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

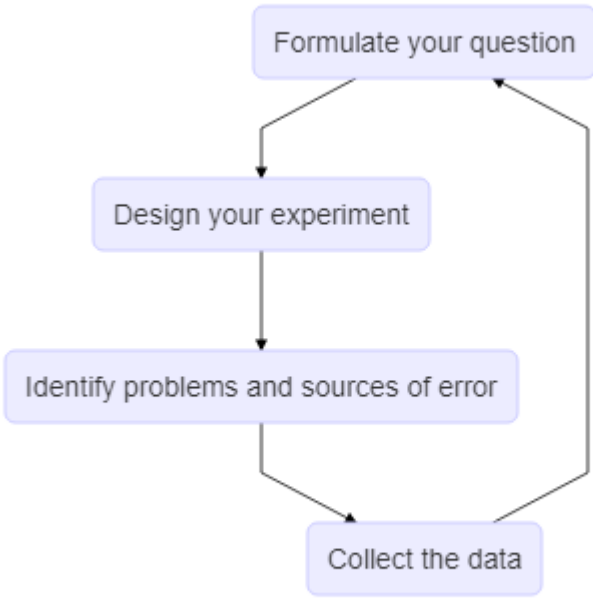
- Gold standard in data analysis
- Often applied to the results on randomized studies that were designed to identify causation
- Usually analyzed in aggregate and observed relationships are usually average effects

Mechanistic

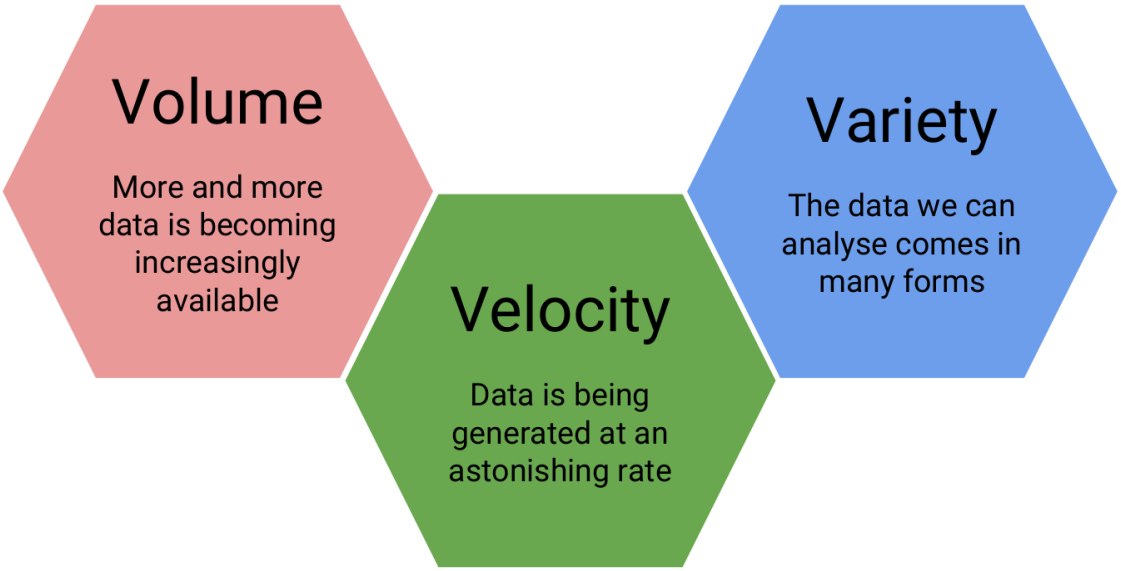
Goal: Understand the exact changes in variables that lead to exact changes in other variables

- 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