Lab 01: Data Preprocessing & Distance and Similarity

PSTAT 131/231, Spring 2018

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 Lab01 for details on packages installation). The `tidyverse` includes many packages that will be utilized repeatedly in this class including `dplyr`, `tidyr`, `tibble` and `ggplot2`. Installing `tidyverse` will a few minutes.

```
# Load packages
# install.packages("hflights")
# Installing tidyverse may take a couple minutes
# install.packages("tidyverse")
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
##  * <int> <int>      <int>     <int>   <int>   <int>         <chr>
##  1  2011     1          1         6    1400    1500            AA
##  2  2011     1          2         7    1401    1501            AA
##  3  2011     1          3         1    1352    1502            AA
##  4  2011     1          4         2    1403    1513            AA
##  5  2011     1          5         3    1405    1507            AA
##  6  2011     1          6         4    1359    1503            AA
```

```
## 7  2011     1          7         5     1359    1509             AA
## 8  2011     1          8         6     1355    1454             AA
## 9  2011     1          9         7     1443    1554             AA
## 10  2011     1         10         1     1443    1553             AA
## # ... with 227,486 more rows, and 14 more variables: FlightNum <int>,
## #   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 he 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}:

```
# 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:

```
# 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:

```
# 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()
```

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
# 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:

```
# 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

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

```
## `summarise_each()` is deprecated.
## Use `summarise_all()`, `summarise_at()` or `summarise_if()` instead.
## To map `funs` over a selection of variables, use `summarise_at()`
```

(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. Distance and Similarity Metrics

(a) Some description of dataset

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

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

```
## Parsed with column specification:
## cols(
##   .default = col_integer(),
##   User = col_character()
## )
## See spec(...) for full column specifications.
```

```r
x
```

```
## # A tibble: 3 x 21
##    User item1 item2 item3 item4 item5 item6 item7 item8 item9 item10
##   <chr> <int> <int> <int> <int> <int> <int> <int> <int> <int>  <int>
## 1 user1   119    73   114    86    82    78    74    78   124     87
## 2 user2    28     8    30    14    12    10     9    10    32      9
## 3 user3    35    35    34    35    35    34    36    36    36     35
## # ... with 10 more variables: item11 <int>, item12 <int>, item13 <int>,
## #   item14 <int>, item15 <int>, item16 <int>, item17 <int>, item18 <int>,
## #   item19 <int>, item20 <int>
```

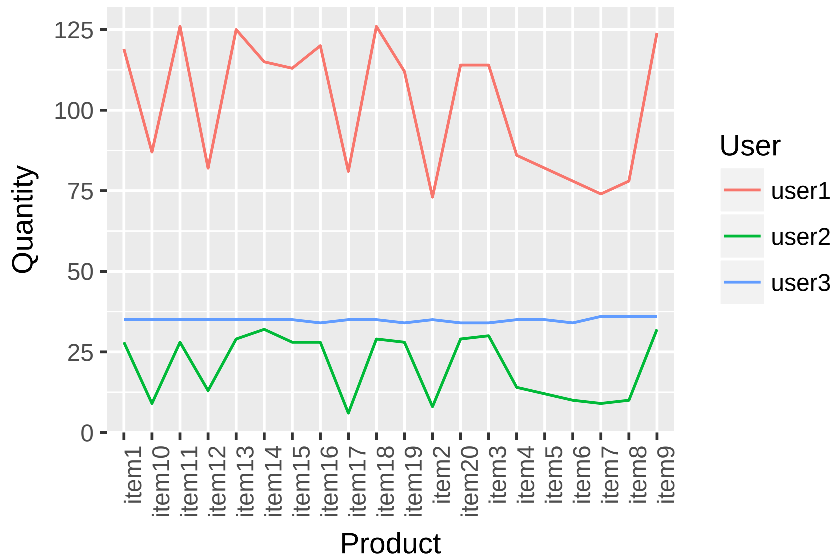
Here are many situations where data is presented in a format that is not ready to dive straight to exploratory data analysis or to use a desired statistical method. The `tidyr` package provided with `tidyverse` provides useful functionality to avoid having to hack data around in a spreadsheet prior to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

```r
# 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
## * <chr> <int>  <int>  <int>  <int>  <int>  <int>  <int>  <int>  <int>
## 1 user1   119     87    126     82    125    115    113    120     81
## 2 user2    28      9     28     13     29     32     28     28      6
## 3 user3    35     35     35     35     35     35     35     34     35
## # ... with 11 more variables: item18 <int>, item19 <int>, item2 <int>,
## #   item20 <int>, item3 <int>, item4 <int>, item5 <int>, item6 <int>,
## #   item7 <int>, item8 <int>, item9 <int>
```

(b) Distances

Various distances can be computed by `dist()` function. This function computes distances between rows of input matrix by user-specified distance measure. Here distances are those between rows of a data matrix. `diag=TRUE` is specified to display diagonal elements in the distance matrix.

