MATHS 7107 Data Taming Practical

Using the package `dplyr` to manipulate data

So far, we have seen some basic ways to manipulate data. Now, we are going to go all out. By the end of this practical, you will have the fundamentals to well and truly twist your data into any form you need.

Choose subjects with filter

In week 1, we explained how to access rows (subjects) and columns (variables) in a data frame. For example, to get the first row of the `MPG` dataset, we use:

```
pacman::p_load(tidyverse, dplyr)
data("mpg")
mpg[1,]
```

```
## # A tibble: 1 x 11
##   manufacturer model displ  year   cyl trans    drv     cty   hwy fl    class
##   <chr>        <chr> <dbl> <int> <int> <chr>    <chr> <int> <int> <chr> <chr>
## 1 audi         a4      1.8  1999     4 auto(15) f        18    29 p     compact
```

If you want to see the first six rows of a data frame you can use:

```
head(mpg)
```

```
## # A tibble: 6 x 11
##   manufacturer model displ  year   cyl trans      drv     cty   hwy fl    class
##   <chr>        <chr> <dbl> <int> <int> <chr>      <chr> <int> <int> <chr> <chr>
## 1 audi         a4      1.8  1999     4 auto(15)   f        18    29 p     compa~
## 2 audi         a4      1.8  1999     4 manual(m5) f        21    29 p     compa~
## 3 audi         a4      2    2008     4 manual(m6) f        20    31 p     compa~
## 4 audi         a4      2    2008     4 auto(av)   f        21    30 p     compa~
## 5 audi         a4      2.8  1999     6 auto(15)   f        16    26 p     compa~
## 6 audi         a4      2.8  1999     6 manual(m5) f        18    26 p     compa~
```

Question: How could you get the last six rows? Can you guess the function? (hint: remember you can always use the ? help function).

This way is fine most of the time, but occasionally I would like to be more precise with my filtering. So, we are going to introduce you to the package `dplyr` in the `tidyverse` package which is oh-so-much nicer. Also, as a treat, we will move away from the `MPG` dataset (everyone, please take a moment to reminisce — we'll miss you `MPG`.... or will we...) and introduce the flights dataset. This dataset is in the `nycflights13` package, so go get it. It contains information on all 336776 flights out of New York City in 2013.

```
## # A tibble: 336,776 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
##  1  2013     1     1      517            515         2      830            819
##  2  2013     1     1      533            529         4      850            830
##  3  2013     1     1      542            540         2      923            850
##  4  2013     1     1      544            545        -1     1004           1022
##  5  2013     1     1      554            600        -6      812            837
```

```
##  6  2013     1     1      554            558        -4      740            728
##  7  2013     1     1      555            600        -5      913            854
##  8  2013     1     1      557            600        -3      709            723
##  9  2013     1     1      557            600        -3      838            846
## 10  2013     1     1      558            600        -2      753            745
## # ... with 336,766 more rows, and 11 more variables: 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>
```

The first verb that `dplyr` introduces is `filter()`. This lets us filter the subjects that we want, i.e., filter rows. So first – let's pause a second and decide what are the subjects in the `flights` dataset?

Go away, you have not paused long enough.

Better.

Yes, each subject is a flight.

Let's start by filtering for just the flights in January (Month 1):

```
filter(flights, month == 1)
```

```
## # A tibble: 27,004 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
##  1  2013     1     1      517            515         2      830            819
##  2  2013     1     1      533            529         4      850            830
##  3  2013     1     1      542            540         2      923            850
##  4  2013     1     1      544            545        -1     1004           1022
##  5  2013     1     1      554            600        -6      812            837
##  6  2013     1     1      554            558        -4      740            728
##  7  2013     1     1      555            600        -5      913            854
##  8  2013     1     1      557            600        -3      709            723
##  9  2013     1     1      557            600        -3      838            846
## 10  2013     1     1      558            600        -2      753            745
## # ... with 26,994 more rows, and 11 more variables: 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>
```

This gives use 27,004 rows (see the top of output).

