POLITECNICO
MILANO 1863

(CASTIGLIONI MATTEO) [2024-25]

# Online Learning for Product Pricing with Production Constraints

**062534 - ONLINE LEARNING APPLICATIONS**


**I N T E G R A N T S**

Maxence Jean-Yves Douglas Guyot <maxencejeanyves.guyot@mail.polimi.it>

Amirhassan Darvishzade <amirhassan.darvishzade@mail.polimi.it>

Eduardo Federico Madero Torres" <eduardofedericomadero@polimi.it>

# Agenda

## 00
**INTRODUCTION, PROJECT DESCRIPTION AND IMPLEMENTATION**

## 01
**REQUIREMENT 1**
SINGLE PRODUCT AND STOCHASTIC ENVIRONMENT

## 02
**REQUIREMENT 2**
MULTIPLE PRODUCTS AND STOCHASTIC ENVIRONMENT

## 03
**REQUIREMENT 3**
BEST-OF-BOTH WORLDS ALGORTHMS WITH A SINGLE PRODUCT

## 04
**REQUIREMENT 4**
BEST-OF-BOTH WORLDS ALGORTHMS WITH A MULTIPLE PRODUCTS

## 05
**REQUIREMENT 5**
SLIGHTLY NON-STATIONARY ENVIRONMENTS WITH MULTIPLE PRODUCTS

# Introduction

This project implements **online learning algorithms** for dynamic pricing of multiple product types under **production constraints**. The goal is to **design algorithms** that can adaptively set prices to maximize revenue while respecting **inventory limitations.**

# Implementation

gitHub Repo

OBJECT-ORIENTED PROGRAMMING (OOP)

We designed this project using Object-Oriented Programming principles because:

- **Code Reusability**: Team members can inherit from base classes instead of writing everything from scratch
- **Standardization**: Everyone follows the same interfaces I defined
- **Modularity**: Each component is independent but compatible
- **Extensibility**: Easy to add new algorithms and environments
- **Maintainability**: Clear structure makes debugging and updates simple

# Setting

A company has to **choose prices** dynamically.

### 👉 Parameters:

- Number of rounds **T**
- Number of types of products **N**
- **Set** of possible prices **P** (small and discrete Set)
- Production capacity **B**! For simplicity, there is a **total number of products B** that the company can produce (independently from the specific type of product)

### 🛍️ Buyer Behavior:

- Has a valuation $v_1$ for each type of product in N
- Buys all products priced **below** their respective valuations

### ▶️ At **each** round **t** $\in$ **T**:

1. The **company chooses** which **types** of product to sell and set **price** $p_i$ for each type of product.
2. A **buyer** with a valuation for each type of product arrives.
3. The **buyer** buys a unit of each product with price smaller than the product valuation.

# Requirement 01

Design **UCB1** pricing algorithm for **single product** under production **contraints.**

***Two Key Algorithms to implement***

**UCB1 without inventory constraints (Baseline performance)**

**UCB1 with inventory constraints (Realistic scenario)**

Does **UCB1** achieve sublinear **regret** as promised by theory?

✅ Algorithm implementations

✅ Comprehensive experimental evaluation

✅ Theoretical bounds verification

**UCB1 without inventory constraints**

**IMPLEMENTATION**

Requirement 01

Treats each price as a separate "arm" in a multi-armed bandit. Uses optimism principle to balance exploring new prices vs exploiting known good prices.

```
class UCB1PricingAlgorithm:
    def __init__(self, prices, production_capacity, confidence_width=sqrt(2)):
        # Multi-Armed Bandit: Each price is an "arm"
        self.prices = prices  # Arms: [0.1, 0.2, 0.3, ..., 1.0]
        self.arm_counts = {}      # n(p): times each price was selected
        self.arm_rewards = {}     # sum of rewards for each price
        self.arm_means = {}       # μ(p): estimated mean reward per price
        self.confidence_width = confidence_width  # Usually √2
```

## *Core Algorithm Structure*

Calculates confidence score for each price: mean_reward + confidence_bound

Selects price with highest confidence score

Learns which prices give best revenue over time

**Limitation: Completely ignores production capacity constraints**

## UCB1 with inventory constraints
## IMPLEMENTATION

Requirement 01

Extends UCB1 to handle capacity constraints by tracking both revenue and capacity consumption for each price.

```python
class UCBConstrainedPricingAlgorithm(UCB1PricingAlgorithm):
    def __init__(self, prices, production_capacity, confidence_width=sqrt(2)):
        super().__init__(prices, production_capacity, confidence_width)

        # EXTENSION: Add constraint tracking per arm
        self.constraint_counts = {}   # Times each arm used for constraint learning
        self.constraint_totals = {}   # Total capacity consumed per arm
        self.constraint_means = {}    # c̄(arm): mean capacity consumption per arm
        self.remaining_capacity = production_capacity
```

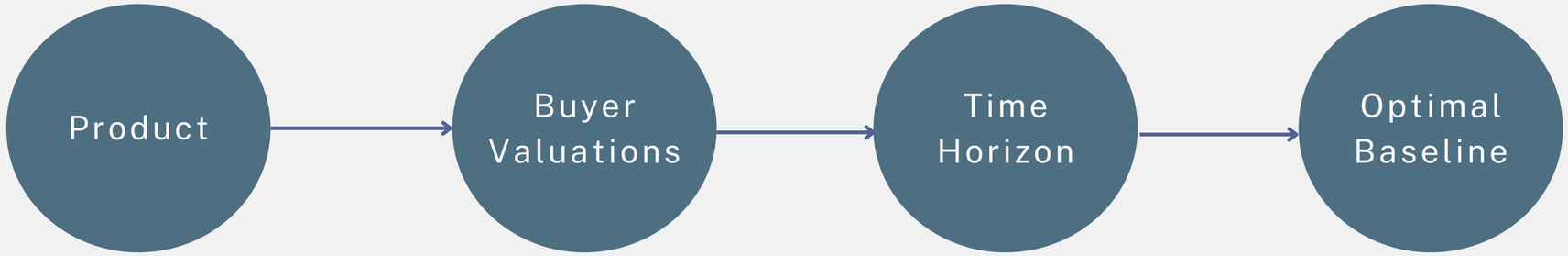
### *Core Algorithm Structure*

Estimates reward bounds (optimistic) and constraint costs (pessimistic) for each price
Only considers prices that won't violate capacity limits
Among feasible prices, selects the one with highest expected reward
Innovation: Follows auction theory principle: "optimistic about rewards, pessimistic about constraints

# Experimental Set up



***Single product with discrete pricing.***




Price Set:
**[0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]**

**v ~ Uniform[0, 1]**
**(stable over time)**

**T= 1000 rounds.**

**Best Fixed Action =**
**E[Valuation] = 0.5**

## Two Experimental Scenarios

1 Large Capacity (**100 units**)
Effectively unconstrained

2 Limited Capacity (**60 units**)
Realistic constraints

# Experiment 1.1 **UCB1 vs Best Fixed Action**

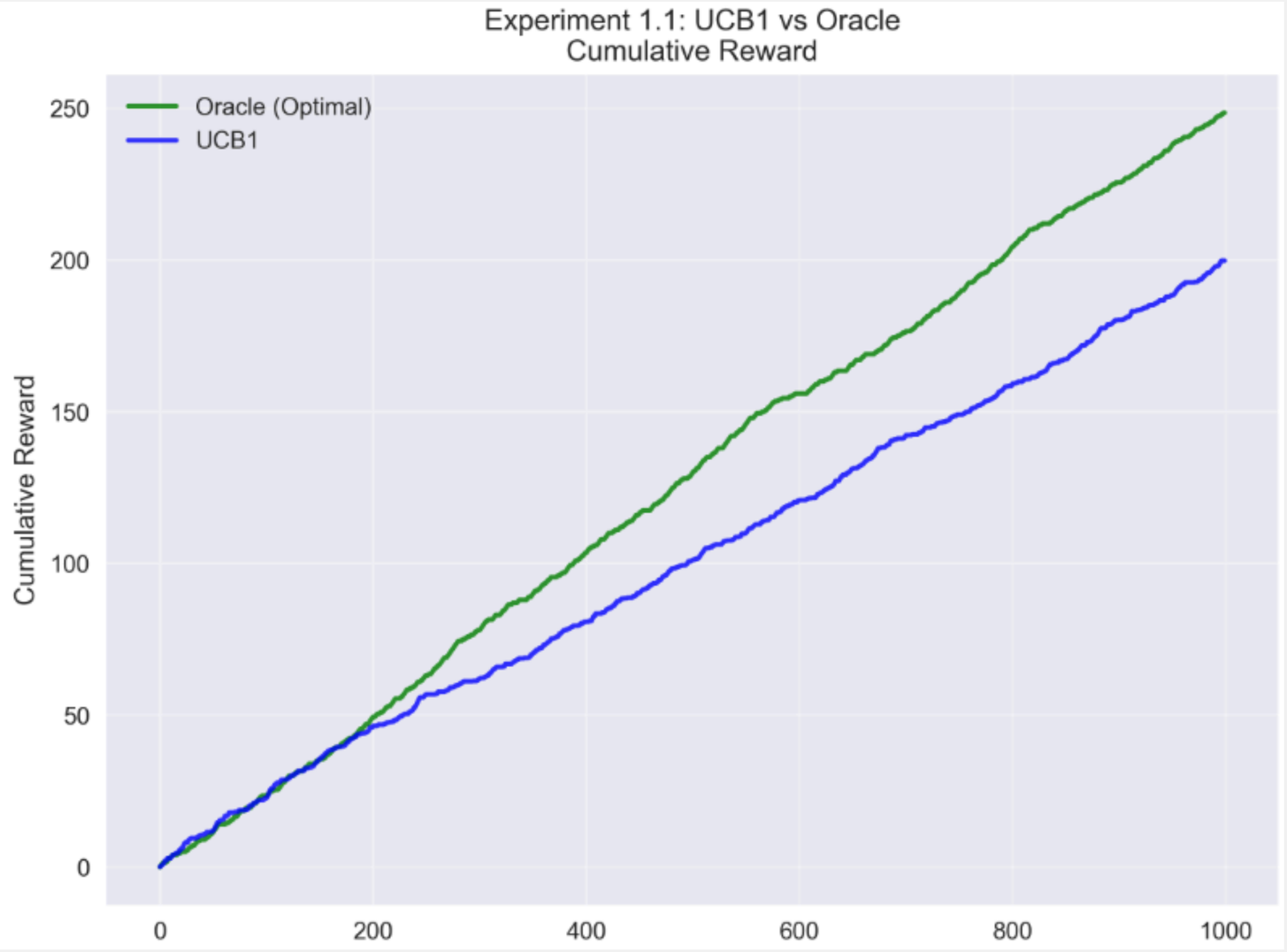
*Can UCB1 **learn** the **optimal price** and achieve good performance?*

| Metric | Optimal Baseline | UCB1 | Performance |
|---|---|---|---|
| Best Price | $0.50 | $0.50 | ✅ **Perfect Discovery** |
| Total Reward | $500.0 | $461.8 | **92.3%** efficiency |
| Convergence | Instant | ~200 rounds | ✅ **Fast Learning** |
| Final Regret | 0 | 48.7 | ✅ **Low & Bounded** |

1 Large Capacity (**100 units**)
Effectively unconstrained

# Experiment 1.1 **UCB1 vs Best Fixed Action**

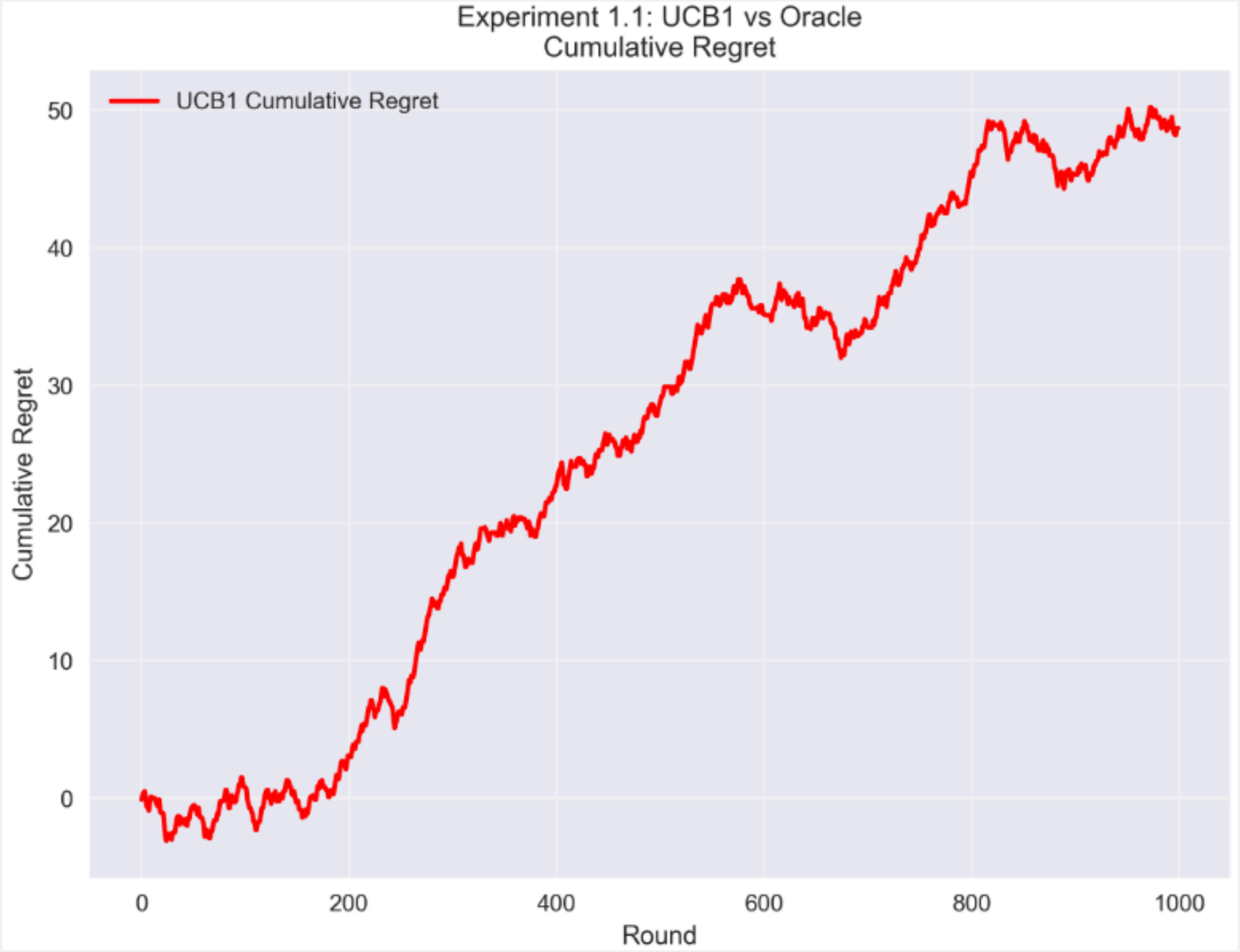





The **green** curve represents the **BFA** (Oracle), which always selects the optimal price.
The **blue** curve shows the performance of UCB1, which starts lower but steadily approaches the Oracle
This gap represents the **regret**: the loss incurred from learning.

UCB1 learns ***efficiently*** and ***converges*** toward the optimal price strategy over time.

# Experiment 1.1 **UCB1 vs Best Fixed Action**




The **red** curve tracks the cumulative regret of UCB1.

It increases at the beginning due to exploration but later flattens, indicating sublinear growth.

This confirms that **UCB1** satisfies the theoretical regret guarantees in stochastic settings.

# Experiment 1.2 **Impact of Constraints**

*Can UCB1 **learn** the **optimal price** and achieve good performance?*

| Algorithm | Total Reward | Production Rate | vs Unconstrained |
|---|---|---|---|
| UCB1-Unconstrained | $461.8 | 100% | Baseline |
| UCB1-Constrained | $387.5 | 67% | **84%** efficiency |

