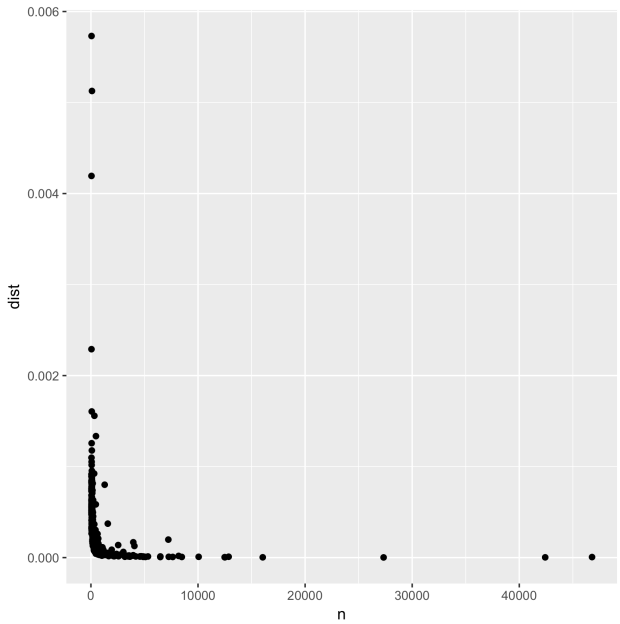
- Create a subset of the `master_hod` data under a new name

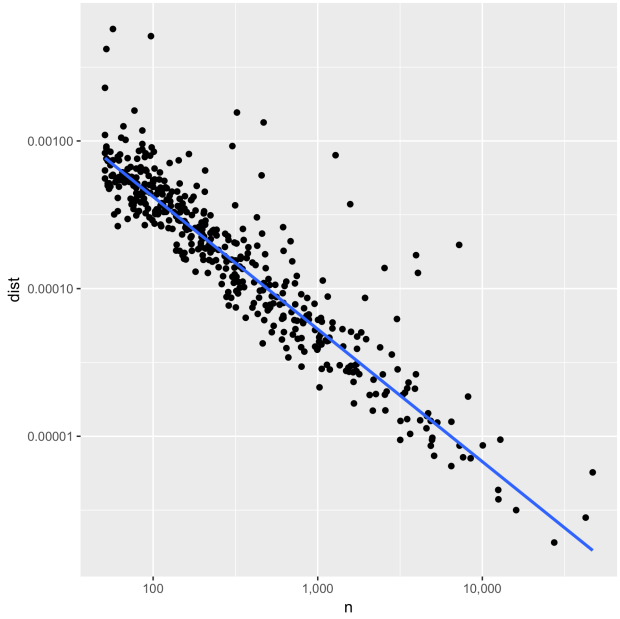
- Use `filter()` to select `cod` being either "I21", "N18", "E84", or "B16" and `hod` being greater or equal to 8 and smaller or less than 11

- Use `select()` to pick columns in a desired order and `arrange()` to sort

Part C. Find outliers

We will reproduce:





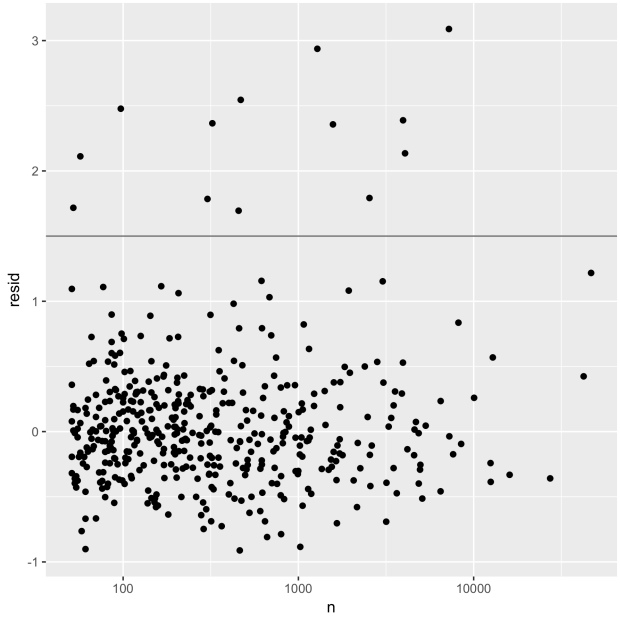
We will create a deviation variable named `dist` by taking the mean of squared differences between `prop` and `prop_all`. The above figures show the the number of observations `n` and this distance measure `dist` by cause of death in the raw scale (left) and in the log scale (right).

