OOS AdX Agent Project Report

Or Sagi, Omer Rotem, Shelly Grossman

Introduction

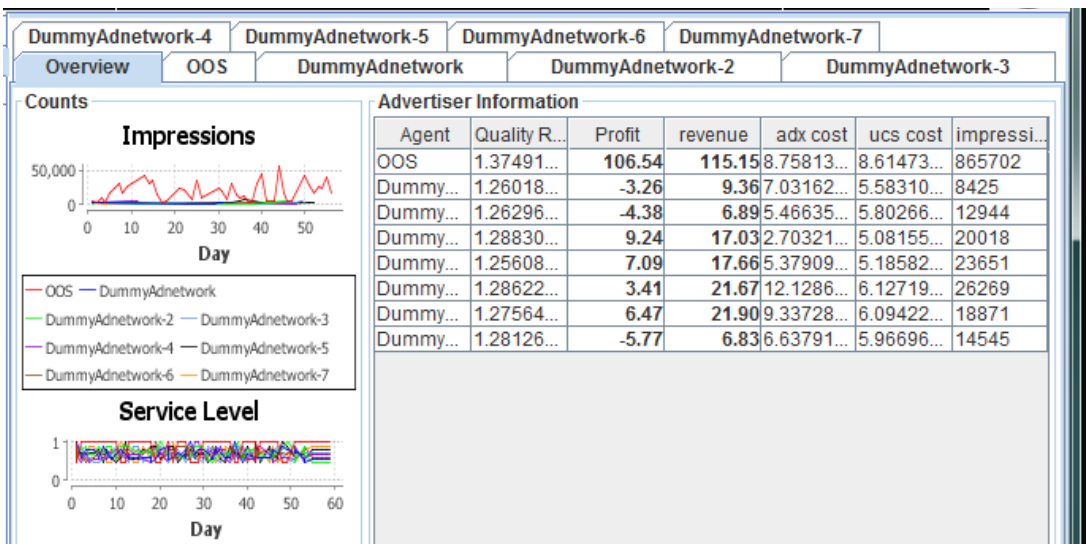
In this workshop we have implemented an agent which plays the game that was introduced at class. In this game, we (i.e. the agent we have implemented) need to play an advertising agency, and the aim of the game is, of course, to make the most money by the end of the game. The game features three main components: Contract Auction, AdX Real-Time Auction and User Classification Auction.

In the first part, the AdX Real-Time Auction, a matrix is filled. The matrix represents one's bids for each market segment (a subset of user space), and how much they are willing to offer for each "unknown" market segment for each site (to be explained later in the first part).

In the second part, The User Classification Auction, a bid is made for user classification. The agent that comes in the ith place (i.e. – the ith highest bid), gets the market segment in probability Loading... for each impression opportunity, where p=0.9. If the agent does not get the relevant market segment, the "unknown" policy is taken.

In the third part, the Contract Auction, a bid has to be given for an advertising campaign. The higher one's rating is and the lower their bid is, the more likely they are to win the campaign

Here are some results we got with our final agent:



Agent	Quality R...	Profit	revenue	adx cost	ucs cost	impressi...
OOS	1.37491...	**106.54**	**115.15**	8.75813...	8.61473...	865702
Dummy...	1.26018...	**-3.26**	**9.36**	7.03162...	5.58310...	8425
Dummy...	1.26296...	**-4.38**	**6.89**	5.46635...	5.80266...	12944
Dummy...	1.28830...	**9.24**	**17.03**	2.70321...	5.08155...	20018
Dummy...	1.25608...	**7.09**	**17.66**	5.37909...	5.18582...	23651
Dummy...	1.28622...	**3.41**	**21.67**	12.1286...	6.12719...	26269
Dummy...	1.27564...	**6.47**	**21.90**	9.33728...	6.09422...	18871
Dummy...	1.28126...	**-5.77**	**6.83**	6.63791...	5.96696...	14545

AdX Real-Time Auction - Bid Bundle calculation

This part comprises 6 parts:

1. Initialization of the ImpressionBidder & Classifier, including choice of the classifier.
2. Calculation of a bid for a new campaign, creating instances for it.
    2b. Analysis of market segments - used in bid calculation.
3. Calculation of the bid bundle given the initial bids - done in 2 tiers.
4. Updating the instances for the next bids to come.
5. Exception handling
6. Different Strategies used