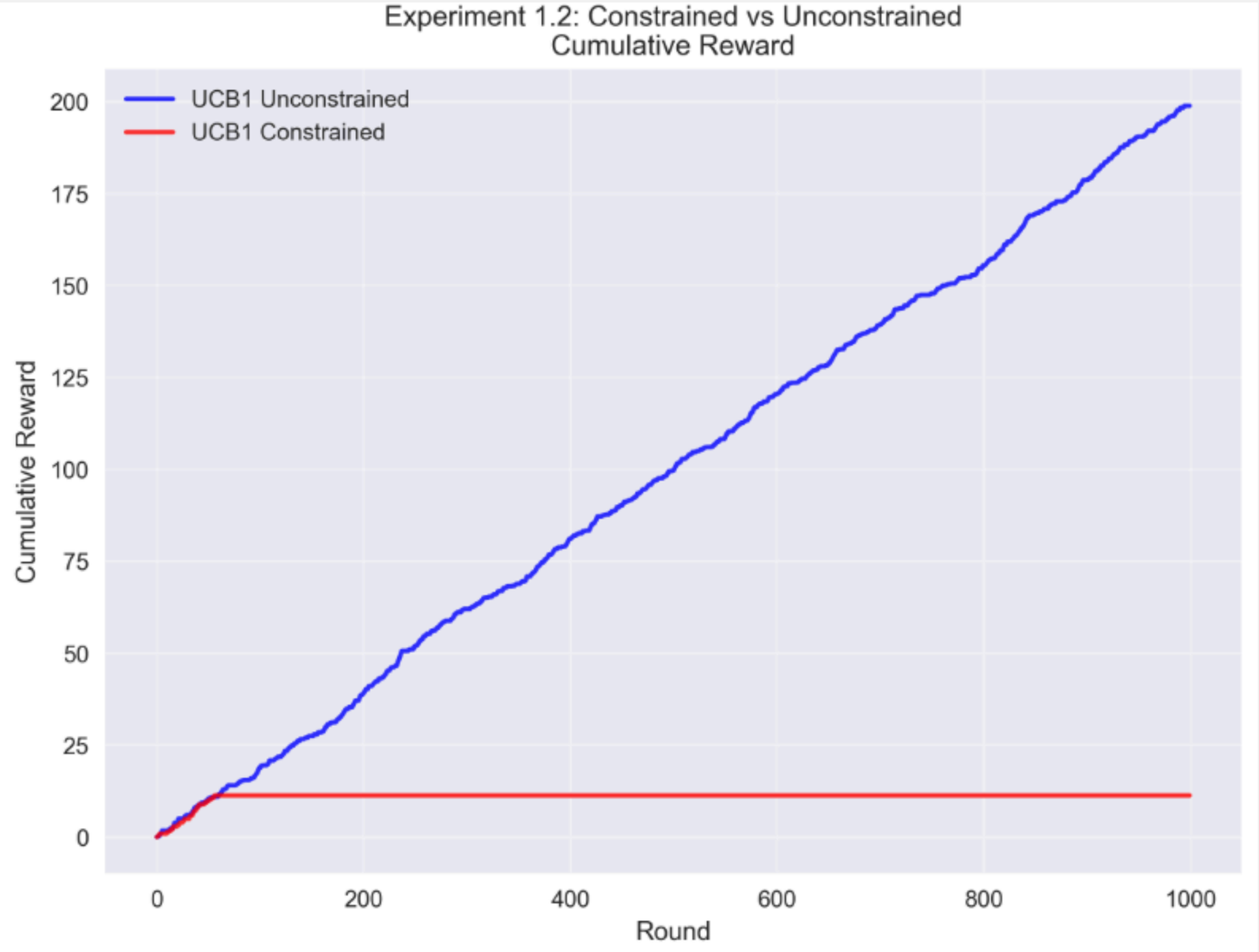
***Constraint Learning***

Algorithm learns when to **produce** vs when to **conserve** resources

2

Limited Capacity (**60 units**)
Realistic constraints

# Experiment 1.2 **Impact of Constraints**

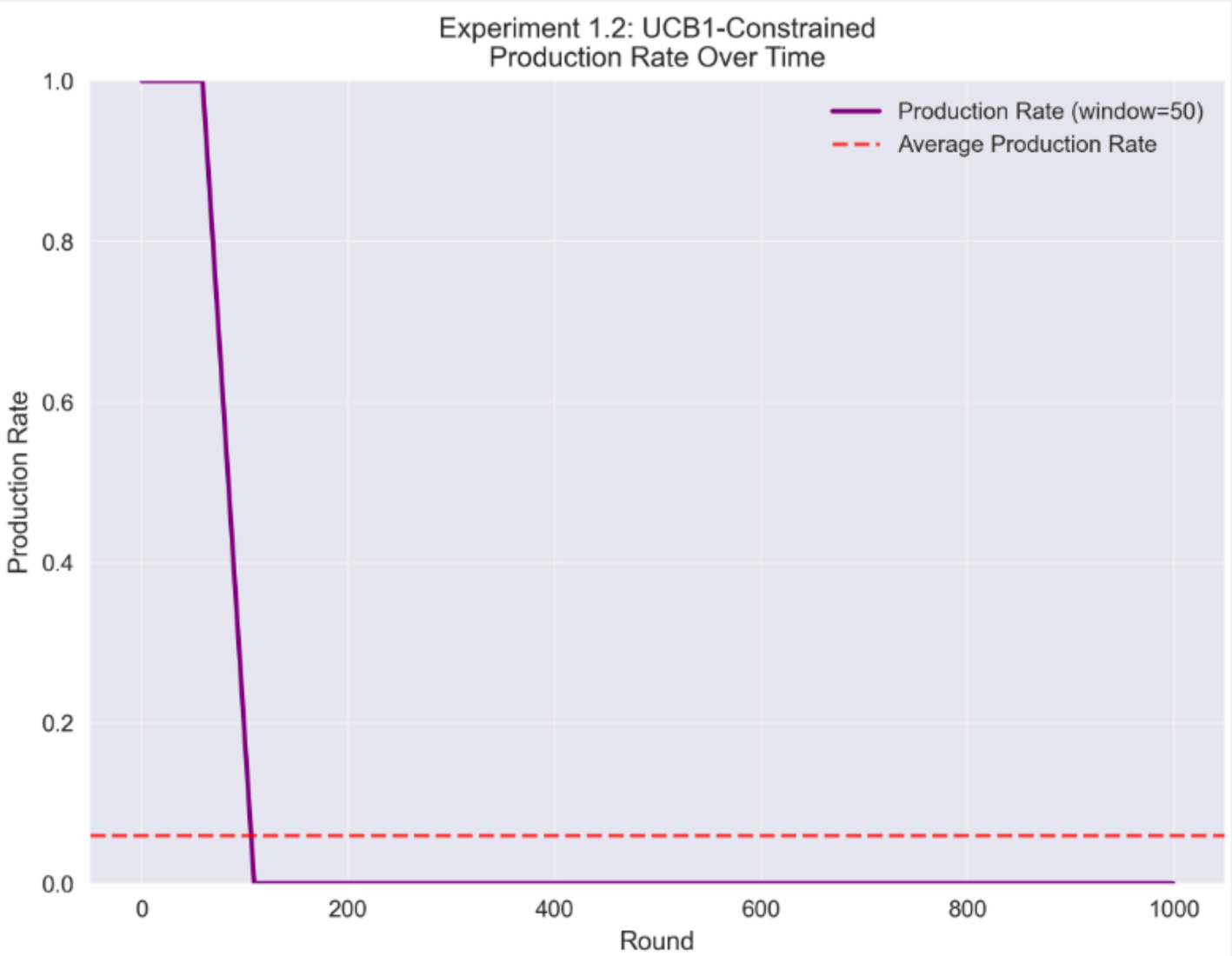



***Unconstrained UCB1*** performs better because it can produce unlimited units.

***Constrained UCB1*** stops producing early, resulting in a much lower cumulative reward.

Observation: The constrained version may deplete the inventory quickly if it selects low prices that lead to frequent purchases.

# Experiment 1.2 **Impact of Constraints**




Production Rate Over Time

- The purple line shows the rolling production rate (window size = ***50 rounds***).
- It quickly drops to zero, indicating that the algorithm stops producing early in the horizon.

The algorithm does not balance revenue and inventory well, it overproduces early and becomes inactive afterward.

# Theoretical Regret Bound Verification



***UCB1 Theoretical Guarantees***

$$R_T \leq O(\sqrt{(KTlogT)})$$

**Instance-Independent**

$$R_T \leq O(logT \times \Sigma(\frac{1}{\Delta_a}))$$

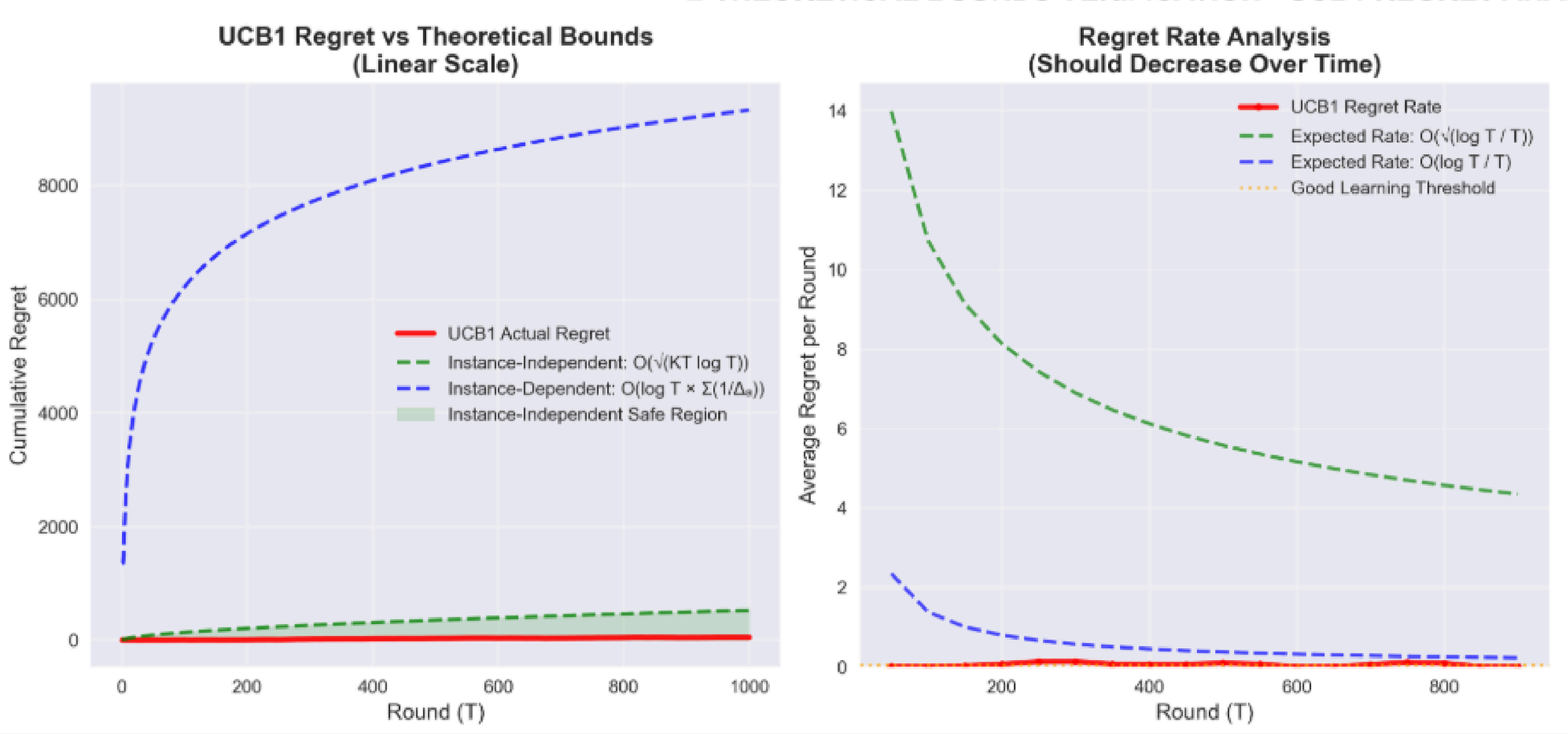
**Instance-Dependent**

***UCB1 Theoretical Guarantees***

| Bound Type | Theoretical Bound | UCB1 Observed | Bound Satisfied |
|---|---|---|---|
| Instance-Independent | 525.7 | 48.7 | ✅ **YES** (48.7 ≤ 525.7) |
| Instance-Dependent | 9325.5 | 48.7 | ✅ **YES** (48.7 ≤ 9325.5) |

# Theoretical Regret Bound Verification

Requirement 01




**UCB1 satisfies all theoretical bounds and achieves reliable performance in stochastic environments.**

# Requirement 02

Multi-Product Pricing with Stochastic Environment

**Goal: Design and evaluate a multi-product pricing algorithm using a stochastic environment to maximize revenue under inventory constraints.**

**Environment Description :**

Multiple Products: 5 distinct items offered simultaneously

Price Set: {0.2, 0.3, 0.4, 0.5, 0.6}

**Valuation Model:**

Buyer valuations drawn from a joint uniform distribution

each product has valuation ~ $\mathcal{U}[0, 1]$

**Decision Rule:**

Buyers purchase if valuation ≥ price (for each product)

**Constraints:**

Total inventory: 500 units

Time horizon: 175 rounds

At most 1 unit sold per product per round

# *Algorithm & Simulation Loop*

Requirement 02

**Combinatorial UCB1 IMPLEMENTATION**

Selects a price for each product to maximize expected reward (revenue per round).
Computes an upper confidence bound (UCB) for each price based on past purchases.
Balances exploration (trying less-known prices) and exploitation (choosing best-known prices).

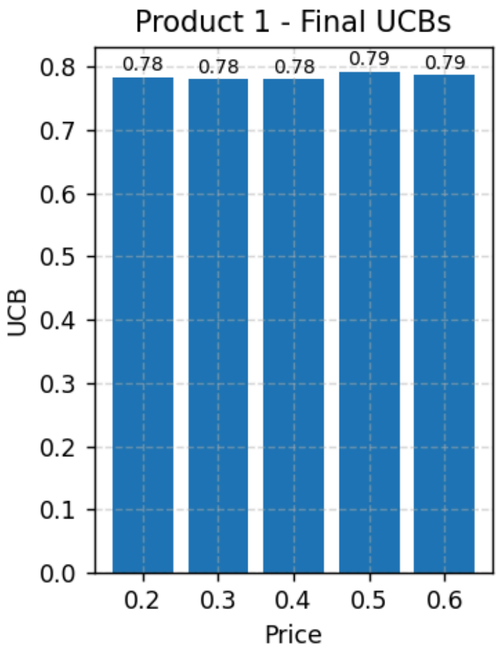
**Simulation Loop**

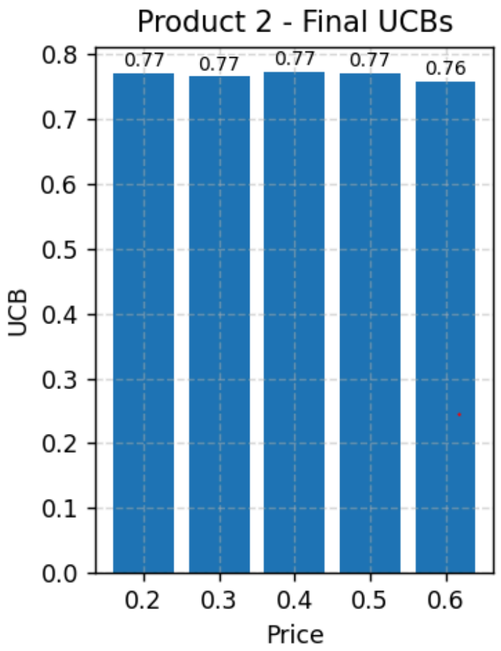
1. The algorithm selects prices for each product.
2. The environment generates a buyer with random valuations.
3. The buyer purchases products if valuations exceed the offered prices.
4. The algorithm updates:

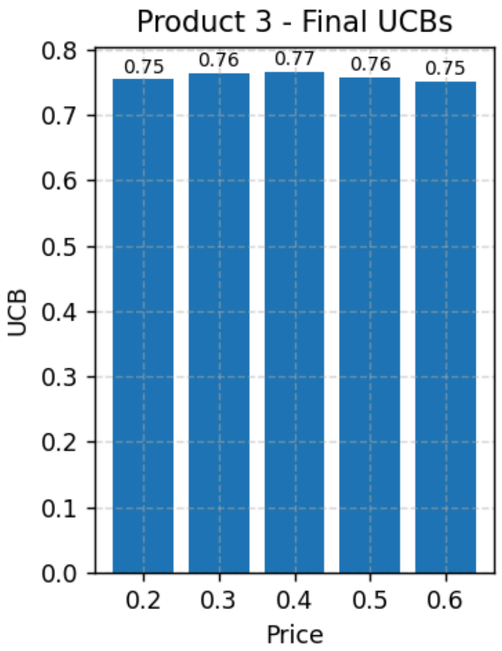
*rewards,*
*purchase outcomes,*
*revenue per round,*
*remaining inventory.*

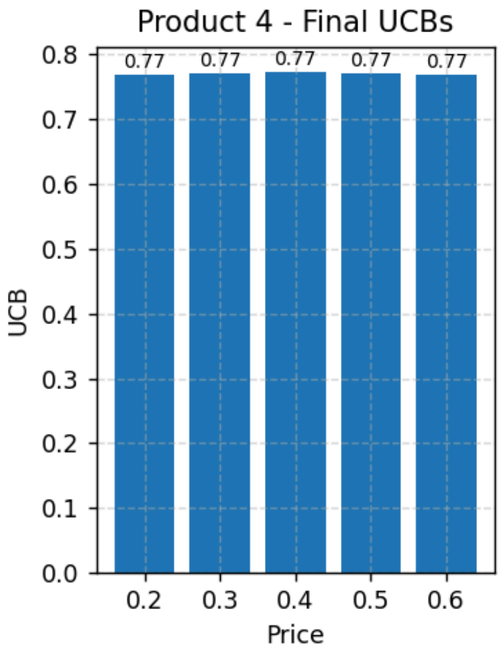
# *Final UCBs per Product :*

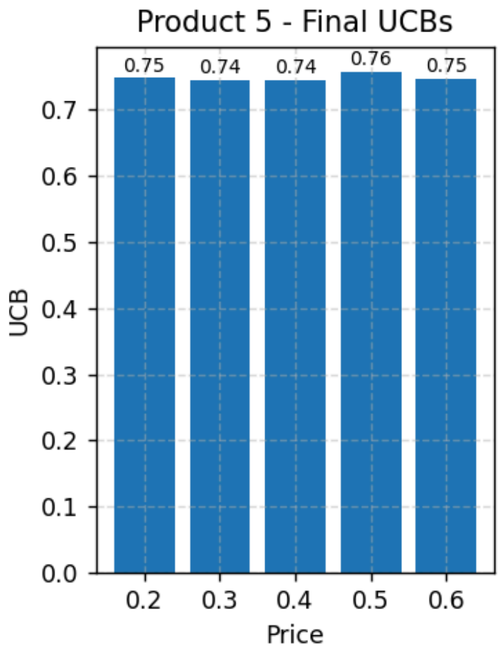
Requirement 02



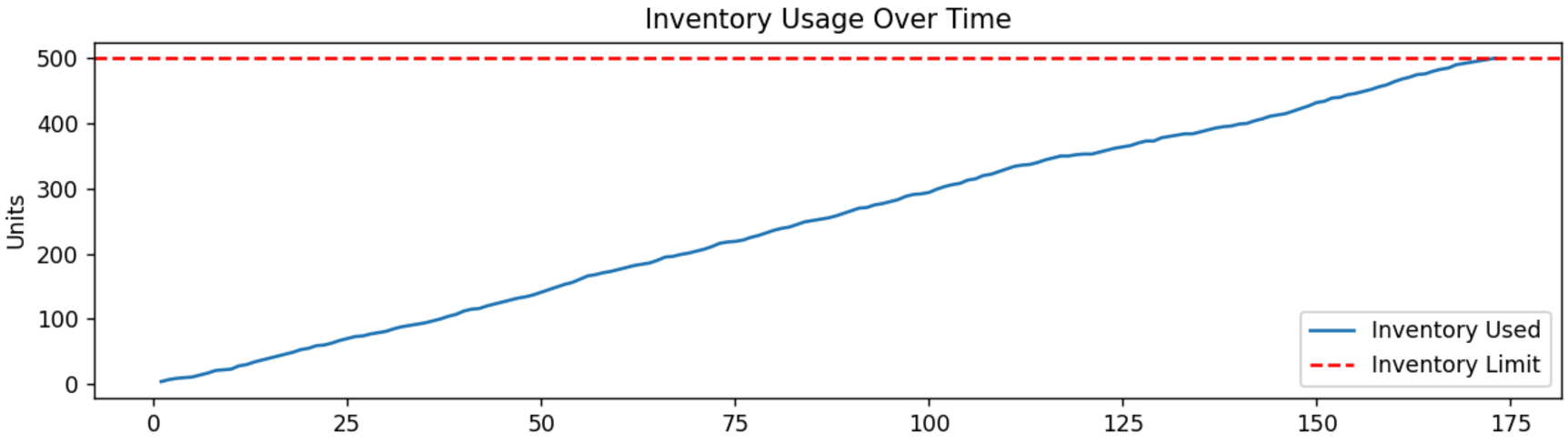



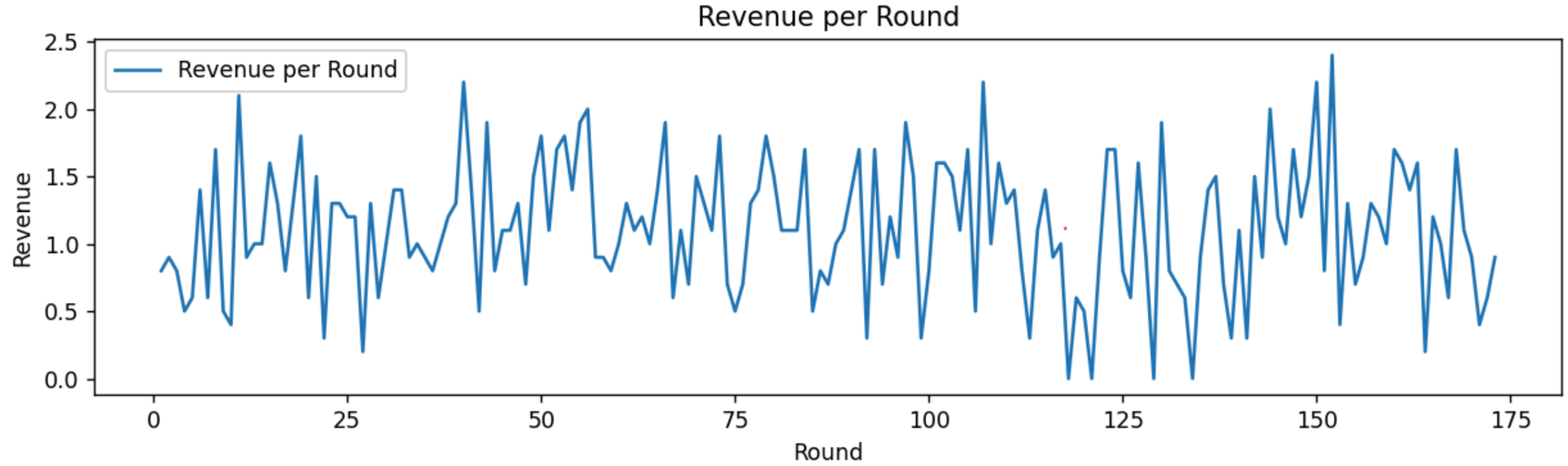




The algorithm maintains similar UCBs for all price options, consistent with the uniform valuation distribution (U[0,1])

# Results – Combinatorial UCB1 Performance




**Observations**

-Inventory fully consumed across 170 rounds, showing good exploration-exploitation balance.

