1 Introduction

In this lab, you will continue to develop agent strategies for bidding in simultaneous auctions. 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 Setup

You can find the stencil code for Lab 7 here. Once everything this lab is set up correctly, you should have a project with files for six Python classes:

- `marginal_values.py`
- `local_bid.py`
- `independent_histogram.py`
- `single_good_histogram.py`
- `competition_agent.py`
- `scpp_agent.py`

3 Recap of the Last Lab

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

A key component of successful AI game-playing programs, dating back to one of the earliest, a Checker-playing program written by Arthur Samuel in 1959[1]—is learning in self-play. As the name suggests, learning in this way means that an agent plays against itself repeatedly, each time improving its behavior slightly. In

[1] If you are interested in learning about the history of AI game-playing programs, we refer you to a AAAI 2020 Panel video moderated by Professor Greenwald.

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 best response; and so on. This process continues until convergence (or forced 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 a more compact representation of the relevant information they contain, namely price predictions, against which it optimizes: i.e., finds a (near) best 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.[2] 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 finite 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:

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

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

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

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.

[2] 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.

Our proposed fix to this shortcoming for marginal value bidding is 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 all 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.,

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

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 Price Predictions

There are multiple ways to represent and learn a distribution over a vector of random variables—in our case, prices. In today's lab, we will use arguably the simplest representation, a **independent histograms** (meaning we will ignore possible correlations among prices), and the simplest learning algorithm, counting.

6.1 Representation

A **histogram** is a special kind of bar chart for plotting a frequencies. 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 very quickly encounter the curse of dimensionality. Whereas learning m independent histograms of, say, 5 bins each, requires that we learn $5(m - 1)$

parameters, learning a joint histogram over m goods requires that we learn $5^m - 1$ parameters. For all but a very small number of goods and bins, it would be difficult to gather enough data for accurate learning in the latter case. That said, if possible, it is 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.

6.2 Learning

The "learning" algorithm that we will use 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 learning epoch, m new histograms will be learned, say P_i^{new}, for all $i \in [m]$. These new histograms (i.e., the new information) can then either replace, or be used to update, the old histograms, say P_i^{old}. *Smoothing* interpolates between these possibilities by way of a parameter $\alpha \in [0, 1]$ (typically close to 0): i.e., $P_i^{\text{old}} = (1 - \alpha)P_i^{\text{old}} + \alpha P_i^{\text{new}}$. The parameter α can be adjusted to prioritize more recent or older data, as appropriate.

6.3 Implementation

Navigate to `single_good_histogram.py`, where we have provided stencil code for a histogram of the prices of a single good. This histogram is implemented as a `Dict<Integer, Double>`. The integer keys are the lower bounds of the bins (all assumed to be the same size), and the double values are the frequencies of prices falling in them. (Note that the terms bins and buckets are used interchangeably in this context.)

We have provided `int get_bucket(double price)`, which returns the key for the bucket where a `price` falls.

Task: Implement the following methods:

- `add_record(double price)` adds a data point to the histogram. You should increment the frequency of the bucket corresponding to `price` and the total frequency across all buckets.
- `smooth(double alpha)` smooths the histogram. You should iterate over each bucket and multiply its frequency by `(1 - alpha)`.
- `update(SingleGoodHistogram new_hist, double alpha)` represents the step of updating a histogram with new data. This will be called 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 (`self.smooth()`), and then for each bucket, it should increase its frequency by `alpha` times the corresponding frequency in the same bucket of `new_hist`. You can assume `self.buckets` and `new_hist.buckets` have the same keys.
- `sample()` should return a sample of the price of a good, based on the frequencies in the histogram. We are leaving the implementation of this method open-ended, but our recommended approach is to generate a random number z between 0 and 1, and return the value at the z^{th}-percentile. (To avoid sampling from an empty histogram, you can initialize all bucket counts to 1.)

Now, navigate to `independent_histogram.py`. This code builds on the `SingleGoodHistogram` you just implemented to create a multiple-good, independent, smoothing histogram. `independent_histogram.py` is 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). Thus, it maintains a histogram for each good independently.
- `sample()` returns an `Dict<String, Double>`, generated by looping over your `SingleGoodHistogram.sample()` method, for all goods.
- Similarly, `add_records` and `update` treat each good independently.

7 LocalBid with Price Sampling

Now that you have a way to represent and learn 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. We provide pseudocode below.

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

totalMV $\leftarrow 0$

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
  totalMV += $[v(\text{bundle} \cup \{g_j\}) - v(\text{bundle})]$
end for

avgMV $\leftarrow$ totalMV / NUM_SAMPLES
return avgMV

Navigate to `marginal_values.py`.

Task: Fill in the following method, to estimate the expected marginal value of `good`:

```
calculate_expected_marginal_value(
        Set<String> goods,
        String selected_good,
        Func<Set<String>, Double> valuation_function,
```

```
        Dict<String, Double> bids,
        Histogram price_distribution,
        int numSamples)
```

7.2 LocalBid: Determining the Bid Vector

Next, you will use `calculate_expected_marginal_value` as a subroutine within LocalBid. This new version of Local Bid takes as input a price distribution P, rather than a price vector $\boldsymbol{p}$.

We provide pseudocode below.

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

for NUM_ITERATIONS or until convergence **do**
  $\boldsymbol{b}_{\text{new}} \leftarrow \boldsymbol{b}_{\text{old}}$.`copy`() ▷ Initialize a new bid vector to the current bids.
  for each $g_k \in G$ **do**
    MV $\leftarrow$ `CalcExpectedMarginalValue`$(G, g_k, v, \boldsymbol{b}_{\text{old}}, P)$
    $\boldsymbol{b}_{\text{new},k} \leftarrow$ 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.
  $\boldsymbol{b}_{\text{old}} \leftarrow \boldsymbol{b}_{\text{new}}$
end for

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

Navigate to `localbid.py`.