Breaking this command down:

- `filter()`: function to do all of the work,
- `flights`: first argument in the brackets gives the dataframe name,
- `month ==`: this is the criteria to filter on. In this case, the variable `month` is equal to `==` the first month `1`. Notice the use of a double `=` to indicate equal to. A single `=` will cause an error. (Try it.... I dare you)

How about the first day of January?

```
filter(flights, month == 1, day==1)
```

```
## # A tibble: 842 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
##  1  2013     1     1      517            515         2      830            819
##  2  2013     1     1      533            529         4      850            830
##  3  2013     1     1      542            540         2      923            850
##  4  2013     1     1      544            545        -1     1004           1022
```

```
##  5  2013     1     1      554            600        -6      812            837
##  6  2013     1     1      554            558        -4      740            728
##  7  2013     1     1      555            600        -5      913            854
##  8  2013     1     1      557            600        -3      709            723
##  9  2013     1     1      557            600        -3      838            846
## 10  2013     1     1      558            600        -2      753            745
## # ... with 832 more rows, and 11 more variables: 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>
```

We can just keep adding criteria.

Question: How many flights were scheduled to depart at 7am?

Using pipes (`magrittr`) to make my code easier to read

As we build up more verbs in our data manipulation tool bag, we are going to end up with lots of nested functions. Instead, we are going to use the `magrittr` or 'pipe' symbol %>%.

This command can be read as 'then', and is used to join verbs. For example, to get the first of January, we can rewrite the above command as:

```
flights %>% filter(month == 1, day == 1)
```

```
## # A tibble: 842 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
##  1  2013     1     1      517            515         2      830            819
##  2  2013     1     1      533            529         4      850            830
##  3  2013     1     1      542            540         2      923            850
##  4  2013     1     1      544            545        -1     1004           1022
##  5  2013     1     1      554            600        -6      812            837
##  6  2013     1     1      554            558        -4      740            728
##  7  2013     1     1      555            600        -5      913            854
##  8  2013     1     1      557            600        -3      709            723
##  9  2013     1     1      557            600        -3      838            846
## 10  2013     1     1      558            600        -2      753            745
## # ... with 832 more rows, and 11 more variables: 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>
```

We read this as:

- get the dataframe: `flights`
- then: %>%
- filter for Jan 1: `filter(month == 1, day == 1)`.

OK, your turn. Filter for all American Airlines flights (AA). Make sure you use the pipe symbol!

Choose variables with Select

The thing about the `nycflights13` dataset (or many 'Big' datasets) is that it is too wide to fit onto the screen. In all the examples of filtering above, we couldn't even see all of the variables — there were always 12 columns that didn't fit on the screen. It would be nice if we could make our data frame a bit narrower so that we can fit the information we're interested in (and nothing else) onto the screen. This is what `select` does — it's essentially a `filter`, but for columns rather than rows.

So, let's say I was just interested in departure times and arrival times for all American Airways flights on the 1st of January; then I could just add to our `filter` example from before:

```
flights %>% filter(carrier == "AA",month == 1, day == 1) %>% select(flight,dep_time,arr_time)
```

```
## # A tibble: 94 x 3
##    flight dep_time arr_time
##     <int>    <int>    <int>
##  1   1141      542      923
##  2    301      558      753
##  3    707      559      941
##  4   1895      606      858
##  5   1837      623      920
##  6    413      628     1137
##  7    303      629      824
##  8    711      635     1028
##  9    305      656      854
## 10   1815      656      949
## # ... with 84 more rows
```

Now we only have 94 x 3 = 282 pieces of information in our data frame. This is starting to look more manageable!