Part 1. Initialization

- During initialization (first day of bid bundle calculation) we init the bidder with our current campaigns (that is, the initial campaign) and the chosen concrete classifier.
- We first began with Weka's implementation of LinearRegression. Later, as we saw the bids are not accurate enough, and tend to converge very quickly for all different instances given to it, we started testing other classifiers. We tried SMOReg (Support Vector Machine for Regression) but it was very slow, that it was late to send the bid bundle.
  Then we tried PaceRegression which gave no better results than the LinearRegression. At the end, RegressionByDiscretization was picked, which gave us both quick and more accurate results.
- At each day, when we received a publishers' report, we updated the bidder with the received data. According to this data, the initial bids are being calculated for new campaigns.

Part 2. Calculation of a bid for a new campaign

- When a new campaign arrives we generate its instances, that are to be classified by the concrete classifier picked.
- To calculate the bid, we consider mainly 2 parameters:
    - The **priority** of the campaign, relative to the other active campaigns (active - which has started and not ended yet, and the impressions got for it has not passed yet the threshold as defined later).
    - The **weight of the market segment and the device for a certain publisher**

(as we remember each bid is given to an AdxQuery which includes the publisher and the market segment).

- These 2 parameters help us set the initial bid (which is the class value of the created instance), as follows: Loading... Loading..., where:
    - Loading... (like in the sample agent).
    - Loading...
      Loading...
      Where "|*|" stands for the size of the set (i.e. the more specific the target segment, the higher the bid should be, as it is harder to satisfy).
      Notice that the weights and their median are relevant for the specific publisher.
        - Note: There was a difficulty with fulfilling campaigns with 3 market segments, so if Loading..., we multiply by 6, not 3.
    - Loading...
    - Loading...
    - **slowStartFactor** - a constant factor only added to the initial bid since we saw bids were not competitive enough until a few reports were received, leading to loss of critical impressions. If the campaign is the initial campaign, an even larger factor is used (again, after seeing that the initial slow-start factor is still too low for the initialCampaign, probably due to many intersections with the other agents' initial campaigns).
- Each instance created, with an initial bid, is kept in a map, so we are able to update the bid later when the AdNetReports arrive, and retrain the classifier.

Part 2b. Analysis of market segments

- The userAnalyzer class is responsible for aggregating publishers' catalogs and daily statistics in order to create better bids. It is initialized (hard-coded) with the data taken from the tables added as appendices to the official Specification Document of the AdX game.
- It has an API letting the clients query for:
    - The weight of a certain device for a publisher.
    - The weight of a certain market segment for a publisher.
    - Calculate median of weights of market segments for a publisher.
- In case no publisherStats are available for a certain publisher, it returns an average over all available publishers. This is used to calculate the adTypeOrientation. If no data is available the assumption will be the orientations are equal (using the popularity of the publisher).
- The class is also capable of returning a weights vector of all the possible Impression Parameters for a certain publisher - i.e. The weights of <market segment, device, ad type> for the publisher, for any market segment, device or ad type. This is used during the 1st tier bid calculation.