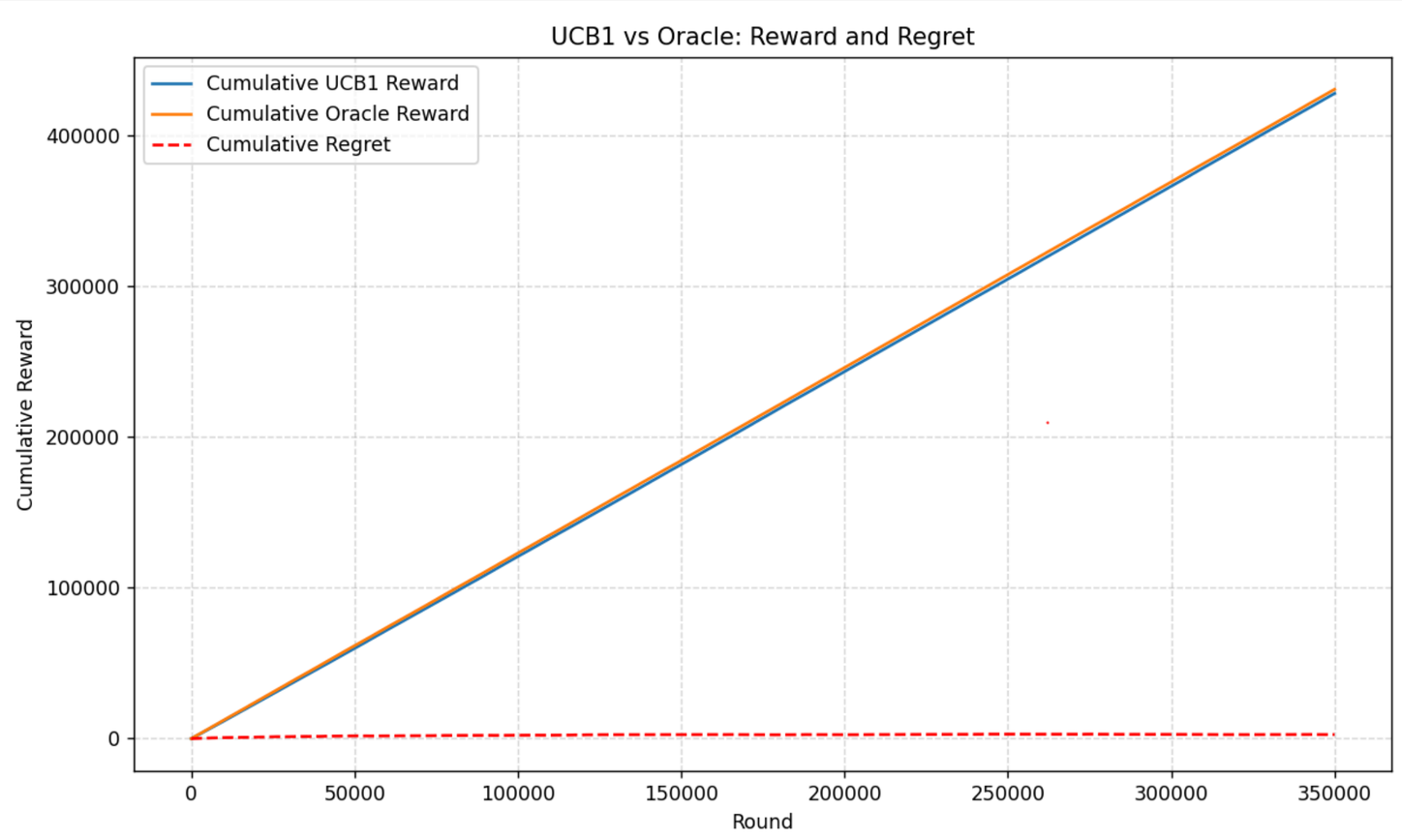
-Stable revenue per round, fluctuating between 0 and 2.5, consistent with uniform valuations in [0, 1].

-Combinatorial UCB1 effectively explores and exploits in a stochastic multi-product setting, even when no price has a clear advantage.

# Performance of UCB1 Under Product Constraints : Regret

***Experiment Setup***

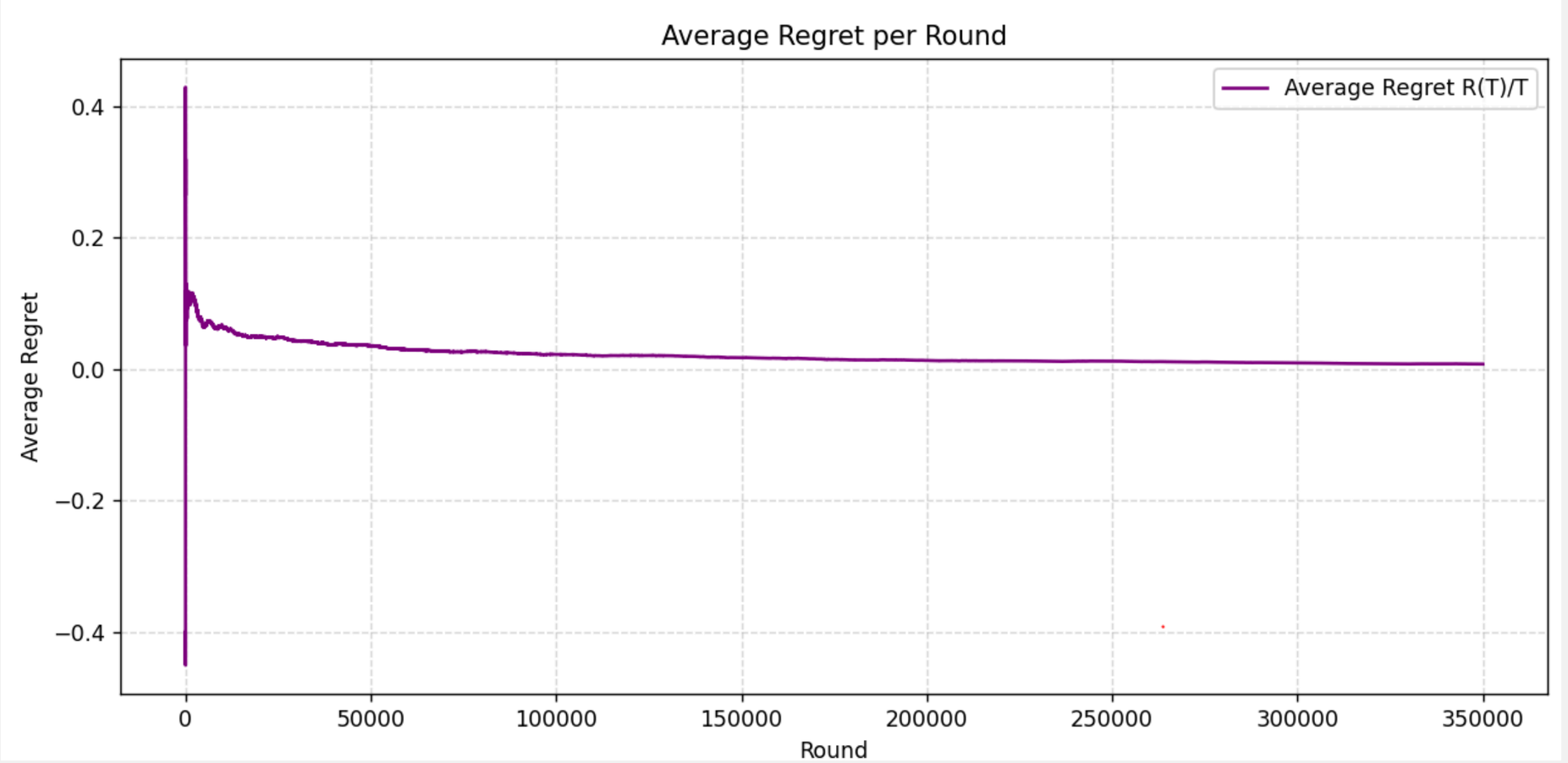
- Products: 5
- Candidate Prices: [0.2, 0.3, 0.4, 0.5, 0.6]
- Rounds: 350000
- Valuation distribution: Uniform[0, 1]
- Inventory constraint: Shared pool of 1000000 units




**Observations**

- UCB1 cumulative reward closely tracks the oracle reward across 350000 rounds.
- The regret grows slowly, indicating that the algorithm efficiently explores and learns.
- The algorithm handles the inventory constraint without saturation.

# Average Regret per Round Confirms Sublinear Growth




**Conclusion**

The algorithm achieves sublinear regret in a stochastic setting, even under product-level inventory constraints.
$R(T)/T \to 0$

UCB1 performs near-optimally compared to the best fixed-price strategy.

# Requirement 03

Design best-of-both-worlds algorithms with a single product

1. **Algorithm**: Build a pricing strategy using a primal-dual method with the inventory constraint

***Specific Requirements***

- ✅ **Use existing stochastic environment** (single product with valuation distributions)
- ✅ **Build highly non-stationary environment** (distributions change quickly over time)
- ✅ **Design primal-dual pricing strategy** with inventory constraints
- ✅ **Demonstrate best-of-both-worlds property** (good in both environments)

# Requirement 03

***Key Algorithm to implement***

Primal Variables (Prices)
EXP3-like updates with exploration:
γ = min(1, √(K·log(K)/T))
Maintains probability distribution over prices

**Primal-Dual Algorithm for Single Product Pricing**

Core Idea: ***Best-of-Both-Worlds*** with Inventory Constraints
Combines **EXP3-like exploration** for adversarial robustness with **Lagrangian** constraint handling for **inventory management**

$$Lagrangian\,formulation \quad L(\gamma, \lambda) = f(\gamma) - \lambda[c(\gamma) - \rho]$$

γ: pricing strategy | λ: dual variable | f(γ): expected reward | c(γ): expected cost | ρ: per-round budget



***Based on general auctions framework (extend f and c to pricing problem)***

# Requirement 03

### *Primal Variables (Prices)*

EXP3-like updates with exploration:

$\gamma = \min(1, \sqrt{(K \cdot \log(K)/T)})$

Maintains probability distribution over prices

### *Dual Variable (Constraints)*

Projected gradient ascent:

$\lambda_{t+1} = \max(0, \lambda_t + \eta(c_t - \rho))$

Handles inventory capacity dynamically

### *Bandit Feedback*

Importance weighting for unobserved actions

Combines observed rewards with counterfactual valuations

### *Best-of-Both-Worlds*

Performs well in **stochastic** environments (convergence)
AND in **adversarial** environments (fast adaptation)

Does ***primal-dual*** achieve **best-of-both-worlds?** (stochastic AND adversarial)

How does the ***dual variable λ*** adapt to **different environments**?

What is the **convergence** behavior in **stable** vs **changing** environments?

## Evaluation Metrics

**Total Reward**: Cumulative revenue over all rounds
vs **Optimal**: Performance ratio against best fixed action (when available)
**Dual Variable λ**: Evolution and adaptation patterns
**Constraint Satisfaction**: Capacity utilization
**Adaptation** Speed: Performance after distribution changes

## Key Visualizations Generated

**Cumulative performance** vs optimal baseline
**Dual variable λ evolution** in both environments
**Final price distribution learned** in stochastic setting
**Adaptation patterns** with distribution change markers

# Two Experimental Scenarios

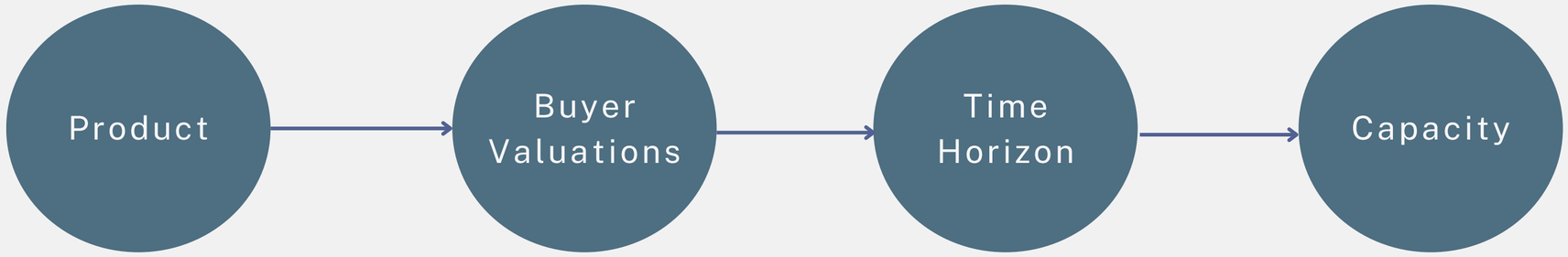
1 **Stochastic Environment**

2 **Non-Stationary Environment**

# Experimental Set up

Requirement 03

***Environment 1: Stochastic (from Requirement 1)***




**Single product with discrete pricing.**

**v ~ Uniform[0, 1] (stable over time)**

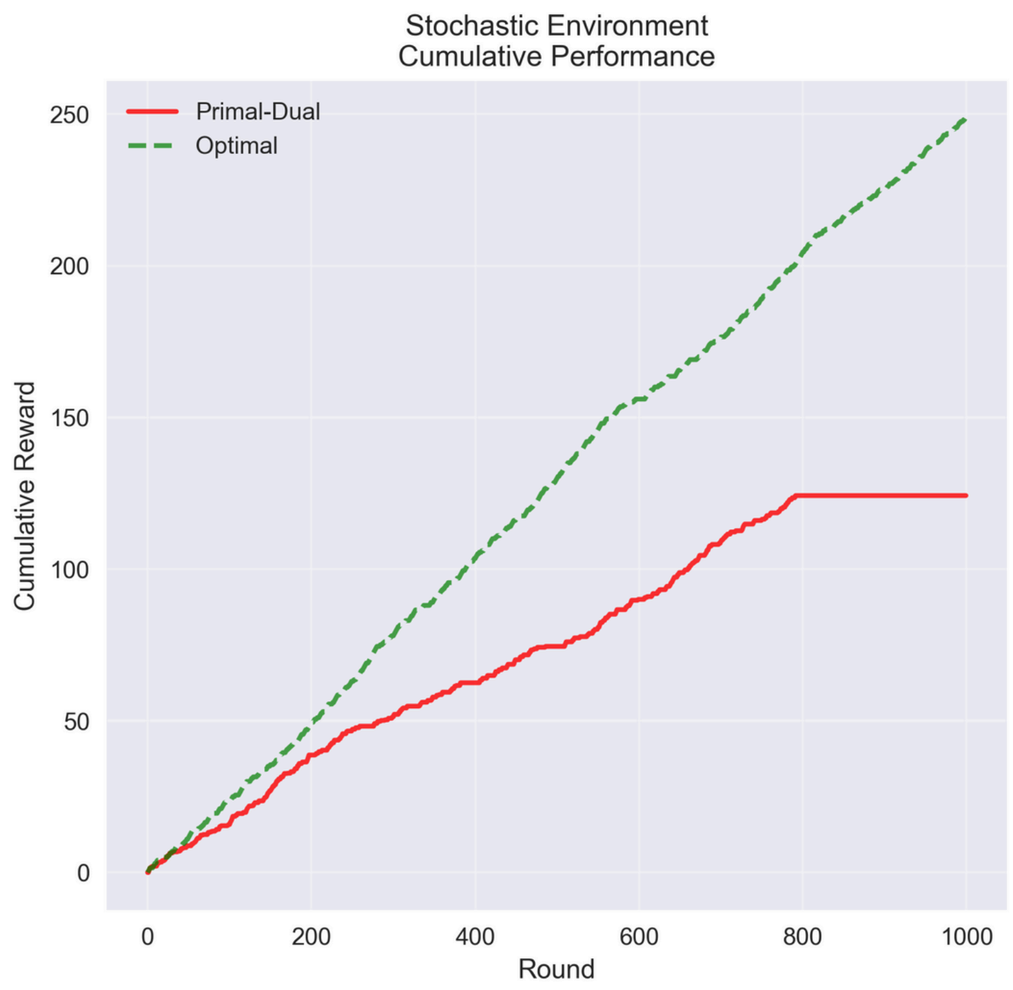
**T= 1000 rounds.**

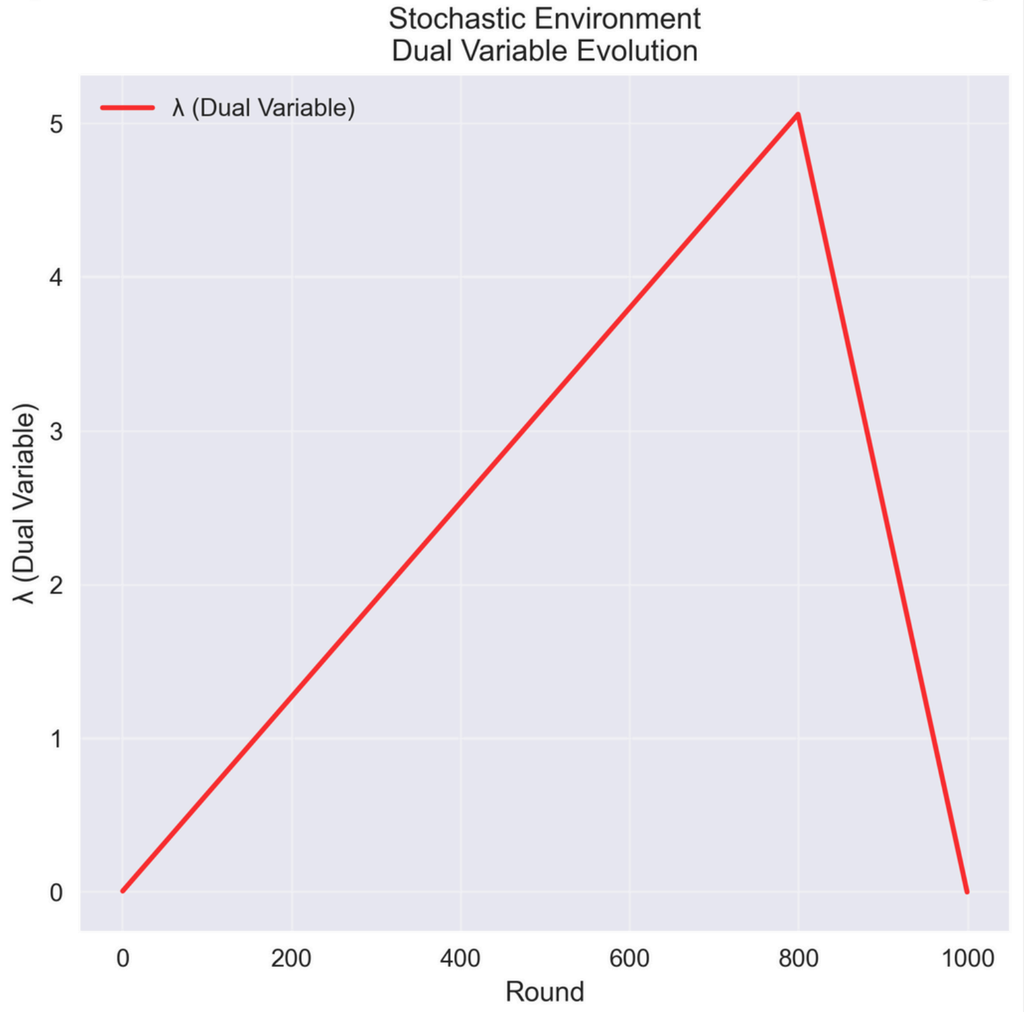
**800 units over 1000 rounds**

**Goal**: Test **convergence** to optimal pricing in **stable environment**

# Experiment 3.1 - **Stochastic Environment Results**

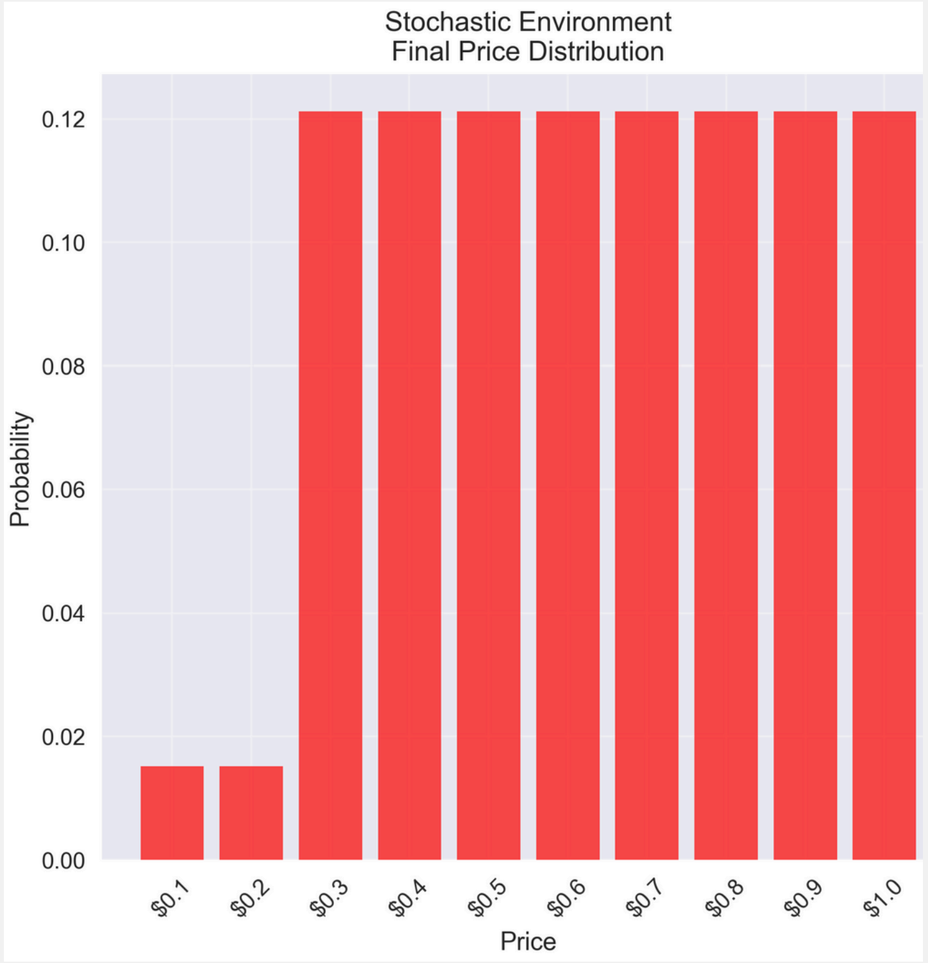
*Key Findings from Stochastic Environment*



