Subject Model Simulation

Maciej Jankowski

I. Introduction

NOWADAYS, as video streaming and similar services consume our time and bandwidth. Both providers and clients want to ensure that those valuable resources aren't wasted. That is exactly why QoS's brother - *Quality of Experience* (QoE) - is crucial. Testing software is relatively easy - one just (hires interns and) writes lots of unit tests that are simple and repetitive. In more sophisticated projects *Quality Assurance* (QA) engineers are needed in order to break everything to pieces and find so-called edge-cases. All of the above is quite simple. This simplicity disappears when humans - their opinions in particular - are involved. That is the case with assessing *Quality of Experience*. Testing sessions are long and place-dependent - they require time, money and space.

Since tests have to provide unbiased results and both money and time are limited resources a compromise needs to be reached. Based on previous experiments in the field, some distributions and parameters were acquired. Therefore simulations can be ran to estimate how many subjects and video sequences are required to eliminate subject bias.

II. Modelling Subjective Tests

Let's with some basic terms.

- **Processed Video Sequence** (PVS) – sample from the original dataset with intentionally changed (downgraded) quality [1].
- **Absolute Category Rating** (ACR) – grading system where e.g grades vary from "Bad" to "Excellent" and have only integer values from a 5-level scale. Note that ACR values are only integers (1 to 5), so random variables from any continuous distribution need to be discretized.
- **Mean Opinion Score** (MOS) – mean value of ACR grades for every PVS.

Results of a testing session can be written as a matrix $U_{J\times I}$ with columns representing subjects and rows representing PVSs. Every element of this matrix is:

$$U_{ji} = Disc(N(\psi_j + \Delta_i, \sigma_j))$$

where:

- ψ_j – true quality value of j^{th} PVS,
- Δ_i – opinion bias of i^{th} subject,
- σ_j – standard deviation,
- $i = 1, 2, \ldots, I$ and I is an even number,
- $j = 1, 2, \ldots, J$.

According to Janowski and Pinson [2]:

- There is a true quality value $\psi_j \sim U(1, 5)$ for every PVS and it is a random variable from uniform distribution on range [1, 5].
- $\Delta_i \sim N(0, 0.25)$ is a shift between true value ψ_j and i^{th} subject's scores. Notice that it is the same for every PVS that subject graded and has mean value equal to 0. That enables us to consider subjects giving grades both too high and too low.
- $\sigma_j \sim \Gamma(N = 30, \lambda = \frac{0.7}{30})$

III. Modelling in Python

As it's easy to observe – the whole matrix seems trivial – there are a few random variables and a pretty simple function (namely three *if* statements). The biggest challenge is hidden in code optimization. *Python* is simple and self-explanatory. Yet everything comes with a price – and the price here is the overhead. *Python* is a high-level language with dynamic typing, so the amount of additional class-specific data is bigger in comparison to Java and much, much bigger comparing to C++. Nevertheless *Python* has *modules* with optimized statistical functions such as the *Gamma distribution*. So the speed was traded for simplicity and now in order to cut a better deal the code should be optimized wherever it is possible.

The model based on theory above was implemented in *Python 3.7* and consists of following parts and functions.

A. Distributions

Random variables ψ, were generated using *rvs* method of appropriate distribution with given parameters from *SciPy.stats* module. Every model with J PVSs has J values of ψ. Since *rvs* function generates multiple random variables (and is optimized to do so) we can generate a single list of J variables (*rvs* used only once) instead of J vectors of size 1 (*rvs* used J times). Theoretically a single Gamma distribution can be replaced by a sum of exponential distributions.

$$X_n \sim exp(\lambda), n = 1, \ldots, N$$

$$\sum_{n=1}^{N} X_n = Y \sim \Gamma(N, \lambda)$$

Generating 1 million variables from a single *Gamma distribution* ($N = 30, \lambda = \frac{0.7}{30}$) took $89ms$, when a sum of 30 exponentials ($\lambda = \frac{0.7}{30}$) took $79s$. Clearly this option is no longer worth considering.

B. Discretization

ACR values are only integers in range [1, 5], therefore every value x from a continuous distribution needs to be *clipped* i.e. discretized as follows:

$$Disc(x) = \begin{cases} 1 & \text{if } x \leq 1 \\ [x] & \text{if } 1 < x < 5 \\ 5 & \text{if } x \geq 5 \end{cases}$$

C. Model function

Single model is represented by a function which returns a *tuple* with random variables, means, and *p-values* from t- and U- tests. Input parameters are:

- **I** – even number of subjects,
- **J** – number of PVSs,
- **scores** – initially empty array of size $J \times I$ for scores (one for all simulations),
- **indices_pairs** – list with every combination of 2 distinct PVSs indices (one for all U-tests).

D. Means

In order to visualize the impact of subject bias, means for each PVS were calculated for two groups, each with exactly $\frac{I}{2}$ subjects.

$$\mu_{j1} = \frac{2}{I}\sum_{i=1}^{\frac{I}{2}} U_{ji} \quad \text{and} \quad \mu_{j2} = \frac{2}{I}\sum_{i=\frac{I}{2}+1}^{I} U_{ji}$$

Let's consider a function f, so that $f(\mu_{j1}) = \mu_{j2}$. Its plot shows differences in MOSs for each PVS. Since ψ_j values remain unchanged, the only variable here is Δ_i – the subject bias. The closer scores μ_{j1} and μ_{j2} are, the more f's plot resembles a straight line through the origin at an angle $\frac{\pi}{4}$.

