Lab 0: Tidyverse

PSTAT 131-231

Objectives

- An introduction to `tidyverse`
  - `filter()`
  - `select()`
  - `%>%` (chaining)
  - `mutate()`
  - `summarize()`
  - `group_by()`
  - `pivot_wider()`
  - `pivot_longer()`
  - `ggplot()`

Instructions: work through the .Rmd file and fill in the 'Your turn' portions. Once complete, knit to pdf and submit your pdf to Gradescope. Your work will be evaluated based on whether you completed all of the 'Your turn' portions.

1. Data wrangling with `tidyverse`

We will use the epynomous dataset in the `hflights` package and a simulated shopper behavior dataset to illustrate use of basic `tidyverse` commands for data manipulation.

The `hflights` dataset contains all flights departing from Houston airports IAH (George Bush Intercontinental) and HOU (Houston Hobby) in 2011. It is in 'tidy' format: every row is an observation (a flight) and every column is a feature (flight attribute).

The data comes from the Research and Innovation Technology Administration at the Bureau of Transportation statistics (source details).

In RStudio Cloud, package installs are managed for you as part of the workspace setup. If you are working on your own machine, you'll need to install `tidyverse` and `hflights`.

```
# Load packages
library(hflights)
library(tidyverse)
library(pander)
```

In the `tidyverse`, "tibbles" are used instead of data frames. They have greater flexibility and better display behavior. The `hflights` dataset is stored as a data frame, so we'll convert it to a tibble.

```
# convert to tibble
flights <- as_tibble(hflights)

# print a few rows and columns
preview <- head(flights, c(3, 6))
pander(preview)
```

Year	Month	DayofMonth	DayOfWeek	DepTime	ArrTime
2011	1	1	6	1400	1500
2011	1	2	7	1401	1501
2011	1	3	1	1352	1502

The verbs introduced below all perform operations on tibbles (and will also work on data frames).

Subsetting with `filter()`

`filter()` returns rows of a tibble that match specified conditions, and is a useful way to extract subsets of data. Run the following several examples of its use:

```
# flights in january
filter(flights, Month == 1)

# flights on january 1
filter(flights, Month == 1, DayofMonth == 1)

# all AA or UA flights
filter(flights, UniqueCarrier %in% c("AA", "UA"))

# flights delayed more than an hour
filter(flights, DepDelay > 60)
```

Your turn (1) Write filter commands to find the following:

```
# all flights longer than 3 hours
filter(flights, ActualElapsedTime > 180)

# all cancelled flights from HOU
filter(flights, Cancelled == 1, Origin == "HOU")

# all UA flights in march departing IAH in the morning
filter(flights, UniqueCarrier == "UA", Month == 3, DepTime < 1200, Origin == "IAH")
```

Select columns with `select()`

`select()` is used to pick a set of columns by their names, often with helper functions. Run the following examples to get a sense of its use:

```
# select departure and arrival times
select(flights, DepTime, ArrTime)

# return all adjacent variables between year and weekday
select(flights, Year:DayOfWeek)

# select all variables whose names include 'Time'
select(flights, contains('Time'))

# select all variables whose names start with 'Day'
select(flights, starts_with('Day'))

# both of the previous two sets of variables
select(flights, contains('Time'), starts_with('Day'))
```

```r
# exclude year
select(flights, -Year)

# select all numeric variables
select(flights, where(is.numeric))

# all variables with no 'NA' entries
select(flights, where(~ sum(is.na(.x)) == 0))
```

Your turn (2) Write `select()` commands to obtain the following variables:

```r
# delay-related variables
select(flights, contains('Delay'))

# all character variables
select(flights, where(is.character))

# date information, origin, destination, and delay information
select(flights, Year:DayOfWeek, Origin, Dest, contains('Delay'))
```

Chaining with %>%

Filtering and selecting (and other data manipulations) are often performed in conjunction. The usual way to perform multiple operations in one line is by nesting them:

```r
select(filter(flights, DayofMonth == 1), contains('Time'))
```

This is hard to read. Chaining alleviates the trouble. `x %>% f(...)` performs `f(x, ...)`, passing `x` to the first argument in `f`. Rewriting the above:

```r
# show time variables for all flights in january
filter(flights, Month == 1) %>%
  select(contains('Time'))

# even better...
flights %>%
  filter(Month == 1) %>%
  select(contains('Time'))
```