Part 3. Calculation of the bid bundle

- For every publisher we fill the bid bundle with the relevant queries.
- The 1st step is to calculate for the known market segments.
    - First we get the impression parameters weight vector for the publisher.
    - Then we filter the relevant campaigns for the impression parameters.
    - Calculating the first bid when the classifier is fed only with the weight of the impression parameters.
    - Generating a weight vector out of the relevant campaigns, which is normalized such that its sum is 1, and also calculating the factor used for the normalization (as the bid bundle accepts weights as ints, not as fractions, and we want to keep the same ratios).
    - “Enrich” the first instance with the priority (which is updated according to the relevant campaigns only and thus is higher than the priority given in the original instance)
    - Bid for the enriched instance
    - In debugging sessions we saw that normally the classifier will not give a different bid between the 2 tiers: initial and enriched.
    - The final bid given for all these relevant campaigns is the weighted average of the bids determined for each relevant campaign separately, where the relative priorities serve as the weights for average calculation.
    - We treated the bid bundle as it was keyed by the query and so we had to average our bids.
- The 2nd step is to calculate a bid for the unknown market segment.
    - This is done only if the UCS level is < 1.
    - We calculate again the priority-weight vector such that the most urgent campaigns will have the highest weight (these campaigns are of interest to us here).
    - We calculate the minimal priority in this weight vector.
    - We create an “unknown” instance for classification. The minimal priority is used to update the priority for the campaigns like this:
        - Loading...
    - Therefore each campaign gets an increased priority except for the least urgent campaign (since Loading...), and the more urgent the campaign the greater the increase in its priority.
    - The unknown instance is also kept in the map of instances (it will be updated only if there’s a “hit”, as will be explained later).
    - The unknown bid is a weighted average of the bids for each campaign, divided by a constant factor as we do not want to gamble and spend too much on “luck” - i.e. on unknown market segments.
- Last but not least, we set the daily spend limit, and set it to be the sum of the remaining budgets for all active campaigns.

Part 4. Updating

- When we receive the AdNetReport we use it to retrain our classifier. This phase is critical to the success of the agent.
- We get the AdNetReport and use the data from it:
    - Fetching the campaign from yesterday's active campaigns.
    - Bid count.
    - Win count.
    - Cost.
- If Loading... we calculate the second-price bid: Loading....
    - As we know that our win could affect the other agents and make them increase their bids as well, we add a random factor of 0-0.1 plus the Loading... ratio - such that if we had a really small number of wins, we will double the bid more or less (as Loading...).
- Otherwise, if Loading..., we increase our bid by a factor of Loading... (randomly, where x is constant and preset).
- The actual instance update process is a very technical one, involving taking all possible market segments out of the AdNetReport and updating only those instances which are relevant.
- Then we retrain the classifier with all corrected bids.

Part 5. Exception handling

- During the work, especially in the beginning of the implementation, several scenarios led to exceptions and to no bid bundle sent.
- Therefore, a method that mimics the behavior of the sample agent was implemented, and it did not throw any exception, so if a fatal error occurred in the classifier, and no bids could be generated, at least the default bid bundle would be sent.
- As the agent stabilized, this method was not used anymore as we encountered no fatal exceptions in the classifier.
- For non-fatal failures, for example in calculation of the initial bid, we are returning the last bid for the same publisher and the same impression parameters in the previous bid-bundle sent.
    - If there's no such previous bid, then it is "fatal" and we fallback to the default bid-bundle calculation as specified before.
- Improvement: we could have made a bid-bundle that is containing default bids only for those queries where it failed, but as we encountered no exceptions as we stabilized the agent, it was redundant.
- The only case when we saw the fallback mechanism get into actions is when we entered in the middle of the game, where our data structures weren't initialized properly.

Part 6. Different Strategies used

- We started with instances that included many attributes, for example we thought we would use nominal attributes for Device and AdType.
    - The regressions we chose did not cope well with the increase in dimension, leading to quickly converging values which were not optimal and did not allow for granularity in the bids.
- We also saw that serializing nominal values was quite disastrous and led to “explosions” in the bid, as there was no relation between the serialization of the nominal value to the bid itself.
    - Therefore the number of attributes was reduced, instead of serializing them to numeric values, and to reduce the dimension of the regression, and more importantly, to cut on values that may throw the class value off due to serialization. We were even surprised to discover that some bids became negative. But we also think that if the bid the classifier gives is so low, that it returns a negative number, then we should give 0, so not bidding at all. It didn’t happen often enough in practice to require more sophisticated strategy.
- During the competitions it was discovered that bids that were effective against the dummy agents were not competitive “in the real world”.
    - Furthermore, the problems were apparent right on the beginning when we saw that the first and second campaign were not satisfied at all.
    - This has lead to the definition of the slow start factor - whose purpose is to give a boost in the beginning of each campaign, to be fixed according to the wins as described above.
    - We also added an initial campaign slow start factor to boost even more the initial campaign, which was particularly problematic, since all agents got such a campaign and many times, there were intersections between the campaigns’ market segments, making it harder to win impressions.
    - Losses during the 1st days have affected the rest of the game - when the agent was unsuccessful in the first days, it was very difficult to recover.
    - Our conclusion was that the first days are critical, and as we only rarely encountered issues of too-high AdX costs, it was decided to increase the first bids by a much larger factor than we thought we would need in the beginning (2000 instead of 10 for the slow start factor!).

