Final Project: TAC AdX Game

1 Introduction

This final project option is derived from the TAC AdX Game, ultra-simplified versions of which we explored in Labs 7 and 8.

2 Game Description

There are several differences between this version of the game and the earlier versions from the labs, which we explain in-depth in this section. In short, the game will be extended to 10 days, campaigns will vary in length, and most importantly, **campaigns will be allocated endogenously via auctions** rather than exogenously (according to some distribution), so that one agent may seek to fulfill multiple campaigns simultaneously.

2.1 Advertising Campaigns and Impression Opportunities

Your agent's main task in this game is to bid on (and hopefully win) **advertising campaigns**, based on which it will again bid on (and hopefully win) **impression opportunities**, namely opportunities to exhibit ads to Internet users as they browse the web.

Each day of the simulation, a random number of Internet users browse the web. These users hail from multiple market segments, of which there are a total of 26, corresponding to combinations of `{Male, Female}` x `{HighIncome, LowIncome}` x `{Old, Young}`. A market segment might target only one of these attributes (for example, only `Female`) or two (`Female_Young`) or three (`Female_Old_HighIncome`). Users' market segments drawn from the distribution described in Appendix A.

For each user, a **second-price sealed-bid auction** is run to determine which agent to allocate that user's advertising space to, and at what price. Ties are broken randomly, so if there are two winning bidders in a market segment, each will be allocated (about) half the users in that segment at the price they bid.

Ad networks are motivated to participate in these auctions by their desire to fulfill advertising campaigns. A campaign is a contract of the form, "The ad network will show some number of ads to users in some demographic. In return for these impression opportunities, the advertiser agrees to pay the ad network said budget." More specifically, each campaign is characterized by:

- A **market segment**: a demographic(s) to be targeted.
- A **reach**: the number of ads to be shown to users in the relevant market segment.
- A **budget**: the amount of money the advertiser will pay if this contract is fulfilled.
- A **start day** and an **end day**: the time during which the campaign is active. This range is inclusive; you will have the opportunity to bid on a campaign on its start day, its end day, and every day in-between.

Here is an example of a campaign:
`[Segment = Female_Old, Reach = 500, Budget = $40.0, Start_Day = 4, End_Day = 6]`

To fulfill this campaign, your agent must show at least 500 advertisements to older women between days 4 and 6. If successful, it will earn $40. To show an advertisement to a user, you must win the auction for that user. (Yes, we, as Internet users, are regularly auctioned off!) But note that winning an auction for a user who does not match a campaign's market segment does not count toward fulfilling that campaign.

2.2 Ad Auctions: Bids and Spending Limits

Unlike the sealed-bid auctions we have studied in class, which are one-shot auctions, the auctions in this game are repeated, as users arrive repeatedly. However, agents can only bid in these auctions once!—before their simulation begins. Consequently, agents must reason in advance about how events might unfold over the course of the day, and perhaps make contingency plans. The AdX game provides a mechanism for making a contingency plan in the form of spending limits. These limits are upper bounds on the amount an agent is willing to spend in either a specific market segment, or in total on a campaign, meaning overall, across all market segment associated with that campaign.

If your agent has a campaign whose market segment is very specific (e.g., `Female_Old_HighIncome`), then it won't really have a choice about which users to bid on; it has to bid for users in precisely that market segment, or it cannot earn a positive profit. However, if its market segment is less specific (e.g., `Female`), it can bid different amounts in the `Female_Old` and `Female_Young` markets, for example, based on how much competition it thinks there will be in each. Keep in mind, though, that the order in which users arrive is random. So if it bids more on `Female_Old` than `Female_Young`, but then if all `Female_Old` users arrive before any `Female_Young`, it may end up spending its entire budget for that campaign on `Female_Old` users. For this reason, when bidding on a market segment, your agent might want to specify a spending limit each market segment.

