Importing and Manipulating Dates with Lubridate

Bailey Joseph

2017-10-31

```
## You'll need to have the following packages installed and loaded:
library(lubridate) # For dealing with dates
library(ggplot2) # For creating visuals and plots
library(dplyr) # For working with the data
library(readr) # For importing csv data files
```

Introduction and Motivation

We have many ways of writing down and understanding dates in English. I'm sure you know what I mean when I say this draft is being written on October 24th 2017 and that I hope to be finished on 2017-10-31 at 11:59 PM. If I asked you how long I have to finish, you could mentally *subtract* those two dates. This is no problem even though I wrote them in different formats and they're not even really numbers. It's more complicated to do this operation in R because R has no idea how to subtract characters.

We want R to "understand" that a character string representing a date is a particular type of object that can be manipulated with the same methods that humans use, but much faster. This will allow us to greatly expand the scope of the analysis that we can do because we're able to look at changes over time. For example, we could take data from a business and determine which times of the day and which day of the week is the most popular for shopping. We could take data from the NBA and figure out which top players start to produce more as the season progresses.

This post will explain how to turn a column of character values (with a variety of starting formats) into dates and then give some interesting examples of what we can actually do with these date objects.

Converting to Objects of Class "Date"

R's large user base allows us to do a most tasks in multiple different ways. Converting to dates is no different, and we have the choice to use base R or any of a variety of packages. I'll be mostly focusing on the "lubridate" package, but feel free to explore other options if you're so inclined. The Berkeley Stats department has a great guide on how to do many of these same processes using base R. However, in my opinion, lubridate is more intuitive and far easier to use than the base R alternative.

During your time as a data analyst, you'll surely encounter dates in many formats, some much nicer than others. For reference, if you're ever collecting data, it's recommended that you input them in a YYYY-MM-DD format to make your life (and the lives of anyone else who might analyze your data) easier.

The first step in importing dates, just like with most data analysis, is to take a look at your data. Figure out the ordering of the parts of the dates – are they written year month day (2017-10-24), month day year (10-24-2017), or something else? Don't worry about the delimiting character (- and / are common ones, but lubridate is smart enough to figure it out in most reasonable cases) or the way the month is described. Lubridate with handle full month names, abbreviated month names, or the number of the month with no issues.

Be careful about ambiguous formats, such as 08-09-2017, where you can't distinguish between the month and the day. In this case, you can look deeper into your list of dates and see if you can find one that's more clear or examine any information presented with the data in a data dictionary or on the source website to see there are any clues.

Once you've figured this out, you're ready to start importing your dates with lubridate. Let's see how.

Lubridate

Lubridate is a package written by Garrett Grolemund and Hadley Wickham which includes a series of functions used to change a character vector into a date. Order the abbreviations below according to the format in your character vector:

Abbreviation	Object
y	year
m	month
d	day

You should end up with a function of the form `ymd()`, which takes in a character vector.

If you have information more specific than just the day, you can add an underscore and then abbreviations from the following chart:

Abbreviation	Object
h	hour
m	minute
s	second

In this case, you'll end up with a function that looks something like `ymd_hms()`.

Let's try a straightforward example.

```
## Convert this character vector to date objects:
char_days <- c("Oct 24 2017", "Oct 25 2017", "Oct 26 2017", "Oct 27 2017",
          "Oct 28 2017", "Oct 29 2017", "Oct 30 2017")
# First examine the structure -- it looks like we have month, day, year.
fixed_days <- mdy(char_days) # mdy comes from the first chart above
# Let's take a look at our new objects
head(fixed_days, 3)
```

```
## [1] "2017-10-24" "2017-10-25" "2017-10-26"
```

Here are a few important things to notice:

- The input had “Oct” for the month but the output has changed this to a 10.
- The input was in a `mdy` format, but the output is in the recommended YYYY-MM-DD. This will happen with successful parsing *no matter* the format of the input.

Let’s go through some more examples:

```
## Includes time information
dates_with_times <- c("October 25 2017 11:40PM", "January 18 2017 3:45AM")
# Here we have the same month, day, year format,
# but we also have the hour and the minute.
fixed_dates_with_times <- mdy_hm(dates_with_times)
fixed_dates_with_times
```

```
## [1] "2017-10-25 23:40:00 UTC" "2017-01-18 03:45:00 UTC"
```

Now we have “UTC” added to the end of our dates. That’s an abbreviation for the time zone lubridate chooses by default. We didn’t have this before because days of the year don’t depend on time zones directly. We can change the output to specify our own time zone with the optional `tz` argument inside any of the lubridate functions. According to the cran documentation of lubridate, the `tz` argument uses information in your computer becuase R has no internal list of time zones. You can find a full list of commonly accepted time zone arguments at this Wikipedia page.