User Classification Auction

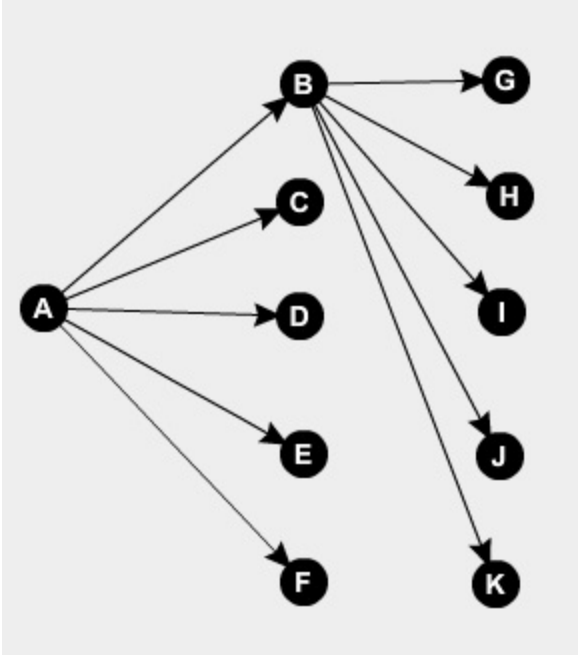
We have decided to implement the UCS bidder using a Q-learning algorithm. We have decided to use it because a reinforcement learning algorithm is favorable for our purposes – we ourselves don't know what the ideal bid for the UCS is, but a posteriori we can tell whether our bid was satisfactory or not.

Q learning is a state-based reinforcement learning algorithms that in each state, it can perform several actions. Based on these actions, the agent gets reinforcements (can be either positive or negative). The agent, based on the reinforcements, builds and maintains a table of the score it gives each action performed in each state. The greater and quicker the reinforcements the agent gets after performing a certain action in a specific state, the greater the score this action in this state gets in the table. The exact updating formula is as follows:

$$Q_{t+1}(s_t, a_t) = \underbrace{Q_t(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha_t(s_t, a_t)}_{\text{learning rate}} \times \left[\overbrace{\underbrace{R_{t+1}}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \underbrace{\max_a Q_t(s_{t+1}, a)}_{\text{estimate of optimal future value}}}^{\text{learned value}} - \underbrace{Q_t(s_t, a_t)}_{\text{old value}} \right]$$

To use Q-learning, we need to divide possible environments to states. To do so, we decided to use the following function from all possible environments to real numbers: For each active campaign (all campaigns whose end day is not overdue and which still have a positive number of impressions left to reach), and we divide the impressions left to reach by the total number of impressions needed to be reach. Then, we sum over all those campaigns and get a real number. To divide the environments to states, we simply divide the real numbers to ranges. The major question is – how many ranges and what will each range be? To decide, we used a method of implementing a few agents, with similar campaign bidders and impression bidders, but changed a few parameters in the UCS bidder – one bidder had 4 ranges, a second had 4 different ranges, another one had 5 different ranges etc. Then, we simply let them compete with one another for several matches and then picked the one which did the best (It's important to mention we did not choose our ranges completely randomly – we first run the agent to see what the function from possible environments to real numbers would return, and noticed that it rarely exceeded 4 (as we rarely have 4 active campaigns at once), so we understood we had to divide the range between 0-5 in a few possible ways.)

The Q-learning algorithm requires a Markov Decision Process. The MDP we have constructed is as follows:



The initial state is A. Then, based on the active campaigns we have - we move to one of the states B to F. The more active campaigns we have, the more we need the UCS and the “closer” to F we will go (i.e. - the state we will go to will have a letter closer to F in the alphabet). Then, based on the reinforcement we get, we advance to one of the states G to K, where the greater our reinforcement is the “closer” to G we will get. Note that there should have been edges from every vertex in {C,D,E,F} to every vertex in {G,H,I,J,K}, but these were omitted in order to keep the graph readable.