This also works with base commands. For example:

```r
# calculate Euclidean distance
x1 <- 1:5
x2 <- 2:6
(x1-x2)^2 %>% sum() %>% sqrt();
```

Your turn (3) Write a chain of commands to accomplish the following:

```r
# display arr/dep delay times for all UA flights
flights %>%
  filter(UniqueCarrier  == 'UA') %>%
  select(ArrTime, DepTime)

# display flight times for all flights from HOU to DFW
flights %>%
```

```
  filter(Origin == 'HOU', Dest == 'DFW') %>%
  select(DepTime)
```

Defining new variables with `mutate()`

`mutate()` creates new variables as functions of existing variables. Run the following examples of its use:

```
# calculate average airspeed
flights %>%
  mutate(AvgSpeed = Distance/AirTime*60)

# proportion of time in the air
flights %>%
  mutate(FlightTimeProp = AirTime/ActualElapsedTime)

# indicator for whether a flight has 'NA' for any variable
flights %>%
  mutate(has.na = complete.cases(flights))

# add observation number
flights %>%
  mutate(obsNum = 1:nrow(flights))
```

To store a newly created variable with the original data, it is necessary to assign the result of the operation to a named object:

```
# store as a new tibble
flights_mod <- flights %>%
  mutate(AvgSpeed = Distance/AirTime*60)

# replace original tibble
flights <- flights %>%
  mutate(AvgSpeed = Distance/AirTime*60)
```

Your turn (4) Define the following new variable:

```
# indicator for whether a flight was delayed
flights_mod3 <- flights %>%
  mutate(has.delay = DepDelay > 0 | DepDelay < 0)
head(flights_mod3)
```

Data aggregation with `group_by()` and `summarize()`

`summarize()` applies user-specified functions to columns within (row-wise) groupings defined by `group_by()`. Run the following examples of their use:

```
# average arrival delay (no grouping)
flights %>%
  summarize(AvgDelay = mean(ArrDelay, na.rm=TRUE))

# average arrival delay by destination
flights %>%
    group_by(Dest) %>%
    summarize(AvgDelay = mean(ArrDelay, na.rm=TRUE))

# average and maximum delay times by destination
```

```
flights %>%
  group_by(Dest) %>%
  summarize(AvgDelay = mean(ArrDelay, na.rm=TRUE),
            MaxDelay = max(ArrDelay, na.rm=TRUE))
```

A helper verb, `across()` allows you to apply the same summary function to multiple columns at once. Try the following:

```
# calculate percentage of diverted and cancelled flights for each carrier
flights %>%
    group_by(UniqueCarrier) %>%
    summarize(across(c(Cancelled,Diverted), mean))

# calculate percentage and total diverted and cancelled flights for each carrier
flights %>%
    group_by(UniqueCarrier) %>%
    summarize(across(c(Cancelled,Diverted),
                     list(pct = mean, total = sum)))
```

Your turn (5) Compute the following summaries (you may also need to filter/mutate):

```
# average delay time by carrier in january
flights %>%
    group_by(UniqueCarrier) %>%
    filter(Month == 1) %>%
    summarize(AvgDepDelay1 = mean(DepDelay, na.rm=TRUE))

# average delay time for all morning flights on UA by month
flights %>%
    group_by(Month) %>%
    filter(UniqueCarrier == "UA", DepTime < 1200)  %>%
    summarize(AvgDepDelay2 = mean(DepDelay, na.rm=TRUE))
```

Counting with `count()`

Often it's helpful to know how many observations are in different groups. The function `count()` sums up the number of observations in each group. Run the following examples:

```
# number of flights per carrier
flights %>%
  group_by(UniqueCarrier) %>%
  count()

# number of flights by month
flights %>%
  group_by(Month) %>%
  count()

# number of flights per carrier per month
flights %>%
  group_by(UniqueCarrier, Month) %>%
  count()
```

Your turn (6) Count the following:

```r
# number of flights delayed by more than an hour per carrier
flights %>%
  group_by(UniqueCarrier) %>%
  filter(DepDelay > 60) %>%
  count()

# number of cancelled flights by month
flights %>%
  group_by(Month) %>%
  filter(Cancelled == 1) %>%
  count()
```