An agent can also specify an overall campaign spending limit to ensure that it does not spend more than some pre-specified total across *all* market segments associated with a campaign. In sum, the key decisions an agent must make in the ad auctions are what bids to place on what market segments, and what spending limits should accompany those bids.

2.3 Effective Reach and Quality Score

At the end of each day, after all of the users have browsed the web and all of the ad auctions have been run, the server will calculate the **effective reach** of all the agents for each of their active campaigns. This value captures how much of a campaign has been fulfilled via a sigmoidal function that relates the number of matching impressions won to the campaign's total reach.

Effective reach is used to calculate an agent's immediate profit on a campaign. Given a campaign C with budget B, cost K, and effective reach $\rho(C)$, profit is calculated as $\rho(C) \cdot B - K$. Thus, achieving a high effective reach, in addition to keeping costs low, is key to maximizing profits.

The specific function for the effective reach of a campaign C is given by:

$$\rho(C) = \frac{2}{a}\left(\arctan\left(a\left(\frac{x}{R}\right) - b\right) - \arctan(-b)\right),$$

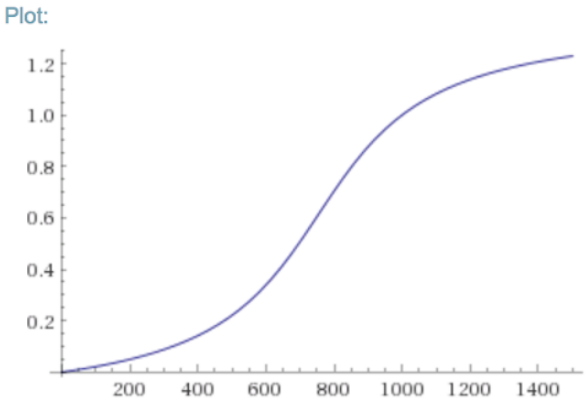
where $a = 4.08577$, $b = 3.08577$, x is the amount of impressions achieved, and R is your campaign's reach.

The following plot depicts the effective reach function $\rho(C)$ of a campaign with reach of $R = 1000$ and x impressions achieved. Note that $\rho(0) = 0$, $\rho(R) = 1.0$, and $\lim_{x\to\infty} \rho(C) = 1.38442$. The plot shows that the value of obtaining the first few impressions on the road to fulfilling a campaign is relatively low, compared to the value of obtaining the middle and final impressions.

Effective reach is also used to calculate **quality score**, which is a measure of an agent's reputation (used in the campaign auctions), and hence impacts its long-term profits. Quality score is initialized to 1, and then updated at the end of each day as a moving average of the average effective reach, say $\bar{\rho}$, of all campaigns that end on that day. More specifically, an agent's quality score is iteratively updated as follows:

$$Q_{\text{after}} = (1 - \alpha)Q_{\text{before}} + \alpha\bar{\rho}.$$

Here, α controls how quickly the quality score changes with each day's effective reach.



2.4 Campaign Auctions

On the first day of the game, your agent will be assigned one campaign at random. Then, on each subsequent day, multiple (randomly-generated) campaigns will be put up for auction. Campaign auctions are conducted as **second-price reverse auctions**. As opposed to a traditional auction where the good goes to the highest bidder, *in a reverse auction, the contract goes to the lowest bidder*. Reverse auctions are common in procurement. Think of the "bids" as "budgets" for the contract; the lowest bidder is the one who is willing to take on the contract given the lowest budget, which in AdX means at the lowest cost to the ad agency.

Your agent should submit bids for each campaign it is interested in. These bids, for a campaign with reach R, must fall within the range $[0.1R, R]$.[1] The bids are then divided by the agents' respective quality scores to arrive at **effective bids**, before they are entered into the auction. For example, for a campaign with reach 100, if two agents bid at the bottom of the acceptable range (i.e., 10), and one's a quality score is 0.9, and the other's is 1.1, then their effective bids are $10/0.9 > 10/1.1$, respectively, so the winner is the agent with the higher quality score. The budget is then set to $(10/0.9)(1.1)$; in other words, the second-lowest budget times the winning agent's quality score. If there is only one bidder, say A, in the auction, A wins the campaign with a (maximum) budget of $(R/Q_{\text{low}})Q_A$, where Q_{low} is the average quality score among the three agents with the lowest quality scores on that day, and Q_A is bidder A's quality score.

