1 Introduction

In this lab, you will continue to develop agent strategies for bidding in simultaneous auctions on our TradingPlatform. Recall that many successful agent strategies are two-tiered, consisting first of a price prediction method, and second of an optimization method. Last week, you implemented an optimization method, namely **LocalBid**. This week you will be incorporating price prediction into LocalBid by first computing so-called **self-confirming price predictions (SCPP)** for LocalBidders. When multiple LocalBidders optimize with respect to SCPP for LocalBidders, they are indeed correct: i.e., self-confirming.

2 Code Installation

Important: Just like last week, you should create a new Eclipse project and add the following JAR file to your build path:

`/course/cs1951k/pub/2020/trading_platform/TradingPlatform.jar`.

If you have any issues with the setup, please ask a neighbor or a TA for help.

3 Recap of Last Time

Last week, we explored an optimization method called **LocalBid**, which iteratively calculates the marginal values of each good in a simultaneous auction, relative to the current bid vector and a predicted price vector. More specifically, for a good $g_j \in G$, LocalBid compares a bid vector $\boldsymbol{b}$ to a price vector $\boldsymbol{p}$ to determine which bundle of goods, other than g_j, the bidder can expect to win. Using its valuation function v, it then compares the value of this bundle of winnings, including and excluding g_j. The difference in these values represents the bidder's marginal value for g_j. LocalBid iterates this process, updating the bid vector with the newly calculated marginal value of each good, for a set number of iterations, or until convergence.

Note: If this description feel unfamiliar, we recommend reviewing the last lab before moving on.

The LocalBid algorithm takes as input the bidder's valuation function v. In addition, the algorithm initializes the bid vector $\boldsymbol{b}$ somehow. But where does the price vector $\boldsymbol{p}$ come from?

Last week, we provided your agents with a price vector $\boldsymbol{p}$ as input. But ordinarily, the agent is responsible for estimating $\boldsymbol{p}$ itself. Indeed, your goal today is to generalize your implementation of LocalBid from last week to one that estimates price vectors.

4 Learning in Self-Play

You may recall from Professor Greenwald's AAAI Panel video that many successful game-playing AI programs learn via self-play. As the name suggests, learning in this way means that an agent plays against itself repeatedly. In particular, in a two-player game, the agent assumes that its opponent is playing its own best strategy to date, and estimates a best response to that strategy. Then, during the next iteration, it assumes its opponent's strategy is the best response it learned during the previous iteration, and estimates a best response to *that* strategy; and so on. This process continues until (near) convergence, at which point the algorithm has learned a (near) symmetric equilibrium: i.e., a strategy that is a (near) best-response to itself.

Observe that the methodology at the heart of this training loop mimics our two-tiered agent architecture for bidding in auctions, namely first predict, and then optimize. When learning in self-play, an agent *predicts* its

opponent strategy, which it takes to be its own best strategy so far, and then *optimizes* against it. Likewise, we can wrap a training loop around our bidding agent architecture, in which an agent predicts that its opponents will bid according to its own best strategy to date, and then computes a near-best response to their collective behavior. The only minor caveat is that our agent architecture does not represent opposing strategies explicitly, but rather collapses them into price predictions, against which it optimizes its response.

Self-confirming price predictions in an auction are prices that are realized when all agents bid according to a two-tiered strategy (price prediction plus optimization), and the input to the optimization routine equals its output.[1] In other words, this strategy forms a symmetric equilibrium (i.e., it is a best response to itself).

5 Expected Marginal Values

Recall that equilibria are not guaranteed to exist in games unless agents are allowed to "mix" their strategies: e.g., in the context of auctions, agents often randomize their bids. Consequently, it is insufficient to predict deterministic price vectors $\boldsymbol{p}$. Rather, agents would do better to predict *distributions* over prices, and ideally, joint distributions, which can represent correlations in prices that reflect correlations in bidders' valuations for goods (i.e., complements and substitutes).

Generalizing the notion of marginal value from last week, we can compute **expected marginal values**, by taking expectations over marginal values with respect to a distribution of prices.