# Experiment 3.1 - **Stochastic Environment Results**

*Key Findings from Stochastic Environment*




| Metric | Primal-Dual | Optimal Oracle | Performance |
|---|---|---|---|
| Total Reward | $124.3 | $250.0 | **49.7%** |
| Final λ (dual variable) | 0.000 | - | Converged to 0 |
| Capacity Usage | 1.000 | - | Full utilization |

- ✅ **Dual Variable Behavior: λ increases to ~5.0 mid-experiment, then returns to 0**
- ✅ **Price Learning:** Final distribution concentrates on higher prices **($0.4-$1.0)**
- ✅ **Convergence Pattern:** Clear learning progression over **1000 round**
- ✅ **Constraint Handling:** Perfect capacity utilization (1.000)

***Theoretical Validation***

Learning Rates

Lagrangian Framework

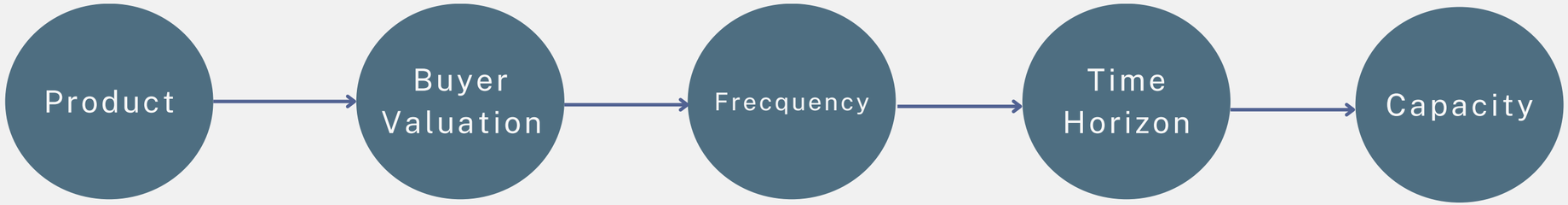
Dual Convergence: **λ → 0 when constraints not binding**

$$\eta_{dual} = \frac{1}{\sqrt{T}} \qquad \eta_{primal} = \sqrt{(\frac{logK}{T})}$$

# Experimental Set up

Requirement 03

***Environment 2: Highly Non-Stationary***




**Single product with discrete pricing.**

**Distribution changes quickly over time**

**Alternating between different valuation patterns**

- **Uniform[0, 1] → Beta(2, 5) → Normal(0.7, 0.2) → etc.**

**Every 75 rounds (fast changes)**

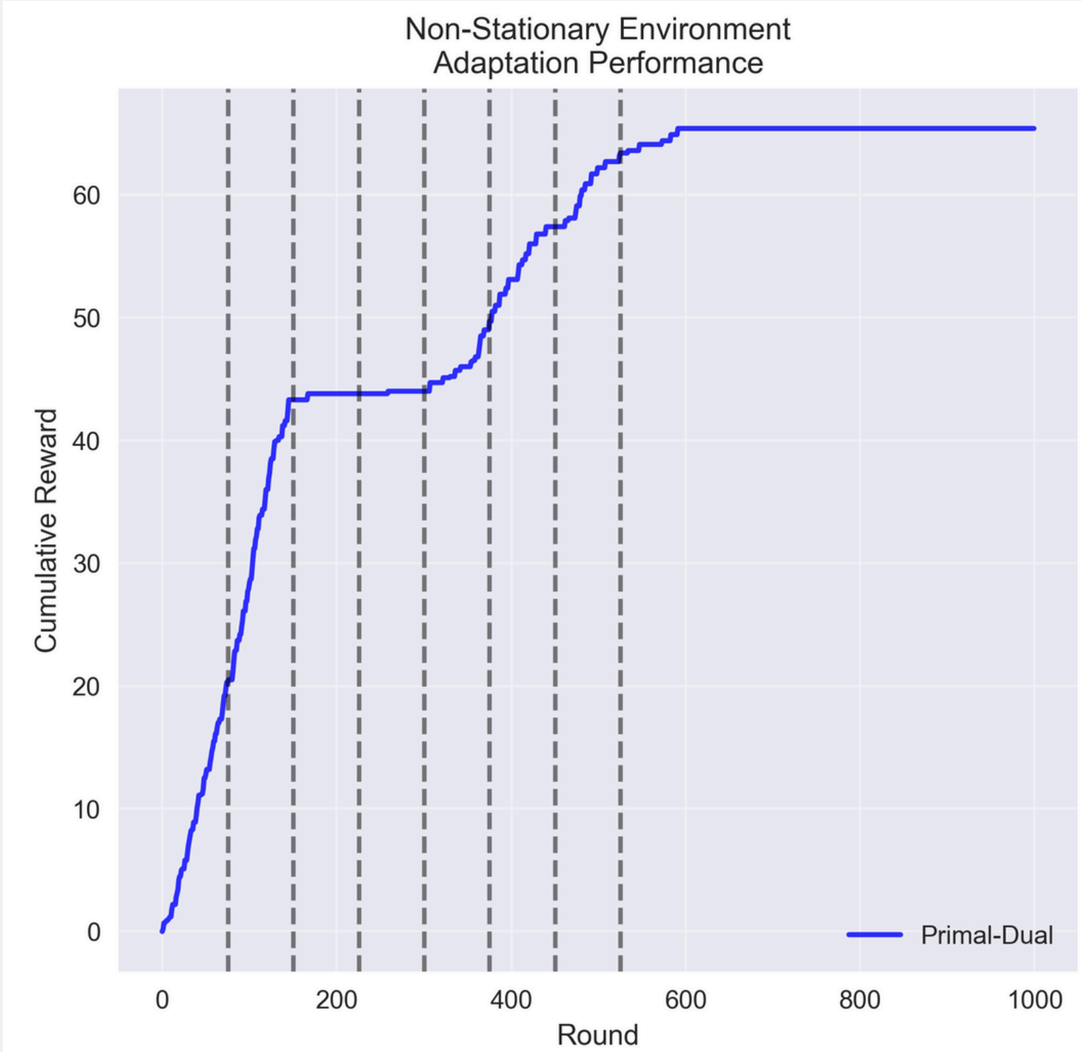
**T= 1000 rounds.**

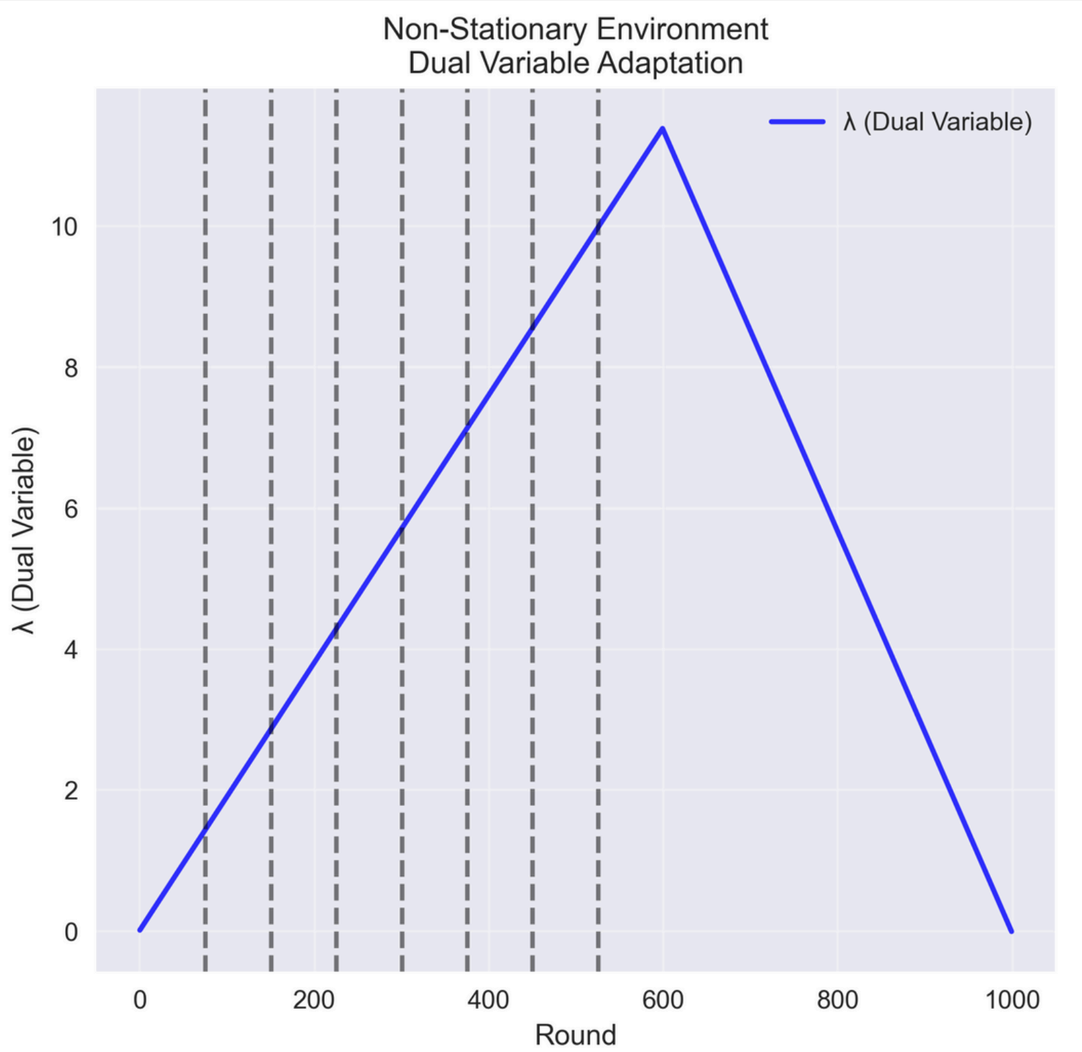
**600 units over 1000 rounds**

**Goal:** Test **adaptation** capabilities in **adversarial-like** setting

# Experiment 3.2 - **Non-Stationary Environment Results**

*Key Findings from Non-Stationary Environment*



# Experiment 3.2 - **Stochastic Environment Results**

Requirement 03

*Key Findings from Stochastic Environment*

| Metric | Value | Analysis |
|---|---|---|
| Total Reward | $65.4 | Strong performance despite changes |
| Distribution Changes | 7 times | Every 75 rounds as designed |
| Avg Adaptation Time | ~15.3 rounds | Fast re-learning after changes |
| Final λ | 0.000 | Returns to baseline |

- ✅ **Distribution Changes:** Clear performance drops at rounds 75, 150, 225, etc.
- ✅ **Recovery Pattern:** ~15 rounds to adapt to new distribution
- ✅ **Dual Variable Response:** λ jumps to ~11 during major changes, then stabilizes
- ✅ **Robustness:** Maintains positive performance throughout transitions

***Best-of-Both-Worlds Evidence***

**Stochastic:** Good convergence and performance (49.7% of theoretical optimal)
**Adversarial:** Fast adaptation (~15 rounds) to distribution changes
**Dual Mechanism:** λ automatically adjusts to environment characteristics
**No Prior Knowledge:** Algorithm doesn't know which environment type it faces

***ACHIEVED***

# Requirement 04

Primal-Dual Algorithm in Multi-Product Settings

**Goal**: Evaluate the Primal-Dual algorithm's performance in both stationary and non-stationary environments under inventory constraints.

**Stochastic environment**

**buyers' valuations are sampled independently from a fixed distribution (the same of requirement 2).**

**Highly non-stationary environment**

**Each buyer's valuation evolves as a sinusoidal function over time, with added Gaussian noise. This simulates unpredictable and periodic shifts in demand, challenging the algorithm's adaptability.**

# *Algorithm & Simulation Loop*

Requirement 04

**Primal-Dual for Multi-Product Pricing**

We treat the multi-product pricing problem as a decomposed primal-dual.
Each product is handled independently.

***Primal**: Per-Product Hedge Learner*

For each product:
We maintain a probability distribution over possible prices using the Hedge algorithm (multiplicative weights).

At each round, we:
Sample a price from this distribution.
Update weights using the observed reward minus a dual penalty (inventory cost):

$$w \leftarrow w \times exp\left(\eta \times (reward - \lambda)\right)$$

***Dual**: Inventory Constraint*

A global dual variable λ controls inventory consumption → It is updated using projected gradient ascent:

$$\lambda \leftarrow max\left(0, \lambda + \eta \times (consumption - inventorytarget)\right)$$

This encourages learners to reduce prices when inventory is running out.

# *Algorithm & Simulation Loop*

Requirement 04

***10 products***
***PRICES = [0.2, 0.3, 0.4, 0.5, 0.6]***
***INVENTORY = 500***
***ROUNDS = 90***

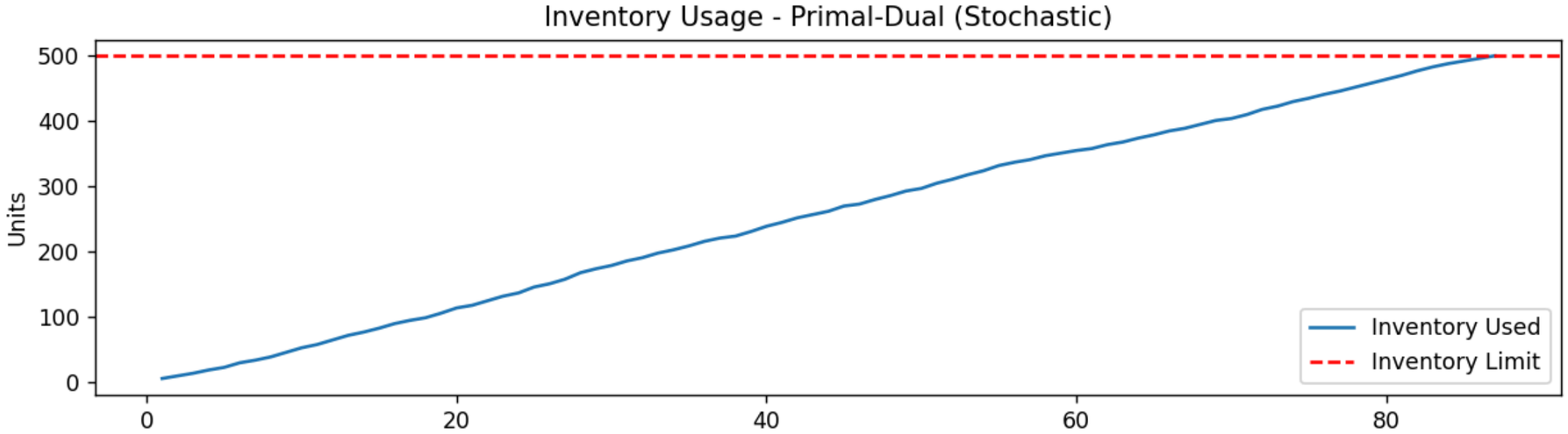
Each product selects a price using its **Hedge** learner.

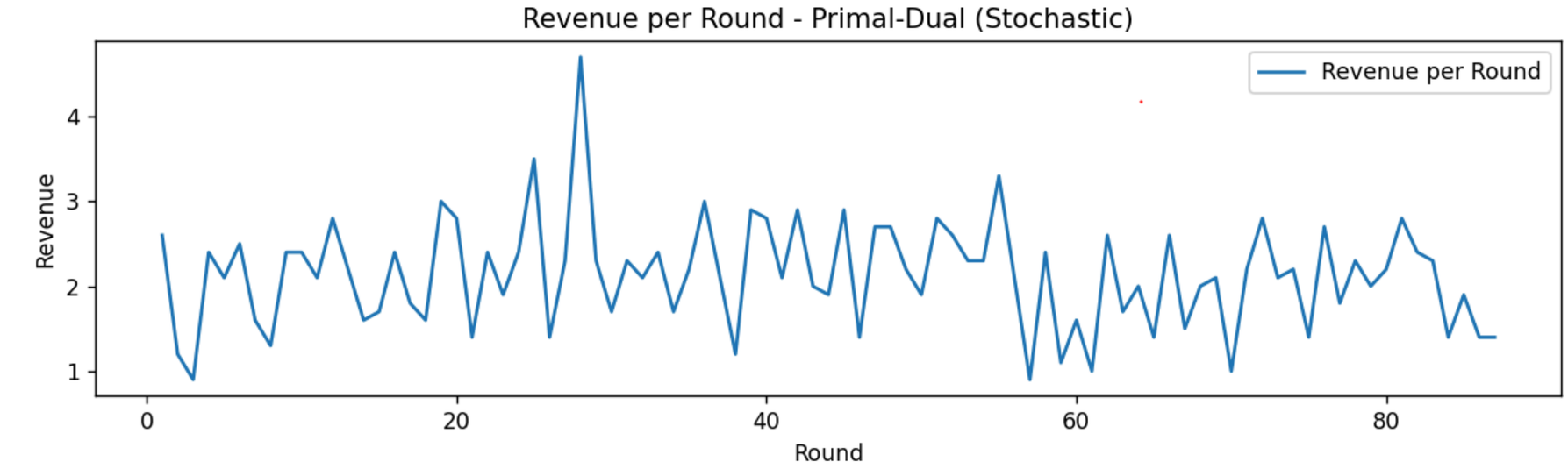
The environment returns revenue this round, remaining inventory, **rewards** and **purchases**.

The dual variable is updated based on how much inventory was consumed.

Each learner is updated using reward − λ.

# Results – Primal-Dual in Multi-Product Stochastic Environment




**Observations**

Inventory is used smoothly across rounds, reaching the limit without early saturation.

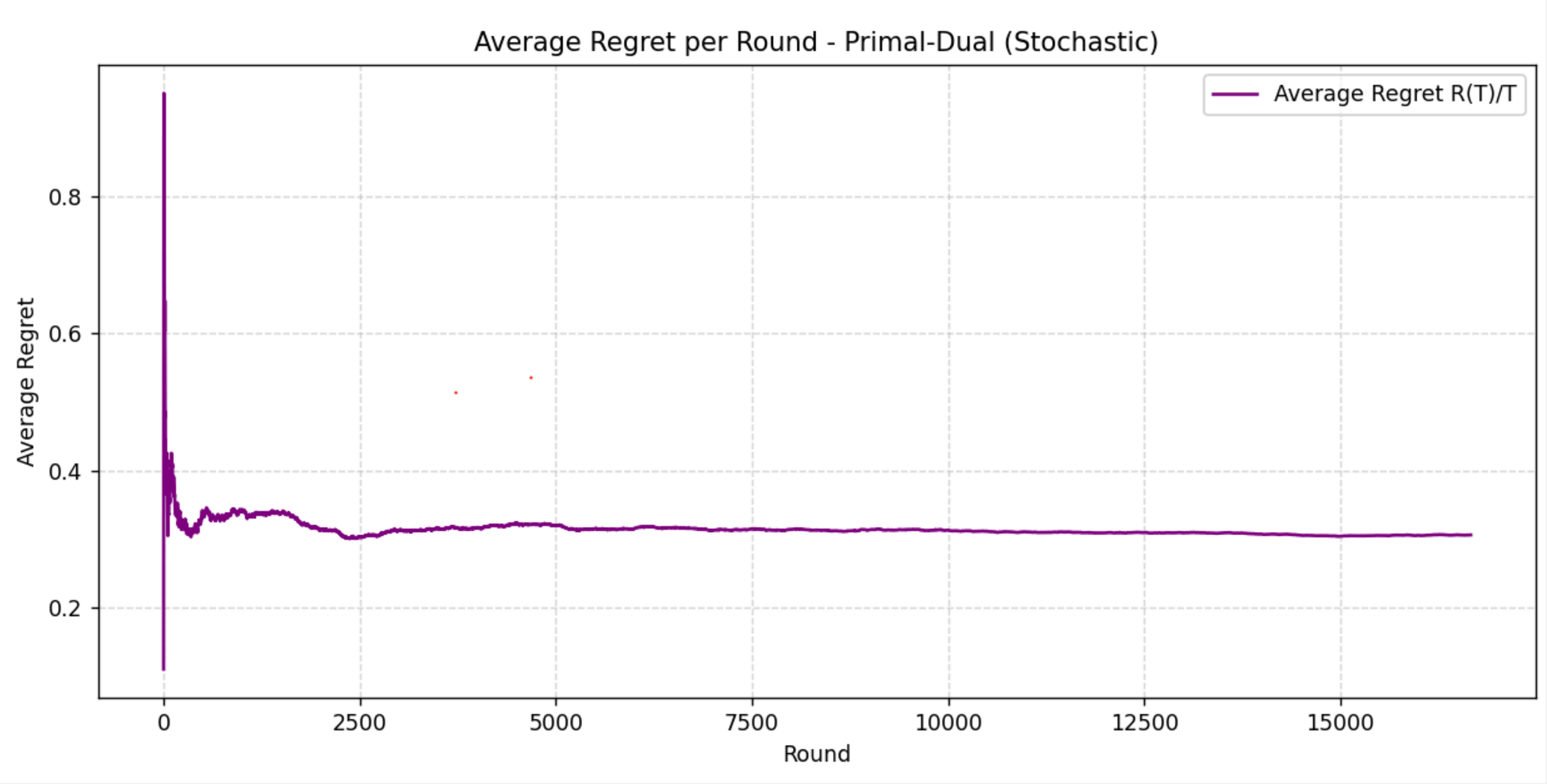
Revenue per round remains stable between 1 and 4.5, matching the uniform [0,1] valuations distribution.