We can get fancy with `select`: if I wanted to grab just the variables from `flights` that have something to do with 'time', then I could use the `contains` command:

```
select(flights, contains("time"))
```

```
## # A tibble: 336,776 x 6
##    dep_time sched_dep_time arr_time sched_arr_time air_time time_hour
##       <int>          <int>    <int>          <int>    <dbl> <dttm>
##  1      517            515      830            819      227 2013-01-01 05:00:00
##  2      533            529      850            830      227 2013-01-01 05:00:00
##  3      542            540      923            850      160 2013-01-01 05:00:00
##  4      544            545     1004           1022      183 2013-01-01 05:00:00
##  5      554            600      812            837      116 2013-01-01 06:00:00
##  6      554            558      740            728      150 2013-01-01 05:00:00
##  7      555            600      913            854      158 2013-01-01 06:00:00
##  8      557            600      709            723       53 2013-01-01 06:00:00
##  9      557            600      838            846      140 2013-01-01 06:00:00
## 10      558            600      753            745      138 2013-01-01 06:00:00
## # ... with 336,766 more rows
```

Or,

```
flights %>% select(contains("time"))
```

```
## # A tibble: 336,776 x 6
##    dep_time sched_dep_time arr_time sched_arr_time air_time time_hour
##       <int>          <int>    <int>          <int>    <dbl> <dttm>
##  1      517            515      830            819      227 2013-01-01 05:00:00
##  2      533            529      850            830      227 2013-01-01 05:00:00
##  3      542            540      923            850      160 2013-01-01 05:00:00
##  4      544            545     1004           1022      183 2013-01-01 05:00:00
##  5      554            600      812            837      116 2013-01-01 06:00:00
##  6      554            558      740            728      150 2013-01-01 05:00:00
##  7      555            600      913            854      158 2013-01-01 06:00:00
##  8      557            600      709            723       53 2013-01-01 06:00:00
```

```
##  9      557            600      838            846      140 2013-01-01 06:00:00
## 10      558            600      753            745      138 2013-01-01 06:00:00
## # ... with 336,766 more rows
```

In a similar vein, you might be able to guess what `starts_with()` or `ends_with()` do. I can also select, say, the columns from year to day:

```
select(flights, year:day)
```

```
## # A tibble: 336,776 x 3
##     year month   day
##    <int> <int> <int>
##  1  2013     1     1
##  2  2013     1     1
##  3  2013     1     1
##  4  2013     1     1
##  5  2013     1     1
##  6  2013     1     1
##  7  2013     1     1
##  8  2013     1     1
##  9  2013     1     1
## 10  2013     1     1
## # ... with 336,766 more rows
```

Type `?select` at the console to see some more examples.

But... how about if we want to make some new variables? It's time for us — like a statistical X-Men character — to mutate.

Making new variables with `mutate()`

In the dataset `flights`, we have the departure delay `dep_delay`, which is the difference between the scheduled departure time (`sched_dep_time`) and the departure time (`dep_time`). Let's assume that this is not given and that we would like to calculate it. We can do that with:

```
flights %>% mutate(delay = dep_time - sched_dep_time)
```

```
## # A tibble: 336,776 x 20
##     year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
##  1  2013     1     1      517            515         2      830            819
##  2  2013     1     1      533            529         4      850            830
##  3  2013     1     1      542            540         2      923            850
##  4  2013     1     1      544            545        -1     1004           1022
##  5  2013     1     1      554            600        -6      812            837
##  6  2013     1     1      554            558        -4      740            728
##  7  2013     1     1      555            600        -5      913            854
##  8  2013     1     1      557            600        -3      709            723
##  9  2013     1     1      557            600        -3      838            846
## 10  2013     1     1      558            600        -2      753            745
## # ... with 336,766 more rows, and 12 more variables: 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>,
## #   delay <int>
```

Let's work through this:

- Take the dataset: `flights`
- then: %>%
- add a new column: `mutate()`
- called delay: `delay =`
- which is calculated as the difference: `dep_time - sched_dep_time`. If you look at the last row of the output, you see the new column `delay <int>`, but wait — that is a trick!