Pivoting wide data to long data and vice-versa

Suppose data consist purchase history of three users of an online shopping site.

```r
# import data
shopping <- read_csv('data/online-shopping.csv')
shopping %>% head(c(3, 6)) %>% pander()
```

User	item1	item2	item3	item4	item5
user1	79	77	119	112	117
user2	7	12	28	27	28
user3	35	34	35	35	35

The data are formatted as a wide table, where rows are the unique users and columns are the item variables and the entries are the quantities purchased. A single observation is a purchase event. For each purchase event, there are three pieces of information: a quantity, a user and an item type.

`pivot_longer` can help us put the data in tidy format. The following stacks all but the first columns on top of one another, preserves the value of the first column, and creates a named variable for the values stacked on top of one another and another named 'key' variable with the column names.

```r
shopping %>%
  pivot_longer(cols = -1,
               names_to = 'itemtype',
               values_to = 'quantity') %>%
  head() %>%
  pander()
```

User	itemtype	quantity
user1	item1	79
user1	item2	77
user1	item3	119
user1	item4	112
user1	item5	117
user1	item6	117

`pivot_wider()` converts data from long format to wide format. It does the opposite of `pivot_longer()`, and can be used to undo the operation above:

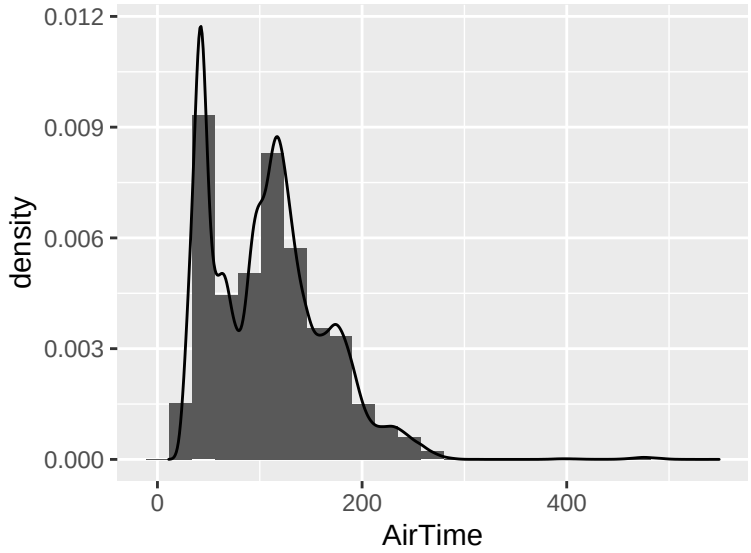
```
shopping %>%
  pivot_longer(cols = -1,
               names_to = 'itemtype',
               values_to = 'quantity') %>%
  pivot_wider(names_from = 'itemtype',
              values_from = 'quantity') %>%
  head(c(3, 6)) %>%
  pander()
```

User	item1	item2	item3	item4	item5
user1	79	77	119	112	117
user2	7	12	28	27	28
user3	35	34	35	35	35

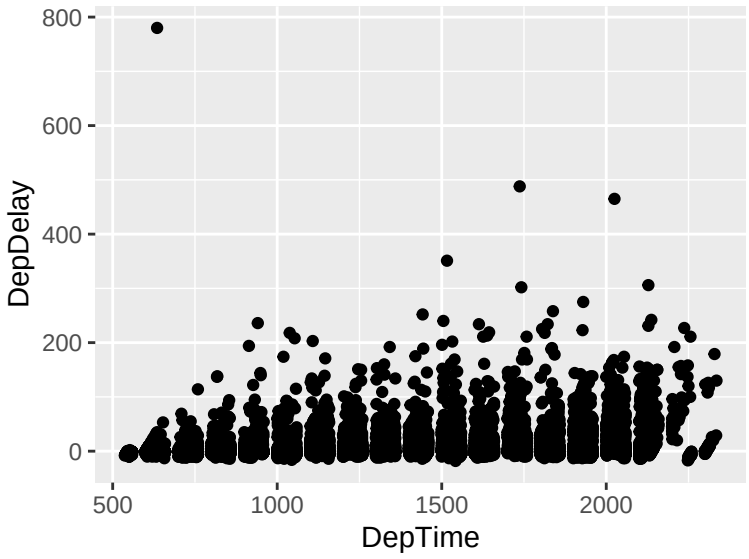
Visualization with `ggplot()`

The `gg` in `ggplot()` stands for grammar of graphics. For all but very specialized situations, ggplot visualizations are much easier to work with than base R plots. In `ggplot()`, variables are mapped to plot aesthetics with the `aes()` function and layers are added to the resulting plot one-by-one with various other commands. Here are some examples:

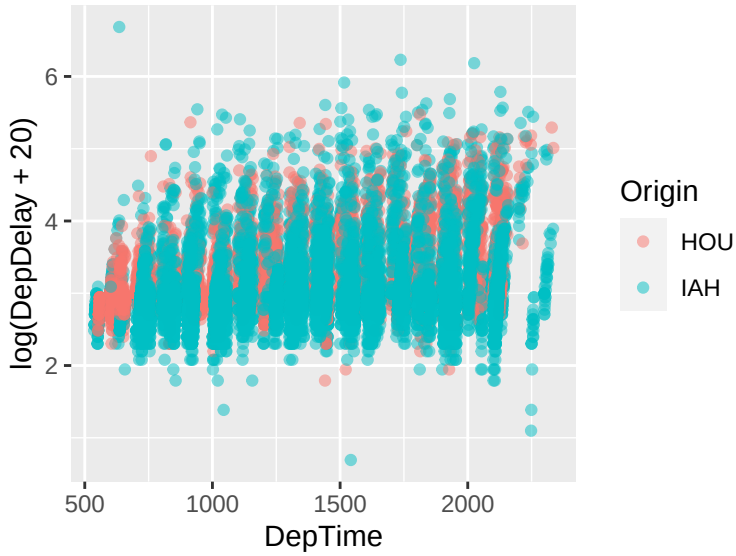
```
# histogram of flight times and added smooth density estimate
flights %>%
  na.omit() %>%
  ggplot(aes(x = AirTime, y = ..density..)) +
  geom_histogram(bins = 25) +
  geom_density()
```



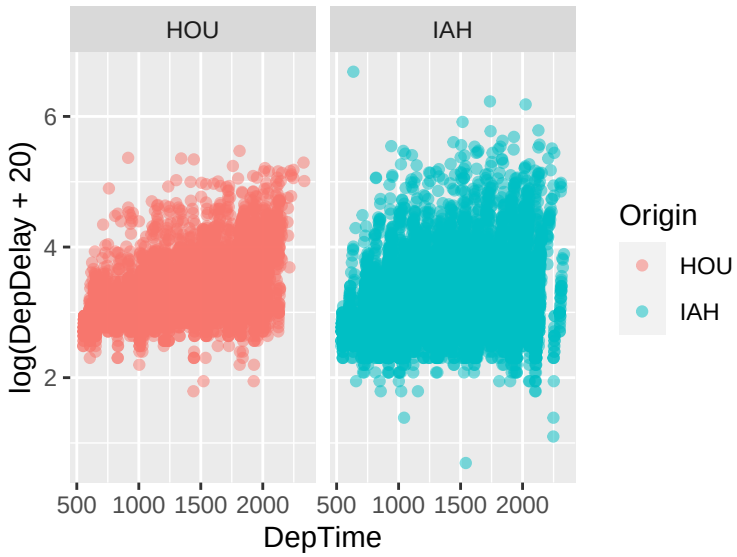

```
## scatterplot of departure delay against departure time
flights %>%
  na.omit() %>%
  filter(Month == 1, DepTime >= 500) %>%
  ggplot(aes(x = DepTime, y = DepDelay)) +
  geom_point()
```