Let’s update the dates we just made to specify that I’m on the US West Coast. As far as I can tell, there is no time zone that specifically mentions San Francisco, so I’ll use “America/Los_Angeles.” To check the time zone setting for your current computer, you can use the base R function `sys.timezone()`.

```
# Adding the correct time zone:
fixed_time_zones <- mdy_hm(dates_with_times, tz = "America/Los_Angeles")
fixed_time_zones
```

```
## [1] "2017-10-25 23:40:00 PDT" "2017-01-18 03:45:00 PST"
```

Lubridate was even smart enough to distinguish that one of the dates was during daylight savings!

Inconvenient or Uncommon Cases

Okay, so we’ve learned enough to handle most common cases. Up until now, the data has been in a convenient format, but what if your dataset has some digit information left off? For example, let’s examine what lubridate does if the years are in a two digit ambiguous format:

```
## Ambiguous years
dates_without_digits <- c("13/12/89", "9/5/17", "30/10/68", "21/3/69")
# Here we can tell that we have day, month, year,
# but we've left off some information (century and 0s from months and days)
fixed_dates_without_digits <- dmy(dates_without_digits)
fixed_dates_without_digits
```

```
## [1] "1989-12-13" "2017-05-09" "2068-10-30" "1969-03-21"
```

The results may be surprising – two digit years less than or equal to 68 are parsed as being from the 21st century, while numbers 69 or larger are parsed as being from the 20th century. This is a quirk of lubridate, and honestly I don’t agree with that default. In my opinion it’s more likely that you’ll be looking at time data from the mid 20th century than data from the future. Hopefully your datasets will be more well behaved, but if you do need to load 20th century data with two digit years, check out this stack overflow page for a well written solution.

The last important case is when you have a group of dates in different formats. This will commonly come up if you’ve merged together datasets prepared by different people. Lubridate has a function `parse_date_time()` which allows you to handle this with the argument `orders`. The `orders` argument takes in a *character vector* where each element is one of the usual lubridate functions such as “ymd”. Lubridate will use the first element of `orders` to parse the first element of your date vector. This will continue through the length of the two vectors.

The easiest way to show this is with an example:

```
# Two datasets with different formats
format_1 <- c("9/4/98", "10/28/97", "12/26/95") #mdy format
format_2 <- c("17/3/98", "12/7/97") #dmy format
# Combine the vectors into one longer vector
mixed_formats <- c(format_1, format_2)
# Use the parse_date_time function:
not_mixed <- parse_date_time(mixed_formats, # Use the combined dates
              orders =
                c(rep("mdy", length(format_1)), # Uses mdy for first vector's elements
                rep("dmy", length(format_2))), # Uses dmy for second vector's elements
              tz = "America/Los_Angeles")

not_mixed
```

```
## [1] "1998-04-09 PDT" "1997-10-28 PST" "1995-12-26 PST" "1998-03-17 PST"
## [5] "1997-07-12 PDT"
```

One final note for importing dates is that lubridate has no `md()` function to use when your dates have no year information at all. If this happens, here's a stack overflow question which explains how you can use `parse_date_time` to force the parsing anyway:

```
solstices <- c("3/20", "6/21", "9/23", "12/21") # month day format for season solstices
solstices <- parse_date_time(solstices, orders = "%m-%d") # changes to date format
solstices
```

```
## [1] "0000-03-20 UTC" "0000-06-21 UTC" "0000-09-23 UTC" "0000-12-21 UTC"
```

I won't get into the details of the syntax because it's unintuitive and models after the base R way to import dates. This is a degenerate case – dates need a year to be complete. This is why you can see lubridate added a default "missing value" of 0000 to each date, so do be careful with calculations. If you're given data with no years, I recommend doing some data preprocessing to add the correct years.

Armed with lubridate, you should now be able to coerce characters to dates. But what next? I'll show you how you can perform a variety of operations on dates to get useful results.

Operations with Dates

Functions that normally work on numbers work on dates now! Suppose my two best friends and I want to share a joint birthday celebration at the average of our three birthdays. I could probably figure this out on paper, but it will be faster this way:

```
# Import our birthdays using lubridate
birthdays <- mdy(c("10/15/1997", "10/28/1997", "12/26/1997"))
# Use the base R mean() function to get the average
party <- mean(birthdays)
party # This is still an object of class date!
```

```
## [1] "1997-11-12"
```

We're obviously not going to go back in time to 1997 celebrate, so let's extract only the month and day from this average using the lubridate functions `day()` and `month()`. When we use month, the default output will be the month's number (from 1 to 12), but we can specify a more human readable version by using the optional argument `label = TRUE`. This will change the output to an ordered factor with abbreviations for the months of the year as levels.

```
party_month <- month(party, label = TRUE)
party_month
```

```
## [1] Nov
## 12 Levels: Jan < Feb < Mar < Apr < May < Jun < Jul < Aug < Sep < ... < Dec
```

```
party_day <- day(party)
party_day
```

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

It turns out that there are a lot of things we can extract! Try out the following functions:

Function Name	Output	Class of Output
`second()`	Second (defaults to 0)	Numeric
`minute()`	Minute (defaults to 0)	Numeric
`hour()`	Hour (defaults to 0)	Numeric
`day()`	Day of month (1 through 31)	Numeric
`wday()`	Day of week (1 through 7)	Numeric
`yday()`	Day of year (1 through 365	Numeric
`week()`	Week of year (1 through 53)	Numeric

Function Name	Output	Class of Output
`month()`	Month (1 through 12)	Numeric
`year()`	Year	Numeric
`tz()`	Time zone abbreviation	Character

Note: The `wday()` function also has the optional `label = TRUE` argument.

