Wrangling Data

BSDS 100 - Intro to Data Science with R

Outline

- Tidying Data
  - The `tidyr` package
  - The `dplyr` package
- Review: Piping and the magrittr package

Reference: Chapter 12 in *R for Data Science* book

Additionally Complex Analysis

- In many cases, data doesn't come in the structure we'd like it to for easy analysis and plotting
- It is not uncommon that a majority of the time spent visualizing data is not writing the visualization code, e.g, `ggplot`, but rather *restructuring* data (mainly data frames) so as to be able to visualize the data

Additionally Complex Analysis

- There are functions in base `R` such as `aggregate()` and `merge()` that can help with this, as well as functions such as `melt()` and `cast()` from the `reshape2` package
- Here, we will examine the use of the `tidyr` and `dplyr` packages

Tidy Data

- Data should always be tidy before you begin to work with it!
- All data should be organized such that
  1. **each column is a variable**
  2. **each row is an observation**
- `tidyr` provides four main functions for tidying messy data
  1. `pivot_longer()`
  2. `pivot_wider()`
  3. `separate()`
  4. `unite()`

The `pivot_longer()` function

- `pivot_longer()` takes multiple columns and pivot_longers them into key-value pairs
- It makes *wide* data *longer*
- The `melt()` function in the `reshape2` package is equivalent to this function

An Example

- In an experiment, you compare the *normal* weekly sales to sales when using *fancy* lighting and signage in a grocery store in three different cities
- The data may be stored as follows

```
> myDataFrame = data.frame(
    City = c("Austin", "Georgia", "Vancouver"),
    Fancy = c(35000, 43000, 106000),
    Normal = c(30000, 44000, 77000)
    )


> myDataFrame
       City  Fancy Normal
1    Austin  35000  30000
2   Georgia  43000  44000
3 Vancouver 106000  77000
```

Example, continued

- **Goal**: Create a nice ggplot comparing the `Fancy` sales with `normal` sales from each location.
- In this case, we cannot directly use the data frame as it is formatted
- We can use the `pivot_longer()` function to reshape the data frame into two columns so that we can directly compare.

Example, continued

```
library(tidyr)
library(dplyr)
?pivot_longer

my_tidy_df1 = pivot_longer(my_df1,
cols = c(Fancy, Normal),
names_to = "lightSign",
values_to = "Sales")

my_tidy_df1
           City lightSign  Sales
    1    Austin     Fancy  35000
    2   Georgia     Fancy  43000
    3 Vancouver     Fancy 106000
    4    Austin    Normal  30000
    5   Georgia    Normal  44000
    6 Vancouver    Normal  77000
```

Now plot with ggplot

```
> library(scales)

ggplot(my_tidy_df1 ,
    aes(x = City, y = Sales, group = lightSign, colour = lightSign))
+ geom_line()
+ geom_point()
  + scale_y_continuous(labels = comma)
  + ylab("Sales")
  + xlab("\n City")
  + theme(legend.title = element_blank())
```

The Result

100,000
75,000
50,000
25,000
0
Sales
Austin
Georgia
Vancouver
City
Fancy
Normal

The `separate()` function

- Sometimes variables are clumped together in a single column but we'd like to *separate* them
- The `separate()` function allows you to parse them
- The `extract()` function works similarly but uses regular expressions groups instead of a splitting pattern or position
- The opposite of `separate()` is `unite()`

Example

- There is an experiment which measures how much time (in hrs) a person has spent on their mobile phones at specific times in the day. The experiment controls for the mobile operating system (iOS or Android). Each subject has two readings taken taken at work and at home in the AM and PM.
- Goal Generate a plot with the mean time spent on mobile device (y), time of day (x) by location (work/home) and operating system (iOS/Android)

Example: Generating the Data

```
> set.seed(1979)

 (myDataFrame = data.frame(
  uniqueId = 1:4,
  treatment = sample(rep(c('iOS', 'Android'), each = 2)),
  work_am = runif(4, 0, 1),
  home_am = runif(4, 0, 1),
  work_pm = runif(4, 1, 2),
  home_pm = runif(4, 1, 2)
))
```

```
  uniqueId treatment   work_am    home_am   work_pm   home_pm
1        1   Android 0.1655928 0.47930296 1.912491 1.068928
2        2       iOS 0.2641590 0.36626685 1.276391 1.317642
3        3       iOS 0.6108628 0.34724530 1.011851 1.572617
4        4   Android 0.3877454 0.06951258 1.222138 1.759057
```

pivot_longering the data first

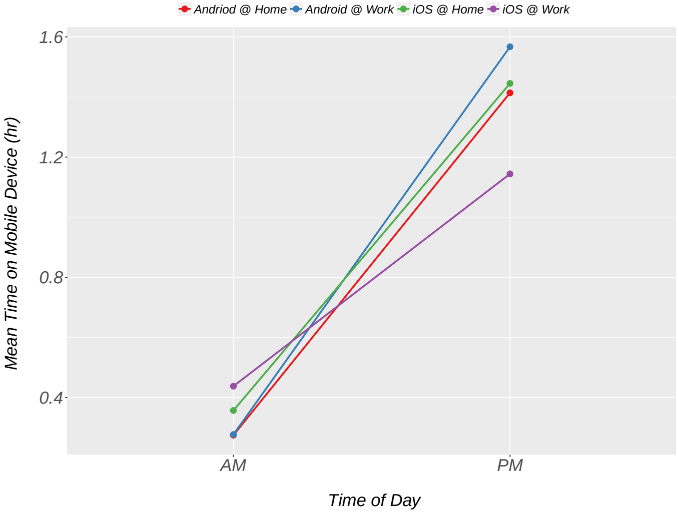
- The raw data from the previous slide is *not* tidy, i.e., each column should be a variable and each row should be an observation