If each bidder i ascribes value $v_i(X)$ to $X \subseteq G$, and if $q(X) = \sum_{k \in X} q_k$, then the marginal value $\mu_{ij}(\boldsymbol{q})$ is:

$$\mu_{ij}(\boldsymbol{q}) = \max_{X \subset G \setminus \{j\}} v_i(X \cup \{j\}) - q(X) - \max_{X \subset G \setminus \{j\}} v_i(X) - q(X) \tag{1}$$

More generally, if the prices $\boldsymbol{q}$ are drawn from distribution $\boldsymbol{Q}$, the **expected marginal value** of good j is:

$$\bar{\mu}_{ij}(\boldsymbol{q}) = \mathbb{E}_{\boldsymbol{q} \sim \boldsymbol{Q}} \left[\max_{X \subset G \setminus \{j\}} v_i(X \cup \{j\}) - q(X) - \max_{X \subset G \setminus \{j\}} v_i(X) - q(X) \right] \tag{2}$$

Last week, we constructed examples in which bidding marginal values on all goods was not a good idea. Analogously, bidding *expected* marginal values on all goods is also not a good idea.

Question: Let $v(g_j) = v(g_k) = v(g_j g_k) = 2$. Assume the prices of goods g_j and g_k are independently distributed s.t. either is 1 or 101, each with probability $1/2$. Compute the marginal values of g_j and g_k under all four realizations of the price vector, and then take expectations to compute expected marginal values. What is the expected utility of bidding expected marginal values, given this price distribution?

Answer: Bidding expected marginal values yields expected utility $-1/4$. Bidding zero, which generates no utility, but likewise, no loss, dominates expected marginal value bidding in this example.

Our fix to this shortcoming for marginal value bidding was the **LocalBid** optimization routine, which bids marginal values, but not marginal values computed independently per good, rather marginal values such that each one depends on the marginal values of the others. We now generalize LocalBid's marginal value calculation to an analogous *expected* marginal value calculation.

The **marginal value** of a good j to bidder i, given a vector of bids $\boldsymbol{b}_i$ as well as a (deterministic) price vector $\boldsymbol{q}$, is simply the difference in value between having good j and not having it: i.e.,

$$\mu_{ij}(\boldsymbol{b}_i, \boldsymbol{q}) = v_i(x_i(\boldsymbol{b}_i, \boldsymbol{q}) \cup \{j\}) - v_i(x_i(\boldsymbol{b}_i, \boldsymbol{q}) \setminus \{j\}) \tag{3}$$

[1] In general, SCPP are defined relative to a *vector* of optimization routines, not just one; but for simplicity, we consider the symmetric case, in which all agents' valuation distributions and bid optimizers are the same.

More generally, if the prices $\boldsymbol{q}$ are drawn from distribution $\boldsymbol{Q}$, the **expected marginal value** of good j is:

$$\mu_{ij}(\boldsymbol{b}_i, \boldsymbol{q}) = \mathbb{E}_{\boldsymbol{q}\sim\boldsymbol{Q}}\left[v_i(x_i(\boldsymbol{b}_i, \boldsymbol{q}) \cup \{j\}) - v_i(x_i(\boldsymbol{b}_i, \boldsymbol{q}) \setminus \{j\})\right] \tag{4}$$

In today's lab, you will generalize your implementation of LocalBid in exactly this way—to bid based on expected marginal values, where the expectation is computed with respect to distributional price predictions, instead of bidding based on marginal values relative to deterministic price predictions. But first, we must build a representation of distributional price predictions.

6 Representing Price Predictions

There are multiple ways to represent a distribution. In today's lab, we will use arguably the simplest representation: a **histogram**. Moreover, while the quantity of interest is a vector of random variables—the price vector, specifically—we will assume independence among prices. That is, we will represent the joint distribution of a vector of m prices as m **independent histograms**.

A **histogram** is a special kind of bar chart for plotting a frequency distribution. For example, in the case of a single good whose price falls somewhere in the range $[0, 50)$, we could bucket prices into bins, such as

$$[0, 1), [1, 5), [5, 15), [15, 30), [30, 50)$$

The width of each bin is the magnitude of the range of possible outcomes it represents. These bins (which must be contiguous) are plotted on the x-axis. The height of each bin, plotted on the y-axis, is the corresponding density, meaning the frequency of outcomes that fall in this bin, divided by the bin's width. Note that the sum of the areas (widths times heights) in a histogram is proportional to the sample size (or 1, if the histogram has been normalized), so that a histogram is the discrete analog of a pdf.