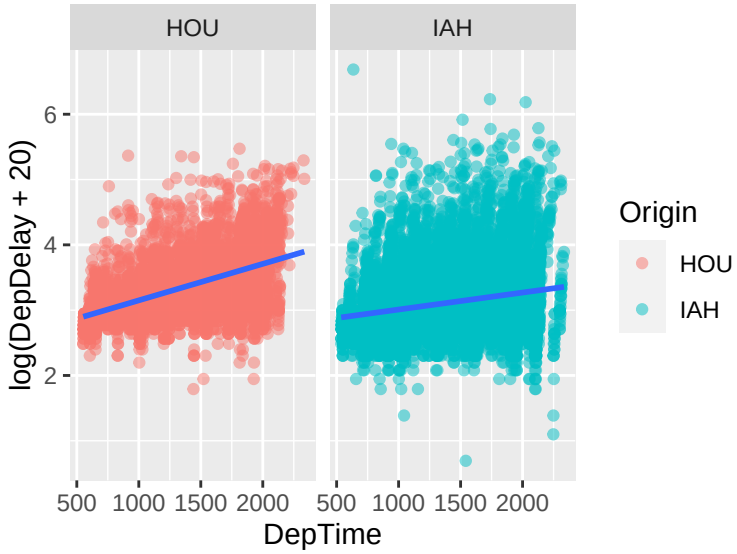

```
## map color to origin, transform y variable, adjust point tranparency
flights %>%
  na.omit() %>%
  filter(Month == 1, DepTime >= 500) %>%
  ggplot(aes(x = DepTime, y = log(DepDelay + 20))) +
  geom_point(aes(color = Origin), alpha = 0.5)
```




```
## separate the plot into facets
flights %>%
  na.omit() %>%
  filter(Month == 1, DepTime >= 500) %>%
  ggplot(aes(x = DepTime, y = log(DepDelay + 20))) +
  geom_point(aes(color = Origin), alpha = 0.5) +
  facet_wrap(~ Origin)
```



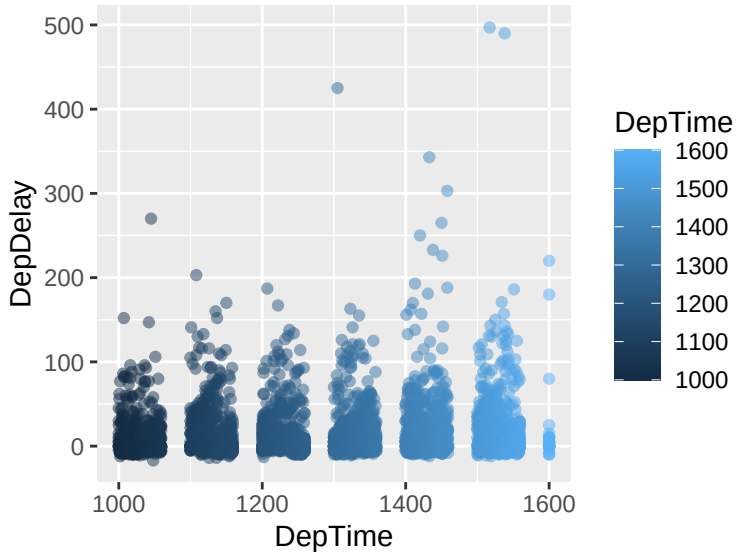

```
## add a trend line
flights %>%
  na.omit() %>%
  filter(Month == 1, DepTime >= 500) %>%
  ggplot(aes(x = DepTime, y = log(DepDelay + 20))) +
  geom_point(aes(color = Origin), alpha = 0.5) +
  facet_wrap(~ Origin) +
  geom_smooth(method = 'lm', se = F)
```



Your turn (7) Make a scatterplot of departure delay against departure time for flights in April departing between 10am and 4pm. Modify the plot by adding at least one other aesthetic or layer to help show whether the relationship seems to change depending on the day of the week.

```
# scatterplot of delay vs departure time
flights %>%
  na.omit() %>%
  filter(Month == 4, DepTime >= 1000 & DepTime <= 1600) %>%
  ggplot(aes(x = DepTime, y = DepDelay)) +
```

```
  geom_point(aes(color = DepTime), alpha = 0.5)
```




```
# delay vs departure time by day of week
flights %>%
  na.omit() %>%
  group_by(DayOfWeek) %>%
  filter(Month == 4, DepTime >= 1000 & DepTime <= 1600) %>%
  ggplot(aes(x = DepTime, y = DepDelay)) +
  facet_wrap(~ DayOfWeek) +
  geom_point(aes(color = DayOfWeek), alpha = 0.5)
```