Dynamic Pricing Challenge – Submission Report

Zia Muhammad

Date: September 2025

Repository: github.com/zia-muhammad/dpc-duopoly-bot

Introduction:

This project implements a dynamic pricing bot for a duopoly market. The required deliverable is a Python file duopoly.py exposing a function p(...) that, given historical data and competitor capacity information, outputs a price decision and a state object. Beyond the core functionality, I also included tests, continuous integration, and simple local simulation scripts to make the solution easier to validate and maintain.

Approach

The pricing logic is based on a few guiding principles:

- Bounded memory: Only a fixed-size history of recent periods is retained, ensuring predictable memory use.
- Incremental learning: Demand is estimated using an online OLS regression that updates in constant time with each new observation.
- Policy function: Prices are chosen using a combination of: Exploitation of the fitted demand model. ε-greedy exploration to avoid local traps. Smoothing (damping) to prevent sudden price jumps.Monopoly uplift when the competitor is out of capacity.
- Clean interface: All logic is encapsulated inside the p(...) function in duopoly.py

Implementation

- RingBuffer – a simple circular buffer storing the most recent (my_price, competitor_price, demand) triples.
- OnlineOLS3 – maintains sufficient statistics for an OLS regression of demand on own price and competitor price. Each update is O(1).
- choose_price(...) – pure function that selects the next price using model coefficients, exploration, and safety bounds.
- p(...) – the required entrypoint. Restores state, records last observation, updates the model, and returns the next price with updated state.

Tuning knobs (exposed as constants in duopoly.py):

- WINDOW – history size for online learning (default 50).
- DAMP – smoothing factor for stability (default 0.20).
- UNDERCUT – relative price when model is not reliable yet (default 0.98).
- EXPLORE_BAND – ε-exploration band vs competitor price (0.85–1.15).
- MONOPOLY_UPLIFT – markup when competitor has no capacity (default 1.10).

Testing

A lightweight but meaningful test suite is included under tests/. It covers:

- Smoke test – ensures p(...) runs and respects bounds.
- State tests – check that RingBuffer rolls over correctly and that OnlineOLS3 recovers expected coefficients.
- Policy tests – verify that choose_price(...) respects price bounds.
- End-to-end test – simulates a few periods with dummy demand and ensures state is preserved and outputs remain valid.

All tests pass locally and in CI.

Continuous Integration

A GitHub Actions workflow (`.github/workflows/ci.yml`) is configured. On every push or pull request:

- Python is set up (version 3.9).
- Dependencies are installed (numpy, pytest).
- The test suite is executed.

This ensures any change is automatically validated.

Simulation and Benchmarks

To complement the tests, I added two small utilities:

- simulate_local.py – runs multi-season simulations with a synthetic competitor and demand model. Reports per-season and overall revenue. Can be run with TIMING=1 to also show latency stats.
- bench_local.py – a micro-benchmark that calls p(...) 1000 times with synthetic data and reports average and max latency.

On local runs, the bot's average per-call latency is around 1–2 ms, with maximums well under 5 ms. This is far below the 200 ms average / 5 s max requirement.

Conclusion

This solution satisfies the requirements of the Dynamic Pricing Challenge:

- Correct interface – p(...) implemented in duopoly.py.
- Bounded state – no unbounded growth in memory.
- Exploration and stability – ensures robustness to noise and avoids erratic behavior.
- Validation – comprehensive unit tests and automated CI pipeline.
- Performance – sub-millisecond average latency, safe under platform constraints.

All supportive files (tests, CI, simulator, benchmark, README) are included for clarity, but the only required deliverable for the platform is duopoly.py.

Acknowledgements

During the development of this task, I made extensive use of **ChatGPT** as a coding assistant. It helped me structure the implementation step by step, follow best practices, and refine details such as function naming, test design, and CI setup. All final design decisions and integration were done by me, but the solution reflects significant support from AI-assisted programming.