Hints

- Use `group_by()` and `summarise()` on the `master_hod` data frame to generate `n` with function `sum()` and `dist` by `mean((prop - prop_all)^2)`
- Filter this summary for `n > 50` and call it `devi_cod` (deviations by cause of death)
- Use `ggplot() + geom_point()` with `data = devi_cod` to produce the raw-scale figure
- Additionally use `scale_x_log10()`, `scale_y_log10()`, and `geom_smooth(method = "rlm", se = FALSE)` to produce the log-scale figure
- See help for `scale_x_log10()` to adjust axis labels (look for "comma")
- Technically speaking, we should change the axis labels to indicate the logarithmic transformation, but we skip it here.
- Let's not worry about reproducing the exact grids as they appear in the paper

Part D. Fit data by a regression and plot residuals

We will reproduce:

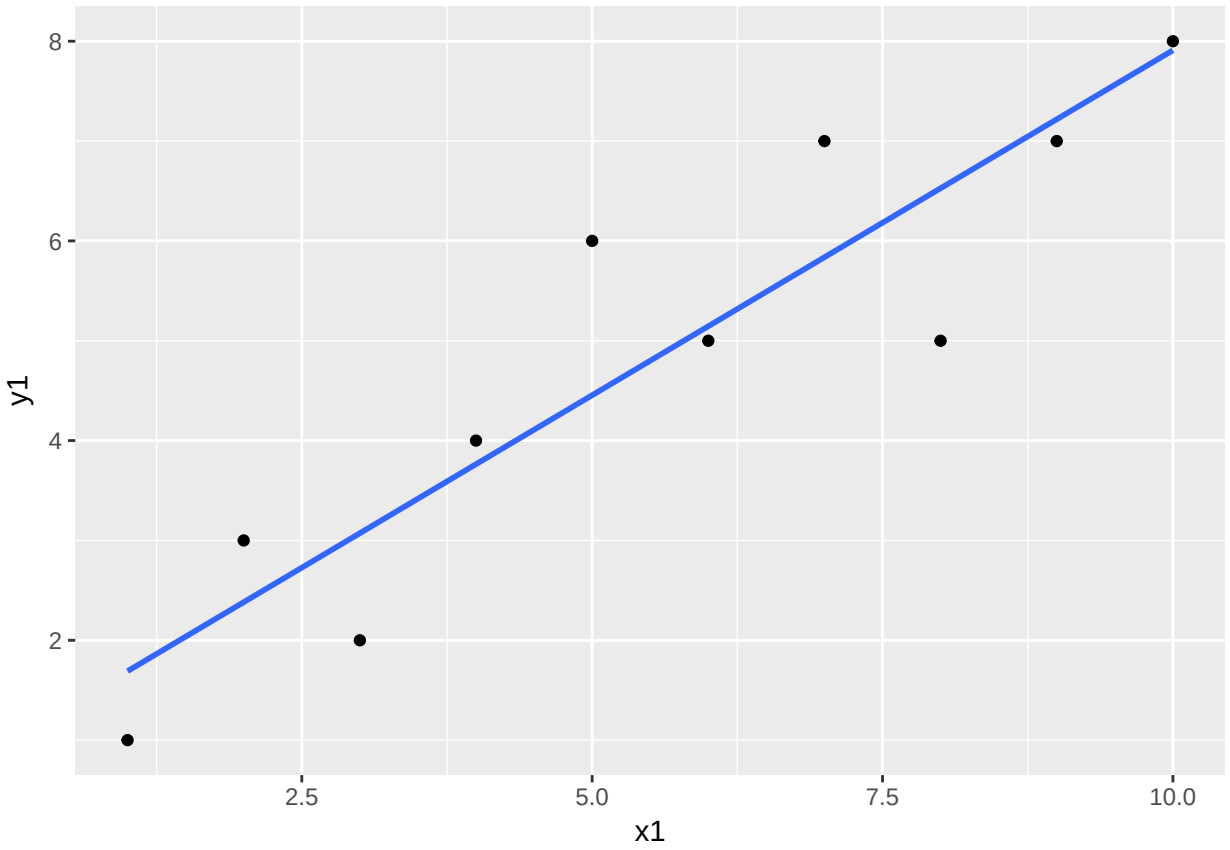


The figure is a plot of the regression residuals `resid` of `log(dist)` on `log(n)`. By visual inspection, the points lying above the holizontal line at `resid=1.5` are considered to be "unusual causes of deaths" by the author.

Here the author used the robust linear model (`rlm()`) regression, but the syntax is mostly the same as that of the standard linear model regression (`lm()`).