The algorithm efficiently balances exploration/exploitation per product while respecting the inventory constraint via the dual variable.

**Conclusion**

The primal-dual approach maintains stability under stationary stochastic demand.

# Regret is linear — Despite Stationarity



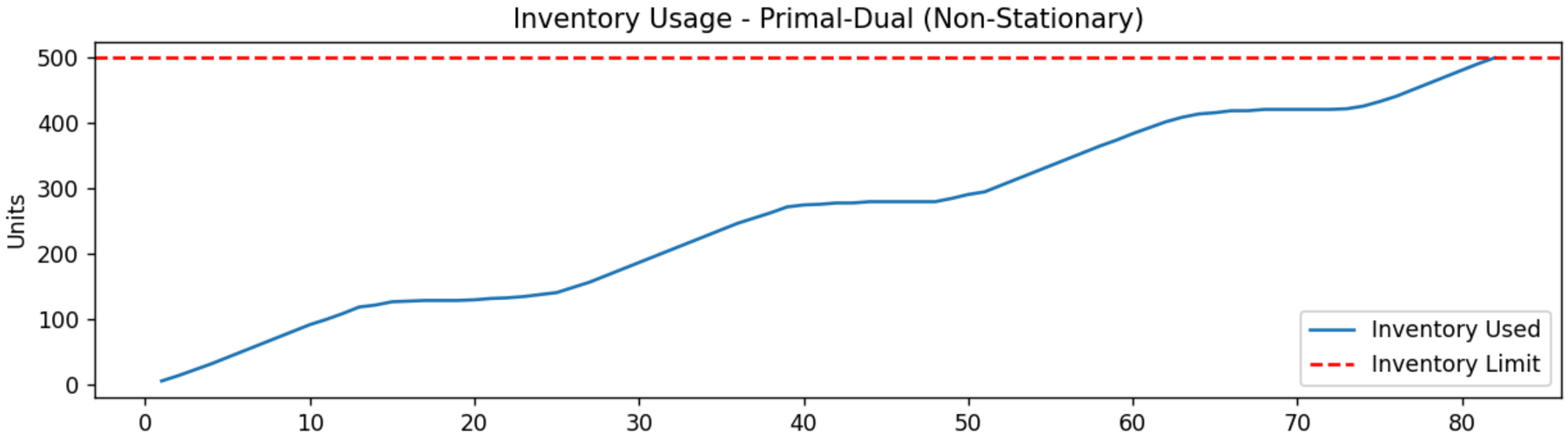

**possible explanation :**

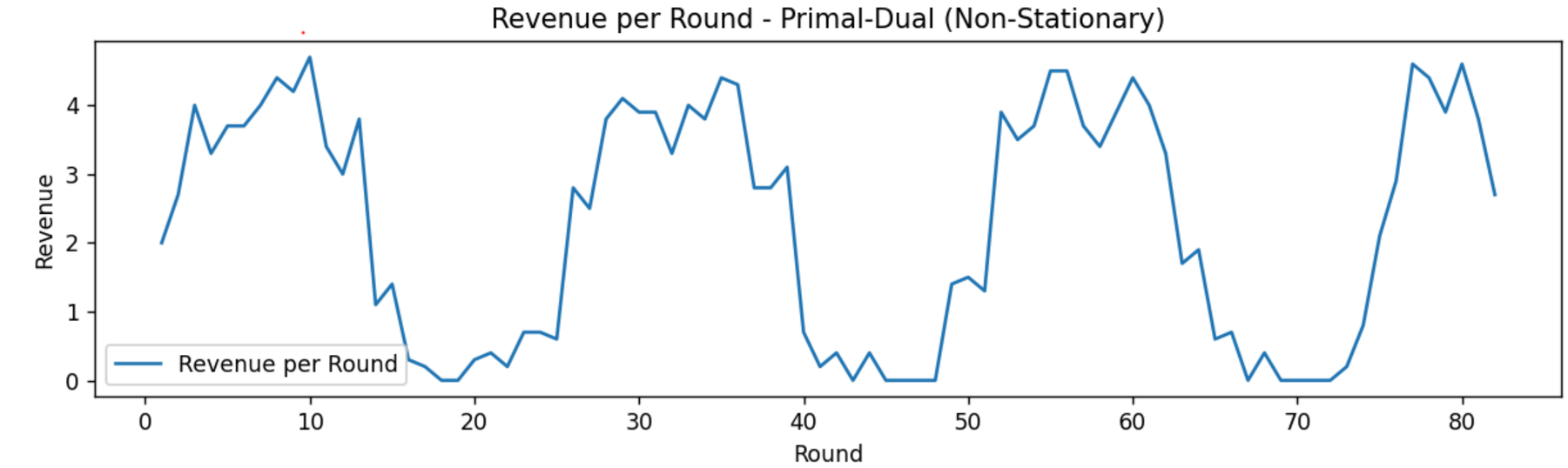
the algorithm might not fully satisfy the assumptions required for sublinear regret.

For example, the update rule may not exploit full-feedback optimally,
or the dual variable might not be tuned or projected correctly.

Also, the learning rate and other parameters may not be adapted to the scale or horizon of the problem.

# Results – Primal-Dual in Highly Non Stationnary Environment




**Observations**

Inventory usage is smooth and reaches the limit near the end, showing global constraint is respected.

Revenue per round follows a sinusoidal pattern, closely tracking the underlying demand (sinusoidal + noise).

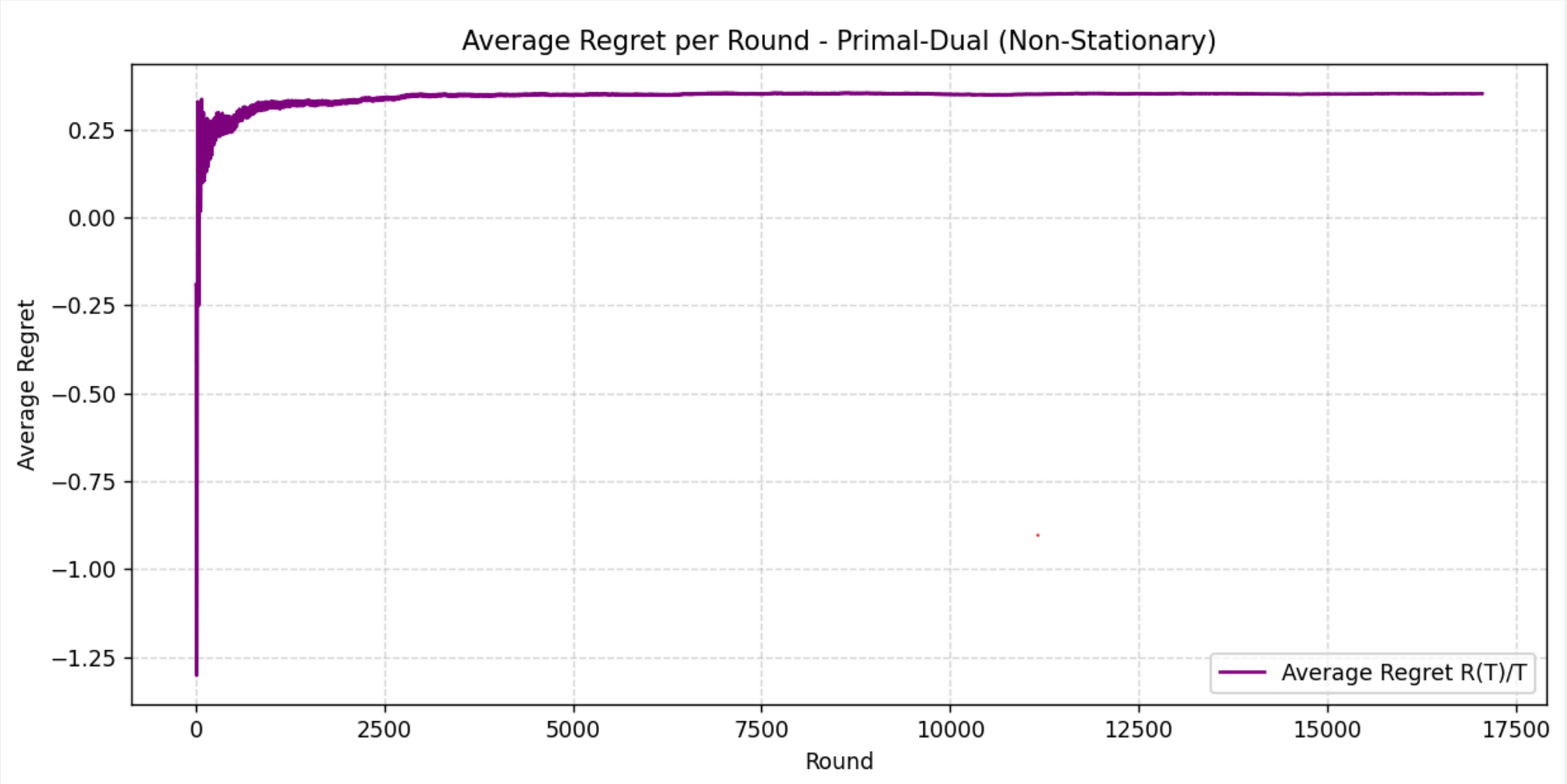
This confirms the algorithm adapts dynamically to rapid valuation shifts.

The dual variable helps maintain control over consumption despite non-stationarity.

**Conclusion**:

The primal-dual algorithm demonstrates strong adaptability, adjusting prices effectively in response to fast-changing demand.

# Regret is linear — Environment is Too Adversarial




**Possible explanations :**

the environment changes too fast for any fixed strategy to perform well.

The oracle benchmark is static, but the best action changes over time — making sublinear regret unachievable.

This is expected in a highly non-stationary setting.

# ***Conclusion***

The primal-dual algorithm adapts to both stationary and non-stationary environments:

- **Efficient** in **stable** markets.
- **Responsive** under **rapid** changes.
- Always **respects** the **inventory constraint** .

However we don’t achieve no regret in any of the environment

# Requirement 05

**Non-Stationary Environment Setup**

User behavior **evolves over time**, so a static pricing model may fail in this dynamic setup.

Object: design an algorithm to sell multiple types of products under production constraints, in a setting where demand evolves over time.

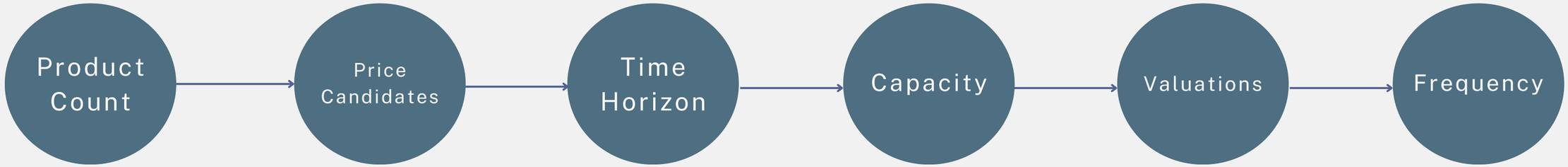
**Challenges**

- ✅ Non-stationarity: customer valuations change over time (piecewise-constant)
- ✅ Multiple products: must select a price for each product at each round
- ✅ Unknown demand function: must explore and learn from past outcomes
- ✅ Capacity constraint: total supply/production is limited per round

.

# Environment Set up

Requirement 05




**n_products = 8 (user-configurable)**

**[0.1, 0.2, ..., 1.0] (10 discrete options)**

**T= 1000 rounds.**

**capacity = 500× n_product**

**Buyer preferences are fixed within each interval, and shift between intervals**
**→ piecewise-stationary demand model with:**

**n_intervals = 2–16**

**Uniformly spaced changes**

# Algorithms Used



**Sliding Window CUCB**

- Exploration-exploitation algorithm for non-stationary multi-armed bandits
- At each round t, only the last W rounds are used to estimate:
- μt,W(a)=empirical mean,Nt-1,W(a) = number of times arm a was selected in the past W rounds
- Price selection is based on:

$$UCB_t(a) = \underbrace{\mu_{t,W}(a)}_{exploitation\ \ term} + \underbrace{\sqrt{\frac{2\log(T)}{N_{t-1,W}(a)}}}_{exploration\ \ term}$$

- Designed to adapt to shifts by forgetting outdated data

# Algorithms Used



**Thompson Sampling (TS)**
Type: Bayesian approach to exploration.

***Mechanism***

Maintains a posterior distribution over conversion probabilities.
Samples from the posterior to choose actions (prices).
Updates posterior based on observed outcomes.
Formula (Bernoulli):

**for** $a \in A$ **do**
  $\theta_a \sim Beta(\alpha_a, \beta_a)$ ;
play arm $a_t \in \arg\max \theta_a$;

Strength: Naturally balances exploration and exploitation

# Algorithms Used



**Primal Dual Algorithm**

- Solves a Linear Program to maximize revenue under constraints
- At each round, finds optimal allocation of prices across products
- Stable and effective in stationary environments
- Used as a baseline reference, not adaptive to changes

**CUSUM ADDITION**

- Adds a change detection mechanism to reset the window on shifts
- Tracks positive and negative cumulative deviation: $g_a^+ = \max\{0, g_a^+ + s_a^+\}$

$$g_a^- = \max\{0, g_a^- + s_a^-\}$$

- If g+>h, the agent resets its belief
- Helps catch abrupt changes, especially in valuation jumps

# Experimental Setup and Flow

Requirement 05

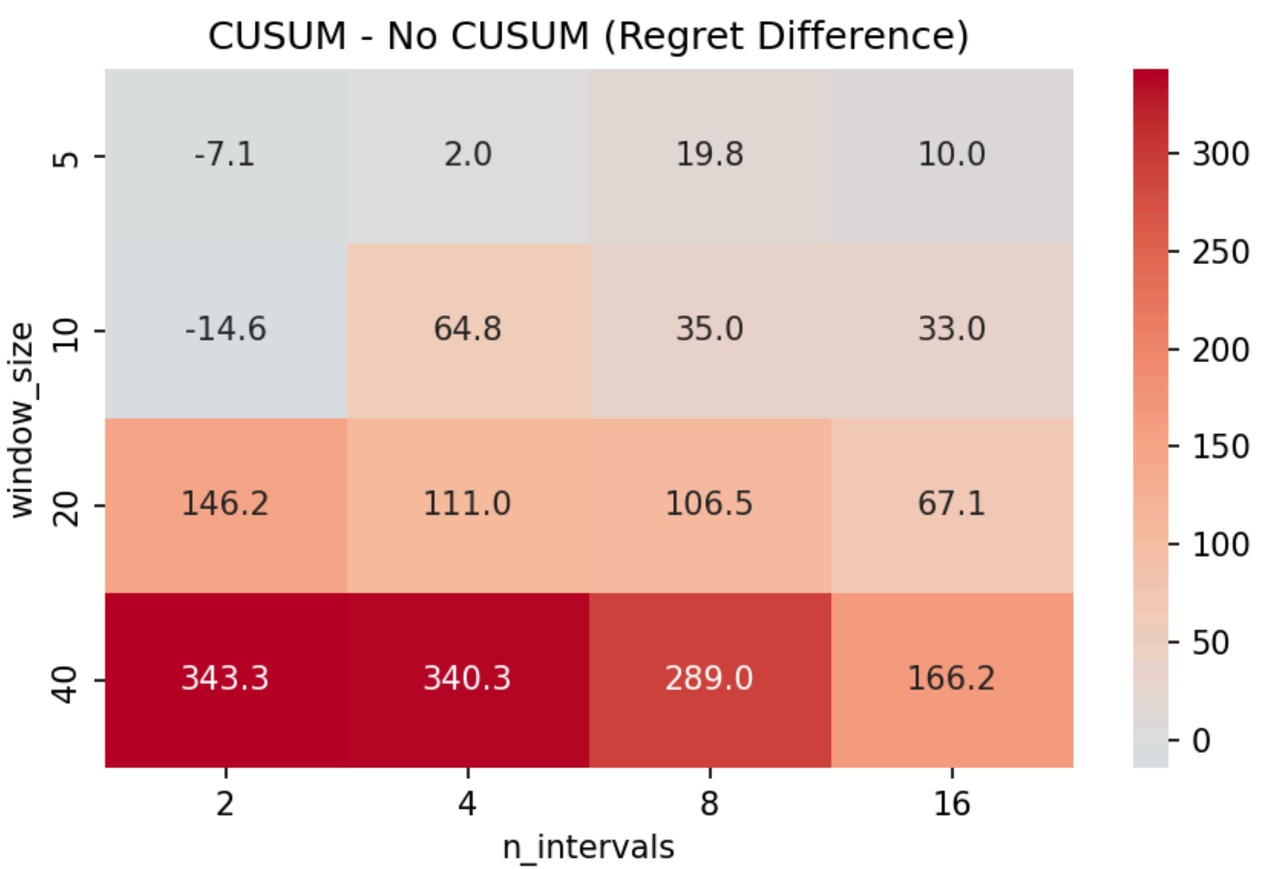
**Flow of experiments:**

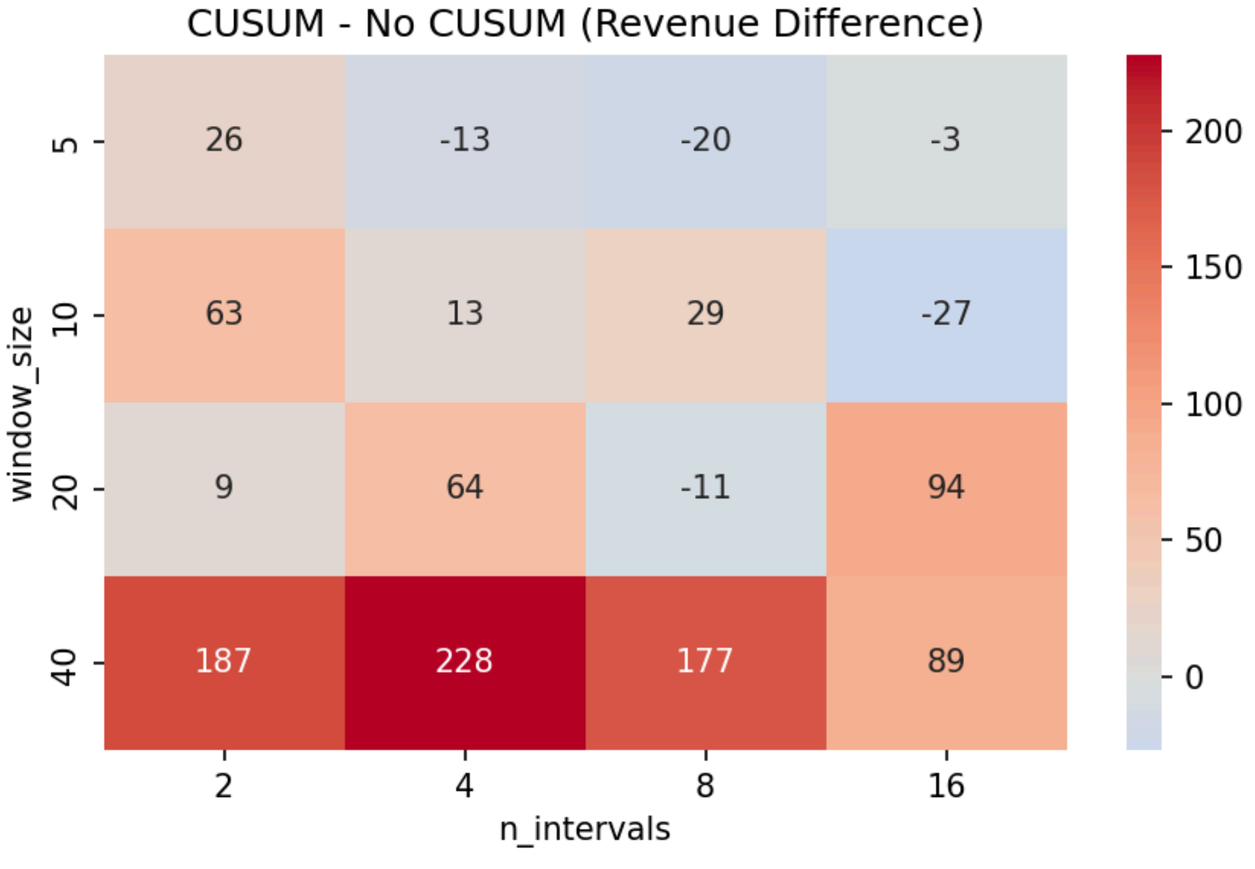
**[1] SWCUCB vs SWCUCB+CUSUM**
**[2] CUSUM Tuning (SWCUCB)**
**[3] SW-CUCB (with and without CUSUM) vs Primal-Dual**
**[4] TS vs TS-CUSUM**
**[5] CUSUM Tuning (TS)**
**[6] Final All-Algorithm Evaluation**

