Final Report 2025

Part 1: Demand estimation and price optimization (without competition)

Demand Estimation

I framed demand estimation as a binary classification task. Given a price and three customer covariates, predict the probability of purchase. My initial model, developed independently in the *old_model.ipynb* file, used logistic regression with standardized features. While simple and fast, this baseline could not capture the nonlinear price-demand relationship.

After establishing that baseline and speaking with Professor Garg, I broadened to random forests and gradient-boosted trees, performing systematic hyperparameter sweeps for each model family. These experiments are documented in my *create_model.ipynb* file. To evaluate my model, I used ROC-AUC values that came directly from 5-fold cross-validation, where the model was repeatedly trained on 80% of the data and evaluated on the remaining 20%. Random forests achieved strong accuracy but were slower at inference, which mattered given my need to evaluate many prices per customer. Gradient boosting provided the best tradeoff between accuracy and speed. My best model was a GradientBoostingClassifier with 100 estimators, learning rate 0.1, and max depth 7, achieving an ROC-AUC of 0.994.

Once I selected the boosted model, I retrained it on the full dataset and saved both the model and the fitted scaler. These are the exact files loaded in my deployed agent, ensuring consistency between training and inference. Before moving to pricing, I used the model to evaluate a dense price grid for each training customer and calculated expected revenue for every candidate price. This yielded per-customer revenue-maximizing prices. I also computed the average maximum expected revenue across customers from the training dataset, approximately $63.62, which later served as the baseline for my opportunity-cost threshold.

Compared to HW 3, which focused on static, per-customer pricing, my final agent incorporated several major additions. I moved from logistic regression model to gradient boost (changed feature complexity), evaluation moved from revenue based only to a cross-validated ROC-AUC, and I introduced fully personalized pricing by estimating a complete demand curve for each customer. I also replaced the Bellman-style dynamic programming approach with a heuristic threshold policy to manage inventory under capacity constraints.

Price optimization

With my demand model in place, I turned to designing a price-optimization strategy that could run efficiently under strict timing and inventory constraints. The core of my approach was to compute expected revenue as price x predicted purchase probability. This entire step is

implemented in a fully vectorized manner. Rather than constructing a new DataFrame or recomputing feature structures for every customer, I preallocated a fixed NumPy array (_base_features) during initialization and reused it throughout the simulation. Only the covariate columns are updated per customer. This avoided repeated memory allocation and DataFrame overhead and was one of the main contributors to my exceptionally low per-customer runtime. Having a low per-customer run-time was important to me to match real-world applications.

My price-optimization logic was originally developed while experimenting with the logistic regression baseline. When I transitioned to the gradient-boosted model, I made only minimal changes to this logic. With leaderboard evaluations no longer running frequently, and with only one more run left, I wanted to avoid introducing unnecessary risk. The boosted model fit directly into the same vectorized evaluation framework, meaning all of my runtime improvements and expected-revenue computations carried over without needing redesign.

Because the environment replenished inventory every 20 customers, I did not use the revenue-maximizing price automatically. Selling to a low-value customer early could prevent me from serving a high-value customer later, while being too conservative could leave inventory unused by the time replenishment occurs. To account for this, I incorporated a dynamic threshold mechanism. The threshold represents the minimum expected revenue a sale must exceed to justify using up one unit of inventory. Its baseline is derived from the average optimal expected revenue computed using the training dataset ($63.62), as explained in the demand estimation section. I then scaled this value so that it is comparable to predicted expected revenue magnitudes. The threshold is adjusted based on current inventory and customers remaining until replenishment. When inventory is high, the threshold is lowered. When inventory is scarce relative to remaining customers, the threshold rises. When replenishment approaches, the threshold decreases automatically to prevent leftover stock. If the customer's best expected revenue exceeds this threshold, I offer the corresponding price. Otherwise, I effectively reject the customer using a very high "reject" price.

I evaluated this strategy using offline analysis, small local simulations, and the course leaderboard. Offline, the expected-revenue curves from my model (in create_model.ipynb) exhibited the anticipated single-peaked structure. Locally, simulations confirmed sensible pricing decisions while maintaining low per-customer run-time. A challenge with these local tests was that they evaluated only a small number of randomly chosen customers (100 customers per run), which caused noticeable variability between runs and made it difficult to distinguish true improvements (seen at bottom of run_gym.ipynb). Final validation came from the course leaderboard, where my agent achieved competitive profit with no timed-out actions.

