Coursera - Data Science & R Course

Daniel O.

15/12/2020

Types of Data Science Questions

Descriptive

Goal: To describe or summarize a set of data

- Early analysis when receiving new data
- Generate simple summaries about the samples and their measurements
- **Not** for generalizing the results of the analysis to a larger population

Exploratory

Goal: To examine the data and find relationships that weren't previously known

- Explore how different variables might be related
- Useful for discovering new connections
- Help to formulate hypotheses and drive the design of future studies and data collection

Inferential

Goal: Use a relatively small sample of data to say something about the population at large

- Provide your estimate of the variable for the population and provide your uncertainty about your estimate
- Ability to accurately infer information about the larger population depends heavily on sampling scheme

Predictive

Goal: Use current and historical data to make predictions about future data

- Accuracy in predictions is dependent on measuring the right variables
- Many ways to build up prediction models with some being better or worse for specific cases

Causal

Goal: See what happens to one variable when we manipulate another variable

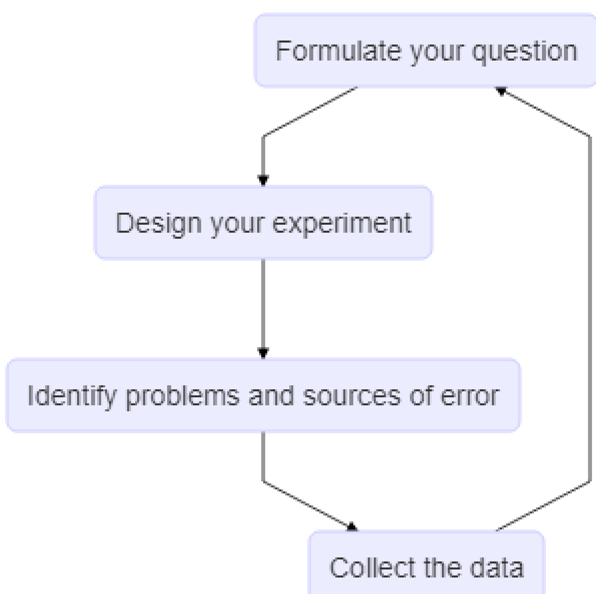
- Gold standard in data analysis
- Often applied to the results on randomized studies that were designed to identify causation
- Usually analyzed in aggregate and observed relationships are usually average effects

Mechanistic

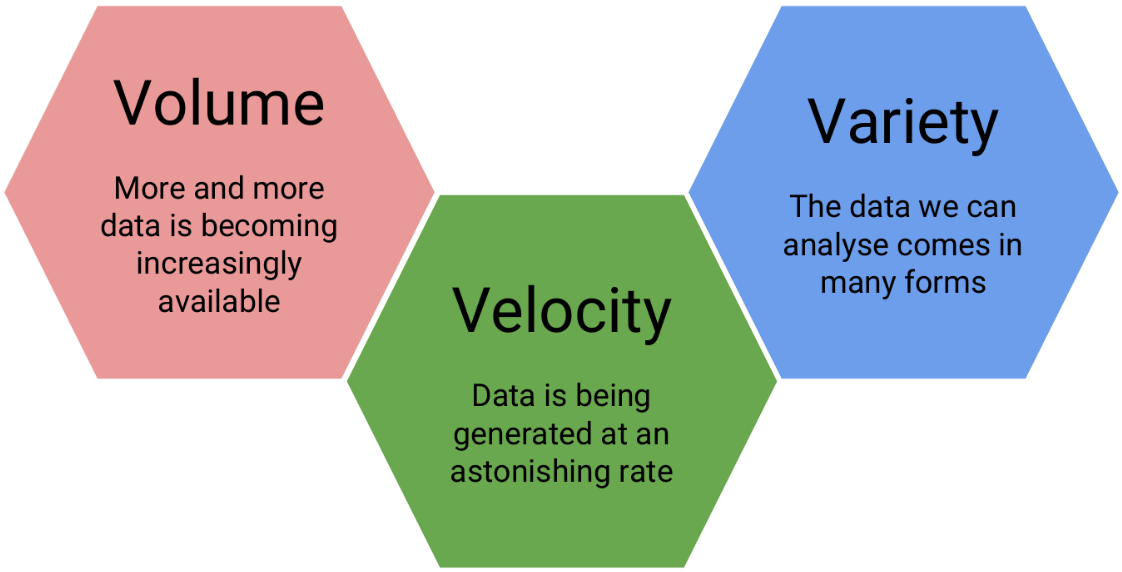
Goal: Understand the exact changes in variables that lead to exact changes in other variables