  STEP 1: Make the data tidy using `pivot_longer()`

Now separating the data

STEP 2: Split `sample` into location (work, home) and time of day (AM, PM) using `separate()`

The Resulting Plot



The `pivot_wider()` function

- The opposite of the `pivot_longer()` function is the `pivot_wider()` function, which takes *long* data and makes it *wide*
- Recall the example from a few slides ago, where *wide* data was made *long*

```
my_tidy_df1 = pivot_longer(my_df1,
cols = c(Fancy, Normal),
names_to = "lightSign",
values_to = "Sales")
```

```
       City lightSign  Sales
1    Austin     Fancy  35000
2   Georgia     Fancy  43000
3 Vancouver     Fancy 106000
4    Austin    Normal  30000
```

The `pivot_wider()` function

- Undo what was done to the data and return it to it's original form using `pivot_wider()`

```
> (pivot_wider(my_tidy_df1,
  names_from = lightSign,
  values_from = Sales))
       City  Fancy Normal
1    Austin  35000  30000
2   Georgia  43000  44000
3 Vancouver 106000  77000
```

A Brief Review: `magrittr`

- `magrittr` is package which allows for the use of the piping operator `%>%`
- There are various piping operators, but for the purposes of this brief introduction, the focus will be solely on `%>%`; you are encouraged to read the `magrittr` documentation to learn more about the different piping operators
- Piping allows for a significant reduction in typing, as well as making code intuitive to external code reviewers

The `magrittr` package

- Piping is very useful when using data manipulation packages such `tidyr` and `dplyr`
- Flow: data is piped into a function and the data argument in the function can be dropped
  - Without the piping operator: `mean(x)`
  - With the piping operator: `x %>% mean()`
- Piping operator keystoke: `SHIFT + COMMAND + M`

Piping: For Readability and Organization

- Imagine taking the sum of the square root of a vector of numbers X. We could simply write

```
> sum(sqrt(X))
```

- But, we have to read this from the inside out, which can seem a little unintuitive.
- Piping allows us to write this more naturally:

```
> X %>% sqrt() %>% sum()
```

- Obviously things can get much more complicated, but this can be really useful for complicated flow

Another example

```
# Without magrittr
my_tidy_df1 = pivot_longer(my_df1,
cols = c(Fancy, Normal),
names_to = "lightSign",
values_to = "Sales")
ggplot(myTidyDataFrame,
       aes(x = City, y = Sales, group = lightSign, colour = lightSign))
   + geom_line(size = 1)
#-----------------------
# With magrittr
library(magrittr)
myDataFrame %>%
  pivot_longer(cols = c(Fancy, Normal),
    names_to = "lightSign",
    values_to = "Sales") %>%
  ggplot(aes(x = City, y = Sales, group = lightSign, colour = lightSign))
    geom_line(size = 1)
```

Manipulation of Data Frames with `dplyr`

- `dplyr` is a package which provides a set of tools for efficiently manipulating datasets in `R`
- `dplyr` is designed to have a set of functions defined by verbs
  1. `filter()`: keep rows with matching criteria
  2. `select()`: select columns by name
  3. `arrange()`: reorder rows
  4. `mutate()`: add new variables
  5. `summarise()`: reduce variables to values
- How it works
  1. First argument is a data frame
  2. Subsequent arguments say what to do with data frame
  3. Always returns a data frame

Using the `filter()` function

- Load `flights.csv`
- Find all flights
  1. in May
  2. that were not delayed
  3. that departed between 4pm and 5pm
  4. to SFO

Solutions

1. `flights %>% filter(date < "2011-06-01" & date >= "2011-05-01")`
2. `flights %>% filter(arr_delay <= 0 & dep_delay <= 0)`
3. `flights %>% filter(dep >= 1600 & dep <= 1700)`
4. `flights %>% filter(dest == "SFO")`

Example of the `select()` function

```
myDF = data.frame(myNum = c(1,2,3,4,5),
    myColor = c("blue", "yellow", "yellow", "blue", "blue"))

    # select a column from a data frame
    my DF %>% select()

    ### using flights.csv data

    flights %>% select(arr_delay:dep_delay)

    flights %>% select(ends_with("delay"))

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

Example of the `arrange()` function

```
#arrange demo
head(flights)
?arrange
demo_df %>% arrange(color)
demo_df %>% arrange(-num)
flights %>% arrange(dist)
flights %>% arrange(dest)
```

Example of the `mutate()` function

```
demo_df %>% mutate(squared = num^2)

flights = flights %>% mutate(dep_hr = dep %>%
str_pad(width = 4, pad = "0") %>%
substr(start = 1, stop = 2))
```