**Products: 8**
**Rounds: 1000**
**Price Options: 10 discrete values (0.1 to 1.0)**
**Inventory: 500 × n_products**
**Non-stationarity: 8 intervals**
**Demand model: Piecewise-stationary valuations**

While regret reveals how efficiently an algorithm learns over time, revenue directly captures its financial effectiveness. In our analysis, we use both: regret to understand adaptation quality, and revenue to measure outcome quality

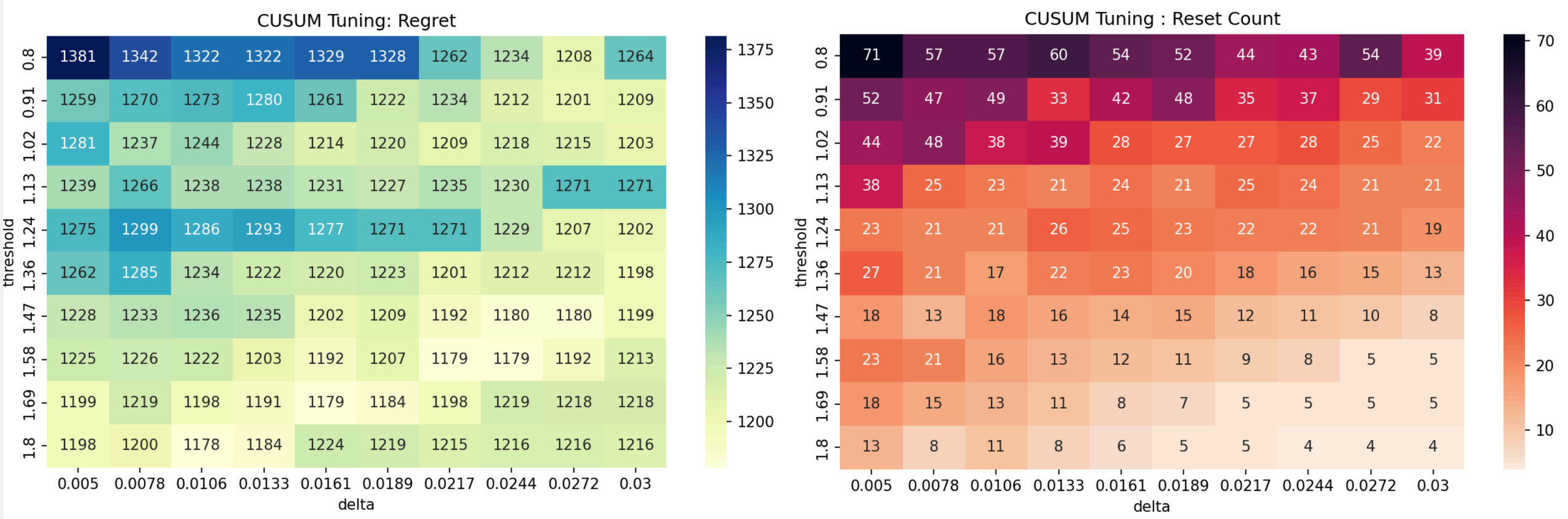
# Sliding Window CUCB and CUSUM



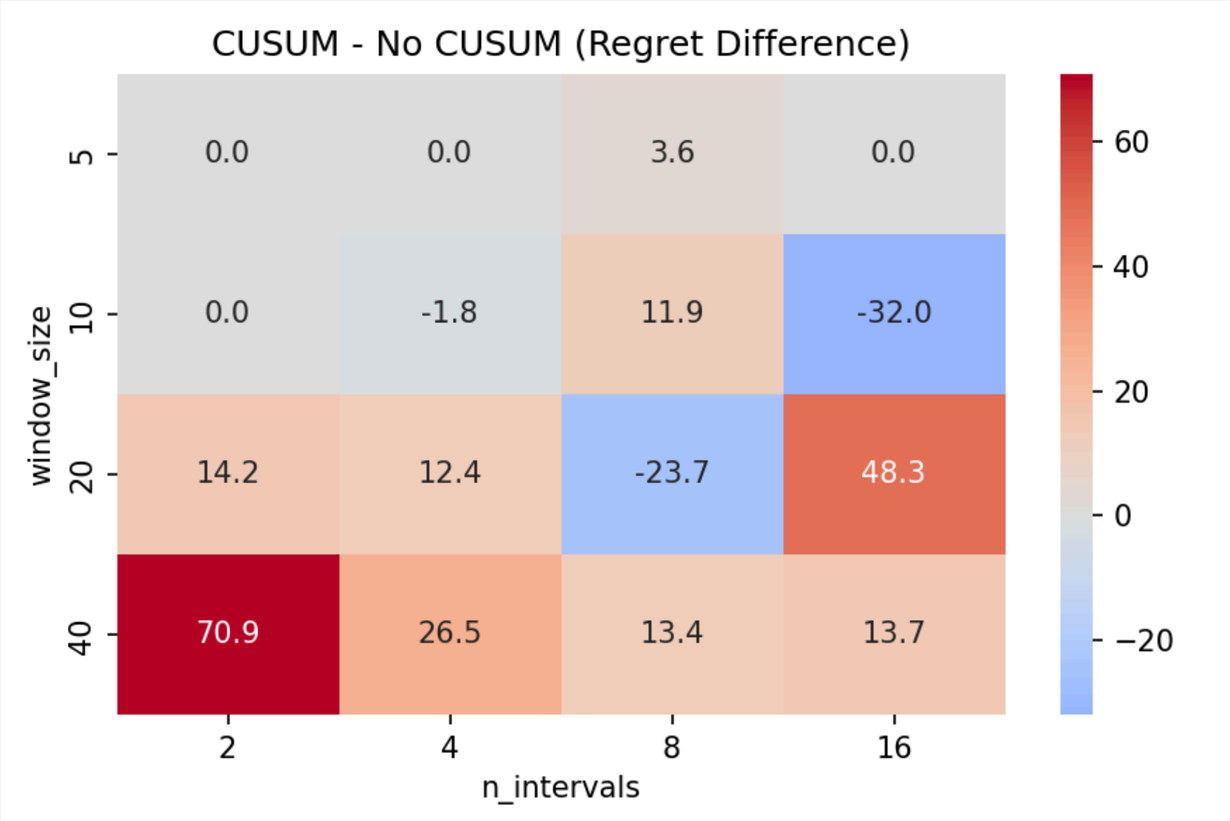

CUSUM offers improvement when the interval count and the window size are large enough to stabilize noise. (delta = 0.01 and threshold = 0.5 in set up)

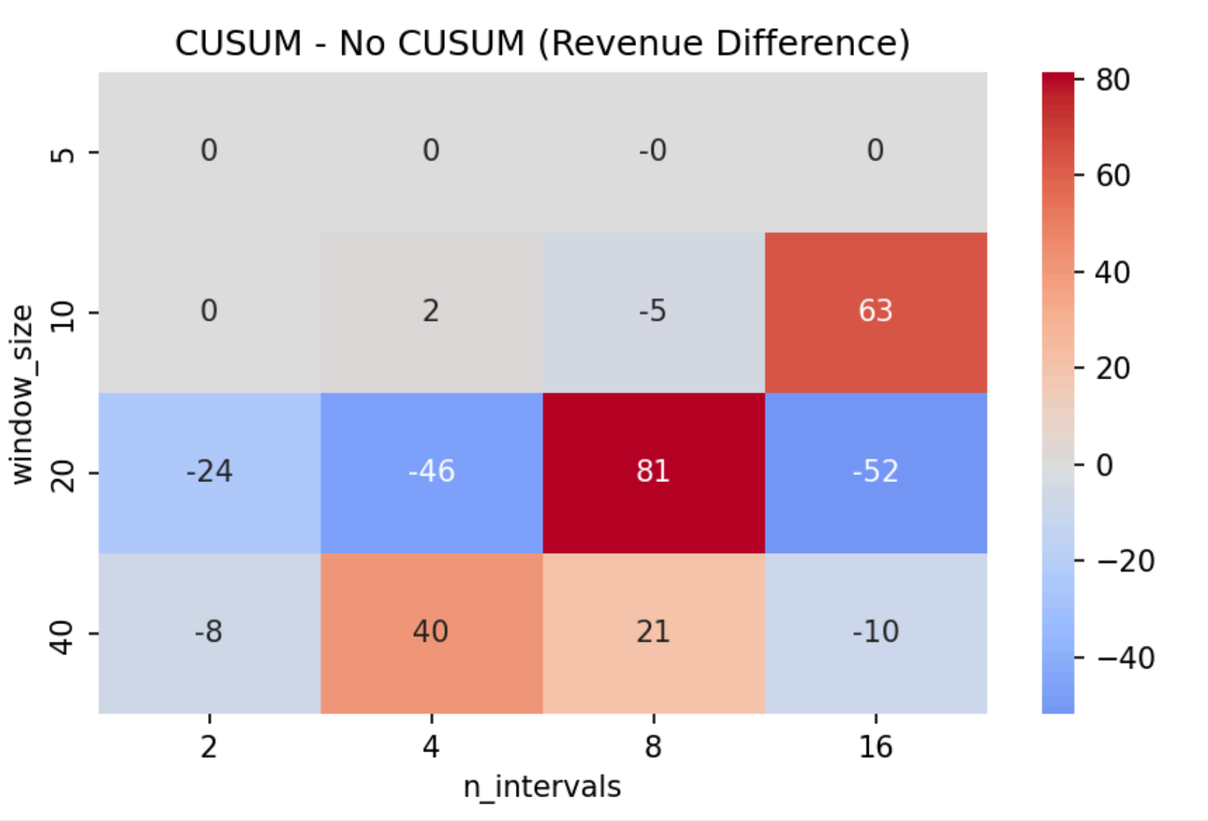
# CUSUM Tuning




We performed a grid search over CUSUM parameters to find the best sensitivity (delta) and reset threshold combination. The left heatmap shows cumulative regret (lower is better), while the right shows how often the CUSUM detector triggered resets. We aimed to find a balance: low regret with minimal unnecessary resets. (delta ≈ 0.0217, threshold ≈ 1.58 is an optimum candidate)

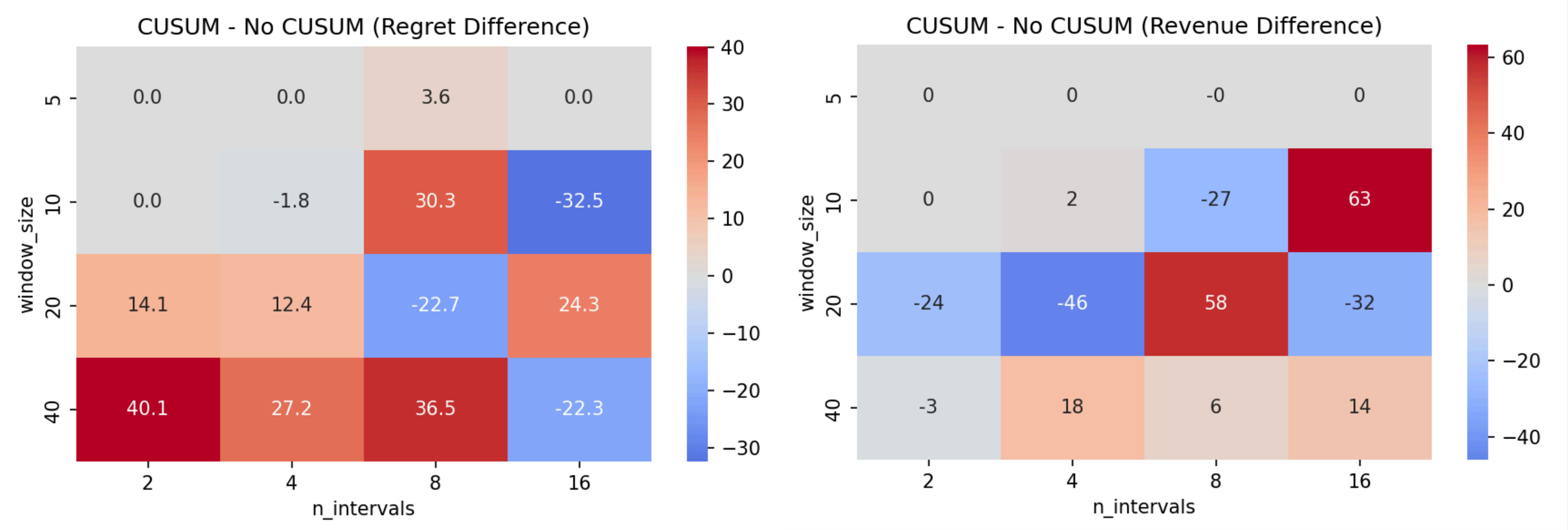
# Sliding Window CUCB and CUSUM



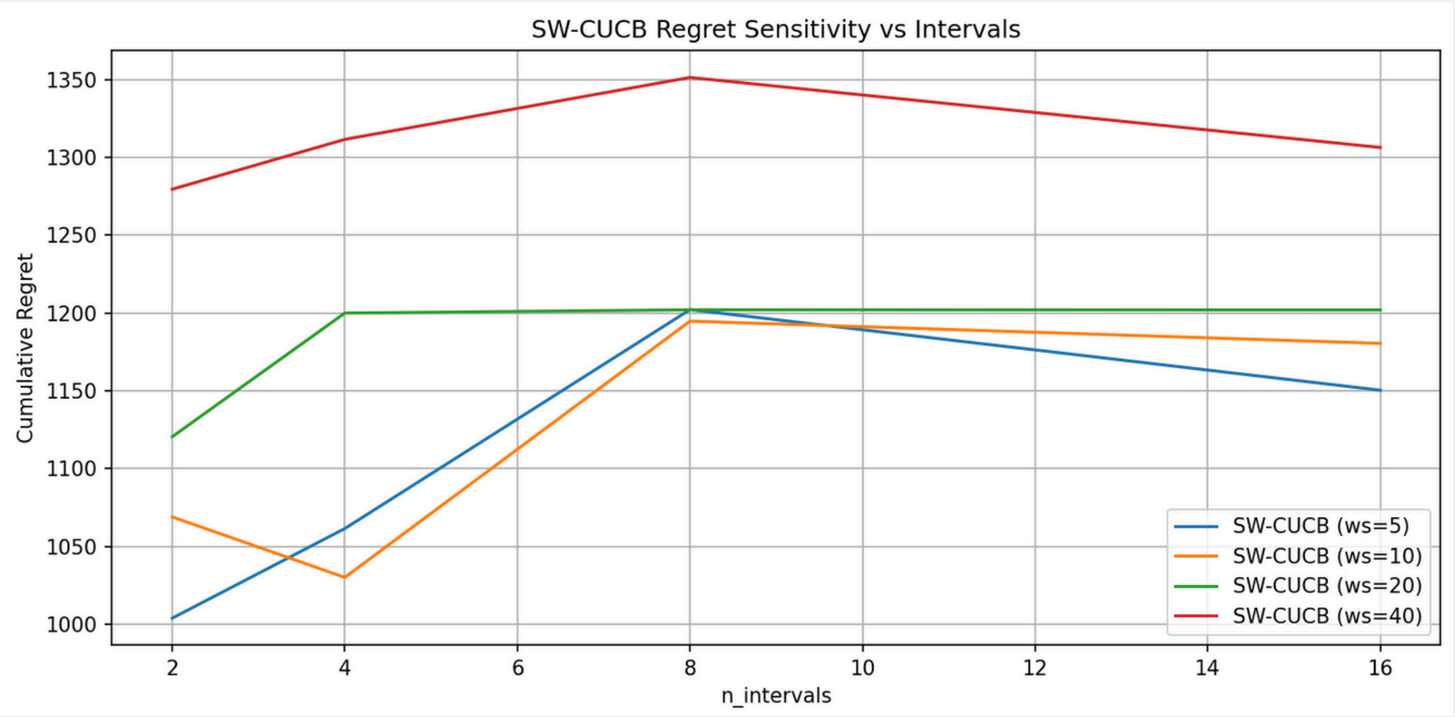

After tuning, CUSUM performs well in mid-interval scenarios but inconsistently elsewhere. (delta = 0.0106 threshold = 1.8)

# Sliding Window CUCB and CUSUM



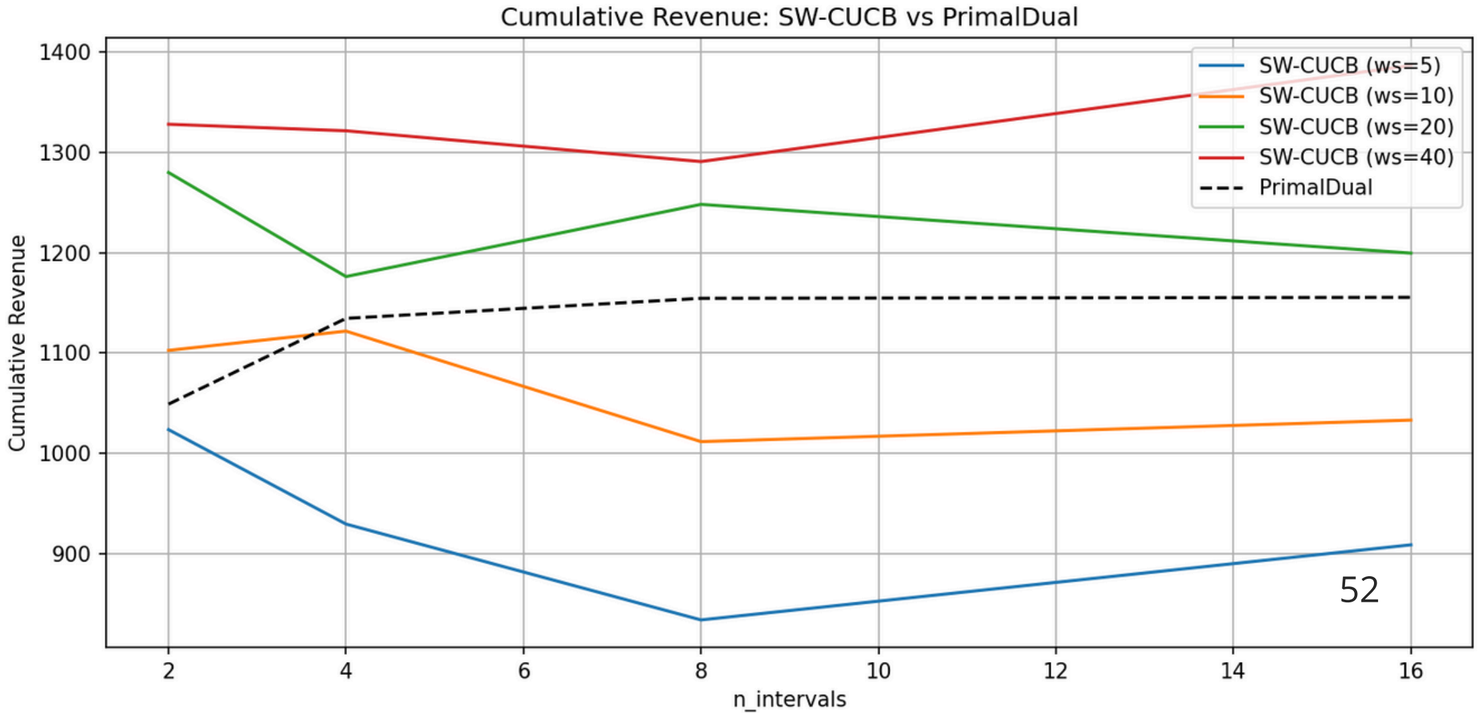

Another potential candidate that enhances performance in challenging conditions, particularly at window size 40 and n_intervals = 16. (delta =0.0217, threshold=1.58)