- Applied to simple situations or those that are nicely modeled by deterministic equations
- Commonly applied to physical or engineering sciences
- Often, the only noise in the data is measurement error

Experimental design



Big Data



Getting started with R

Directories

- getwd()
- setwd()
- dir()
- dir.create()
- file.create()
- file.exists()
- file.infor()
- file.path()
- file.rename()
- file.copy()

Looking at data

`ls`: List the variables in the environment

`nrow` & `ncol`: Number of rows/columns in a dataframe (`dim` for both)

`object.size`: How much space the object occupies in memory

`names`: Names of columns (variables)

`head` & `tail`: Preview the top/bottom of the dataset

`summary`: Provides different output for each variable, depending on its class. For **numeric** data summary() displays the minimum, 1st quartile, median, mean, 3rd quartile, and maximum. These values help us understand how the data are distributed. For **categorical** variables (called 'factor' variables in R), summary() displays the number of times each value (or 'level') occurs in the data.

`str`: Shows the structure of the data.

Control Structures - if/else

if(condition) {

<do something>

} else("condition2") {

<do something else>

}

–

if(condition) {

<do something>

{ else if(condition2) {

<do something different>

} else {

<do something else>

}

Control Structures - for loop

for(i in vect){

<function>[i]

}

Control Structures - while

while(finite condition){

`<do something>`

}

Control Structures - repeat, next, break, return

`repeat` is a construct that basically initiates an infinite loop. The only way to exit a `repeat` loop is to call `break`

`next` is basically used in any time of looping construct when you want to skip an iteration.

`return` signals that a function should exit and and return a given value

Functions

Basic format:

f <- function(x, y =

){

x+y

}

User-defined binary operators have the following syntax:

```
%[whatever]%
```

where [whatever] represents any valid variable name.

Dates and Time

Dates are represented by the `Date` class and times are represented by the `POSIXct` and `POSIXlt` classes. Internally, dates are stored as the number of days since 1970-01-01 and times are stored as either the number of seconds since 1970-01-01 (for `POSIXct`) or a list of seconds, minutes, hours, etc. (for `POSIXlt`).

`strptime()` converts character vectors to `POSIXlt`.

If you want more control over the units when finding the above difference in times, you can use `difftime()`, which allows you to specify a 'units' parameter.

Loop functions

‘The Split-Apply-Combine Strategy for Data Analysis’

`lapply`: Loop over a list and evaluate a function on each element » `lapply(data, fun)`

`sapply`: Same as `lapply` but try to simplify the result

`apply`: Apply the function over the margins of an array » `apply(data, dimension, fun)`

`mapply`: Multivariate function of `lapply` *e.g.* `list(rep(1,4), rep(2,3), rep(3,2), rep(4,1))` can be done with `mapply(rep, 1:4, 4:1)`

`tapply`: Apply a function over the subsets of a vector » `tapply(data, factor, fun)`

`split`: Takes a vector and splits it according to a factor(similarly to `tapply` but without applying any type of summary statistics)

`sprintf` originally comes from C and all formatting rules are taken from it as well

e.g. sprintf(“%03d”, 7) “%03d” is a formatting string, which specifies how 7 will be printed.

d stands for decimal integer (not double!), so it says there will be no floating point or anything like that, just a regular integer.

3 shows how many digits will the printed number have. More precisely, the number will take at least 3 digits: 7 will be ___7 (with spaces instead of underscores), but 1000 will remain 1000, as there is no way to write this number with just 3 digits.

0 before 3 shows that leading spaces should be replaced by zeroes. Try experimenting with sprintf(“%+3d”, 7), sprintf(“%-3d”, 7) to see other possible modifiers (they are called flags).