2.5 Scores

At the end of each simulation, the server will compute the profit earned by each agent/ad network. The agent with the highest total profit wins. Because of the randomness inherent in the game, it will be simulated repeatedly, and scores averaged over multiple simulations, to determine an overall winner.

2.6 Putting It All Together

All in all, here is a summary of the AdX game:

2.7 Game Specifics

- The game will last for 10 days.
- The quality score adjustment rate α will be 0.5.

[1] The non-zero lower bound on bids is intended to avoid bidding wars, where all agents' effective bids are zero.

Algorithm 1 Run the n-day AdX game

```
Give each agent an initial campaign
Set each agent's quality score to 1

for each day 1 . . . n do
    List m new campaigns for auction
    for each agent a do
        Solicit bids for the campaigns which are up for auction
        Solicit bids on market segments for a's currently active campaigns
    end for
    Simulate users, run ad auctions, allocate users, calculate costs
    for each campaign that ends today do
        Compute the relevant agent's effective reach
        Compute that agent's profit (effective reach less cost)
    end for
    Run campaign auctions, allocate the new campaigns
    Update agents' quality scores
end for
The agent with the highest total profit wins
```

- Initial campaigns will be assigned uniformly at random.
- Each day, there will be 5 new campaigns up for auction, also chosen uniformly at random.
  - When the ad agency draws these new campaigns, they remove all campaigns that end after day 10. So, towards the end of the game (specifically, on days 9 and 10), there is a (good) chance that there will be fewer than 5 new campaigns.
- The game will consist of 10 agents, so your agent will have 9 opponents.

The reach and start and end days of campaigns are drawn from distributions (see Appendix A). A campaign's budget (i.e., potential revenue for winning impressions) is set during a campaign auction (see Section 2.4). TAC AdX is a game of incomplete information, as each agent knows (with certainty) the budgets of its own campaigns only, not those of its competitors.

3 Strategy and Considerations

This is a very complex game and requires several strategic considerations beyond what was in play for Labs 7 and 8.

First, your agent can have multiple active campaigns at once, and via its bids in campaign auctions, has some control over which campaigns it is (or at least hopes to be) allocated. If your agent already owns a campaign for `MALE_YOUNG`, should it bid low for a campaign for `MALE_OLD`, so as to not compete too heavily with itself for `MALE` users? Or should it try to corner the market?

Second, the reverse auction mechanism, and how it determines a campaign's budget, introduces some interesting considerations. How low is your agent willing to bid? If it bids too low, it may end up stuck with a low budget, limiting your agent's profits. This is exacerbated by the fact that the first part of the effective-reach curve is essentially flat—profits don't pick up until about half of a campaign's reach is achieved. On the other hand, if your agent bids too high, it may have trouble winning any campaigns in the first place, and it cannot profit at all without any campaigns.

Third, if your agent wins two campaigns with overlapping market segments, which should it prioritize? Should it place higher bids for the campaign that ends sooner, since it has less time to fulfill that campaign? Or

should it pay equal mind to the campaign that ends later, as fulfilling that campaign early would allow it to focus on other overlapping campaigns in the future? Perhaps it can balance its interests by setting precise spending limits based on how much of each campaign its already fulfilled?

If your agent procures only a small percentage of a campaign's reach, then its effective reach for that campaign will be low, which means your agent's profit on that campaign will be low, since the lower part of the effective reach curve is basically flat. But worse still, its quality score will decrease, which means that all else being equal, it must bid lower to win campaign auctions, so it will have to settle for lower budgets on its future campaigns. In short, your agent must choose wisely when deciding how to bid on campaigns, because it cannot safely ignore any of its active campaigns. Once it wins a campaign, it is stuck with it.

