Lab 1-2: Data Preprocessing & Distance and Similarity

PSTAT 131/231, Fall 2023

Learning Objectives

- Complete installation of `tidyverse`
- First steps using `tidyverse` - `filter()` - `select()` - `chaining()` - `mutate()` - `summarise()`
- Data prepocessing
- Distances - Euclidean distance - Manhattan distance
- Similarity - Correlation - Spearman rank Correlation

---

1. Preprocessing in the `tidyverse`

We will use the dataset called `hflights`. This dataset contains all flights departing from Houston airports IAH (George Bush Intercontinental) and HOU (Houston Hobby). The data comes from the Research and Innovation Technology Administration at the Bureau of Transportation statistics: hflights.

Make sure that you have installed the packages `hflights` and `tidyverse` before using them. (See Lab 1-1 for details on packages installation). The `tidyverse` includes many packages that will be utilized repeatedly in this class including `dplyr`, `tidyr`, `tibble` and `ggplot2`.

Please note that although basic R commands could also achieve these functionality, they are usually much harder/messier to write. `tidyverse` is usually considered as an `modern` way of using R for data analysis. Using `tidyverse` is not mandatory in homework, but is highly recommended since it will make things a lot easier.

Installing `tidyverse` will take a few minutes.

```
# install.packages("hflights")
# Installing tidyverse may take a couple minutes
# install.packages("tidyverse")

# Load packages
library(hflights)
library(tidyverse)

# Explore data
data(hflights)
flights = as_tibble(hflights) # convert to a tibble and print
flights
```

```
## # A tibble: 227,496 x 21
##     Year Month DayofMonth DayOfWeek DepTime ArrTime UniqueCarrier FlightNum
##    <int> <int>      <int>     <int>   <int>   <int> <chr>             <int>
##  1  2011     1          1         6    1400    1500 AA                  428
##  2  2011     1          2         7    1401    1501 AA                  428
##  3  2011     1          3         1    1352    1502 AA                  428
##  4  2011     1          4         2    1403    1513 AA                  428
##  5  2011     1          5         3    1405    1507 AA                  428
##  6  2011     1          6         4    1359    1503 AA                  428
```

```
##  7  2011     1          7         5    1359    1509 AA                  428
##  8  2011     1          8         6    1355    1454 AA                  428
##  9  2011     1          9         7    1443    1554 AA                  428
## 10  2011     1         10         1    1443    1553 AA                  428
## # i 227,486 more rows
## # i 13 more variables: TailNum <chr>, ActualElapsedTime <int>, AirTime <int>,
## #   ArrDelay <int>, DepDelay <int>, Origin <chr>, Dest <chr>, Distance <int>,
## #   TaxiIn <int>, TaxiOut <int>, Cancelled <int>, CancellationCode <chr>,
## #   Diverted <int>
```

Note that by default tibble only prints the first few rows and columns. Beneath the variable names (columns) it includes the data type

(a) `filter()`

`filter()` helps to return **rows** with matching conditions. Base R approach to filtering forces you to use the data frame's name repeatedly, yet `dplyr` approach is simpler to write and read.

The command structure (for all dplyr verbs):

- First argument is the data frame you're working on
- Return value is a data frame
- Nothing is modified in place

Note: `dplyr` generally does not preserve row names

View all flights on January 1^{st}:

```r
# Base R approach
flights[flights$Month==1 & flights$DayofMonth==1, ]

# dplyr approach
# Note: you can use comma or ampersand to represent AND condition
filter(flights, Month==1, DayofMonth==1)
```

View all flights carried by American Airlines *OR* United Airlines:

```r
# Base R approach
flights[flights$UniqueCarrier=="AA" | flights$UniqueCarrier=="UA", ]

# Use pipe for OR condition
filter(flights, UniqueCarrier=="AA" | UniqueCarrier=="UA")

# You can also use %in% operator for OR condition
filter(flights, UniqueCarrier %in% c("AA", "UA"))
```

(b) `select()`

`select()` is used to pick a set of **columns** by their names. Base R approach is awkward to type and to read. `dplyr` approach uses similar syntax to select columns, which is similar to a SELECT in SQL.