Independent histograms means that we maintain a separate histogram to represent the price of each good. In contrast, a joint histogram representing the prices of two goods would populate two-dimensional bins: e.g.,

$$[0, 1) \times [0, 1), [0, 1) \times [1, 5), [0, 1) \times [5, 15), [0, 1) \times [15, 30), [0, 1) \times [30, 50)$$

$$[1, 5) \times [0, 1), [1, 5) \times [1, 5), [1, 5) \times [5, 15), [1, 5) \times [1, 5), [1, 5) \times [30, 50)$$

$$\vdots$$

$$[30, 50) \times [0, 1), [30, 50) \times [1, 5), [30, 50) \times [5, 15), [30, 50) \times [1, 5), [30, 50) \times [30, 50)$$

Using a joint, rather than an independent, representation, we would very quickly encounter the curse of dimensionality. Whereas learning m independent histograms of, say, 5 bins each, requires that we learn $5m$ parameters, learning a joint histogram over m goods requires that we learn 5^m parameters. For all but a very small number of goods and bins, it would be difficult to gather enough data for accurate learning.

That said, if possible, it is certainly preferable to model price predictions jointly, rather than independently.

Question: What are the advantages and disadvantages of representing the uncertainty in price predictions as a joint distribution over a vector of m prices rather than as m independent distributions. Construct an example in which an agent grossly overestimates or underestimates its expected value of its winnings (i.e. $\mathbb{E}_{\boldsymbol{q}\sim\boldsymbol{Q}}\left[v_i(x_i(\boldsymbol{b}_i, \boldsymbol{q})\right]$, given bid vector $\boldsymbol{b}_i$) because it chooses to represent uncertainty using independent distributions. **Hint:** Assume perfect complements or perfect substitutes, the former meaning an agent accrues no value whatsoever if it does not win all the goods in a bundle, and the latter meaning an agent accrues no additional value whatsoever for winning any additional good beyond just one.

6.1 Data Generation and Learning

Now that we have chosen a representation for our agent's uncertainty, we must discuss how to populate that representation; that is, how to generate data based on which your agents can learn. As discussed at the outset, your agents will learn in self-play. That means, that they will repeatedly simulate multiple auctions in which they bid against themselves—each set of which comprises one **epoch**—and then they will learn from the data collected during each epoch. More specifically, the agents will collect price data during each simulation. As these data are meant to summarize the behavior of the *other* agents in the simulation—not the learning agent—the relevant statistics are the highest bids on each good among all the *other* agents (i.e., not including the agent that is doing the learning).

After collecting an epoch's worth of data, the "learning" algorithm that is used to build a histogram is simply counting—counting up the number of times each good's price falls into the various bins of that good's histogram. Thus, after each epoch, m new histograms are learned. Then, these new histograms (i.e., the new information) are merged into the old. Specifically, they are "smoothed" into the old, meaning that before updating, the frequencies in the old histograms are reduced by some factor $(1 - \alpha)$, where $\alpha \in [0, 1]$ is a parameter generally chosen to be close to 0. Smoothing serves to prioritize more recent data over older data, which assists in convergence.

6.2 Implementation

Navigate to `SingleGoodHistogram.java`.

You will notice that the histogram is implemented as a `Map<Integer, Double>`. The integer keys are the lower bounds of the buckets (all assumed to be the same size), and the double values are the frequencies of prices falling in the respective buckets. For the tasks that follow, you may find the following patterns useful:

```
// loop through the buckets
for (int bucket = 0; bucket <= this.maxBucket; bucket += this.bucketSize) {
    double freq = this.buckets.get(bucket)  // get the frequency of prices in a bucket.
    this.buckets.put(bucket, ...);          // edit the frequency of a bucket.
}
```

You have also been provided with `int getBucket(double price)` which returns the key for the bucket a `price` falls into.

You need to implement the following methods:

- `addRecord(double price)` adds a data point to the histogram. You should increment the frequency of the correct bucket for `price`.
- `smooth(double alpha)` smooths the histogram. You should iterate over each bucket and multiply its frequency by `(1 - alpha)`.
- `update(SingleGoodHistogram newData, double alpha)` represents the step of updating a histogram with new data. This will occur every few simulations, when you have a new histogram full of data, and you want to update the old histogram to incorporate the new information (more on this later). This method should first smooth the old histogram (`this.smooth()`), and then for each bucket, increase its frequency by the corresponding frequency in the same bucket of `newData`. You need not worry about the bucket bounds not aligning; it is safe to assume `this.buckets` and `newData.buckets` have the same keys.