Part 2: Pricing under competition

Pricing Under Competition Strategy

The strategy for the second part of this project was built on top of the threshold based expected-value framework from the first part. I added opponent aware code on top of the preexisting baseline. While still using expected revenue, $p^* = p \times P(\text{purchase} \mid p,x)$, as my baseline objective and applying a dynamic threshold to decide when to sell to a customer, I made several edits to make this baseline adaptive to opponent behavior. This was achieved by modeling the opponent's prices using a ridge regression model. This model was trained on historical data including their previous prices, the customer's features, and the outcomes (whether I won or lost that round). While this did not perfectly forecast the actions of my opponents it helped me understand if they were using aggressive, or moderate pricing tactics. I also used α to adjust the baseline price ($\alpha \in [0.5,1]$), and the value of α was based on how aggressive or moderate the opponents' price prediction tactics were deemed to be. If the opponent was predicted to have a more laid back pricing method, then my agent priced closer to p^*. If the opponent was found to have an aggressive pricing tactic, meaning they would constantly undercut my price, then my agent lowered the price accordingly. This allowed me to build a model that was able to keep the initial high prices when possible, but also able to shift the price and price lower when needed.

Changes from Part 1

There were several changes that were made to the first part in order to adapt to the new problem posed. By creating a new variable for the opponent's price prediction I was able to track information like what their proposed price was, what were the buyer's covariates and whether or not my agent beat them for that round. As previously mentioned, a ridge regression model was then trained on this data after enough was collected. Additionally, instead of having a binary accept or reject approach to the price (p^*), I changed my decision making strategy to follow a continuous flow of price adjustments by adjusting the price using α. This allowed my agent to move freely between fair pricing and undercutting my opponent when necessary. As previously mentioned I also categorized each opponent as fair or aggressive, based on their pricing strategy, and this information was updated each round. This allowed my agent to have a more dynamic reaction to the decisions of its opponents. Another difference within the agent from its construction in part one was how inventory was considered. While in part one I relied only on the inventory parameter θ, in part two I relied on changes in α that would reflect how urgent it was to sell to a given customer. If it was predicted that the opponent would price aggressively, then my agent would price lower to avoid losing several rounds in a row and becoming less competitive.

Additionally, in order to improve price prediction of the agent new prediction models were tested, notably gradient boosting trees and neural networks. These machine learning models

are more powerful than the logistic regression previously employed in part one. I tested several variations of each of these models with different hyperparameters such as learning rates and max depth. I then trained these models on the given pricing data and evaluated the accuracy of their predictions. The best performing model ended up being Neural Networks with three hidden layers (128, 64, 32). I saved this model and the scaler into a pickle file and it was used for price prediction in my agent.

Evaluating the Results

My evaluation focused on validating the accuracy of the core demand model and determining the optimal parameters for computational efficiency.

First, I tried different classification models to predict the purchase probability (which is the foundation of the expected revenue calculation). I used 5-fold* cross-validation and the ROC_AUC score to compare performance. The best model from Part 1 (Logistic Regression) achieved a ROC-AUC of 0.9366. After testing XGBoost and Neural Networks, the Neural Networks with three hidden layers showed the best performance with a ROC-AUC score of 0.9953. Because of this significant increase, I decided to replace the Logistic Regression with the Neural Network model.

Second, I evaluated the necessary density of the price search grid(Ngrid) to find the true revenue maximizing price p* without having excessive computational time. Testing grid sizes from 50 to 400 showed that using Ngrid=200 results in a very low average regret (0.5355%) with a reasonable runtime of 4.98 seconds, using Ngrid=400 only improved 0.0395 but increased the runtime significantly (7.06 seconds). I decided to use Ngrid=100 for the competitive scenario as a good balance between speed and accuracy to prioritize speed in this time-constrained environment

Other Things Tried or Considered

I considered several benchmarks to ensure my strategy was robust. Even in the competition, I tested my agent against the assumption of a purely greedy p* maximizer (the part 1 baseline). Then I tested it against a model assuming aggressive competitive undercutting. My agent was designed to revert to p* when the predicted model opponent price was high (in order to effectively maximize revenue against greedy opponents), but to deploy competitive undercutting when the opponent was predicted to price low.

In order to explore, I implemented an exploration phase. In the first 20 rounds, my agent randomly probed the opponent's response by setting a price that deviated from the optimal competitive price in 10% of the rounds. This strategy was essential for gathering rich data points (opponent price, my probe price, customer covariates) to effectively train the ridge regression opponent model.