Using Dates to Manipulate Datasets

Let's put these functions to use on a real dataset. The following data comes from the Bureau of Transportation and has information about the date and delays of many US flights. First let's load the data:

```
# Import using readr
flights <- read_csv("../data/flight_delays.csv", col_types = c("cnnnnnnn"))
# Let's examine a slice of 5 rows uniformly distributed through the data
slice(flights, seq(1, length(flights$FL_DATE), length(flights$FL_DATE)/5))
```

```
## # A tibble: 5 x 8
##      FL_DATE CRS_DEP_TIME DEP_TIME DEP_DELAY_NEW CRS_ARR_TIME ARR_TIME
##        <chr>        <dbl>    <dbl>         <dbl>        <dbl>    <dbl>
## 1 2016-11-01         1640     1644             4         1827     1830
## 2 2016-11-07          725      726             1         1050      958
## 3 2016-11-13         2035     2029             0         2223     2201
## 4 2016-11-18         1000     1002             2         1300     1246
## 5 2016-11-25          930      926             0         1200     1146
## # ... with 2 more variables: ARR_DELAY <dbl>, DISTANCE <dbl>
```

Taking a look at the data we've just loaded, we see that FL_DATE tells us the day of the flight as a character in `ymd`, and the scheduled departure hour and minute information is in CRS_DEP_TIME column as a number. Unfortunately, the time zone information isn't present, so I'll choose to leave it as the default: UTC. The source website mentions that all times are correct in their local time zone, so we don't have to worry about any skewing. Let's convert the day into a date object and then use the `hour()` and `minute()` functions to actually set the hour and minute information into the date:

```
# Convert to date
flights$FL_DATE <- ymd(flights$FL_DATE)
# Extract the hour number from CRS_DEP_TIME
flight_hour <- flights$CRS_DEP_TIME %/% 100 # floor division operator
# Extract the minute number from CRS_DEP_TIME
flight_minute <- flights$CRS_DEP_TIME %% 100 # mod division operator
# Add in the hour information
hour(flights$FL_DATE) <- flight_hour
# Add in the minute information
minute(flights$FL_DATE) <- flight_minute
head(flights$FL_DATE)
```

```
## [1] "2016-11-01 16:40:00 UTC" "2016-11-01 16:50:00 UTC"
## [3] "2016-11-01 11:35:00 UTC" "2016-11-01 10:25:00 UTC"
## [5] "2016-11-01 15:55:00 UTC" "2016-11-01 08:06:00 UTC"
```

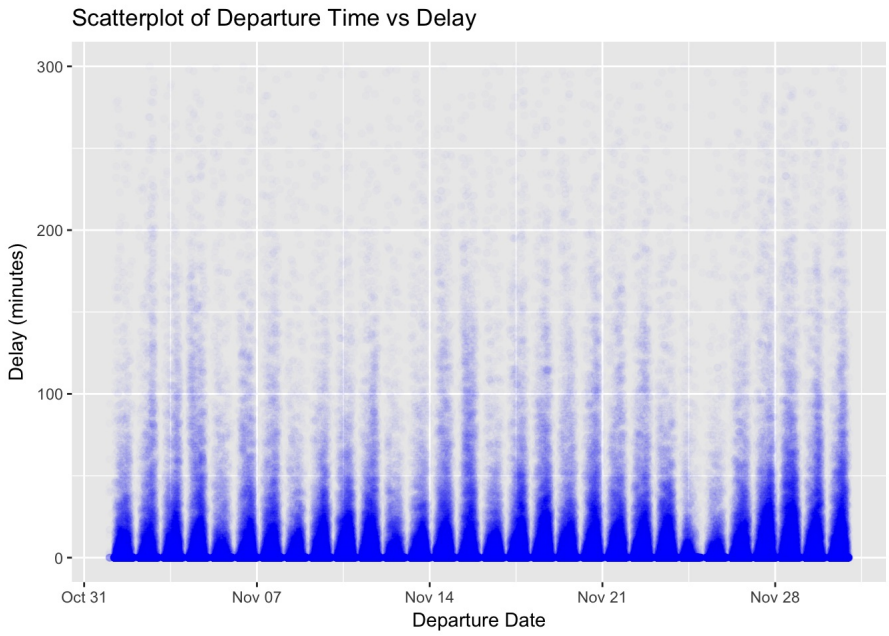
Caution: Much of the rest of this post requires a basic understanding of how to use ggplot2 and dplyr, so feel free to reference guides for ggplot2 and dplyr if you need a refresher or if any step is confusing. Also, unfortunately, dates with times don't play very nicely with dplyr. Throughout this analysis, you'll see me move individually needed date components to other columns and then select off the full date/time column when I use the dplyr function `group_by()`.