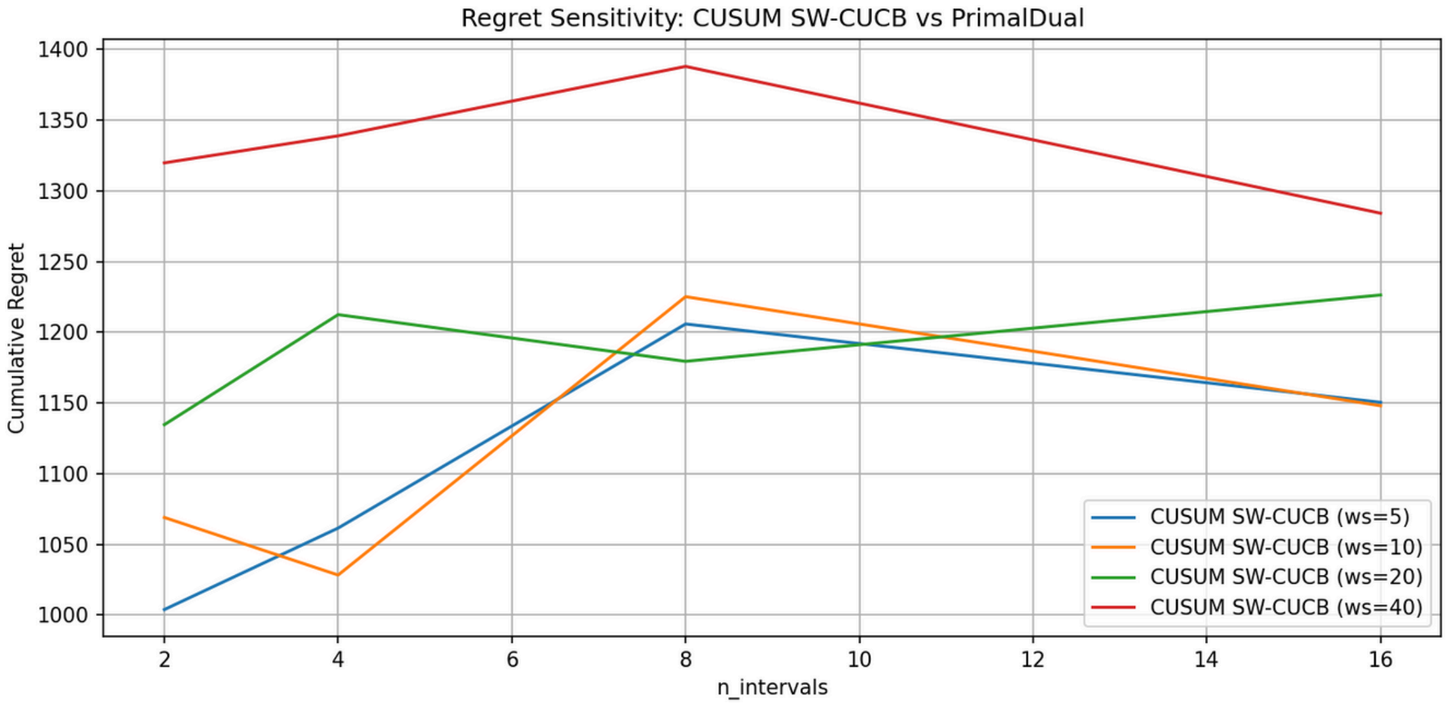
**CUSUM's effect depends on window size and needs tuning accordingly**




- Smaller window sizes (5 and 10) adapt faster to changes but suffer higher variance, leading to non-monotonic regret trends
- Larger window sizes (20 and 40) adapt slowly, resulting in higher regret as the number of intervals increases (40 consistently has the worst regret)

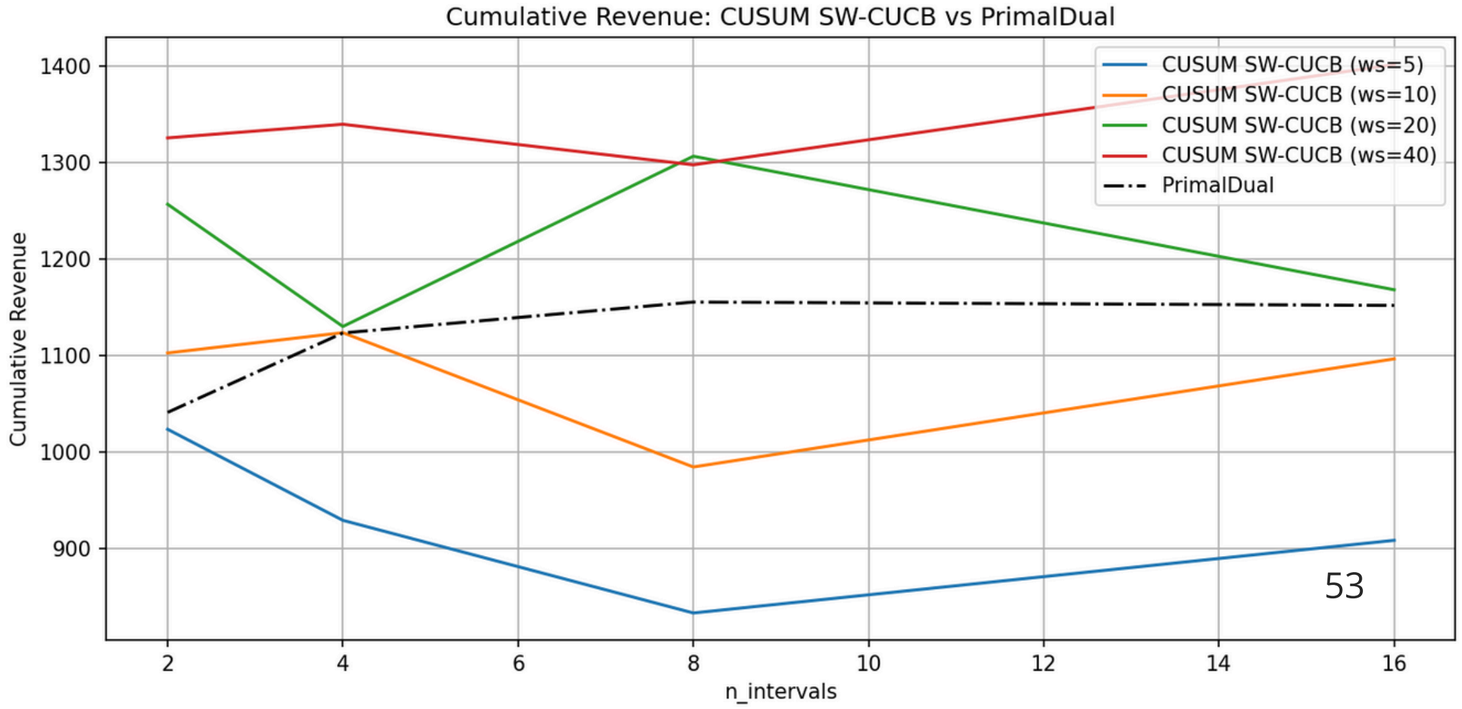
- SW-CUCB with window size 40 achieves the highest overall revenue, benefiting from stable learning. In contrast, smaller window sizes (5, 10) tend to overreact to noise, leading to unstable performance.
- Primal-Dual stays constant as it does not adapt to changes in the environment.



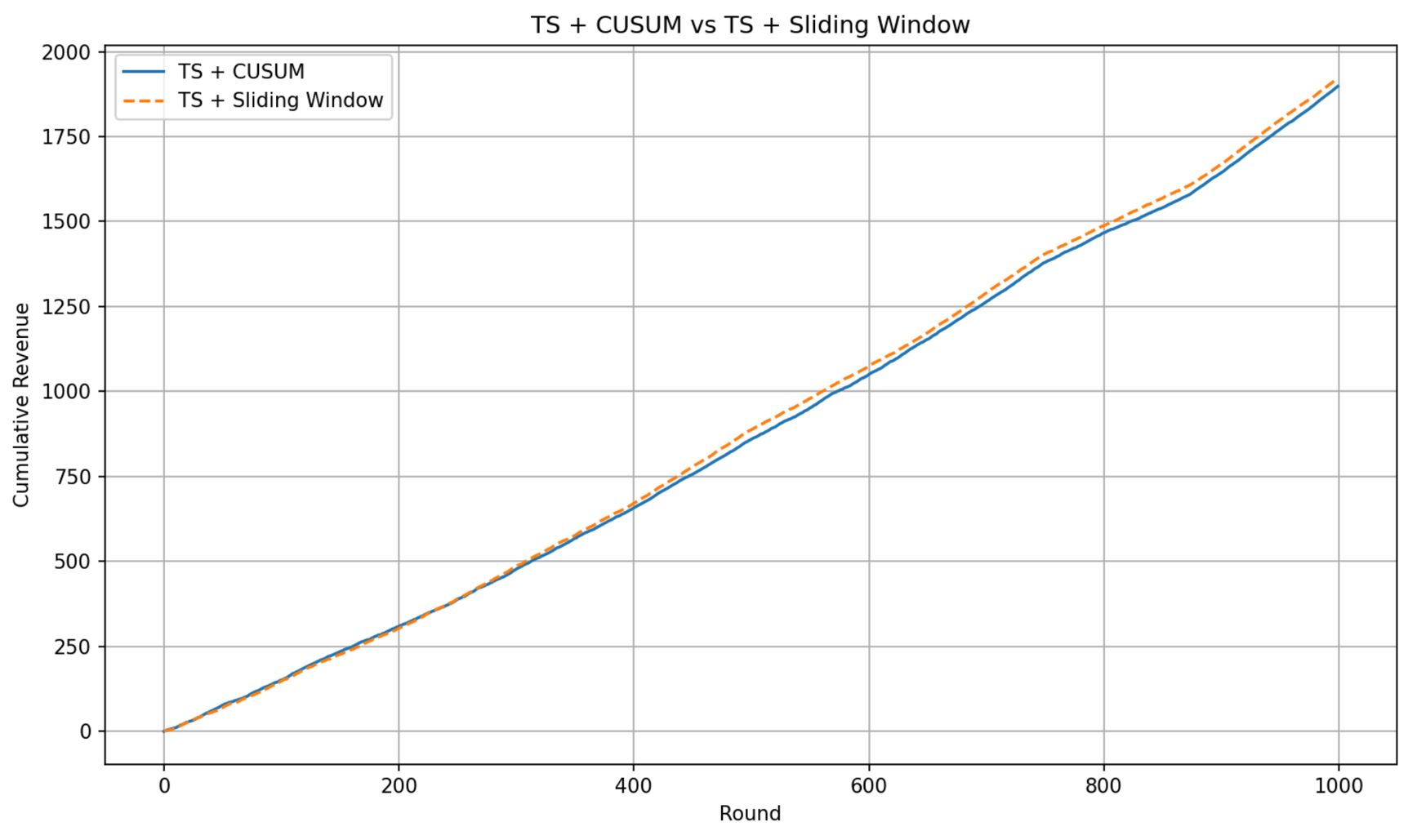



- Small window sizes (ws=5, 10) are quick to detect changes but suffer from overreaction, while larger window sizes (ws=20, 40) show smoother regret evolution but incur higher regret due to slower adaptation to environment shifts.

- Similar to SW-CUCB with sharper changes in intervals due to the change detector presence.

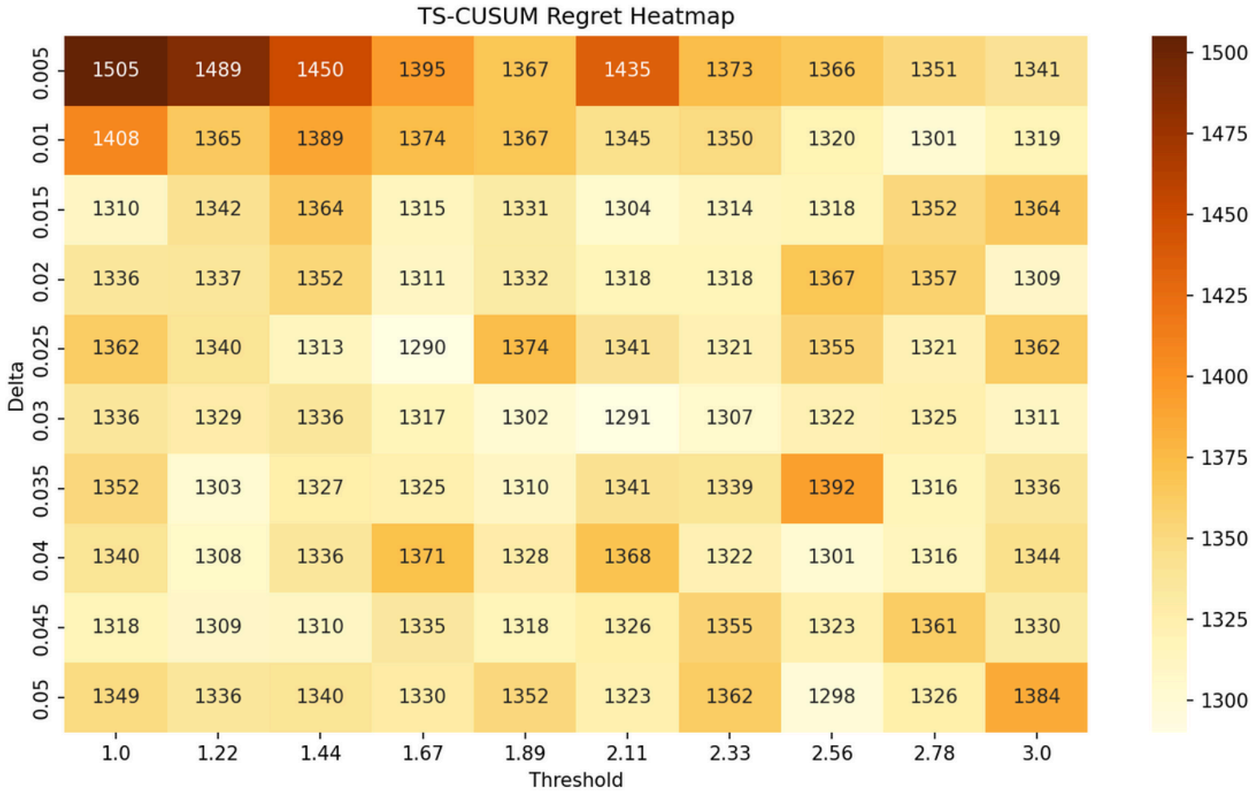


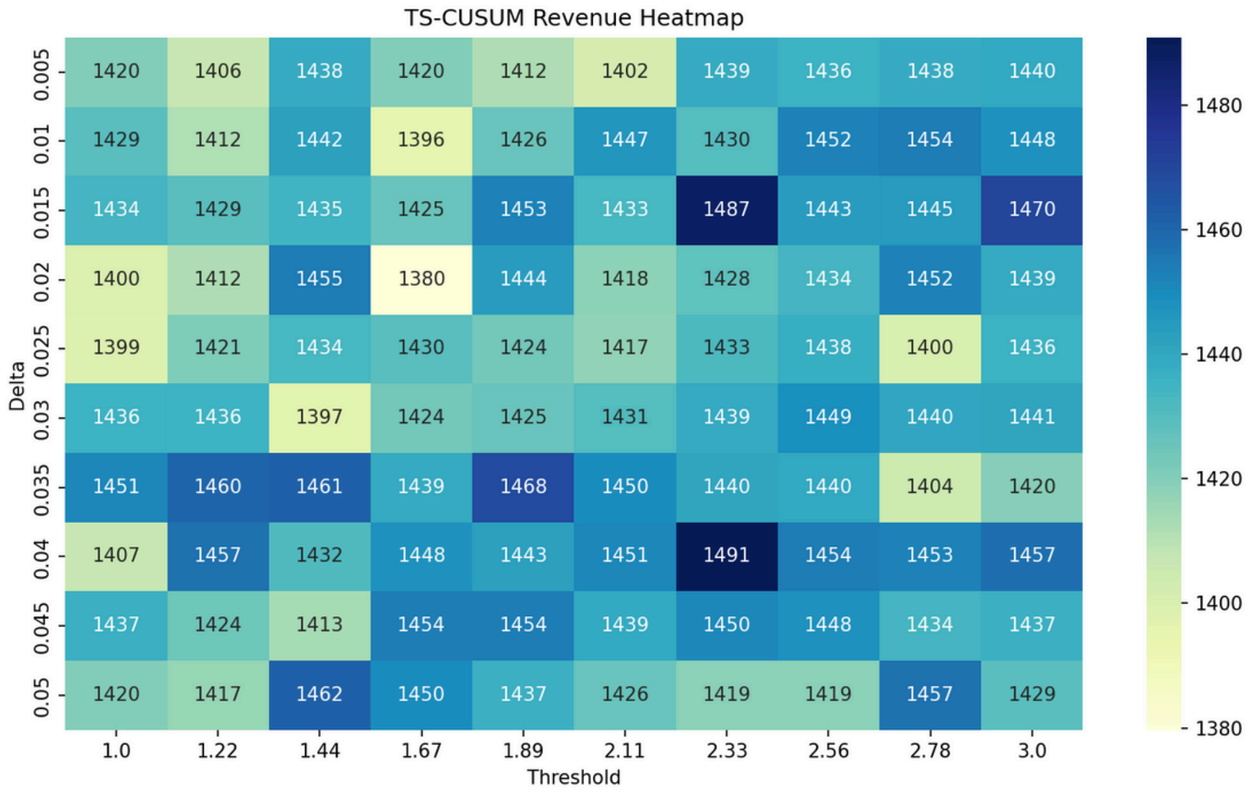
# Sliding Window TS and CUSUM



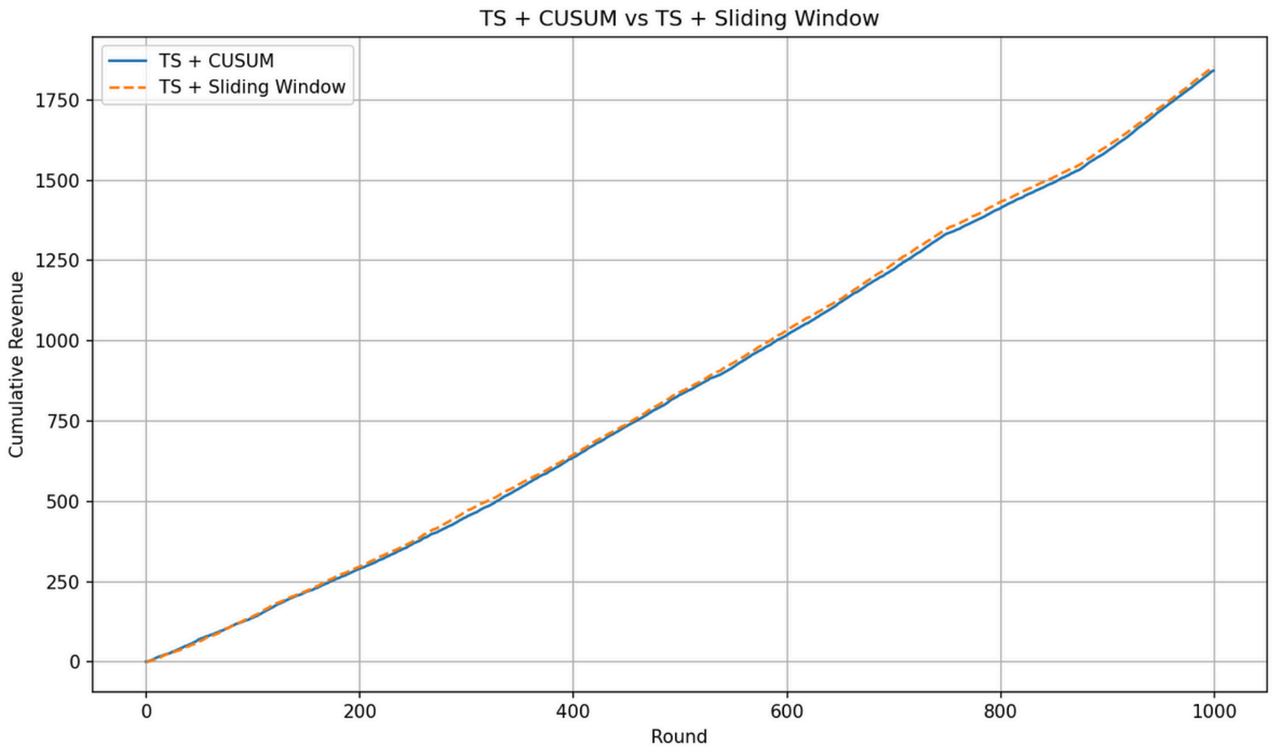

In this configuration, both TS + CUSUM and TS + Sliding Window perform similarly. CUSUM resets do not significantly improve adaptation under these parameters, indicating the need for tuning. (delta = 0.04 cusum_threshold = 1.6, window_size = 20)