Here is an example of regression by `lm()`.

```
df <- data.frame(
  x1 <-  c(1:10),
  y1 <-  c(1,3,2,4,6,5,7,5,7,8)
)

df %>%
  ggplot(aes(x = x1, y = y1)) + geom_point() +
  geom_smooth(method = "lm", se = FALSE)
```



The geom_smooth() is estimating the following linear regression:

$$y1 = intercept + coefficient * x1 + residual$$

The model is estimated by `lm()` as follows;

```
f1 <- lm(formula = y1 ~ x1,  data = df)
```

Now let's see what we get out of the estimation results `f1`.

```
class(f1)    # class "lm"
```

```
## [1] "lm"
```

```
summary(f1)  # summary() knows how to summarise an object of class "lm"
```

```
##
## Call:
## lm(formula = y1 ~ x1, data = df)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -1.52727 -0.57273 -0.02727  0.52273  1.54545
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)   1.0000     0.6924   1.444 0.186656
## x1            0.6909     0.1116   6.192 0.000262 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
##
## Residual standard error: 1.014 on 8 degrees of freedom
## Multiple R-squared:  0.8273, Adjusted R-squared:  0.8058
## F-statistic: 38.34 on 1 and 8 DF,  p-value: 0.0002618
```

```
coefficients(f1)  # coefficient point estimate
```

```
## (Intercept)          x1
##   1.0000000   0.6909091
```

```
vcov(f1)          # coefficient variance-covariance matrix
```

```
##             (Intercept)          x1
## (Intercept)  0.47939394 -0.06848485
## x1          -0.06848485  0.01245179
```

```
predict(f1)       # predicted (fitted) values with the estimated coefficients
```

```
##        1        2        3        4        5        6        7        8
## 1.690909 2.381818 3.072727 3.763636 4.454545 5.145455 5.836364 6.527273
##        9       10
## 7.218182 7.909091
```

```
resid(f1)         # residuals:
```

```
##           1           2           3           4           5           6
## -0.69090909  0.61818182 -1.07272727  0.23636364  1.54545455 -0.14545455
##           7           8           9          10
##  1.16363636 -1.52727273 -0.21818182  0.09090909
```

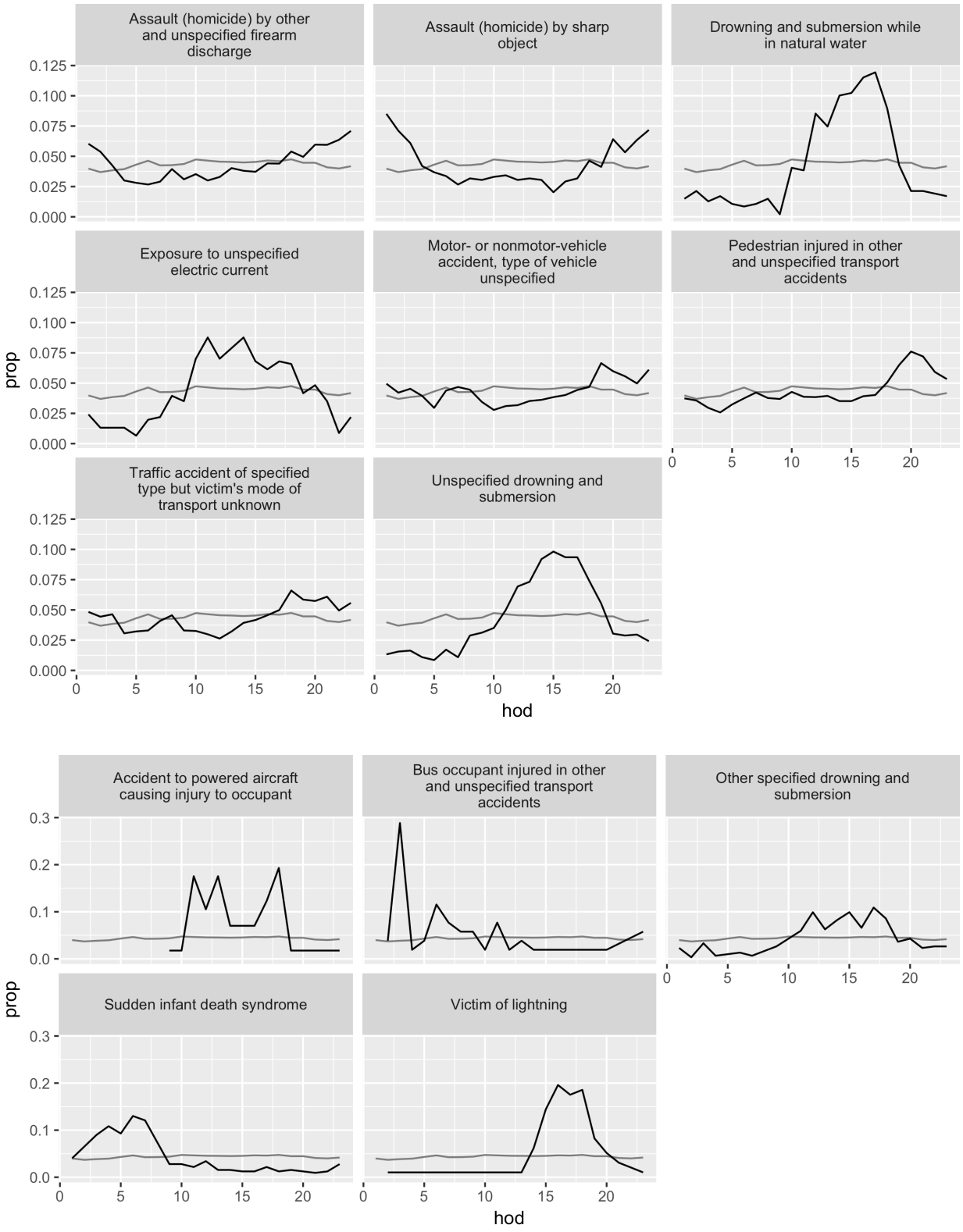
Now let's get back to our exercise and reproduce the figure above.