Tracking Values with Respect to Time

To make sure that our dates are working as they should, I'll graph a simple plot of delays versus time. In general, I recommend putting time on the x axis so that it's easy to see change over time.

```
gg_delay <- (ggplot(flights, aes(x = FL_DATE, y = DEP_DELAY_NEW)) +
               geom_point(alpha = .01, color = "blue") +
               ylim(0, 300) + # Some points have y > 300, ggplot gives us a warning
               ggtitle("Scatterplot of Departure Time vs Delay") +
               labs(x = "Departure Date", y = "Delay (minutes)"))
gg_delay
```

```
## Warning: Removed 1938 rows containing missing values (geom_point).
```



This graph shows us that delays tend to spike during the middle of the day. It might be visually misleading because of the sheer number of points on the graph. To combat this, I chose to make each point very light to distinguish between what delay lengths are likely instead of just showing the range of possible values. Also, I restricted the scale to have a max of 5 hours, thinking that everything else is an outlier/rescheduled flight and including them would only take away from our ability to see the overall trend. It appears that delays spike during the middle of the day – flights early in the morning and late at night seem to be more likely to get off on time. There doesn't seem to be much difference throughout the month, except that there were fewer flights right around Thanksgiving, which was on November 24th.

Let's see if we can confirm that delays are higher in the middle of the day. Start by adding a column `period` which specifies early (times earlier than noon), middle (times between noon and 8:00 PM), or late (times after 8:00 PM). Note these time intervals aren't equal in length. I chose them to reflect what I'd view as being in the beginning, middle, or end of the day. *If this seems too arbitrary for you*, simply pay attention to the process. You can use a similar process to add a categorical variable based on time intervals to a dataset of your choosing.

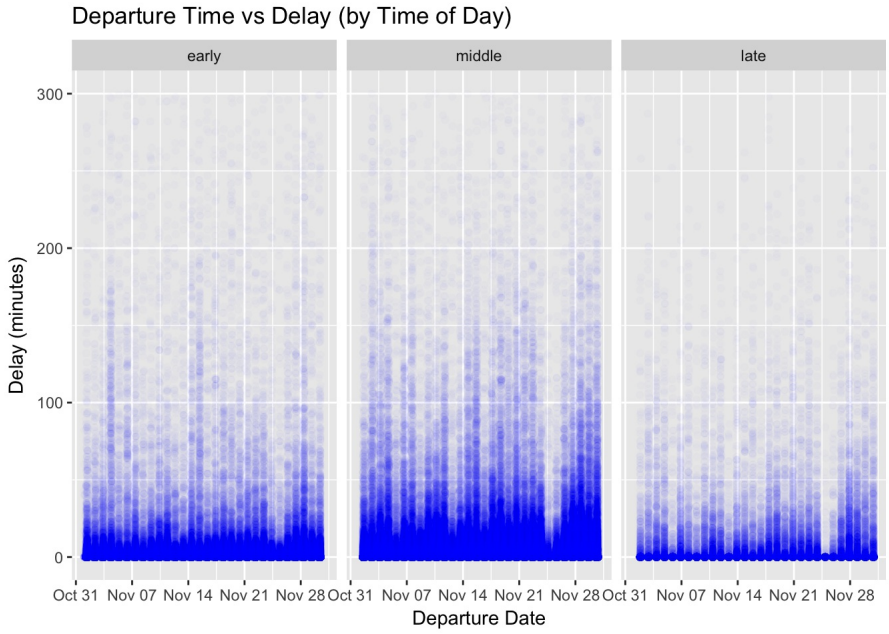
```
part_of_day <- function(dates) {
  period <- rep(NA, length(dates)) # Initialize and dummy vector
  for (i in 1:length(dates)) {
    if (hour(dates[i]) <= 11) { # Includes up to 11:59 AM
      period[i] <- "early"
    } else if (hour(dates[i]) >= 12 & hour(dates[i]) <= 19) { # 12:00 PM to 7:59 PM
      period[i] <- "middle"
    } else { # 8:00 PM to Midnight
      period[i] <- "late"
    }
  }
  return(period)
}
flights$period <- part_of_day(flights$FL_DATE)
head(flights$period, 5)
```

```
## [1] "middle" "middle" "early"  "early"  "middle"
```

Using this new information, let's make a new plot that distinguishes between these categories. You can use the `facet_wrap` layer of ggplot to accomplish this by faceting by period.