- `sample()` should return a sample of the price of the good, based on the frequencies in the histogram. We are leaving this open-ended, but our recommended method is to generate a random number z between 0 and 1, and return the z^{th}-percentile value.

Now, navigate to `IndependentHistogram.java`. This is a multiple-item, independent, smoothing histogram built upon the `SingleGoodHistogram` you just implemented. It is what your agents will use later in the lab to sample their price vectors from, for use in LocalBid.

`IndependentHistogram.java` filled in for you, but we encourage you to explore the implementation. You should notice a few things:

- `IndependentHistogram` is implemented as a `Map<String, SingleGoodHistogram` (where the `String` is the name of a good). This is maintaining a histogram independently, for each good.
- `sample()` constructs an `IPriceVector` by looping over your `SingleGoodHistogram.sample()` method for all goods.
- Similarly, `addRecords` and `update` treat each good independently.

7 LocalBid with Price Sampling

Now that you have a way to represent distributional price predictions and sample price vectors from them, you have the necessary tools in place to generalize last week's LocalBid agent to one that samples its price vectors repeatedly from an input distribution, in order to estimate expected marginal values.

7.1 Estimating Expected Marginal Values

The expected marginal value of a good, given a price distribution, can be estimated by averaging the marginal value of that good across multiple price vectors sampled from the distribution.

Below, we have provided pseudocode.

Navigate to `MarginalValues.java`.

Here, you should fill in the following method:

```
calcExpectedMarginalValue(
        Set<IItem> G, IItem good,
        IGeneralValuation v, IBidVector b, IIndependentDistribution P, int numSamples)
```

This function should estimate the expected marginal value of `good`, as described by the pseudocode.

`ICart` is the Trading Platform's representation of a (possibly singleton) bundle of goods. To look up the valuation of a bundle, you should use `v.getValuation(ICart cart)`, via the following pattern:

```
ICart c = new Cart();
for (IItem g : bundle) {
    c.addToCart(g);
}
double valuation = v.getValuation(c);
```

Algorithm 1 Estimate the expected marginal value of good g_j

INPUTS: Set of goods G, select good $g_j \in G$, valuation function v, bid vector $\boldsymbol{b}$, price distribution P
HYPERPARAMETERS: NUM_SAMPLES
OUTPUT: An estimate of the expected marginal value of good g_j

total_mv $\leftarrow 0$

for NUM_SAMPLES **do**
  $\boldsymbol{p} \leftarrow P$.`sample`() ▷ Sample a price vector.
  bundle $\leftarrow \{\}$
  for $g_k \in G \setminus \{g_j\}$ **do** ▷ Simulate either winning or losing good g_k by comparing bid to sampled price.
    price $\leftarrow \boldsymbol{p}_k$
    bid $\leftarrow \boldsymbol{b}_k$
    if bid $>$ price **then** ▷ The bidder wins g_k.
      bundle.`Add`(g_k)
    end if
  end for
  total_mv += $[v(\text{bundle} \cup \{g_j\}) - v(\text{bundle})]$
end for

avg_mv $\leftarrow$ total_mv / NUM_SAMPLES
return avg_mv

To get the price of an `IItem g` from a price vector `p`, use `p.getPrice(g)`. To obtain a price vector `p`, sample it from your distribution using `P.sample()`.

To get the assumed bid for `IItem g` from a bid vector `b`, use `b.getBid(g)`.

7.2 LocalBid: Determining the Bid Vector

Now, you can use `calcExpectedMarginalValue` as a subroutine within LocalBid. This new version of LocalBid will sample from a price distribution P, rather than a price vector $\boldsymbol{p}$.

We have provided pseudocode below.

Navigate to `LocalBid.java`.

Here you should fill out the following method:

```
localBid(Set<IItem> G, IGeneralValuation v, IIndependentDistribution P,
        int numIterations, int numSamples)
```

This method should return an `IBidVector` representing average marginal values for each good. You should use your `MarginalValues.calcExpectedMarginalValue()` method as a subroutine.

To set a value in an `IBidVector b`, use `b.setBid(IItem good, double bid)`.

Note that other than the call to `CalcExpectedMarginalValue`, and the parameters thereof, this week's LocalBid pseudocode is exactly the same as last week's LocalBid pseudocode. Feel free to borrow code from your implementation last week when writing this week's version.