In addition to these considerations, a successful AdX agent strategy will comprise many others. We suggest building a theoretical model of the game (consider starting your writeup!) before attempting to implement an agent strategy. This model will help you envision even more of the many important factors that your agent's strategy should try to take into account.

4 Game API

4.1 `MyNDaysNCampaignsAgent` class

Your task is to fill in the following methods of the `MyNDaysNCampaignsAgent` class:

- `getAdBids()`: returns a `Set<NDaysAdBidBundle>` representing the agent's bid bundles on the current day. A bid bundle includes a campaign and a spending limit, as well as bids in market segments corresponding to the campaign (and optional spending limits in those market segments as well).

  Note that these bids are *daily* bids; they only last for one day. If an agent has a multiple-day campaign, and it places a bid on the first day of its campaign, it must to place another bid on the second day, if it would like to continue bidding. You should feel free to take advantage of this flexibility; it allows your agent's strategy to adjust to the events of yesterday when planning for tomorrow.

- `getCampaignBids(Set<Campaign> campaignsForAuction)`: returns a `Map<Campaign, Double>` representing the agent's bid on each element of `campaignsForAuction` on which it is bidding.

The server, in playing the role of an ad exchange, calls these two methods every day of the game. Whatever your agent returns is then used by the server when it runs the auctions.

In addition, we have provided the following utility method, which you may fill in, should you find it useful:

- `onNewGame()`: called at the beginning of each simulation of the AdX Game. If you are maintaining any per-game data, this is where you would want to initialize or reset it. For example, if you are using a pre-trained agent, this is where you would read in your saved data. Or, if you are keeping more fine-grained data than that provided by the inherited methods (for example, a map from market segment to the amount of campaigns you have covering it), you would clear it here so that it is empty going into the next game.

Finally, below is a list of methods inherited by `MyNDaysNCampaignsAgent` which you may also find useful:

- `int getCurrentDay()`: gets the current day within the game, a number between 1 and 10.
- `Set<Campaign> getActiveCampaigns()`: gets the agent's active campaigns. This set should correspond to the campaigns the agent bids on in `getAdBids()` (assuming it doesn't ignore any). Only the campaigns owned by your agent will be in this set.

- `double getQualityScore()`: gets the agent's current quality score.
- `int getCumulativeReach(Campaign c)`: gets the number of impressions that the agent has fulfilled for campaign `c` *so far*. This information is particularly useful for multi-day campaigns; the agent can use it to set goals and spending limits as a campaign proceeds.
- `double getCumulativeCost(Campaign c)`: gets the amount the agent has spent on impressions in the ad auctions for campaign `c` *so far*. This information is particularly useful for multi-day campaigns; the agent can use it to set goals and spending limits as a campaign proceeds.
- `double getCumulativeProfit()`: gets the agent's cumulative profit from the start of the game up to the current day.
- `double effectiveReach(int x, int R)`: the effective reach function.
- `boolean isValidCampaignBid(Campaign c, double bid)`: checks whether `bid` is in the range `[0.1 * c.getReach(), c.getReach()]`. We recommend using this function to verify that your agent's bids are valid before submitting them, as invalid bids are rejected by the server.
- `double clipCampaignBid(Campaign c, double bid)`: returns `bid`, clipped into the range of valid bids on campaign `c`. In other words, returns `max(0.1 * c.getReach(), min(c.getReach(), bid))`.
- `double effectiveCampaignBid(double bid)`: returns the *effective bid* that would be derived from bidding `bid` in a campaign auction. In other words, returns `bid / this.getQualityScore()`.

4.2 The `MarketSegment` Enum

`MarketSegment` is implemented as an `Enum`, so to iterate over all of them, you can do something like:

```
for (MarketSegment m : MarketSegment.values()) { ... }
```

The static function `MarketSegment.marketSegmentSubset(MarketSegment m1, MarketSegment m2)` returns a `boolean` indicating whether `m2` is a subset of `m1`, so that users in market segment `m2` are also in market segment `m1`. (**N.B.** market segments are subsets of themselves.)