Suppose we would like check three variables, DepTime, ArrTime and FlightNum:

```r
# Base R approach to select DepTime, ArrTime, and FlightNum columns
flights[, c("DepTime", "ArrTime", "FlightNum")]

# dplyr approach
select(flights, DepTime, ArrTime, FlightNum)
```

You can use colon to select multiple columns, and use `contains()`, `starts_with()`, `ends_with()`, and `matches()` to match any columns by specifying the keywords. For example, we want to select simultaneously all the variables between Year and DayofMonth (inclusive), the variables containing the character string "Taxi" and "Delay", and the variables that start with the character string "Cancel":

```
# Select columns satisfying several conditions
select(flights, Year:DayofMonth, contains("Taxi"), contains("Delay"), starts_with("Cancel"))
```

To select all the columns except a specific column, use the subtraction operator (also known as negative indexing). For instance, select all columns except for those between Year and TailNum:

```
# Exclude columns
select(flights, -c(Year:TailNum))
```

(c) chaining or pipelining

The usual way to perform multiple operations in one line is by nesting them. Now we can write commands in a natural order by using the %>% infix operator (which can be pronounced as "then"). The main advantages of using %>% are the following:

- Chaining increases readability significantly when there are many commands
- Operator is automatically imported from the `magrittr` package
- Chaining can be used to replace nesting in R commands outside of `dplyr`

A toy example to illustrate that chaining reduces nesting commands:

```
# Create two vectors and calculate the Euclidean distance between them
x1 = 1:5; x2 = 2:6
# Base R  will do
sqrt(sum((x1-x2)^2))

# Chaining will do
(x1-x2)^2 %>% sum() %>% sqrt()
```

Note that the result on the left hand side of %>% will be passed as the first argument in the function on the right hand side of %>%.

Suppose we want to filter for all records with delays over 60 minutes and display the UniqueCarrier and DepDelay for these observations.

```
# Nesting method in dyplr to select UniqueCarrier and DepDelay columns and filter for
# delays over 60 minutes
filter(select(flights, UniqueCarrier, DepDelay), DepDelay > 60)

# Chaining method serving for the same purpose
flights %>%
    select(UniqueCarrier, DepDelay) %>%
    filter(DepDelay > 60)
```

(d) mutate()

`mutate()` is helpful for us to create new variables (features) that are functions of existing variables. Create a new column called Speed which is the ratio between Distance to AirTime.

```
# Base R approach to create a new variable Speed (in mph)
flights$Speed = flights$Distance / flights$AirTime*60
flights[, c("Distance", "AirTime", "Speed")]

# dplyr approach
```

```r
# Print the new variable Speed but does not save it in the original dataset
flights %>%
    select(Distance, AirTime) %>%
    mutate(Speed = Distance/AirTime*60)

# Save the variable Speed in the original dataset
flights = flights %>% mutate(Speed = Distance/AirTime*60)
```

Note: all dplyr functions only display the results for you to view but not save them in the original dataset. If you want to make changes in the original dataset, you have to put `dataset =` as illustrated by above example.

(e) `summarise()` (`summarize()`)

`summarise()` is primarily useful with data that has been grouped by one or more features. It reduces multiple values to a single (or more) value(s).

- `group_by()` creates the groups that will be operated on.
- `summarise()` uses the provided aggregation function to summarise each group.
- `summarise_each()` allows you to apply the same summary function to multiple columns at once.

Suppose we are interested in computing the average arrival delay to each destination:

```r
# Base R approaches
with(flights, tapply(ArrDelay, Dest, mean, na.rm=TRUE))
aggregate(ArrDelay ~ Dest, flights, mean)

# dplyr approach
# Create a table grouped by Dest, and then summarise each group by taking the mean of ArrDelay
flights %>%
    group_by(Dest) %>%
    summarise(avg_delay = mean(ArrDelay, na.rm=TRUE))
```

For each carrier, calculate the percentage of flights cancelled or diverted

```r
# dplyr approach
flights %>%
    group_by(UniqueCarrier) %>%
    summarise_each(funs(mean), Cancelled, Diverted)
```

(f). Summary

As seen above, we can use `dplyr` to perform the following data preprocessing procedures:

- Aggregation: examples are computing the mean, standard deviation etc.
- Feature subset selection: drop unnecessary variables
- Dimensionality reduction: delete redundant records
- Feature creation: create new variables