We used the aforementioned method to decide some more critic issues – such as, what would the possible bids be? What are the learning factor and the discount factor (parameters for Q-learning) going to be? (An example of running several agents with different parameters can be seen on the next page)

One more important parameter which had to be decided was, obviously, the reinforcement the bidder would get. We decided to use the misses (the bids for unknown impression opportunities we decided to bid for because we had urgent campaigns which we had to fill its quota) we had had the previous day as our main indicator whether our bid was too low (and we needed a higher UCS rating). At the end, we came up with this formula:

Loading...

The more we paid yesterday, obviously the smaller the reinforcement would be. But we also had to give the bidder negative reinforcements (otherwise, once we perform a certain action in a specific state, this action will always be performed when reached to this state). Consequently, we decided to subtract a constant from the fraction – the specific constant was achieved using the aforementioned method.

Several Agents Run - Example

In the following chart, you can see the results after playing the game with 5 different agents, all with a different Gamma factor (a Q-learning parameter). It can be easily seen that the agent with the Gamma factor=0.9 had the highest profit at the end of the game:

Gamma	Profit
0.95	14.5
0.9	14.76
0.85	7.28
0.8	-0.07
0.75	0.04

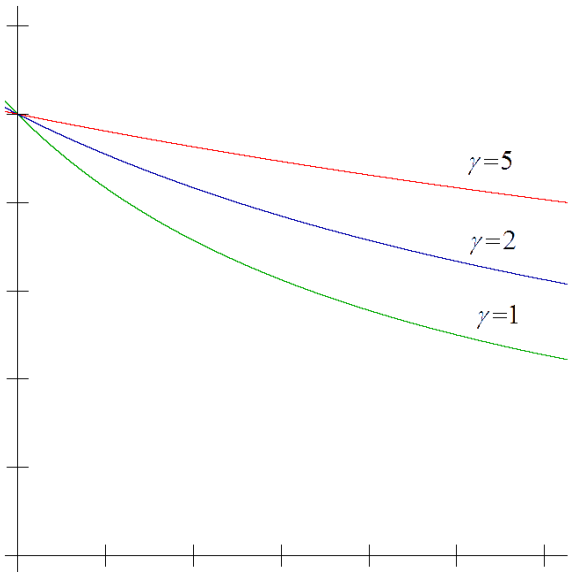
Campaign Auction

Implementation Main Concept

The main idea of the implementation of the campaign bidder was dictated by our initial design. We based our design on the assumption that less machine-learning is needed here, but more of taking all of the current factors into consideration, and then optimize it according to simulations results.