4.3 `Campaign` objects

In both the ad and campaign auctions, you will be working with `Campaign` objects. They provide the following methods:

- `int getID()`
- `int getStartDay()`
- `int getEndDay()`
- `MarketSegment getMarketSegment()`
- `int getReach()`
- `double getBudget()`

4.4 `NDaysAdBidBundle` and `SimpleBidEntry` objects

To submit ad bids, you should construct an `Set<NDaysAdBidBundle>`.

Each `NDaysAdBidBundle` represents an agent's bids on a single campaign, and an overall campaign spending limit. `NDaysAdBidBundle` objects are constructed as follows:

```
NDaysAdBidBundle bundle = new NDaysAdBidBundle(
    int campaignId,
    double limit,
    Set<SimpleBidEntry> simpleBidEntries
);
```

The `limit` field sets a spending limit campaign-wide; that is, it represents a daily spending limit for the campaign. So even if an agent's market-segment-specific limits (described below) total above this limit, its spending on this campaign will still be capped at this limit.

The `simpleBidEntries` field allows an agent to set daily bids and spending limits specific to market segments within a campaign. `SimpleBidEntry` objects are constructed as follows:

```
SimpleBidEntry bidEntry = new SimpleBidEntry(
    MarketSegment marketSegment,
    double bid,
    double limit
);
```

So, a `simpleBidEntries` set should contain one of these objects for each market segment on which the agent is bidding. Likewise, a `NDaysAdBidBundle` set should contain one of these objects for each campaign on which the agent is bidding.

5 Tier 1 Agent

We have provided you with access to our Tier 1 TA bot implementation to test your own agent against. This bot is implemented in the `Tier1NDaysNCampaignsAgent` class, and does the following:

- In all campaign auctions, it bids a random number within the range of valid bids.
- In ad auctions, for each campaign with budget B, reach R, so-far cumulative reach of R_a, and so-far cumulative cost of K_a, it bids $\max(0.1, (B - K_a)/(R - R_a))$ with a spending limit of $\max(1, B - K_a)$.

6 Code: Installation, Testing, and Submission

Please read the following instructions carefully, as it is essential that your code runs successfully with our game server, and that your handin is correctly formatted.

6.1 Installation

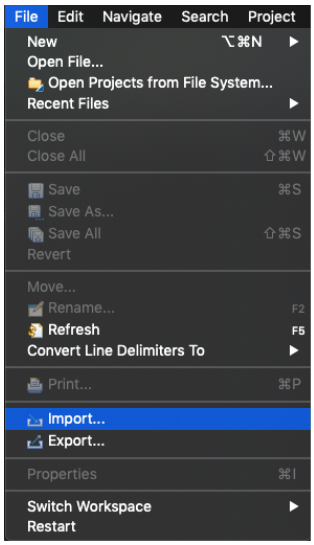
Download the stencil code from the course website. Just as in Labs 7 and 8, the stencil code is an entire Java project, complete with a package structure and a file called `pom.xml`, along with a few python scripts.

We will be using the Apache Maven build system for this project. Put simply, Maven allows us to export our Java projects into runnable JAR files in a standardized fashion (as defined in `pom.xml`, which acts as a configuration file for Maven).