If you run `LocalBid.java`, Java will output a few iterations of your bid vector in a sample case. If your

Algorithm 2 LocalBid with Price Sampling

INPUTS: Set of goods G, valuation function v, price distribution P
HYPERPARAMETERS: NUM_ITERATIONS, NUM_SAMPLES
OUTPUT: A bid vector of average marginal values

Initialize bid vector $\boldsymbol{b}_{\text{init}}$ with a bid for each good in G ▷ E.g., individual valuations.

for NUM_ITERATIONS or until convergence **do**
  $\boldsymbol{b} \leftarrow \boldsymbol{b}_{\text{init}}$.`copy`() ▷ Initialize a new bid vector to the current bids.
  for each $g_k \in G$ **do**
    mv $\leftarrow$ `CalcExpectedMarginalValue`$(G, g_k, v, \boldsymbol{b}, P)$
    $b_k \leftarrow$ mv ▷ Insert the average marginal value into the new bid vector.
  end for

  ▷ You can also try other update methods, like smoothing of the bid vector.
  ▷ This is also where you can check for convergence.
  $\boldsymbol{b}_{\text{init}} \leftarrow \boldsymbol{b}$
end for

return $\boldsymbol{b}_{\text{init}}$

implementation is correct, the marginal values of each good should "converge" somewhere between roughly 30 and 35. ("Converging" will still involve minor fluctuations due to all the randomness.)

8 Self-Confirming Price Predictions (SCPP)

As already noted, this week's implementation of LocalBid is not very different than last week's. In particular, whereas price predictions in the form of vectors were provided to LocalBid last week, price predictions in the form of distributions are provided to LocalBid this week. We are finally ready to address the question—where do these price predictions come from?

SCPP is a prediction method for an agent in an auction, which consists of simulating the auction, pitting the agent against copies of itself. More specifically, SCPP works as follows: it begins with some initial price distribution, and a given optimization routine.[2] It then simulates the auction some number (say T) of times, assuming a set of agents who optimize according to the input optimization routines, given the current price distribution. This simulation process generates T data points, each of which consists of an auction outcome, most notably, a price vector. These data points are then input into a learning algorithm, which incorporates them into the old price distribution to learn a new one. This entire process repeats for some number of iterations, until, hopefully, the price distribution has converged. The algorithm returns this price distribution, which can then be used in a live auction, in conjunction with the agent's optimization routine.

In your simulations, all the agents will play LocalBid, and the prices of the goods from those simulations will be inserted into the agent's histograms. What we mean by "prices" here can vary. The most straightforward approach would be to use the prices at which the goods sell for during the simulations. However, those prices would include the behavior of the learning agent itself. As the goal of learning is to predict the behavior of the *other* agents, not the learning agent, so that your agent's optimization routine can best-respond to that prediction, it should not learn from the actual sell prices, but rather from the other agents' highest bids.

Below, we have provided pseudocode for SCPP.

[2] It is also common to input multiple optimization routines: i.e., to assume different agents optimize differently.

Algorithm 3 SCPP

INPUTS: Set of goods G, optimization routine σ, valuation distribution F_i, initial price distribution P_{init}
HYPERPARAMETERS: NUM_ITERATIONS, NUM_SIMULATIONS_PER_ITERATION
OUTPUT: A learned price distribution

for NUM_ITERATIONS or until convergence **do**
  $P \leftarrow P_{\text{init}}$.`copy()` ▷ Initialize a new price prediction P to the current prediction.
  for NUM_SIMULATIONS_PER_ITERATION **do**
    For each agent $i \in [n]$, draw a valuation function v_i from F_i.
    Simulate an auction, with each agent playing $\sigma(v_i, P_{\text{init}})$.
    Store the resulting prices in the new distribution P.
  end for

  ▷ This is also where you can check for convergence
  $P_{\text{init}} \leftarrow$ `update`(P_{init}, P) ▷ Learn new prices from the simulation data, stored in the distribution P.
end for

return P_{init}

9 Implementing an SCPP/LocalBid Agent

You will now be implementing an SCPP/LocalBid agent which samples price vectors from your independent, smoothing histogram.

Navigate to `MySCPPHistogramAgent.java`.

First, take a look at `bid()`. This method returns the agent's next bid. It is already filled in – you should notice that it uses your LocalBid method to determine its next bid. Thus, if we simulated an auction with many copies of this agent, and updated a distribution, it would be equivalent to running SCPP with LocalBid as the optimization routine σ.

