1. **Brief explanation on Monte Carlo implementation:**

We use Monte Carlo to find the best credit card ordering from the pool of all possible orderings. To explain our process, I will use 5 cards as an example.

With 5 credit cards, we have 5! = 120 total possible orderings. With 1~5 each representing a unique card, we assume that there exists a best ordering such as (1,2,3,4,5). We can then use the numbers and their indices to simulate their clickthrough rates. For example, since (1,2,3,4,5) is the best, then (2,1,3,4,5) should have similar but lower clickthrough rate, as it is very close to the best ordering; while (5,4,3,2,1) is the worst since it’s quite distant from the best ordering.

The second cell in the notebook is doing what’s described above using the highest clickthrough rate in reality: 0.25. You may modify the ‘distance’ function to switch to other ways of simulations, but the method needs to create a constant space for all orderings – don’t assign random rates to them.

1. **How we find the ordering to compare:**

We always start with a guess. Let’s look at the dataframe below:

表格

描述已自动生成

The last four columns are simulated numbers – in reality, we don’t know which ordering is the best, but we need to start from somewhere to guess (and this guess will be our best guess based on field knowledge). To not let the algorithm step randomly in the space, we will switch only two digits at a time to compare. For example, if we have (1,3,2,4,5) at hand, we may go to (1,2,3,4,5) next. Note that for 5-card orderings, there are only (4+3+2+1) = 10 kinds of switches for one ordering.

1. **How we compare two orderings:**

When we have two orderings at hand to compare, we first need basic data on these two orderings. A clickthrough is when a customer opens our page and uses our link to apply for that card; simply open and close is not a clickthrough. We will use 5 times of landings on the page as one sampling and record its clickthroughs.

I started with 20 samplings – total 100 landings for each ordering. And then, I will compare LAC and HAC of two orderings. LAC = higher mean – standard deviation, and HAC = lower mean + standard deviation. If LAC is smaller than HAC, we will keep grabbing new data from our website (i.e., grab one more sampling at each time). We will stop comparing when LAC > HAC and keep the ordering with higher mean. (Please note that the ordering with higher mean could switch between the two when we grab more statistics.)

The main purpose of the step of above is that we want to statistically differentiate two orderings; however, it turns out sometimes two orderings could have too similar clickthroughs that LAC is never larger than HAC. We then have a few solutions:

* We set a mandatory stop point, meaning that if LAC is still smaller than HAC after, for example, 50 samplings (250 landings), then we decide whichever ordering with higher average is better.
* We use a fraction of standard deviation, for example, 1/3.

Both mandatory stop point and fraction of std can be modified in the parameters of MC\_implement function.

1. **Inventory**

Now that we know the general ideology of how to simulate MC, we also have further considerations: what should we do with the dropped (the worse) ordering in the process when we compare pairs?

One solution is to drop them directly, but this could cause the algorithm to step away from the best ordering out of coincidence.

Hence, Anthony suggested a solution: go back to the dropped orderings sometimes.

We do so as follows: for each ordering, the number of comparable orderings is limited (as explained in 2nd part). Therefore, if the algorithm decides to stay with one ordering (such as (2,1,3,4,5)), it will exhaust the pool of comparable orderings first, and once the pool is exhausted, it will restart the pool, sampling, and clickthroughs calculation(as if there were no past data). The only heritage we have from the previous implementation is the best ordering at hand. The advantage of this part is that: 1) we have a best guess using the algorithm, and it could outperform a manual guess as described in the 2nd part), 2) if the customer taste has changed, our algorithm could react to the changes.

1. **Additional notes**

I used a lot of inventories in the middle to record the details of chosen ordering (such as its simulated clickthrough rate/prob, rank, etc) and the dropped ordering at each step, dropped ordering rank, number of new landing required, etc.

I am using an interactive plotly graph here so you will need to install the package to make graphs run. The red dashed vertical lines represent each time when we exhaust a pool and restart with a new one. The graph shows the step and at each step what decision the algorithm makes (should be easy to read).

Do not put set.seed() in the cell where the function or the graph is, that will ruin the simulation.

Even if we tried a fraction of standard deviation, it is most common that LAC is still smaller than HAC in the end (when the number of samplings reaches its stop point), therefore, you might want to explore other methods (two-sample t test does not work well) or remove the complicated calculations but only use the average (if you do so, remember to adjust the stop point as well—because only using the average means you will need as many data as possible to have reliable results).

The use of dictionary to record clickthroughs is necessary, because in reality if we have already compared A and B and decided to stay with B, next time when we compare B and C we don’t need to simulate clickthroughs for B again.

The entire algorithm requires a mandatory stop point as well (stop\_point in the parameters of MC\_implementation, I set it to 60 for this 5 cards trial). It’s because when we deal with larger space, for example, 7 cards, we will have 7! = 5040 total possible orderings, we cannot compare them all.

The most significant cost we consider here is the number of landings. Each landing costs money, so we need to find a point where landings are as few as possible while the results are still reliable. You can keep making experiments on our mandatory stop comparing point and algorithm stop point to find the balance.

Things can get unlucky sometimes and the algorithm might stay with an unsatisfying result when we stop comparing; but it can go to the best results under most of times (as you can see in the last cell, around 75% of all times).

Please let me know if you encountered any problems and I would like to answer. My email address is gaoshaoj@usc.edu.