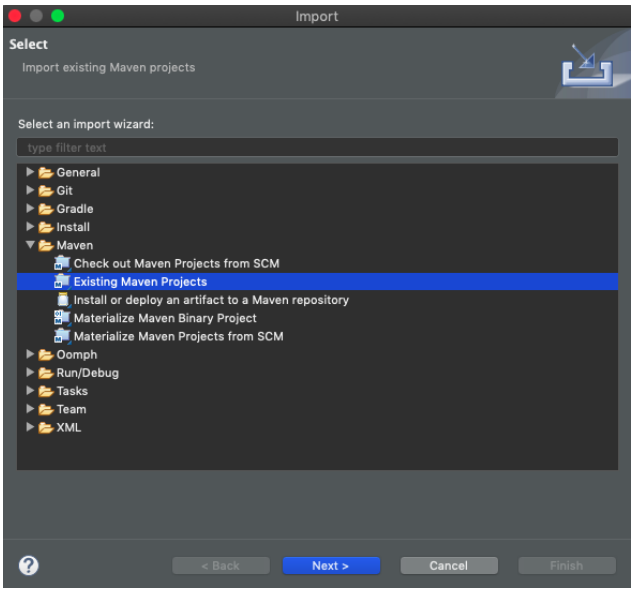
If you do not already have Maven installed, please download and install it. If you are working on a department machine or via FastX, you will already have Maven installed, which you can verify by running `mvn -version`.

If you have not installed Eclipse on your machine, you should download that as well.

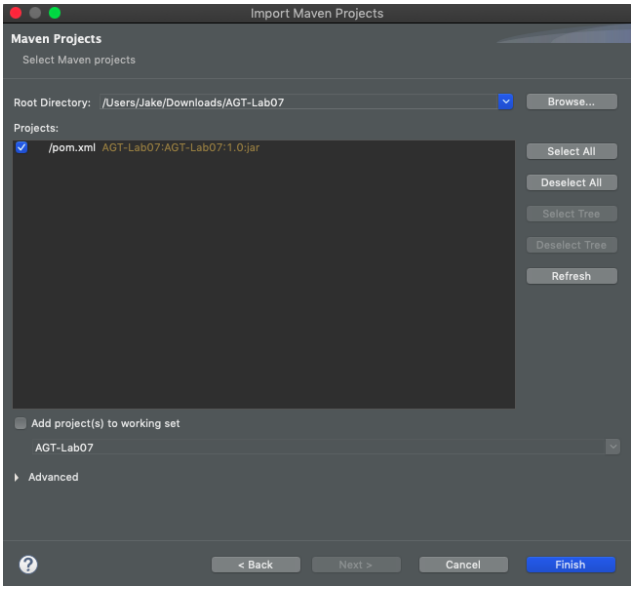
Once you have installed Maven and unzipped the stencil code, open Eclipse and select **File** → **Import**.



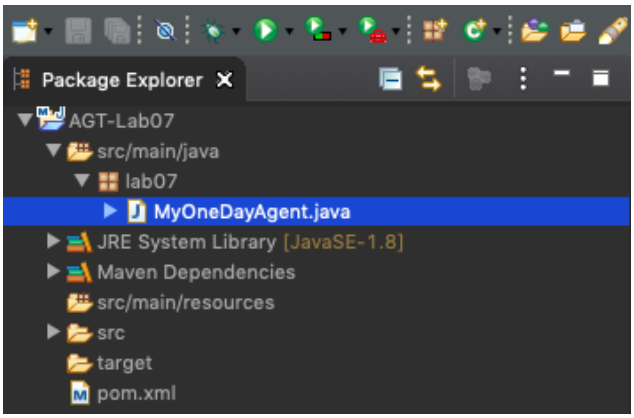
Then, select **Maven** → **Existing Maven Project**.



Finally, navigate to, and select, the stencil project and make sure `pom.xml` is selected. Select **Finish**.



This will create an Eclipse project for this lab. Your task resides in the file `MyNDaysNCampaignsAgent.java` in the package `agent` under `src/main/java`. Feel free to create and use any other packages, classes, etc. within the project, as long as you do not change the location or name of `MyNDaysNCampaignsAgent.java`.



6.2 Local Testing (and Training)

If you would like to test out your agent against other agents, run your `MyNDaysNCampaignsAgent` file directly in Eclipse. The main method instantiates 10 agents, including your own, and runs an offline simulation (in order to not have to deal with server delays and wait times). This simulation consists of 500 iterations of the AdX game.