Look at the flights data frame again. Can you see this new column? The new column is not there. Why? Because you have to tell R to save the new column, like this:

```
flights <- flights %>% mutate(delay = dep_time - sched_dep_time)
```

This is the same as before, but the extra `flights <-` basically says 'do the stuff, then save it as the data frame `flights`', i.e. overwrite it.

Sort using `arrange`

Sometimes I would like to sort a column from smallest to largest. This is done with `arrange()`.

How about an example. We will sort the rows by distance. To make it easier to see, we will also select just the origin, dest, and distance columns:

```
flights %>%
  select(origin, dest, distance) %>%
  arrange(distance)
```

```
## # A tibble: 336,776 x 3
##    origin dest  distance
##    <chr>  <chr>    <dbl>
##  1 EWR    LGA         17
##  2 EWR    PHL         80
##  3 EWR    PHL         80
##  4 EWR    PHL         80
##  5 EWR    PHL         80
##  6 EWR    PHL         80
##  7 EWR    PHL         80
##  8 EWR    PHL         80
##  9 EWR    PHL         80
## 10 EWR    PHL         80
## # ... with 336,766 more rows
```

The shortest flight is EWR to LGA. What about longest distance?

```
flights %>%
  select(origin, dest, distance) %>%
  arrange(desc(distance))
```

```
## # A tibble: 336,776 x 3
##    origin dest  distance
##    <chr>  <chr>    <dbl>
##  1 JFK    HNL       4983
##  2 JFK    HNL       4983
##  3 JFK    HNL       4983
##  4 JFK    HNL       4983
##  5 JFK    HNL       4983
##  6 JFK    HNL       4983
##  7 JFK    HNL       4983
```

```
##  8 JFK    HNL       4983
##  9 JFK    HNL       4983
## 10 JFK    HNL       4983
## # ... with 336,766 more rows
```

Notice the use of `desc` to change from smallest to largest to largest to smallest.

We can also do multiple sorts:

```
flights %>%
  select(origin, dest, dep_time, distance) %>%
  arrange(distance, dep_time)
```

```
## # A tibble: 336,776 x 4
##    origin dest  dep_time distance
##    <chr>  <chr>    <int>    <dbl>
##  1 EWR    LGA         NA       17
##  2 EWR    PHL       1155       80
##  3 EWR    PHL       1240       80
##  4 EWR    PHL       1610       80
##  5 EWR    PHL       1613       80
##  6 EWR    PHL       1617       80
##  7 EWR    PHL       1619       80
##  8 EWR    PHL       1621       80
##  9 EWR    PHL       1829       80
## 10 EWR    PHL       1926       80
## # ... with 336,766 more rows
```

This sorts by `distance` first, and then within each distance by `dep_time`.

Your turn! Obtain the scheduled arrival time and actual arrival time for each carrier and sort by the arrival delay.

Split into Groups with group_by

We are now in a position to do something quite powerful – clump variables together into groups, and then summarise these groups.

Grouping won't look like much just yet, but stick with me on this.

I have a hypothesis about flight delays in New York City: I reckon they're worse in winter.

Snowstorms, ice, wind – I suspect that all of these will make delays in the winter months of December, January, February, worse than in the summer months. To investigate this we'll need to group flights by month, which we can do like this:

```
by_month <- group_by(flights,month)
by_month
```

```
## # A tibble: 336,776 x 20
## # Groups:   month [12]
##     year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
##  1  2013     1     1      517            515         2      830            819
##  2  2013     1     1      533            529         4      850            830
##  3  2013     1     1      542            540         2      923            850
##  4  2013     1     1      544            545        -1     1004           1022
##  5  2013     1     1      554            600        -6      812            837
```

```
##  6  2013     1     1      554            558        -4      740            728
##  7  2013     1     1      555            600        -5      913            854
##  8  2013     1     1      557            600        -3      709            723
##  9  2013     1     1      557            600        -3      838            846
## 10  2013     1     1      558            600        -2      753            745
## # ... with 336,766 more rows, and 12 more variables: 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>,
## #   delay <int>
```

Well that was underwhelming.