Hints

- Run a regression by `rlm` with `formula = log(dist) ~ log(n)` and store the residuals in `devi_cod` data. To read more about linear regressions, see the help file of `lm()` (type `?lm`). For adding a column of residuals, you can use assignment `devi_cod$resid <- your_residuals`.
- Plot the residual against log-scale `n`
- Note: Check the dataset `devi_cod` for missing values of `dist` and `n` before running a regression (you should not have missing values in this case). Most regression functions, including `lm()` and `rlm()`, drop any row with missing values from the estimation. This becomes an issue if we want to add a new column containing predicted values or residuals to the original dataset. (When rows containing missing values are dropped, the vector generated by `predict()` or `resid()` will be shorter than the number of rows in the original dataset.)
- Use `ggplot() + geom_point()` structure for the plot
- Add `+ scale_x_log10()` and `+ geom_hline(yintercept = 1.5)`

Part E. Visualise unusual causes of death

We will reproduce:



The first figure is the unusual causes of deaths in `devi_cod` with a relatively large number of deaths (`n > 350`) and the second is that of a relatively small number of deaths (`n <= 350`).

Hints

- Using the cuttoff value `resid > 1.5`, filter `devi_cod` and call it `unusual` data frame. Join `master_hod` and `unusual` data frames. Then create two subsets of data with conditions `n > 350` and `n <= 350`.
- Use `ggplot() + geom_line()` structure with `+ facet_warp(~ disease, ncol = 3)`
- To include the overall hourly proportions of deaths (`prop_all`) representing the average of all causes of deaths in a given hour, add another layer by `geom_line(aes(...), data = overall_freq)` with a local `aes()` argument and a data argument. With the data argument, variables in another data frame can be combined (assuming the axes have the same measurements), and here we use the `overall_freq` data frame from the panel (d) portion of Table 16 above.
- `last_plot() %+% another_data_frame` reproduces a plot of the same structure with a different data frame

The Key

The solution will be presented at the workshop and later posted.

3.2 Upcoming topics

- Statistical inferences with simulations
- Linear regressions

Chapter 4

Resources

Here are more resources for learning R.

Free Books

- Official CRAN R Manual
- Quick R
- The Art of R Programming by Norman Matloff
- ModernDive
- Impatient R
- Simple R by John Verzani
- Introduction to Probability and Statistics Using R By Jay Kerns
- R Wikibook
- Cookbook for R by Winston Chang
- OnePageR
- The R Inferno

Videos

- Coursera's four week course videos
- Workflow example video by Jermey Chacon
- Video on Youtube

Tutorials

- R Tutorial
- R Bootcamp - Jared Knowles
- Step-by-step (sequential) interactive tutorial- Try R
- Another step-by-step interactive tutorial - swirl

With Small Fees: Tutorials from DataCamp

- Cleaning Data in R
- Data Manipulation in R with dplyr
- Data Visualization in R with ggvis
- Data Visualization with ggplot2

Introduction to Coding

- Coding Resources for Beginners by Tori Dykes

Bibliography

Matloff, N. (2011). *The Art of R Programming*.

Wickham, H. (2014). Tidy data. *The Journal of Statistical Software*, 59.

Wilkinson, L. (2005). *The Grammar of Graphics*.