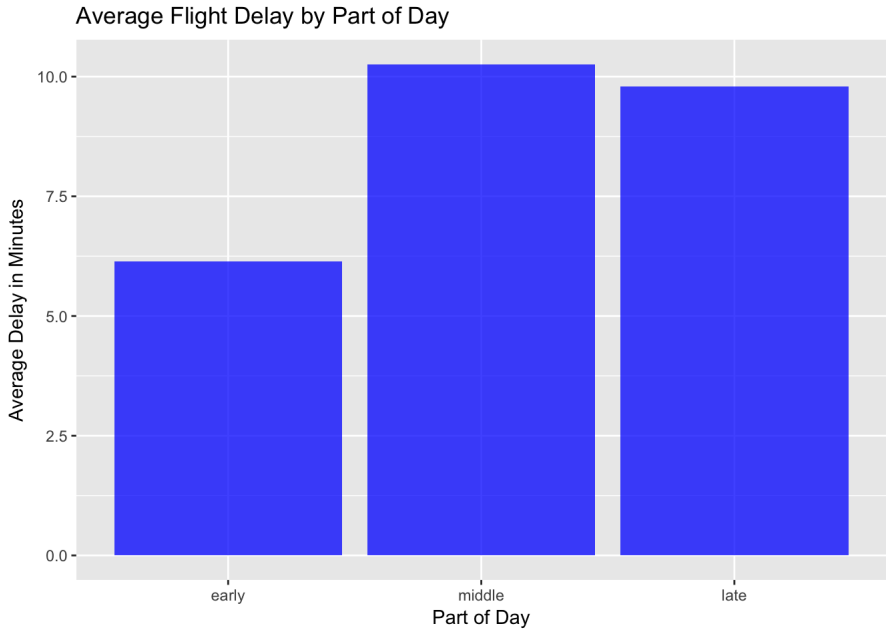
```
# Change the period to a factor to control the displayed order
flights$period <- factor(flights$period, levels=c("early", "middle", "late"))
# Same graph as before but facet wrapped by period
delay_by_period <- (ggplot(flights, aes(x = FL_DATE, y = DEP_DELAY_NEW)) +
                geom_point(alpha = .01, color = "blue") +
                facet_wrap(~ period) +
                ylim(0, 300) +
                ggtitle("Departure Time vs Delay (by Time of Day)") +
                labs(x = "Departure Date", y = "Delay (minutes)"))
delay_by_period
```

```
## Warning: Removed 1938 rows containing missing values (geom_point).
```



It does seem that the delays in the middle of the day tend to be larger. Certainly there are more long delays in the middle of the day, but we still can't tell for sure if this is due to there being more total flights in the early and late parts of the day, so lets control for this by finding and graphing the average delay using dplyr and ggplot2.

```
# Use dplyr to compute the summary statistic for mean delay
ave_del_by_period <- summarise(group_by(flights[ , 2:ncol(flights)], period),
                               ave_delay = mean(DEP_DELAY_NEW, na.rm = TRUE))
# Use ggplot to produce and display the plot
del_by_period_bar <- (ggplot(ave_del_by_period, aes(x = period, y = ave_delay)) +
                      geom_bar(stat = "identity", fill = "blue", alpha = .75) +
                      ggtitle("Average Flight Delay by Part of Day") +
                      labs(x = "Part of Day", y = "Average Delay in Minutes"))
del_by_period_bar
```



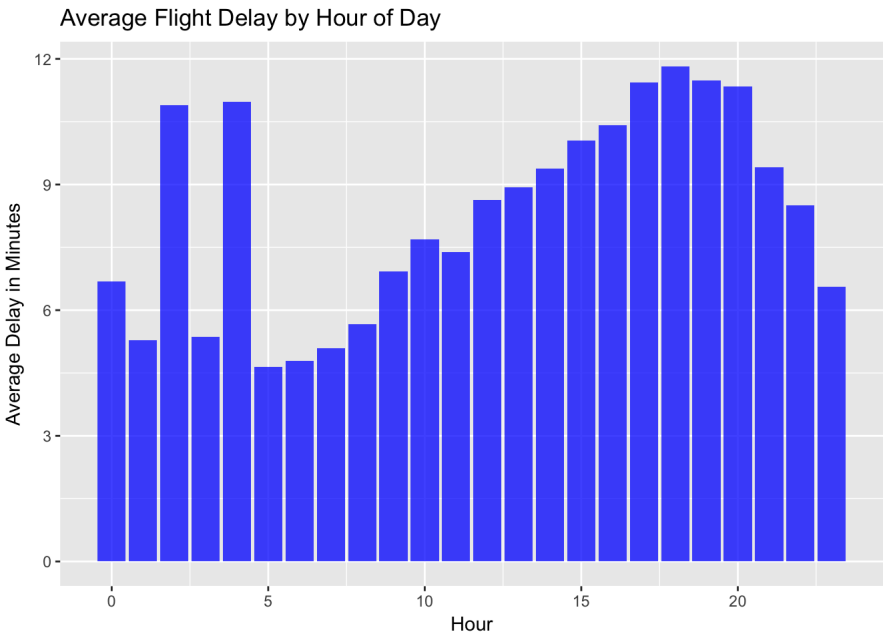
It's true – average delay really is highest during the middle of the day, but not by as much as it appeared in the earlier graph. This is probably because there are more flights in the afternoon, so more points appear visible. It turns out that the middle and end of the day are pretty similar, but delays are low in the morning. Perhaps the airlines simply haven't had the chance to fall behind schedule yet.