This data frame looks pretty much the same as the original, apart from the second line: `Groups: month [12]` .

That tells me that a group for each month has been created, but to explore this further we'll have to learn to `summarise`.

Before we do though, two quick notes about `group_by`:

- You can group by multiple variables: `by_day <- group_by(flights,year,month,day)` will create a dataframe with 365 groups for each day of the year (try it!).
- You can ungroup a grouped dataframe using, you guessed it, `ungroup()`.

Get summaries with `summarise`

To test my hypothesis about flight delays, we need to create a summary statistic about delays for each month.

Let's calculate the mean flight departure delay for each calendar month, which we can do like this:

```
summarise(by_month, delay = mean(dep_delay, na.rm = TRUE))
```

```
## # A tibble: 12 x 2
##    month delay
##    <int> <dbl>
##  1     1 10.0
##  2     2 10.8
##  3     3 13.2
##  4     4 13.9
##  5     5 13.0
##  6     6 20.8
##  7     7 21.7
##  8     8 12.6
##  9     9  6.72
## 10    10  6.24
## 11    11  5.44
## 12    12 16.6
```

This has taken my grouped data frame `by_month`, and for each group has computed the mean of the values in the `dep_delay` column for that group.

The `na.rm = TRUE` argument has told the mean function to remove (`rm` in unix-speak) all values that are not available (`NA`). Basically, some rows in this data frame do not have an entry in the `dep_delay` column, so R puts the symbol `NA` there instead. Trying to calculate the mean of this symbol doesn't work (try it without the `na.rm = TRUE` bit!), so we get rid of them instead.

Secondly, I don't think my hypothesis was correct. Maybe December has slightly worse delays than the preceding months, but January and February really aren't so bad, and by far the worst months are June and July. To drive the point home, let's make a nice plot of the trends over the entire year using ggplot

(which you will learn more about in week 3!). We'll use our skills to group by day this time instead of month, because it looks cooler.

```
by_day <- group_by(flights,year,month,day)
summarise(by_day, delay = mean(dep_delay, na.rm = TRUE)) %>%
  ungroup() %>%
  mutate(day_num = seq_along(delay)) %>%
  ggplot(aes(day_num,delay)) +
  geom_point() +
  geom_smooth()
```

I added a slightly tricky intermediate step here to create a column `day_num` counting the days of the year: `ungroup()` %>% `mutate(day_num = seq_along(delay))` ungroups `by_day`, and then creates a sequence along the column `delay` — essentially counting the row numbers.

Anyway, the middle of the year really does look like a worse time to fly; so much for my hypothesis. Guided by this exploration, I think I have a new one now : June, July, and December would be the busiest months for flying, so maybe it's simply that there are longer delays when there are more flights. A simple modification to our `summarise` command will allow us to explore this relationship:

```
summarise(by_day, delay = mean(dep_delay, na.rm = TRUE), num_flights = n()) %>%
  ggplot(aes(num_flights,delay)) +
  geom_point() +
  geom_smooth()
```

The `num_flights = n()` bit produces a second summary statistic for each group, which is just the number of items in that group. Note that I don't need to index days of the year now, so I can lose the `ungroup` bit.

It looks like there's some relationship between number of flights and delays, but it's not particularly strong. Again, some more investigation is needed.

Your turn (exercises)

1. Load the `mpg` dataset. Hello old friend!
2. Find all cars that are made in 1999. How many are there?
3. Write the command to give just the year and manufacturer columns.
4. Convert the column `hwy` from miles per gallon to km per litre. Hint 1 miles / gallon is 0.42 km / litre.
5. Find the mean `cty` for each year.

Further challenge: in this week's content (week 2), the "Using inspect_df solutions" code is written using `magrittr`. See if you can understand the difference between the instructions in the content and this code and ask us if you have any questions.