(b) (i) Euclidean distance:

Compute L_2 distances among the three users:

```
# L_2 distance between rows
dist.l2 = dist(x, method="euclidean", diag=TRUE)
```

```
## Warning in dist(x, method = "euclidean", diag = TRUE): NAs introduced by
## coercion
```

```
dist.l2
```

```
##           1         2         3
## 1   0.00000
## 2 373.89577   0.00000
```

```
## 3 318.56812  79.62349   0.00000
```

Note that `user2` and `user3` are closest.

(b) (ii) Manhattan distance:

Compute L_1 distances among the three users:

```r
# L_1 distance between rows
dist.l1 = dist(x, method="manhattan", diag=FALSE)
```

```
## Warning in dist(x, method = "manhattan", diag = FALSE): NAs introduced by
## coercion
```

```r
dist.l1
```

```
##         1       2
## 2 1697.85        
## 3 1397.55  300.30
```

Note that `user2` and `user3` are closest.

(c) Similarities

(c) (i) Correlation

We see that general buying pattern are similar between `user1` and `user2`. Correlation is a measure of "similarity" in that fluctuation similarity is quantified. Relative magnitudes do not matter since data is mean centered and variance scaled to one.

Computing correlation between rows of `x`, we get

```r
# By taking transpose of xmat, we're computing the correlation between rows of a
# data matrix -- this is important!

xmat <- x %>% select(-User)
xmat
```

```
## # A tibble: 3 x 20
##   item1 item2 item3 item4 item5 item6 item7 item8 item9 item10 item11
##   <int> <int> <int> <int> <int> <int> <int> <int> <int>  <int>  <int>
## 1   119    73   114    86    82    78    74    78   124     87    126
## 2    28     8    30    14    12    10     9    10    32      9     28
## 3    35    35    34    35    35    34    36    36    36     35     35
## # ... with 9 more variables: item12 <int>, item13 <int>, item14 <int>,
## #   item15 <int>, item16 <int>, item17 <int>, item18 <int>, item19 <int>,
## #   item20 <int>
```

```r
sim.cor = cor(t(xmat))
sim.cor
```

```
##            [,1]       [,2]       [,3]
## [1,]  1.0000000  0.9606775 -0.2373478
## [2,]  0.9606775  1.0000000 -0.2712371
## [3,] -0.2373478 -0.2712371  1.0000000
```

Note that similarity between `user1` and `user2` is high: i.e. close to 1.

Furthermore, a similarity metrics are opposite of what distance metric does: i.e. dissimilarity is a generalization of distance.

```r
dist.cor = 1-cor(t(xmat))
dist.cor
```

```
##            [,1]       [,2]     [,3]
## [1,] 0.00000000 0.03932246 1.237348
## [2,] 0.03932246 0.00000000 1.271237
## [3,] 1.23734785 1.27123709 0.000000
```

Using 1 minus correlation between users buying preferences, we computed a different dissimilarity metric.

(c) (ii) Spearman rank correlation (Spearman ρ statistic)

If attributes of your data are ordinal, difference between two measurements is not meaningful. In this case, data is converted to ranks. Ranks of elements in a vector are ordering of the element if you were to sort the vector from smallest to largest: e.g. ranks of `c(4, 2, 30)` would be `c(2, 1, 3)`: e.g. try `rank(c(4, 2, 30))`.

```r
rank(c(4,2, 30))
```

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

Spearman rank correlation is simply the correlation of the ranks of two vectors. Spearman rank correlation is more appropriate when your measurements are ordinal.

Your turn

Calculate the Spearman rank correlation matrix for the three users. (Hint: read the help file of `cor`)

```r
# Codes start here
```

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