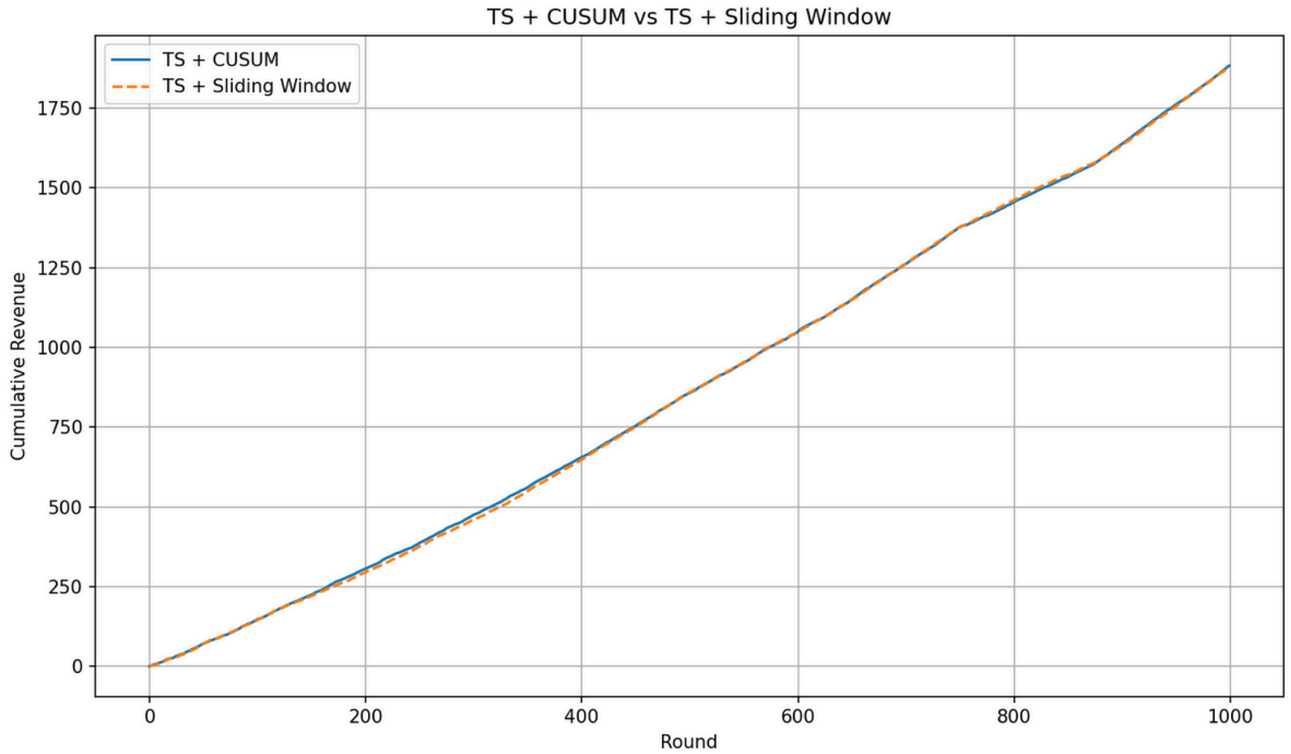
# CUSUM Tuning (for TS)




We targeted configurations that minimize regret while maximizing revenue. So, CUSUM Tuning Parameters Delta (Sensitivity to change, smaller values detect smaller shifts) and Threshold (Reset trigger, higher values delay detection, lower values react quickly) were evaluated across a range of values to identify the optimal balance between responsiveness and stability.



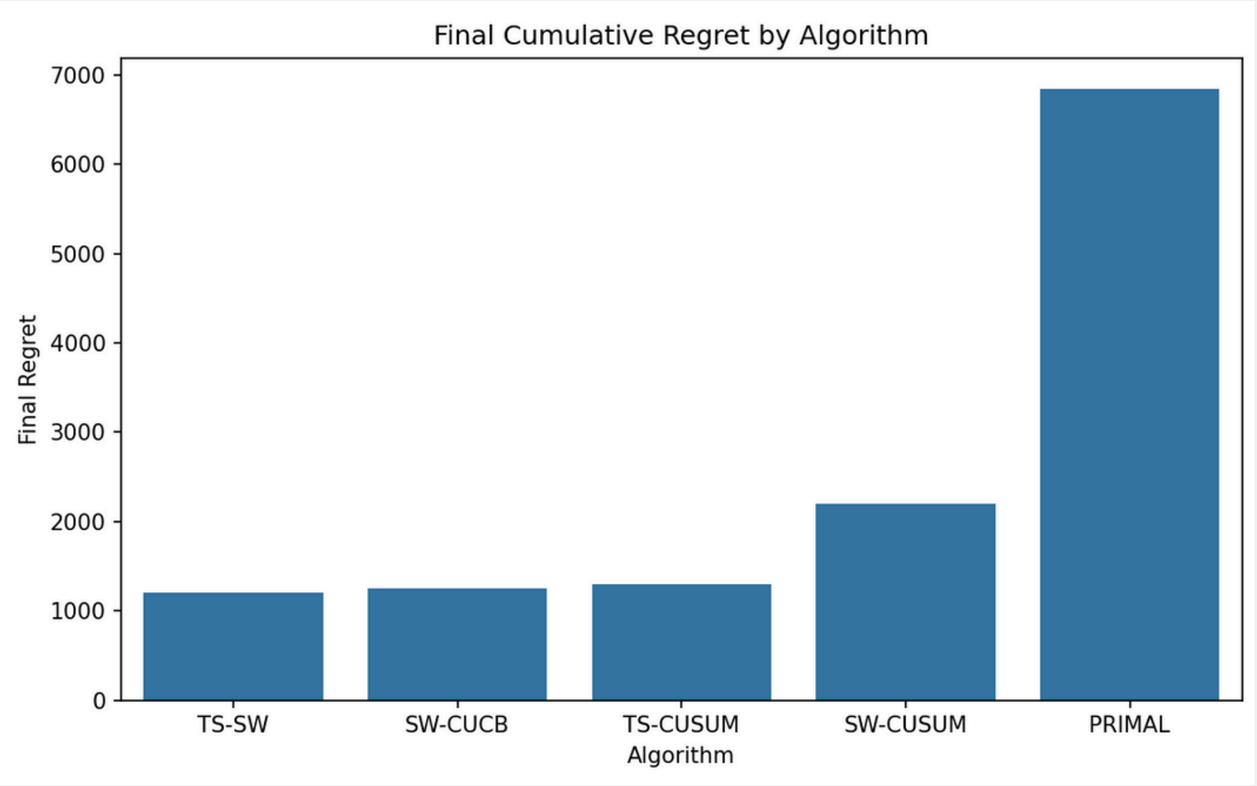



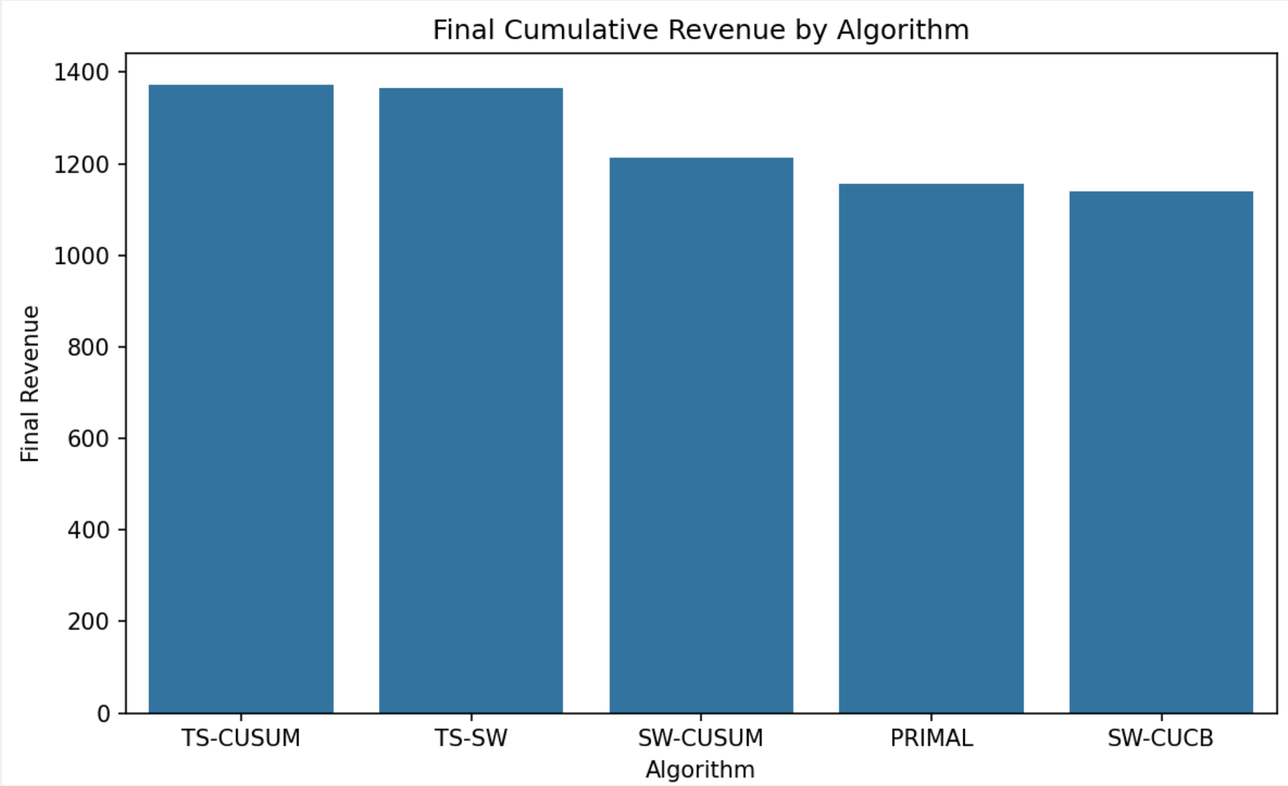

delta = 0.025
cusum_threshold = 1.67
window_size = 20

delta = 0.04
cusum_threshold = 2.33
window_size = 20

Even with tuning, TS + CUSUM closely tracks TS + Sliding Window, indicating CUSUM's marginal gains for TS under this environment.

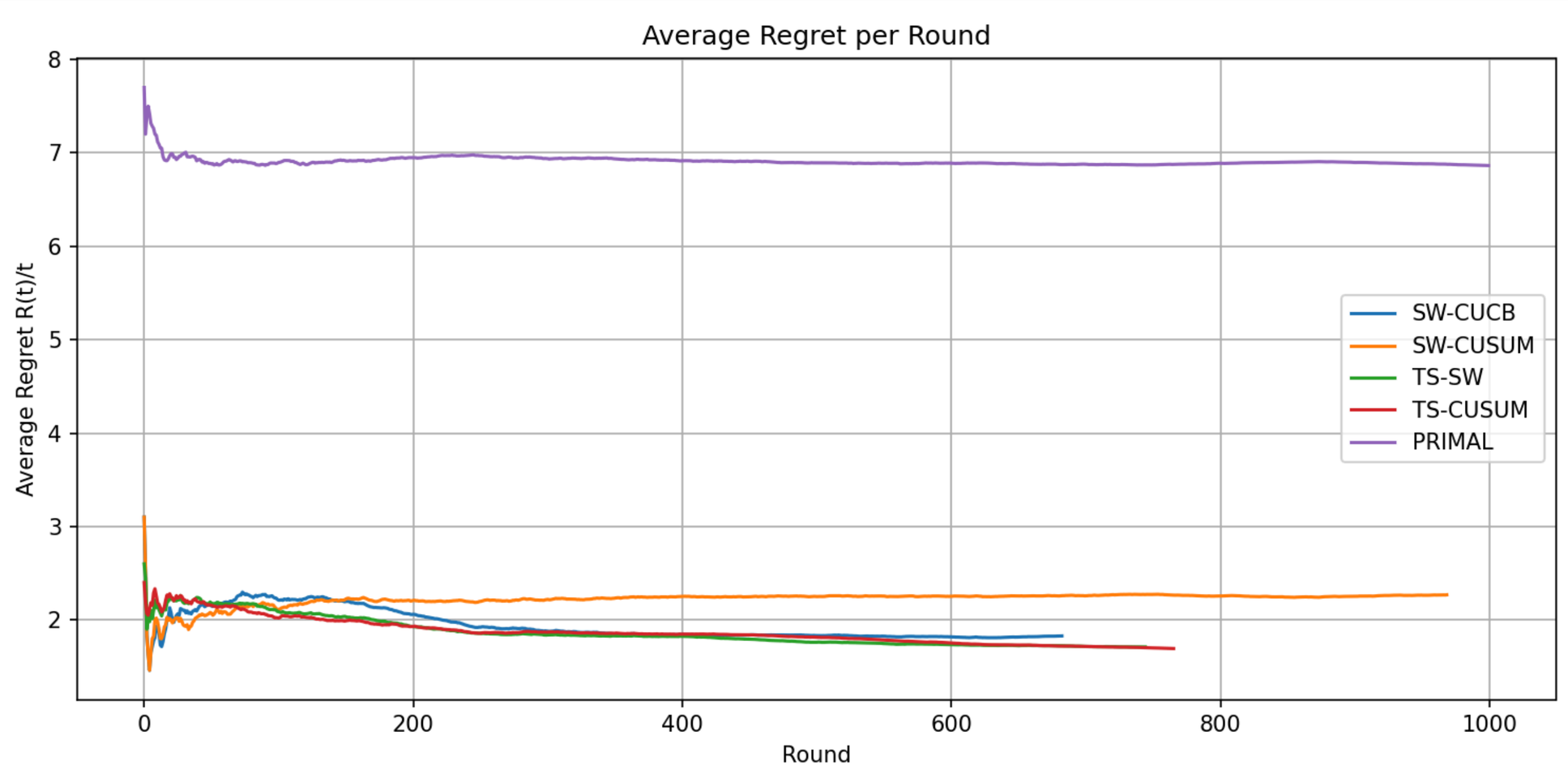
# Final Comparison




TS-based algorithms achieve the best balance of regret and revenue. Sliding Window variants are consistently strong. Primal-Dual, while stable, suffers from high regret due to a lack of adaptation.

# Final Comparison - learning efficiency




This plot shows that TS-based algorithms consistently achieve the lowest average regret per round, indicating the best learning efficiency in a changing environment. SW-CUCB performs reasonably well, while Primal-Dual accumulates regret steadily due to its non-adaptive nature

# THANK YOU

**Questions?**

# APPENDIX

**Some extra things.**

**UCB1 without inventory constraints**

**IMPLEMENTATION**

Requirement 01

```
def select_prices(self) -> Dict[int, float]:
    """Select price using UCB1 formula: μ̂(p) + √(2*log(t)/n(p))"""
    best_arm = None
    best_ucb_score = -float('inf')

    for arm_id, price in enumerate(self.prices):
        if self.arm_counts[arm_id] == 0:
            ucb_score = float('inf')  # Unplayed arms have infinite score
        else:
            mean_reward = self.arm_means[arm_id]
            confidence_radius = self.confidence_width * sqrt(log(t) / self.arm_counts[arm_id])
            ucb_score = mean_reward + confidence_radius

        if ucb_score > best_ucb_score:
            best_ucb_score, best_arm = ucb_score, arm_id

    return {0: self.prices[best_arm]}  # Return selected price
```

# *UCB1 Price Selection*

## UCB1 with inventory constraints
## IMPLEMENTATION

Requirement 01

```python
def select_prices(self) -> Dict[int, float]:
    """Solve: argmax f̄^UCB(arm) subject to c̄^LCB(arm) ≤ remaining_capacity"""

    if self.remaining_capacity <= 0:
        return {}  # No capacity available

    # STEP 1: Calculate reward UCB bounds for all arms
    reward_ucb = {}
    for arm_id in range(len(self.prices)):
        reward_ucb[arm_id] = self.arm_means[arm_id] + sqrt(2*log(t)/self.arm_counts[arm_id])

    # STEP 2: Calculate constraint LCB bounds for all arms
    constraint_lcb = {}
    for arm_id in range(len(self.prices)):
        if self.constraint_counts[arm_id] == 0:
            constraint_lcb[arm_id] = 0.5  # Optimistic estimate for unplayed arms
        else:
            mean_constraint = self.constraint_means[arm_id]
            confidence_radius = sqrt(2*log(t) / self.constraint_counts[arm_id])
            constraint_lcb[arm_id] = max(0.0, mean_constraint - confidence_radius)
```

***UCB-Constrained Selection (Following Auction Theory)***

```python
# STEP 3: Find feasible arms (satisfy capacity constraint)
feasible_arms = [arm for arm in range(len(self.prices))
                 if constraint_lcb[arm] <= self.remaining_capacity]

if not feasible_arms:
    return {}  # No feasible arms

# STEP 4: Among feasible arms, select one with highest reward UCB
best_arm = max(feasible_arms, key=lambda arm: reward_ucb[arm])

return {0: self.prices[best_arm]}
```

## UCB1 without inventory constraints
## IMPLEMENTATION

```
def update(self, prices, rewards, buyer_info):
    # Update parent UCB1 reward statistics
    super().update(prices, rewards, buyer_info)

    # UPDATE CONSTRAINT STATISTICS
    if prices:  # If we produced something
        selected_price = prices[0]
        arm_id = self._price_to_arm_id(selected_price)
        capacity_consumed = 1.0  # Each production uses 1 capacity unit

        # Update constraint statistics for this arm
        self.constraint_counts[arm_id] += 1
        self.constraint_totals[arm_id] += capacity_consumed
        self.constraint_means[arm_id] = (self.constraint_totals[arm_id] /
                                         self.constraint_counts[arm_id])

        # Update remaining capacity
        self.remaining_capacity -= capacity_consumed
```

# *Constraint Statistics Update*

# Primal-Dual Pricing Algorithm
## IMPLEMENTATION

```
class PrimalDualPricingAlgorithm:
    def __init__(self, prices, production_capacity, horizon_T):
        # Theoretical learning rates
        self.eta_primal = sqrt(log(K) / T)      # EXP3-like
        self.eta_dual = 1.0 / sqrt(T)           # Dual convergence
        self.gamma = sqrt(K * log(K) / T)       # Exploration
        self.rho = production_capacity / T       # Per-round budget

        # Algorithm state
        self.lambda_t = 0.0                     # Dual variable
        self.price_probabilities = uniform(K)   # Primal variables
```

### *Core Algorithm Structure*

```
def select_prices(self):
    # Sample from learned distribution over prices
    price_idx = np.random.choice(K, p=self.price_probabilities)
    return {0: self.prices[price_idx]} if capacity_available else {}
```

### *Price Selection (Primal)*

## Primal-Dual Pricing Algorithm
## IMPLEMENTATION

Requirement 03

```
def update(self, selected_prices, rewards, buyer_info):
    # Compute Lagrangian values for all prices
    for p in prices:
        estimated_reward = p if buyer_valuation >= p else 0
        estimated_cost = 1 if buyer_valuation >= p else 0
        lagrangian[p] = estimated_reward - λ * (estimated_cost - ρ)

    # EXP3-like primal update with exploration
    self.reward_estimates += eta_primal * lagrangian
    weights = exp(self.reward_estimates)
    self.price_probabilities = (1-γ)*normalize(weights) + γ*uniform

    # Dual variable update with projection
    self.λ = max(0, λ + eta_dual * (actual_cost - ρ))
```

***Learning Updates (Primal + Dual)***