I'll test this and remove the somewhat arbitrary choice of period of day by splitting the dataset into each of the 24 hours. Now that we know a simple scatterplot can be misleading, let's skip that part and go straight to the summary statistics barplot.

```
## The first step is to add a column that captures the hour of the day
flights$hour_of_day <- hour(flights$FL_DATE)

## Use group_by and summarize operations to compute summary statistics
ave_del_by_hour <- summarise(group_by(flights[ , 2:ncol(flights)], hour_of_day),
                                    ave_delay = mean(DEP_DELAY_NEW, na.rm = TRUE))

## use ggplot to produce a plot, as before
del_by_hour_bar <- (ggplot(ave_del_by_hour, aes(x = hour_of_day, y = ave_delay)) +
                        geom_bar(stat = "identity", fill = "blue", alpha = .75) +
                        ggtitle("Average Flight Delay by Hour of Day") +
                        labs(x = "Hour", y = "Average Delay in Minutes"))
del_by_hour_bar
```

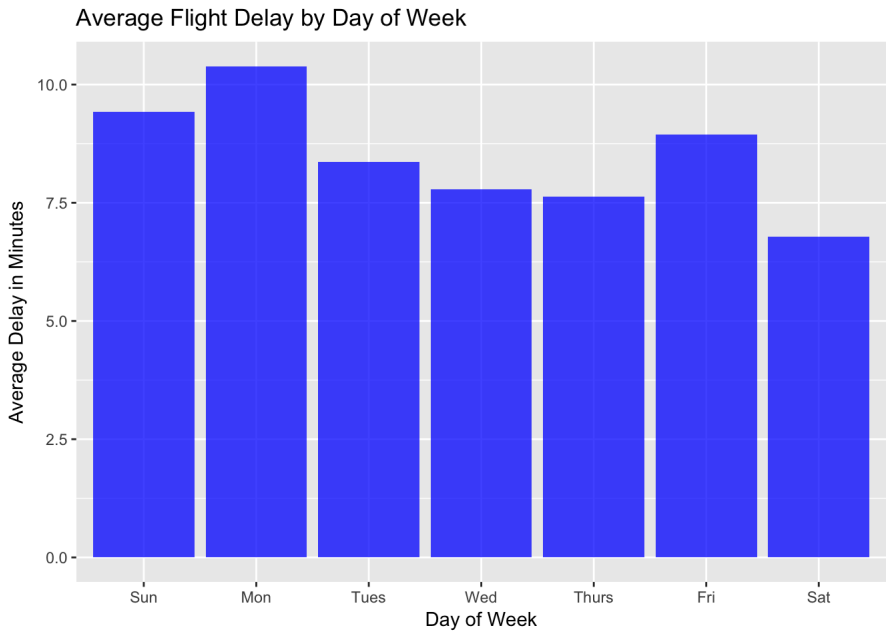


The result is very interesting. Outside of outliers at 2 and 4 am, the average delays build throughout the day until they reach their max around 6 PM, then gradually decline again. Flights that are very early or very late do tend to have the lowest average delays, and the “best” choice for avoiding a delay is around 5 AM.

Okay, so early in the morning is the delay minimizing choice. Personally, I hate early flights, so let’s see if there’s an easier way to minimize delay by repeating our analysis for days of the week.

```
## The first step is to add a column that captures the day of the week.
# Use label = TRUE to automatically convert to a descriptive factor
flights$day_of_week <- wday(flights$FL_DATE, label = TRUE)

## Use group_by and summarize operations to compute summary statistics
ave_del_by_day <- summarise(group_by(flights[ , 2:ncol(flights)], day_of_week),
                                    ave_delay = mean(DEP_DELAY_NEW, na.rm = TRUE))
## use ggplot to produce a plot, as before
del_by_wday_bar <- (ggplot(ave_del_by_day, aes(x = day_of_week, y = ave_delay)) +
                        geom_bar(stat = "identity", fill = "blue", alpha = .75) +
                        ggtitle("Average Flight Delay by Day of Week") +
                        labs(x = "Day of Week", y = "Average Delay in Minutes"))
del_by_wday_bar
```



There is less variation in delay across days of the week, but it looks like the most popular flying days (Friday, Sunday, and Monday) are the most delay prone. Saturday flights have the least delay, possibly because most people taking voluntary trips leave Friday, stay on Saturday, and then return on Sunday or Monday. This is conjecture of course – to prove a relationship between number of flights and average delay we would need to make a scatterplot of average number of flights per day (or hour) and delay and compute a test statistic.