You should also notice the instance variable `learnedDistribution`. This represents the price distribution that the agent has learned: i.e., P_{init} in the pseudocode. You should also notice the instance variable `currDistribution`, which represents the distribution that the agent builds up in the inner loop (i.e. P).

Finally, you should notice the method `onValuationMessage`. This updates the instance variable `valuation` with a new valuation that is received from the auction server, upon each new simulation of the auction. This represents the step in SCPP of drawing valuations from the valuation vector.

9.1 IMPORTANT CAVEAT: READ CAREFULLY

The inner loop of SCPP entails simulating an auction many times. We would like to write SCPP so that it only necessitates starting up the TradingPlatform auction server once. So we will write it so that it all takes place in a single simulation, where the inner loop just represents a few consecutive repetitions of the auction, and then the distribution update in the outer loop is triggered after that amount of simulations.[3]

So the design of our SCPP agent will be similar to:

```
// constant
```

[3] We hope to eventually modify the TradingPlatform to avoid this messiness. If workarounds like this really irritate you, please consider applying to TA this course next year.

```
NUM_SIMULATIONS_PER_ITERATION = ...

// instance variables
simulationCount = 0
learnedDistribution = ...
currDistribution = ...

// playing LocalBid. this part is already filled in.
bid() {
    <play LocalBid with this.learnedDistribution and this.valuation>
}

// called after each simulation, when we have the results (prices) of the simulation.
update(ILinearPrices prices) {
    simulationCount++

    <insert prices into this.currDistribution>

    if (simulationCount % NUM_SIMULATIONS_PER_ITERATION == 0) {
        // update learnedDistribution, reset currDistribution
        this.learnedDistribution.update(this.currDistribution, ALPHA)
        this.currDistribution = this.learnedDistribution.copy()

        <save this.learnedDistribution to disk, for use in live auction>
    }
}
```

Please fill out the `update` method to implement SCPP. It should follow the pseudocode above; you will also find some helpful comments in the code to guide you. Once this is filled out, you will have an agent capable of learning to play LocalBid in a simultaneous auction.

10 Running your Agent

This week, your agent will compete in a very simple four-good, three-agent simultaneous second-price auction. (Next week, you will compete in a GVSM auction.) The valuation function includes some complements and substitutes, so hopefully a strategy that estimates marginal values will give you an edge. To compete in this auction, you will first train your agent (i.e., learn self-confirming price predictions), and then you will run it in a live auction.

At top of `MySCPPHistogramAgent.java`, you will find a constant `MODE`. It should be set to `TRAIN` when training your agent, and `RUN` when running it.

10.1 Training your Agent

Run `MySCPPHistogramAgent.java`, making sure that `MODE` is set to `TRAIN`. This will launch a 100-simulation training phase, in which your agent runs SCPP, training against LocalBid agents which use the same price distribution. Each time your agent updates its `learnedDistribution`, it will be written to disk, and immediately loaded by the opposing agents in order to ensure that the agents in the simulations are all optimizing from the same price distribution (once again, a workaround for TradingPlatform). Once this simulation

completes, your histogram will exist in a file, to be loaded in the next step.

At the end of the simulation, you will see a utility report (similar to those from the repeated games in our first few labs). If your implementation is correct, all of the agents should have positive utilities – the TA implementation usually lands near 3000, but there can be a lot of variation. The utility values may be quite close to each other, and your own agent may not come in first place. This is perfectly fine, as it is just the training phase – it is playing against agents with the exact same strategy, so you would expect equal outcomes (down to the randomness of the valuations, of course).

10.2 Running your Agent

Run `MySCPPHistogramAgent.java` again, this time with `MODE` is set to `RUN`. This will launch another 100-simulation run of the same auction, except this time you will be competing against two mystery agents, rather than copies of your own agent. At the start, your agent will load the histogram created in the training stage. It will then use that histogram as a representation of price predictions within the LocalBid strategy. Hopefully, the learned prices are good enough to estimate your agent's marginal values well, so that it can outperform the mystery agents.

Once again, you will see a utility report at the end of the simulations. This time, your agent's learned strategy should vastly outperform the strategies of the mystery agents – your agent should come in first place and receive at least a thousand more utility over the 100 runs than the next-best agent.