E. t-test

In real experiments the visualization is not enough, a more trustworthy measure is needed. To check if μ_{j1} and μ_{j2} are statistically different we need to perform a *t-test*, also known as *Student test* [3]. This test is implemented in *SciPy.stats* as *ttest_ind()*. Each test returns a *p-value* describing the likelihood of means being equal. Since there are J PVSs, J *t-tests* need to be performed.

F. U-test

U-test, also known as *Mann-Whitney test* [4], is performed on pairs of MOS vectors per PVS in order to check how many statistically unique PVSs there are. As in the *t-test*, the output is a *p-value* describing the similarity. It is easy to observe that there are J PVSs, $\frac{J(J-1)}{2}$ *U-tests* need to be performed. Apart from that all those combinations have to be generated. However J remains constant during the simulation so *combinations of indices* can be found only once and then passed to the *Model function* instead of finding combinations of PVS vectors each time. A simple test was made in order to compare times when performing operations on every pair in a subset using two methods described above. It turns out that for a matrix $A_{160\times24}$ passing only indices made *U-tests* on pairs of rows in 100 different matrices more than 100 times faster (4780 ms vs. 40 ms - mean for each matrix).

```
result1 = [stats.mannwhitneyu(a, b)
        for (a, b) in combinations(A, 2)]

indices_pairs = combinations(range(rows), 2)
result2 = [stats.mannwhitneyu(A[a], A[b])
        for (a,b) in indices_pairs]
```

G. Separation loops

To plot μ_{j2} in terms of μ_{j1} they need to be put in separate vectors beforehand. There are two ways to do it, supposing there is a matrix $M_{J\times2}$. First way (let's call it **common loop**) is to declare two empty lists of length J (A1 and B1) and then write $A1_j = M_{j1}$ and $B1_j = M_{j2}$ for $j = 1, 2, \ldots, J$.

```
A1 = []
B1 = []
for j in range(J):
  A1.append(M[j][0])
  B1.append(M[j][1])
```

Implementing **separate loops** is more *pythonic* and much more elegant.

```
A2 = [M[j][0] for j in range(J)]
B2 = [M[j][1] for j in range(J)]
```

For J equal to 1 million, **common loop** took 881 ms, when **separate loops** were faster – 539 ms.

IV. Simulation script

In order to simulate many models, gather and combine their outputs, a simulation loop is needed. The one used here is fairly simple - an ordinary *for* with N iterations. The whole simulation *script* does what follows:

- generates and passes global constants that don't need to be calculated in each iteration,
- calls the *Model function*,
- appends each output *tuple* with a proper index to output list,
- creates an output *.csv* file,
- counts overall time of both simulation and writing to file.

V. Demonstration & Tasks

A. Other ways to calculate mean value

Compare different methods to calculate mean value of a given vector. Is there a way to swiftly calculate means for two groups without using any loops? [5]

B. Distributions & Plotting

Generate random variables of your choice with *SciPy.stats* [6] module and draw plots/histograms using *matplotlib.pyplot* [7].

C. Single Gamma vs. multiple exponentials

Compare times for 10^3 and 10^5 $X \sim \Gamma(N = 30, \lambda = 0.7)$ and sum of 30 $R_n \sim exp(\lambda = 0.7)$.

D. Titanic

Import the Titanic dataset (an .xlsx file) using the *Pandas* module [5]. Check the header and delete useless columns. Conduct simple data analysis e.g. survival rate based on age.

VI. Further Challenges

A. Other modules for distributions

TensorFlow Probability is a module for *probabilistic machine learning* in *TensorFlow* and *probabilistic reasoning* in general. Using it to generate random variables in this project could lead to an increase in speed.

B. Multithreading

Although some functions used here are already designed to use multiple threads, performing models separately could increase the speed.

References

[1] L. Janowski and M. H. Pinson, "The accuracy of subjects in a quality experiment: A theoretical subject model," *IEEE Trans. Multimedia*, vol. 17, no. 12, pp. 2210–2224, 2015.

[2] L. Janowski and M. H. Pinson, "Subject bias: Introducing a theoretical user model," in *Sixth International Workshop on Quality of Multimedia Experience, QoMEX 2014, Singapore, September 18-20, 2014*, pp. 251–256, 2014.

[3] "Student test." www.encyclopediaofmath.org/index.php/Student_test.

[4] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *The Annals of Mathematical Statistics*, vol. 18, no. 1, pp. 50–60, 1947.

[5] W. McKinney, "Data structures for statistical computing in python," in *Proceedings of the 9th Python in Science Conference* (S. van der Walt and J. Millman, eds.), pp. 51 – 56, 2010.

[6] E. Jones, T. Oliphant, P. Peterson, *et al.*, "SciPy: Open source scientific tools for Python," 2001–. [Online; accessed <today>].

[7] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing In Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.