Linear Regression Using Time Data

Let's see if the times with the longest delays are also the times with the most flights. To do this, start by using the dplyr function `count()` on the flight data, to count the number of flights by the hour of the day.

```
counts_by_hour <- count(flights[ , 2:ncol(flights)], hour_of_day)
head(counts_by_hour, 3)
```

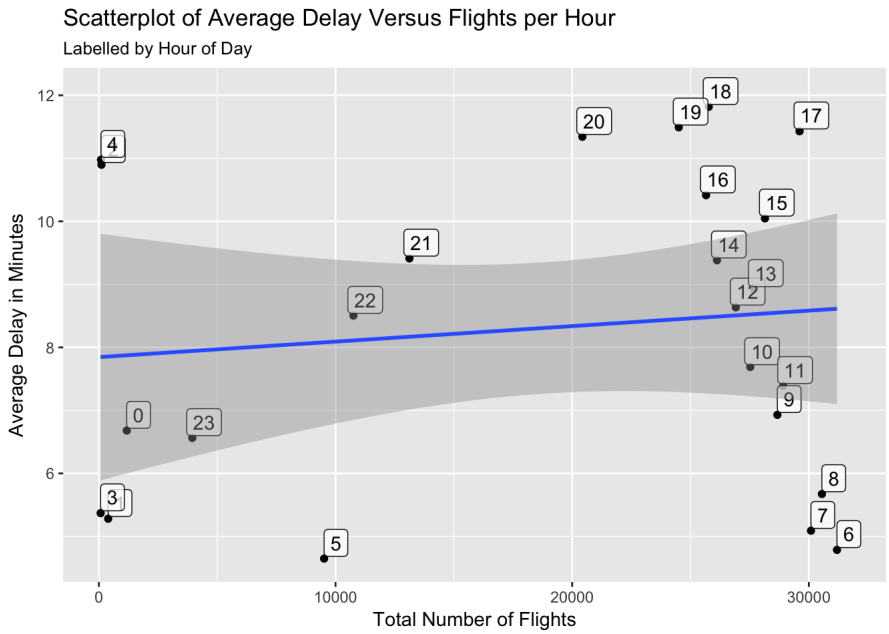
```
## # A tibble: 3 x 2
##   hour_of_day     n
##         <int> <int>
## 1           0  1178
## 2           1   392
## 3           2   109
```

Next, add the count column (which is called `n` by default) onto the data frame we already made with average delay by hour and then create a scatterplot of the delay against the count (using the hour of the day as a label).

```
# Add count column
ave_del_by_hour$count <- counts_by_hour$n
head(ave_del_by_hour, 3)
```

```
## # A tibble: 3 x 3
##   hour_of_day ave_delay count
##         <int>     <dbl> <int>
## 1           0  6.679422  1178
## 2           1  5.280612   392
## 3           2 10.898148   109
```

```
# Make the scatterplot
delay_vs_count <- (ggplot(ave_del_by_hour, aes(x = count, y = ave_delay)) +
                     geom_point() +
                     geom_label(aes(label = hour_of_day), alpha = .75,
                                nudge_x = 500, nudge_y = .25) +
                     geom_smooth(method = lm) +
                     ggtitle("Scatterplot of Average Delay Versus Flights per Hour",
                             subtitle = "Labelled by Hour of Day") +
                     labs(x = "Total Number of Flights",
                          y = "Average Delay in Minutes"))
delay_vs_count
```



The first thing to notice when looking at this graph is that there are very few flights at the very end or very beginning of the day. The bottom 5 hours in terms of quantity are in a continuous range from 11:00 PM to 4:59 AM. The bottom 8 hours are from 9:00 PM to 5:59 AM. These 8 hours seem pretty far offset from the rest of the data. In any case, there is no obvious pattern that comes out of this graph. It seems that quantity of flights is not driving the delays. Just to make sure, let's calculate the correlation coefficient:

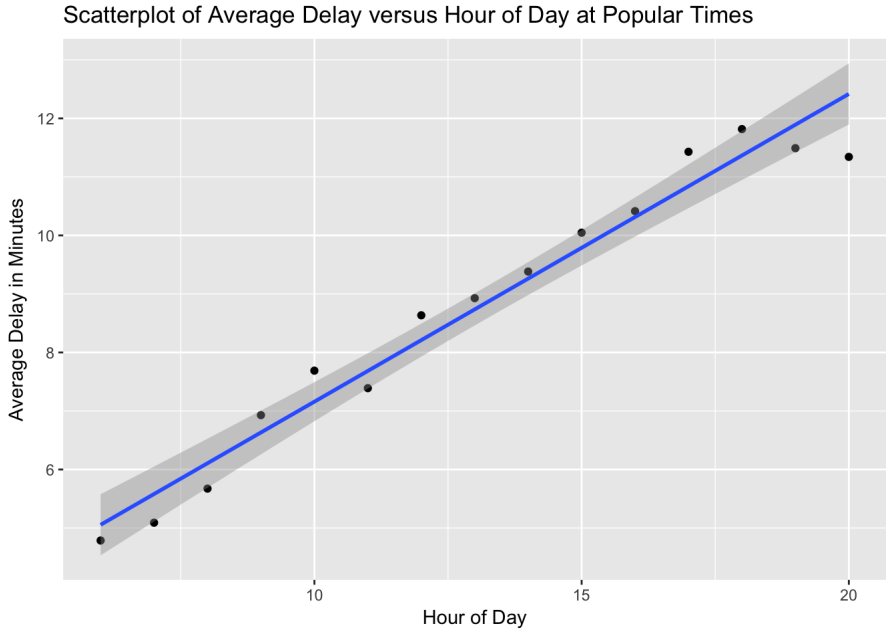
```
cor(ave_del_by_hour$count, ave_del_by_hour$ave_delay)
```

```
## [1] 0.1220106
```

The number of flights per hour and average flight delay have almost no linear correlation! Even though we haven't found what we're looking for, don't be discouraged by negative results. This conclusion is still useful, and now we can move on to checking another possible cause. Perhaps, as we thought earlier, the airlines fall behind as the day progresses. Let's remove the bottom 8 hours where there are less than half as many flights per hour on average and then check this by making a scatterplot of hour against delay.

```
# Slice to get only the hours we want
popular_times_with_delay <- slice(ave_del_by_hour, 7:21) # 7th row is hour 6
                                                         # 21st row is hour 20

delay_vs_hour <- (ggplot(popular_times_with_delay, aes(x = hour_of_day,
                                                       y = ave_delay)) +
            geom_point() +
            geom_smooth(method = lm) +
            ggtitle("Scatterplot of Average Delay versus Hour of Day at Popular Times") +
            labs(x = "Hour of Day",
                 y = "Average Delay in Minutes"))
delay_vs_hour
```



This looks much more correlated, and it is now clear delays do tend to increase as the day progresses. Of course, this only considers the 16 hours of the day with the most flights. As we can see in the count scatterplot, all of these hours are fairly close to each other in terms of number of flights, so in some way this controls for the busyness of the airport. The clear linear association visible in the scatterplot is confirmed by a

much larger correlation coefficient:

```
cor(popular_times_with_delay$hour_of_day, popular_times_with_delay$ave_delay)
```

```
## [1] 0.9803656
```

The next time a friend asks you what time to get to the airport, you'll know that delays are about twice as long on average at the end of the day than at the beginning!

Conclusion

I hope you now see the power of being able to do analysis that depends on dates and times and feel that you have all the tools necessary to investigate the topics that interest you. Being able to visualize how things in the world change over time will allow you, as a statistician, to make more accurate predictions about the future.

References:

- The referenced guide from the Berkeley Stats department about using base R to handle dates: https://www.stat.berkeley.edu/~s133/dates.html
- The referenced guide to formatting your datasets in a convenient format: http://kbroman.org/dataorg/
- The referenced full lubridate package on github: https://github.com/tidyverse/lubridate
- The referenced cran documentation for lubridate: https://cran.r-project.org/web/packages/lubridate/lubridate.pdf
- The referenced Wikipedia page including a list of time zones: https://en.wikipedia.org/wiki/List_of_tz_database_time_zones
- The referenced stack overflow page about converting two digit years: https://stackoverflow.com/questions/12323693/is-there-a-more-elegant-way-to-convert-two-digit-years-to-four-digit-years-with
- The referenced stack overflow page about reading dates without years: https://stackoverflow.com/questions/41173724/how-can-i-read-dates-without-year-using-readr
- The US Bureau of Transportation data website – the source of my flight data: https://www.transtats.bts.gov/DL_SelectFields.asp?Table_ID=236&DB_Short_Name=On-Time
- The referenced tutorial for ggplot2: http://r-statistics.co/ggplot2-Tutorial-With-R.html
- The referenced tutorial for dplyr: https://cran.r-project.org/web/packages/dplyr/vignettes/dplyr.html
- A vignette I used to help me learn lubridate from the ground up: https://cran.r-project.org/web/packages/lubridate/vignettes/lubridate.html
- A tutorial I used to help me understand how to import heterogeneous date formats: https://skgrange.github.io/date_handling.html#heterogeneous_formats
- A page I used to help me do floor and mod division in R: https://stat.ethz.ch/R-manual/R-devel/library/base/html/Arithmetic.html
- A page I used to help me order facets in ggplot2: https://stackoverflow.com/questions/14262497/fixing-the-order-of-facets-in-ggplot
- A page I used to understand the dplyr function `count()`: https://rdrr.io/cran/dplyr/man/tally.html