As with any programming project, you should test your programs extensively. In addition to the usual unit tests, etc., you should play test games against other agents. You can test against 9 copies of your own agent, 9 copies of our Tier 1 TA Agent (see Section 5), or any combination thereof. You can also create your own dummy agents to test your own agent against; just create a class extending `NDaysNCampaignsAgent`. To set the agents to test against, you should edit the agent types in `src/main/resources/offline_test_config.json`.

If you design a strategy that involves training, you can submit a pre-trained agent. Then, to use a pre-trained agent that reads its input from a file, please place the file in `src/main/resources` and read it using `getResourceAsStream()`; see this guide. By including your file in the resource folder, you will ensure that it is included in the executable JAR we use to run your submission; file-reading is thus independent of the specifics of the filesystem.

6.3 Submission

You will be submitting your code via the course handin script.

First, if you are not already working on a department machine or via FastX, you will need to transfer your files to the department filesystem. We have provided a utility to do this upload for you. You are free to transfer the files any way you'd like—this utility is just an attempt to simply your life.[2]

To use this utility, from your project's root directory, run `python3 upload.py`.[3] It will transfer your files to `/course/cs1951k/student/<cslogin>/AdXFinal`. (We have created student directories for you so that you don't need to use up tons of space in your personal filesystem.)

Next, SSH to Brown, and navigate to your project's root directory (if you used the upload script, `/course/cs1951k/student/<cslogin>/AdXFinal`).

To make sure that your agent will work as intended when being run the way we run handins, we have provided a script that launches the server, along with your agent and 9 opponent bots. You should run this script to make sure your agent connects to the server properly, doesn't crash, and properly submits its bids.

The command is `/course/cs1951k/pub/2020/AdXFinal/test`. It should be run from your project's root directory, and it runs 5 iterations of the game, and should take about one minute, or less (but it may take a bit longer the first time, if Maven needs to download any packages).

If it succeeds, you'll be able to see the results of your game against the bots. As such, feel free to also use this script to debug your agent's strategy.

Finally, run `/course/cs1951k/pub/2020/AdXFinal/submit` to submit your agent. We only store your most recent handin, so if you wish to update your agent and submit a newer version at some later date, just repeat these steps and your submission will be replaced.

A Campaign and User Distributions

Each campaign lasts between 1 and 3 days (chosen uniformly at random), and targets one of 20 possible market segments (a combination of at least two attributes). A campaign's reach is given by the average number of users in the selected segment (listed in the tables below), scaled by the campaign's duration in days, times a random reach factor, selected from the set $\{\delta_1, \delta_2, \delta_3\}$, where $0 \leq \delta_i \leq 1$, for all i. The exact values of these factors are tailored to the number of agents in the game. In particular, for the ten-agent games we plan to run, we will use $\delta_1 = 0.3$, $\delta_2 = 0.5$, and $\delta_3 = 0.7$.

[2] Well, not really. Just this very very small part of it.

[3] To run this script, you will need Python 3. You may also need to install it, as well as the `pysftp` library, via `pip`.

User Frequencies

Segment	**Average Number of Users**
`Male_Young_LowIncome`	1836
`Male_Young_HighIncome`	517
`Male_Old_LowIncome`	1795
`Male_Old_HighIncome`	808
`Female_Young_LowIncome`	1980
`Female_Young_HighIncome`	256
`Female_Old_LowIncome`	2401
`Female_Old_HighIncome`	407
Total	**10000**

User Frequencies: An Alternative View

	Young	**Old**	**Total**
Male	2353	2603	4956
Female	2236	2808	5044
Total	4589	5411	10000

	Low Income	**High Income**	**Total**
Male	3631	1325	4956
Female	4381	663	5044
Total	8012	1988	10000

	Young	**Old**	**Total**
Low Income	3816	4196	8012
High Income	773	1215	1988
Total	4589	5411	10000