The competition objective of maximizing the overall revenue, strongly influenced my decision to use the threshold price logic θ. This logic represents the opportunity cost of selling now and allowed me to be highly selective. If a customer's expected revenue was below θ, I rejected them to save inventory for later (hopefully for higher-value customers). This was thought in order to maximize the profit over the entire competition period rather than just optimizing for individual round wins.

I used basic concepts from behavioral game theory by incorporating a Cooperation Score (based on Prisoner's Dilemma as part of strategy instructions for Part 2) . This score, updated each round based on the opponent's pricing behavior, modulated my competitive price adjustment. If the opponent showed cooperative(high) pricing, my agent would price closer to p*, rewarding and encouraging continued high prices, which ultimately maximised joint revenue and then my individual revenue.

The need to account for capacity resulted in a key aspect to my θ calculation. The threshold θ was adjusted not only by my remaining inventory but also by the opponent's remaining inventory. For instance, if the opponent was out of stock, θ was lowered significantly, encouraging me to price more aggressively as the opponent couldn't compete.

Post-Leaderboard Learning and Impact

Following the leaderboard, I learned the importance of the balance between being aggressive against high value customers when necessary but maintaining prices against non-responsive opponents. This confirmed the justification for my dial logic of using p* as a high starting point and only lowering it when the opponent's predicted price forced a competitive undercutting.

An additional change was the addition of a smoothing factor (alpha=0.25) to the opponent price prediction. This was critical because the opponent's price could fluctuate, and smoothing the prediction prevented my agent from overreacting to a single aggressive or passive move. I also applied a final step to the competitive price to ensure it stayed within a reasonable range (between 0.65 x p* and 1.0 p*). This was done to ensure stability in competitive pricing while still avoiding a deep race-to-the-bottom (as talked about in class), then optimizing my revenue in a competitive environment.

I continuously monitored the competition leaderboard and adapted my agent on daily results. This iterative fine-tuning of hyperparameters and core logic was crucial for maximizing performance against the actual opponents. Some changes included adjusting the inventory threshold scale (using a more conservative 0.01), simplifying the price grid used for calculating the optimal price (moving from a segmented grid based on percentiles, to a single, uniformly spaced grid), redefining the competitive price rules to be less aggressive in order to come away from high win-rate and towards maximizing the revenue per successful transaction.

LLM Usage

I took help from Claude to generate test scripts for my agent, and to suggest improvements to the opponent prediction in part 2 to avoid falling for the “race to the bottom" price-undercutting cycle problem as described in the project instructions. Claude was also used during part 1 to help structure *create_model.ipynb* so that I could systematically test different hyperparameters for the models I explored after logistic regression. I made sure I understood any code and improvements suggested by Claude and tried to manually code it based on LLM suggestions, so that I would be able to calculate on my own the way my agent priced. I never copied and pasted code from an LLM, instead I would manually type any code that the LLM would provide to ensure that I understood and implemented the logic myself. Finally, I used Co-pilot for debugging which was a feature as part of VSCode.

Conclusion

I are very satisfied with my final outcome, having successfully implemented an algorithmic pricing strategy that maintained a strong position on the leaderboard, especially in part 2 where the competition was introduced. I learnt that my success was likely attributed to the fact that simplicity and robust implementation often surpass overly complex logic that could not be reliably deployed in real time. My strategy consisted of 3 major components: the use of ridge regression for price prediction, a 3-tier response system to handle passive, moderate and aggressive competitor scenarios, and lastly, the tit-for-tat cooperation mechanism that encouraged mutually beneficial pricing. This project structure, competing on the class leaderboard, gave me a sense of urgency and motivation where I implemented multiple rounds of iteration and refinement, allowing me to effectively merge theory from class into a real time scenario.

Given more time and information, I identified a few alleys I would have liked to explore for improvement. I would have liked to implement Bayesian updation for the cooperative scoring system instead of just simple additive adjustments. I also considered possibly experimenting with reinforcement learning to create a self-learning agent capable of discovering even more interesting and optimal price strategies.

Working on this tournament style project has sparked a significant interest in different domains including game theory, multi-agent competitions in the real world (such as Doordash vs Uber Eats, or AWS vs. Azure). Ultimately, this project has sparked curiosity in the world of revenue and inventory management, inspiring me to look into career paths in this high-impact field.