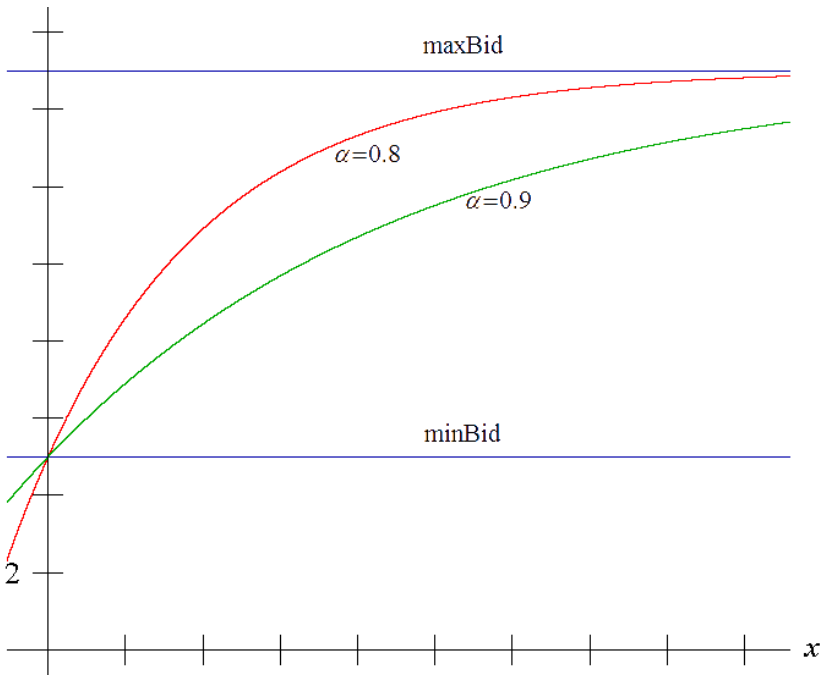
Formula Explanation

- We define a minimum and a maximum to bid:
  Loading...
  Loading...
  RCampaignMin and RCampaignMax are constants which are defined at *The Ad Exchange Game Specification*.
- Obviousely, *bid* needs to satisfy: *Loading...*
  Therefore, we use:
  Loading...
  The above assures that:
  Loading...
  Loading...
- Loading...controls how fast we want to raise our bids closer to maxBid (that is why a typical Loading...would be very close to 1.
- Loading... is the core calculation that contains game factors (formerly presented in our design specification):
  Loading...
- Loading... rates the chances to get impressions of the campaign's Target Segment. it is calculated by:
  Loading...
- Loading...takes the role of calibrating the effect (score decrease) of existing campaigns for the same target audience. bigger Loading...gives bigger drop gradient as can be shown in the graph:



Fixes and Optimization

Campaign bidder is optimized by the parameter Loading.... According to the formula, bigger Loading... means lower bid as can be shown:



Simulations summaries:

1. Beta variation. (Loading...)
5 games were executed in order to test the performance of OOS Agents with different Beta values against each other

Game 1

Agent	BETA	Profit	Revenue	Quality
OOS_1	930	-16	16	1.13
OOS_2	945	-10	6	1.2
OOS_3	960	-15	4	0.31
OOS_4	975	-22	2	0.22
OOS_5	990	-7	20	0.36

Game 2

Agent	BETA	Profit	Revenue	Quality
OOS_1	930	5	22	1.2
OOS_2	945	34	58	1.2
OOS_3	960	-32	0.1	0.1
OOS_4	975	-34	0.7	0.3
OOS_5	990	-13.5	25	1.2

Game 3

Agent	BETA	Profit	Revenue	Quality
OOS_1	930	-4	1.6	0.9
OOS_2	945	7	9.4	1.2
OOS_3	960	25	36	1.3
OOS_4	975	3	25	0.5
OOS_5	990	1.5	16	1.3

Game 4

Agent	BETA	Profit	Revenue	Quality
OOS_1	930	-11	4	1.1
OOS_2	945	4	15	1.3
OOS_3	960	-0.7	9	1.2
OOS_4	975	-9	1	1
OOS_5	990	-4	0	0.2

Game 5

Agent	BETA	Profit	Revenue	Campaigns Won	Quality
OOS_1	930	-45	2.9	2	0.4
OOS_2	945	-0.5	8.9	1	1.16

OOS_3	960	-16	0	1	0.4
OOS_4	975	-9	14	1	1.17
OOS_5	990	2.5	12	1	1.17

2. Focused Beta Variation -

Taking into consideration the results of the former 5 games, Beta range zoomed in

Game 6

Agent	BETA	Profit	Revenue	Quality
OOS_1	933	-50	8	0.4
OOS_2	939	-45	10	0.6
OOS_3	945	-33	0.4	0.2
OOS_4	952	-43	12	0.9
OOS_5	959	-2	30	1.2

Game 7

Agent	BETA	Profit	Revenue	Quality
OOS_1	933	-19	17	1.2
OOS_2	939	-6	5.5	0.9
OOS_3	945	-10	15.5	1.2
OOS_4	952	4	25	1
OOS_5	959	-14	8	1.3

Game 8

Agent	BETA	Profit	Revenue	Quality
OOS_1	933	-40	11	1.3
OOS_2	939	-44	4	1.25
OOS_3	945	0.93	14	1.23
OOS_4	952	14.5	28	1.27
OOS_5	959	25.12	40	1.33

Game 9

Agent	BETA	Profit	Revenue	Quality
OOS_1	933	-31	1.8	0.2
OOS_2	939	-51	6.6	1.2
OOS_3	945	-32	16.6	0.7
OOS_4	952	-39	3.5	0.3
OOS_5	959	-6	0.6	0.1