---

2. Visualization

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

```
# read in data to tibble format using functions from "readr" package
x = read_csv('online-shopping.csv')
```

```
## Rows: 3 Columns: 21
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr  (1): User
## dbl (20): item1, item2, item3, item4, item5, item6, item7, item8, item9, ite...
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
x
```

```
## # A tibble: 3 x 21
##   User  item1 item2 item3 item4 item5 item6 item7 item8 item9 item10 item11
##   <chr> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>  <dbl>  <dbl>
## 1 user1    86   123   123   123    77   118   121    82   120    115     80
## 2 user2     8    34    32    32     6    27    26    14    29     32     11
## 3 user3    36    34    35    36    34    34    35    36    36     36     35
## # i 9 more variables: item12 <dbl>, item13 <dbl>, item14 <dbl>, item15 <dbl>,
## #   item16 <dbl>, item17 <dbl>, item18 <dbl>, item19 <dbl>, item20 <dbl>
```

Note that `read_csv` returns a `tibble`, while `read.csv` returns a `data.frame`. We use `read_csv` here for better compatability with `tidyverse`.

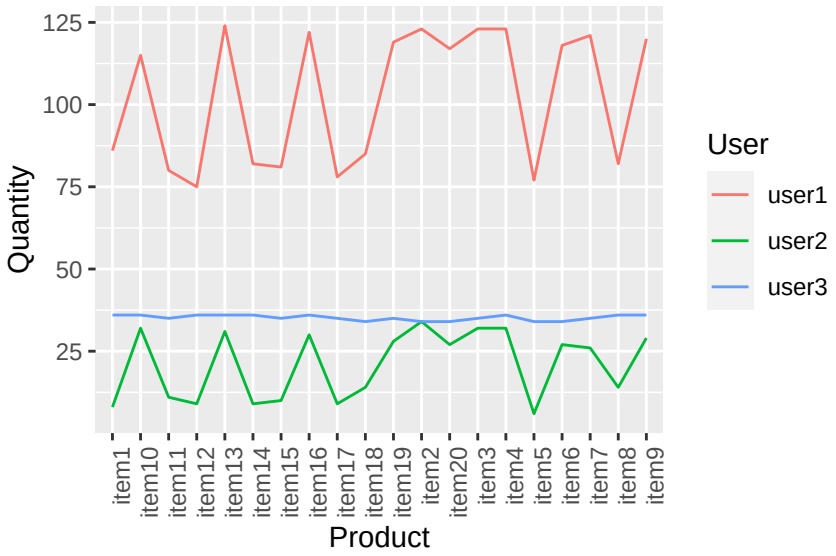
There are many situations where data is presented in a format that is not ready to dive straight to exploratory data analysis or to use a desired statistical method. The `tidyr` package provided with `tidyverse` provides useful functionality to avoid having to hack data around in a spreadsheet prior to import into R.

The `gather()` function takes wide-format data and gathers it into long-format data. The argument `key` specifies variable names to use in the molten data frame.

```
# ggplot2 should load automatically after loading tidyverse.  Otherwise use library(ggplot2)

# Plot the data
# Convert x transpose into a molten data frame
xgathered <- x %>% gather(key='Product', value='Quantity', -User)

# Use ggplot to expand a panel from xgathered; Use geom_line to add three curves representing
# the records of different users; add labels for each axis
xgathered %>% ggplot(aes(x=Product, y=Quantity)) +
    geom_line(aes(group=User, color=User)) +
    theme(axis.text.x = element_text(angle = 90, hjust = 1))
```



Note the use of `gather()` function to reshape data into a format appropriate for `ggplot`. We can convert back to a wide format using the `spread()` function. `gather` and `spread` are complements.

```
# use the spread function convert xgathered back to wide format (xspread will be identical to x)
xspread <- xgathered %>% spread(key="Product", value="Quantity")
xspread
```

```
## # A tibble: 3 x 21
##   User  item1 item10 item11 item12 item13 item14 item15 item16 item17 item18
##   <chr> <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
## 1 user1    86    115     80     75    124     82     81    122     78     85
## 2 user2     8     32     11      9     31      9     10     30      9     14
## 3 user3    36     36     35     36     36     36     35     36     35     34
## # i 10 more variables: item19 <dbl>, item2 <dbl>, item20 <dbl>, item3 <dbl>,
## #   item4 <dbl>, item5 <dbl>, item6 <dbl>, item7 <dbl>, item8 <dbl>,
## #   item9 <dbl>
```

Credit: the original code is from http://rpubs.com/justmarkham/dplyr-tutorial.