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**, and you incorporated price predictions into LocalBid by computing so-called **self-confirming price predictions (SCPP)** for LocalBidders. If price predictions converge when multiple LocalBidders optimize with respect to SCPP for LocalBidders, then they are indeed correct: i.e., self-confirming.

Last week's price predictions were point estimates, meaning they were single numbers, so did not quantify the uncertainty in those predictions in any way. This week you be extending those predictions to probability distributions, thereby quantifying the uncertainty in the predictions.

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.

4 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 over 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.

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.

5 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, **independent histograms** (meaning we will ignore possible correlations among prices), and the simplest learning algorithm, counting.

5.1 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.

5.2 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.

5.3 Sampling

There are multiple ways to sample from histograms, but the simplest way is called **cumulative distribution function (CDF) sampling**. It works as follows:

1. Normalize the counts in the histogram to generate probabilities.
2. Convert the ensuing probabilities into a discrete CDF.
3. Generate a random number z uniformly between 0 and 1.
4. Return the value at the z^{th}-percentile.

5.4 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[int, float]`. 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 `def get_bucket(price: float) -> int`, which returns the key for the bucket in which a `price` falls.

Task: Implement the following methods:

- `def add_record(price: float)` 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.
- `def smooth(alpha: float)` smooths the histogram. You should iterate over each bucket and multiply its frequency by `(1 - alpha)`.
- `def update(new_hist: SingleGoodHistogram, alpha: float)` represents the step of updating a histogram with new data. This method 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). When called, it 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. (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 `dict[str, SingleGoodHistogram]`, where the `str` is the name of a good. Thus, it maintains a histogram for each good independently.
- `sample()` loops over `SingleGoodHistogram.sample()`, and then returns a `dict[str, float]`.
- Similarly, `add_records` and `update` treat each good independently.

6 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.

6.1 Estimating Expected Marginal Values

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

Navigate to `marginal_values.py`.

Task: Fill in the following method, which returns an estimate of the marginal value of `selected_good`, i.e., its average marginal value, averaged over `num_samples` samples:

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

```
def calculate_expected_marginal_value(
        goods: set[str],
        selected_good: str,
        validation_function: set[str],
        bids: dict[str, float],
        price_distribution: Histogram,
        num_samples: int
) -> float
```

6.2 LocalBid: Determining the Bid Vector

Next, you will enhance your implementation of LocalBid to take as input a price distribution P, rather than a price vector $\boldsymbol{p}$. To do so requires that you update your `local_bid` implementation to calculate expected marginal values, instead of marginal values. We provide pseudocode below.

Navigate to `local_bid.py`.

Task: Fill in the following method:

```
def local_bid(
    goods: set[str],
    valuation_function: Callable[[set[str]], float],
    price_distribution: Histogram,
    num_iterations: int,
    num_samples: int
) -> dict[str, float]
```

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., Use 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**
    AMV $\leftarrow$ calculate_expected_marginal_value$(G, g_k, v, \boldsymbol{b}_{\text{old}}, P, \text{NUM_SAMPLES})$
    $\boldsymbol{b}_{\text{new},k} \leftarrow$ AMV ▷ 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 $\boldsymbol{b}_{\text{old}}$

Observe that `local_bid` returns a `dict[str, float]`, which stores the average marginal values of all goods.

Other than the call to `calculate_expected_marginal_value`, 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.

Task: Run `local_bid.py` and observe the bid vectors output over a few iterations. 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.)

7 Self-Confirming Price Predictions (SCPPs)

As already noted, this week's implementation of LocalBid is not very different than last week's. Like last week, your agent will construct its price predictions by learning them in self-play: i.e., from repeated auction simulations, in which its bidding strategy (LocalBid) is employed by all the bidders. 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 (i.e., histograms) are provided to LocalBid this week.

Recall that **self-confirming price predictions** 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. In other words, the optimization routine forms a symmetric equilibrium (i.e., it is a best response to itself).

Below, we provide the same pseudocode as last week for learning SCPPs, which, once again, entails simulating an auction many times. The learned prediction is updated `NUM_ITERATIONS` times, with an actual update triggered only after a fixed number of simulations, namely `NUM_SIMULATIONS_PER_ITERATION`.

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}

8 Implementing an SCPP/LocalBid Agent

This section is nearly identical to Section 6 of Lab 6. All changes are italicized.

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

For starters, take a look at `get_action()`. This method returns the agent's next bid vector, which it finds by running `local_bid` to compute *expected* marginal values given a *distribution* of self-confirming prices.

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 variables:

- `learned_distribution` represents the price *distribution* P_{old} that the agent has learned so far
- `curr_distribution` represents the price *distribution* P_{new} that is constructed from current simulation data

The SCPP agent also has several useful methods, such as `get_valuations()` and `get_opp_bid_history()`. The former retrieves valuations 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.

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 method is filled in, you will have a LocalBid agent capable of learning a self-confirming price *distribution* in simultaneous auctions.

9 Running your Agent

Like last week, your agent will again train in self-play, and then compete against truthful agents. But more than that, we will also 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 a local competition or live auction.

9.1 Training your Agent in Self-Play

Run `scpp_agent.py`, making sure that `--mode` is set to `TRAIN`. This will launch a 500-simulation training phase, in which your SCPP/LocalBid agent trains in self-play: i.e., against multiple copies of itself. Each time an agent updates its `learned_prices`, they will be written to disk, to ensure that the agent optimizes using its current price prediction. Once training completes, the final learned price predictions are likewise written to a file to be loaded when the agent is run in a local competition or a live auction.

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 utility values across agents may be quite close to each other, and your agent may not come in first place. This outcome is perfectly fine, as your agent is playing against itself, so you would expect equal outcomes—up to the randomness in the simulations, of course!

9.2 Running your Agent against Truthful Agents

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 your trained agent will be competing (locally) against two truthful agents, rather than copies of itself. At the start, your agent will load the price prediction created during the training phase. Hopefully, these learned prices are good enough to estimate your agent's marginal values well, so that LocalBid with these predictions can outperform the truthful agents.

Once again, the AGT_SERVER will output a utility report. Your agent should outperform the other agents, earning average utility that is (slightly) higher than that of the other agents over the course of the 100 runs.

Question: Does your agent's performance against the truthful agents vary with the structure of the agents' valuations? That is, how does your agent perform when valuations are additive compared to when they exhibit complementary or substitutable structure?

10 Class Competition

Having simulated a few simultaneous auctions with basic learning strategies, we will finish this lab with a live competition, where your agent will face off against other students' agents. You are free to use any strategy you like in this competition, whether inspired by LocalBid/SCPP or otherwise. Hope you have fun!