Game 10

Agent	BETA	Profit	Revenue	Quality
OOS_1	933	2.87	15	1.3
OOS_2	939	-26	2.8	0.3
OOS_3	945	0	2.4	0.2
OOS_4	952	2.81	40	1.2
OOS_5	959	16.7	44.4	1.3

3. Better Beta focus -

Game 11

Agent	BETA	Profit	Revenue	Quality
OOS_1	950	-37	3	1.2
OOS_2	953	-35	13	0.5
OOS_3	956	-32	8	1.2
OOS_4	959	0.34	20	1
OOS_5	962	-76	1	0.2

Game 12

Agent	BETA	Profit	Revenue	Quality
OOS_1	950	-30		1
OOS_2	953	-4		1.3
OOS_3	956	-31		1
OOS_4	959	3.2		1.3
OOS_5	962	0.7		1.3

Game 13

Agent	BETA	Profit	Revenue	Campaigns Won	Quality
OOS_1	950	-29	9.2	8	1
OOS_2	953	-4	22	7	1.3
OOS_3	956	-31	3.2	6	1
OOS_4	959	3.27	20	6	1.3
OOS_5	962	0.73	26	7	1.3

4. Smallest Beta Range -

Game 15

Agent	BETA	Profit	Revenue	Campaigns Won	Quality
OOS_1	957	-31	5	9	1.3
OOS_2	958	-3	15	8	0.3
OOS_3	959	-37	16	21	1.2
OOS_4	960	-40	2	4	0.2
OOS_5	961	-50	6	7	0.4

Game 16

Agent	BETA	Profit	Revenue	Campaigns Won	Quality
OOS_1	957	-12	14	15	1.2
OOS_2	958	15	35	13	1.2
OOS_3	959	-4	6	5	1.2
OOS_4	960	21	36	14	1.2
OOS_5	961	-3	0.2	1	0.4

Game 17

Agent	BETA	Profit	Revenue	Campaigns Won	Quality

OOS_1	957	18	28	6	1.3
OOS_2	958	-1	5	2	1
OOS_3	959	-5	3	3	1.2
OOS_4	960	8	15	6	1.3
OOS_5	961	13	24	13	1.3

Finally, the chosen Beta is 959.

5. Gamma variation:
Same tests with various Gamma values

Game 119

Agent	GAMMA	Profit	Revenue	Quality
OOS_1	0.5	-1.74	11.1	1
OOS_2	1	0.25	3.9	0.2
OOS_3	1.5	-9.5	11.8	0.6
OOS_4	2	-8.9	21.9	1.25
OOS_5	3	-3.1	9.7	1.2

Game 120

Agent	GAMMA	Profit	Revenue	Quality
OOS_1	0.5	-2.4	5.9	1.26
OOS_2	1	-2.6	5.05	1.24
OOS_3	1.5	11.05	30.6	1.16
OOS_4	2	-1.2	3.9	1.1
OOS_5	3	27.17	36.6	1.23

Game 122

Agent	GAMMA	Profit	Revenue	Quality
OOS_1	0.5	28	35.4	1.31
OOS_2	1	9	19.1	1.34
OOS_3	1.5	20	26.4	1.36
OOS_4	2	4.3	6.9	1
OOS_5	3	49	63	1.2

Game 123

Agent	GAMMA	Profit	Revenue	Quality
OOS_1	0.5	4.5	8.6	1.34
OOS_2	1	14	24.4	1.35
OOS_3	1.5	5.5	12.2	1.32
OOS_4	2	11	18.7	1.37
OOS_5	3	9.3	16	1.36

Game 124

Agent	GAMMA	Profit	Revenue	Quality
OOS_1	0.5	75.7	79	1.3
OOS_2	1	8.3	15	1.3
OOS_3	1.5	2.8	7	1
OOS_4	2	1.1	7.5	1.3
OOS_5	3	23.3	30	1.3

Game 125

Agent	GAMMA	Profit	Revenue	Quality
OOS_1	0.5	31		1.3
OOS_2	1	16		1.3
OOS_3	1.5	27		1.3
OOS_4	2	15		1.3
OOS_5	3	27		1.3

The result is not definite. since a wider Gamma range seem to make relatively little impact, there's no point to "zoom in" as in former tests. However, agent OOS_1 performed slightly better so the chosen Gamma is 0.5.