Task: Fill in the following method:

```
local_bid(Set<String> goods,
          Func<Set<String>, Double>  valuation_function,
          Histogram price_distribution,
          int numIterations,
          int numSamples)
```

This method should return an `Dict<String, Double>` that stores the average marginal values for each good. You should use your `calculate_expected_marginal_value` method as a subroutine.

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 `local_bid.py`, you will see as output a few iterations of your bid vector in a sample case. If your 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 in the setup.)

8 Self-Confirming Price Predictions (SCPP)

As already noted, this week's implementation of LocalBid is not very different than last week's. The only difference is: 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?

Your agents will construct their price predictions (i.e., learn) from 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).

More specifically, SCPP works as follows. Given an initial price distribution, and an optimization routine (e.g., LocalBid),[3] SCPP simulates the auction some number (say T) of times, assuming a set of agents who optimize according to the input optimization routine, 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.

Below, we provide pseudocode for SCPP.

Algorithm 3 SCPP

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

for NUM_ITERATIONS or until convergence **do**
  $P_{\text{new}} \leftarrow P_{\text{old}}$.`copy`() ▷ Initialize a new price prediction P_{new} to the current price 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 $\sigma(v_i, P_{\text{old}})$.
    Store the resulting prices in the new distribution P_{new}.
  end for

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

return P_{old}

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.

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

9 Implementing an SCPP/LocalBid Agent

Your final (and primary) task in this lab is to implement an SCPP/LocalBid agent, which samples price vectors from your independent, smoothing histogram. Navigate to `scpp_agent.py`.

First, take a look at `get_action()`. This method returns the agent's next bid vector, which it finds by running the LocalBid function enhanced with expected marginal values using the current learned price distribution. **N.B.** If we simulate an auction with many copies of this agent, and then learn a distribution from the data generated, this would be an implementation of SCPP with LocalBid as the optimization routine σ.

You should notice two important instance variables:

- `learned_distribution` represents the price distribution P_{old} that the agent has learned so far
- `curr_distribution` represents the distribution P_{new}, which is constructed in the inner loop from simulation data.

The SCPP agent also has several useful methods, such as `get_valuations()` and `get_opp_bid_history()`. The valuations are retrieved from the auction server upon each new simulation of the auction. This corresponds to the step in the SCPP pseudocode where valuations are sampled from their distributions.

SCPP entails simulating an auction many times. The update to the learned distribution is triggered only after a fixed number of simulations (denoted by `NUM_SIMULATIONS_PER_ITERATION`).

```
// constant:
NUM_SIMULATIONS_PER_ITERATION = ...

// instance variables:
simulation_count = 0
learned_distribution = ...
curr_distribution = ...

// the agent computes its bid vector by calling local_bid with learned_distribution
// this code is already filled in
get_action: ...

// called after each simulation, once we have the results (prices) of the simulation
update:
    other_bids_raw = self.get_last_opp_bids()

    // TODO: populate predicted prices per good
    predicted_prices = {}
    for good in self.goods:
        predicted_prices[good] = ???

    self.simulation_count += 1
    // TODO: insert prices into self.curr_distibution

    if simulation_count % NUM_SIMULATIONS_PER_ITERATION == 0:
        // TODO: Update the learned distribution with the newly gathered data

        // TODO: Reset the current distribution

        // TODO: Save the learned distribution to disk (for use in live auction mode)
```

Task: Fill in the `update` method to implement SCPP. Your implementation 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 a price distribution with which to play LocalBid in a simultaneous auction.

10 Running your Agent

This week, your agent will compete in simultaneous second-price auctions against truthful agents. Before doing so, you should train your agents to learn self-confirming price predictions against the truthful agents.

Then, as usual, we will run a live, class-wide competition. When we do so, you may choose to compete with an altogether different strategy; for this purpose, we provide `competition_agent.py`.

When running `scpp_agent.py`, you can run it with an argument `--mode`. It should be set to `TRAIN` when training your agent, and `RUN` when running it in the live auction.

10.1 Training your Agent

Run `scpp_agent.py`, making sure that `--mode` is set to `TRAIN`. This will launch a 100-simulation training phase, in which your SCPP/LocalBid agent trains against other LocalBid agents that use the same predicted price distribution. Each time your agent updates its `learned_distribution`, it will be written to disk, (and the same by the opposing agents) in order to ensure that the agents are all optimizing using their current price prediction. Once training completes, your histogram will have been created and written to a file, to be loaded in the next step.

At the end of training, the AGT_SERVER will output a utility report (similar to those output after playing the repeated games of our first few labs). The actual utility values may be quite close to each other, and your own agent may not come in first place. This is perfectly fine, as your agent is playing against agents employing the exact same strategy, so you would expect equal outcomes (up to the randomness in the simulations, of course).

10.2 Running your Agent

Run `scpp_agent.py` again, but this time set `--mode` 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 during the training phase, to use as the price prediction input to LocalBid. 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, the AGT_SERVER will output a utility report at the end of the simulations. This time, your agent should vastly outperform the other, mystery agents. Your agent should come in first place, and earn average utility that is slightly more than that of the next-best agent over the course of the 500 runs.

11 Class Competition

Having now implemented some advanced learning strategies for a Simultaneous Auction, we will once again play a total of 100 rounds against every other player). This time, instead of competing against TA bots, your bots will face off against those of other students. You are free to use any strategy you like in this competition,

whether inspired by SCPP or otherwise. However, to keep the competition engaging, your strategy should not be constant. Hope you have fun!