Creating the large multivariate dataset

Julian

Contents

Executive summary

The goal of creating the multivariate dataset was to:
- demonstrate various Python commands
- how to potentially import and inspect a large dataset
- creating statistical models for large datasets
- how to author a report using Sweave or R Markdown

Steps

- create subject ids
- set a random seed (makes tracking what was done harder)
- randomly creating subject ages
- simulating subject responses
- generating reaction times and introducing real data
- assigning attempts
- assigning subjects to groups
- adding trial, token phase data in long format

The source code (Python) is contained in the Day 3 directory.

"DO IT?"
DAN, I'M NOT A REPUBLIC SERIAL VILLAIN. DO YOU SERIOUSLY THINK I'D EXPLAIN MY MASTER-STROKE IF THERE REMAINED THE SLIGHTEST CHANCE OF YOU AFFECTING ITS OUTCOME?
I DID IT THIRTY-FIVE MINUTES AGO.

Execution

This document was created in R Markdown. When used in conjunction with `knitr`, you can use other engines such as Python. So you could take this file, change the chunk options to `eval  = TRUE` and replicate the results without having to do so in Python. It much easier however, to examine the source file posted.

Data generation

Import `NumPy`, `pandas`, set random seed

```
import numpy as np
import pandas as pd
np.random.seed(2340875)
```

Create `Subject`, `Age`, `Group` vectors

```
subj_ids = ['subj-' + str(num) for num in range(1, 103)]
groups = ['treatment-1', 'treatment-2', 'control']
age_vec = np.random.uniform(low = 18, high = 52, size = len(subj_ids))
```

Create experimental `DataFrame`

```
cols = ['Subject', 'Age', 'Group', 'Token','Attempts', 'Correct', 'RT']
sample_data = pd.DataFrame(columns = cols)
sample_data['Subject'] = subj_ids * 20 * 40
sample_data.head()
```

Assign subjects to an age

```
age_dict = {}
for subj, age in zip(subj_ids, age_vec):
    age_dict[subj] = age
```

Simulating correct responses

Much slower than generating vector, but harder to figure out what was done

```
correctness = np.random.uniform(low = 0, high = 1, size = sample_data.shape[0])

for idx, prob in zip(range(sample_data.shape[0]), correctness):
    sample_data.ix[idx, 'Correct'] = np.random.binomial(n = 1, p = prob)
```

Generating reaction times

Once again, very slow but an extra hurdle to jump through

```
for idx in range(sample_data.shape[0]):
    sample_data.ix[idx, 'RT'] = \
      np.random.lognormal(mean = 6.0, sigma = 0.5) + 5 + \
      np.random.uniform(low = 10, high = 100) + \
      np.random.normal(loc = 70, scale = 15)
```

Replacing values selectively with real data

```
real_data = pd.read_csv(data_dir + data_file)

real_rt = real_data.absRT

for idx in range(sample_data.shape[0]):
    if sample_data.ix[idx, 'RT'] > 600:
        sample_data.ix[idx, 'RT'] = np.random.choice(real_rt)
    else:
        continue
```

Creating timed out values

```
# replace values > 2200 with 2200
for idx in range(sample_data.shape[0]):
    if sample_data.ix[idx, 'RT'] > 2200:
        sample_data.ix[idx, 'RT'] = 2200
    else:
        continue
```

Mapping ages and attempts

```
# map ages
sample_data['Age'] = sample_data.Subject.map(age_dict)

for idx in range(sample_data.shape[0]):
    sample_data.ix[idx, 'Attempts'] = np.random.random_integers(low = 1, high = 15)
```

Asigning to groups

```
group_dict = {}
for subj, idx in zip(subj_ids, range(1, 103)):
    if idx <= 34:
        group_dict[subj] = groups[0]
    elif idx > 34 and idx <= 68:
        group_dict[subj] = groups[2]
    elif idx > 68:
        group_dict[subj] = groups[1]

sample_data['Group'] = sample_data.Subject.map(group_dict)
```

Sorting data into long format

```
sample_data = sample_data.sort(['Subject', 'Age', 'Group', 'Token'])
```

Changing data to appropriate types

```
sample_data.Attempts = np.array(sample_data.Attempts, dtype = 'float64')
sample_data.Correct = np.array(sample_data.Correct, dtype = 'float64')
sample_data.RT = np.array(sample_data.RT, dtype = 'float64')
```

Add in trials

```
trials = range(1, 21)
sample_data['Trial'] = np.tile(trials, 40 * len(subj_ids))
```

Generate tokens

```
tokens = range(1, 21)
tokens_repeated = np.repeat(tokens, 20)
sample_data['Token'] = np.tile(tokens_repeated, 2 * len(subj_ids))
```

Generate phases

```
phases = np.repeat([1, 2], 400)
sample_data['Phase'] = np.tile(phases, 102)
```

Export to CSV

```
target_dir = '/Users/julian/Github/reproducible-research/Day-2/datasets/'
filename = 'multivariate-exp-data.csv'
sample_data.to_csv(target_dir + filename)
```

Appendix: vectorized functions

The data were created in a rather slow, tedious manner, mostly to introduce control flow and to be explicit about the steps that were taken. The process can be speeded up greatly however, by taking advantage of the fact that `NumPy` and the `SciPy` stack are intended to be fast and not very memory-intensive - hence the lack of deep copies and a lot of modifying in place.

Below are examples of using vectorized functions instead of `for` loops to manipulate data.

Simulating correctness

`NumPy` is designed to be fast and (I believe) all of the function are vectorized. The code to generate correctness, could have just used the vector of probabilities:

```
sample_data['Correct'] = np.random.binomial(n = 1, p = correctness)
```

Simulating reaction time

For the reaction times, just stipulate the size of the final vector:

```
sample_data['RT'] = \
  np.random.lognormal(mean = 6.0, sigma = 0.5, size = 81600) + 5 + \
      np.random.uniform(low = 10, high = 100, size = 81600) + \
      np.random.normal(loc = 70, scale = 15, size = 81600)
```

Replacing simulated values with real values

The function `np.where` could have been used to replace values:

```
np.where(sample_data.RT > 600, np.random.choice(real_rt), sample_data.RT)
```

Creating timed out values

In the same vein as the above, create the timed out values

```
np.where(sample_data.RT > 2200, 2200, sample_data.RT)
```