Iterating the Ultimatum Game: A software package for simulating mixed populations

Maurits Kaptein

Statistics and Research Methods, Tilburg University, The Netherlands.

Abstract

The ultimatum game is a simple game in which the first player (the proposer) proposes to divide a sum of money between herself and the second player (the responder) who subsequently chooses to accept or reject the proposal. If accepted, the money is distributed accordingly, if rejected neither of the players receives a pay-off. Empirical studies demonstrating that human offerers tend to offer relatively even (or "fair") splits of the sum of money and that responders are inclined to reject "unfair" offers, have been taken as a sign for "irrational" behavior in the sense that these choices do not maximize the pay-offs for those involved.

In this work we provide a software library that allows one to easily explore alternative scenarios. The software allows one to compose novel policies (e.g., decision mechanisms that possibly depend on prior data), construct a (possibly mixed) population of agents that utilize these policies, and explore the progression of their offers and responses as multiple rounds of the ultimatum game are played within a population of a set size. In the current poster we devote special attention to sequential learning policies: we create a number of agents for which the pay-offs of the game are initially unclear—corresponding to a human player to whom the game is not explained or does not understand the game—and that strive to learn the best actions (both offers and responses) by some form of trial and error. Here we explore typical multi-armed bandit (MAB) approaches to operationalize agent learning such as Thompson sampling (e.g., choosing an action with a probability proportional to the agents current belief that an action is optimal).

1. Setup

The core of the UltimateSim python library consists of the following files and classes:

- **config.py** Editing the `config.py` file allows one to setup—without further programming efforts—different simulations of the Ultimatum Game with different populations of agents. The following parameters can be set:
  - `popSize` The number of agents (an even Int) in the population; note that agents are first assigned, with probability $\frac{1}{2}$ to be offering or responding in a certain round of the game, and subsequently are (uniformly) randomly matched with another agent to play the game.
  - `offerPolicies` A `numpy` array containing the names (String) of the offer policies in the population.
  - `respondePolicies` A `numpy` array containing the names (String) of the response policies in the population.
  - `probMatrix` A `numpy` matrix containing the probability of occurrence of distinct policies. E.g., when using 2 offerPolicies and 3 responsePolicies, the matrix
  $$\begin{bmatrix} \frac{1}{8} & \frac{1}{8} & \frac{1}{8} \\ \frac{1}{8} & \frac{1}{8} & \frac{3}{8} \end{bmatrix} \quad (1)$$
  would state that agents have a combination of offer-policy 2 and response-policy 3 with probability $\frac{3}{8}$. Note that the sum of elements of this matrix should be 1.
  - `simLength` The length of each simulation (Int). Note that the simulation length is the number of total rounds, and hence the total number of games played is $\frac{1}{2} *$ `popSize` $*$ `simLength`. The number of times an individual agent is on the offering (or responding) side of a game is *in expectation* $\frac{1}{2} *$ `simLenght`.
  - `nSims` The number of simulations (Int) to run. For each new simulations a new set of agents (Population) with offer- and response-policies according to the `probMatrix` is instantiated.
  - Next to the above, there are a number of settings, such as `createPlots` (Bool) which control the output and storage of a simulation. These should be self-explanatory.
- **main.py** File that runs the main simulations.
- `class` **Simulation** The class that controls the full simulation study. This class instantiate a population, and seeds it with agents. Subsequently, the simulation is excuted (by the `Run()` method). The Simulation class also contains methods to create plots and summary statistics.
- `class` **Population** The class that controls a population. A population contains a number of agents, and plays a number of rounds of the iteration game. This class contains methods for creating agents, and playing subsequent rounds of the game.
- `class` **Agent** The Agent class implements the an agent, storing its individual data in a simulation and storing / calling its individual offer and response policies.
- `class` **offerPolicy** The base class for implementing an offer policy; by inheriting from this class new policies can be created. See for a short working example `RANDOMOffer.py`
- `class` **responsePolicy** Similar to the offer Policy, but this time defining the response strategy of an agent.

The above allows users to setup a simulation of agents that each, with some probability have a distinct offer- and response- policy in an iterated version of the ultimatum game. Calling `python main.py` runs the set number of simulations and stores their data.

2. Policies

Currently, we have implemented only a limited number of offer- and response-policies, and we are actively working to create alternatives. Currently implemented are:

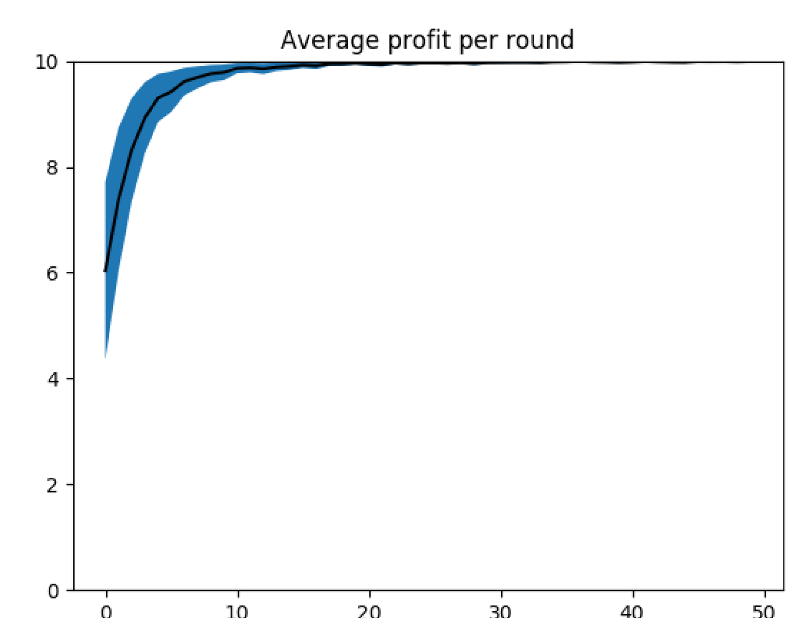
- *RANDOM:* Agents following this strategy exhibit the following behavior:
  - Offer: A uniformly randomly selected number from 0 to 10.
  - Response: Accept the offer with probability p (default $\frac{1}{2}$), reject with probability $(1-p)$.
- *REM:* The "Rational Economic Man" policy. Agents using this offer- and response-strategy exhibit the following behavior:
  - Offer: A certain of of 1
  - Response: Accept any offer higher than 0 (offers of 0 are accepted with probability p, default $\frac{1}{2}$).
- *THOMP1:* The Thompson sampling 1 agent is our first attempt of creating an agent that learns from its interactions; while not having any clear understanding of the game, the agent tries to learn which of the (discrete) choices at her disposal provides the highest pay-off.
  - Offer: For the offer the agents tries to estimate the probability that one of the eleven $[0, 10]$ choices is selected by the responder (ignoring who the responder is) by putting a $\texttt{Beta}(1, 1)$ prior over the probability p_i that offer i is accepted. In each round this prior is updated according to the succes of the offer resulting in a $\texttt{Beta}(1+s, 1+f)$ posterior distribution. We then obtain a draw from each posterior p_i, and subsequently multiply the draw by the actual offer to estimate the value of the offer. Finally, we choose the action with the highest $E[r_i] = \hat{p}_i i$.
  - Response: Here we implement standard Thompson sampling for a 2-armed Bernoulli bandit; any result higher than 0 is counted as a succes for the current arm.

Note that the offer and response sub-policies can be mixed; hence, we can easily create a population of random offerers that use a rational response.

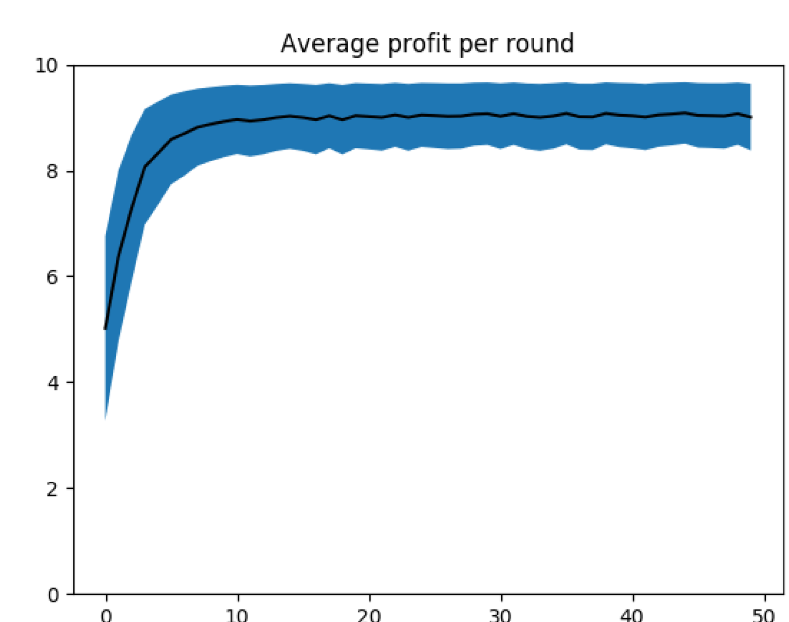
We are primarily interested in further exploring policies akin the *THOMP1* policy: we aim to investigate the behavior over time of populations of self-learning agents.

3. Preliminary results

To briefly demonstrate the utility of the software we present the outcomes of two simple simulation studies. The first Figure shows the progression of the overall profit made (averaged over $m = 100$ simulations, and averaged over agents—the bound present the 5th and 95th percentile) in the different rounds when simulating a population containing 200 agents of which $\frac{7}{10}$ of the agents (in expectation) implement the THOMP1 policy, while the other $\frac{3}{10}$ implement either the REM policy or a mix of the REM and THOMP1 offer/response policies. Hence, there is a small number of rational players, while the other players try to learn the rules of the game as they go along. The Figure shows that the THOMP1 players catch on and learn how to play the game in a "rational" way. However, note that in early rounds of the game, a substantial number of "irrational" offers is made.



The second Figure similarly shows the progression of the overall profit made per round, but this time we substitute the REM policies for RANDOM agents; hence, a number of players just makes a random choice in the game.



Conclusions and Future work

We have briefly introduced UltimateSim, a python library for running simulations of mixed population agents playing an Ultimatum Game. While the software can be used off-the-shelf to run simulations, our core interest is in further developing offer- and response-policies that are stochastic and sequentially learning. Our naive Thompson sampling policy presents a first step in this direction, we aim to further understand the behavior of adaptive learning agents when iteratively playing the ultimatum game in a population of alternative agents.

The software—which is under active development—can be found at `https://github.com/Nth-iteration-labs/ultimatesim`