Debbuging tools

Something’s wrong

`message`: Generic notification; function execution continues

`warning`: Something went wrong but is not fatal; execution of the function is not stopped

`error`: Fatal problem has occurred; execution is halted; produced by the stop function

`condition`: A generic concept indicating that something can occur; programmers create their own conditions

Primary Debugging Tools

`traceback`: Prints out the function call stack after an error occurs and allows to know in which function the error occurred

`recover`: Allows you to modify the error default behavior so that you can browse the function whenever an error occurs

`debug`: Flags a function for debug mode which allows you to step through the function one expression/line at a time

`browser`: Suspends the execution of a function wherever it is called and puts the function in debug mode

`trace`: Allows to insert debugging mode into a function at specific places without altering the function code

Simulation - Generating Random Numbers

`sample`: Samples from vector

Each probability distribution in R has an r*** function (random number generation), a d*** function (density), a p*** (cumulative distribution), and q*** (quantile function).

Getting data

download.file()

read.**file type**()

Manipulating data with dplyr

The first step of working with data in dplyr is to load the data into what the package authors call a 'data frame tibble', using `tibble::as_tibble()` The main advantage to using a tbl_df over a regular data frame is the printing.

The dplyr philosophy is to have small functions that each do one thing well. Specifically, dplyr supplies five 'verbs' that cover most fundamental data manipulation tasks: `select()`, `filter()`, `arrange()`, `mutate()`, and `summarize()`.

select

Select (and optionally rename) variables (columns) in a data frame, using a concise mini-language that makes it easy to refer to variables based on their name (e.g. a:f selects all columns from a on the left to f on the right). You can also use predicate functions like is.numeric to select variables based on their properties.

filter

The `filter()` function is used to subset a data frame, retaining all rows that satisfy your conditions. To be retained, the row must produce a value of TRUE for all conditions. Note that when a condition evaluates to NA the row will be dropped, unlike base subsetting with [.

arrange

`arrange()` orders the rows of a data frame by the values of selected columns.

Unlike other dplyr verbs, `arrange()` largely ignores grouping; you need to explicitly mention grouping variables (or use .by_group = TRUE) in order to group by them, and functions of variables are evaluated once per data frame, not once per group.

mutate & transmute

`mutate()` adds new variables and **preserves** existing ones; `transmute()` adds new variables and **drops** existing ones. New variables overwrite existing variables of the same name. Variables can be removed by setting their value to NULL.

summarize

`summarize()` creates a new data frame. It will have one (or more) rows for each combination of grouping variables; if there are no grouping variables, the output will have a single row summarizing all in the input. It will contain one column for each grouping variable and one column for each of the summary statistics that you have specified.

Grouping and chaining with dplyr

Group by one or more variables

`group_by()` takes an existing tbl and converts it into a grouped tbl where operations are performed "by group". `ungroup()` removes grouping.

Grouping data by variable, allows to summarize data by each value of that variable.

e.g.

```
library(dplyr)
data("iris")
iris_tbl <- tibble(iris)

by_species <- group_by(iris, Species)

plant_sum <- summarize(by_species,
                      count = n(),
                      unique = n_distinct(Species),
                      avg_Petal.Width = mean(Petal.Width))
```

```
plant_sum
```

```
## # A tibble: 3 x 4
##   Species    count unique avg_Petal.Width
##   <fct>      <int>  <int>           <dbl>
## 1 setosa        50      1           0.246
## 2 versicolor    50      1           1.33
## 3 virginica     50      1           2.03
```

Chaining

Chaining makes use of the `%>%` (*pipe*) operator. With this operator, you can chain together different commands without having to provide the base data/vector argument for each one.

e.g.

```
myresult <-
iris_tbl %>% group_by(Species) %>%
  summarize(
            count = n(),
            unique = n_distinct(Species),
            avg_Petal.Width = mean(Petal.Width)) %>%
  filter(avg_Petal.Width > 0) %>%
  arrange(desc(avg_Petal.Width))
```

```
myresult
```

```
## # A tibble: 3 x 4
##   Species    count unique avg_Petal.Width
##   <fct>      <int>  <int>           <dbl>
## 1 virginica     50      1           2.03
## 2 versicolor    50      1           1.33
## 3 setosa        50      1           0.246
```

*** PART 3 OF SWIRL***