RWTH Aachen University

M.Sc. Data Analytics & Decision Science

Vodafone Group Plc

Project Report

Pricing with Learning of Demand Curves under Inventory Constraint

Author and matriculation number:

Vijayan Shinde - 465577

August 14th, 2025

Advisors:

Prof. Dr. Sven Müller
Chair of Data and Business Analytics
RWTH Aachen University

Dr. Guenter Klas
Senior Manager R&D, Research Clusters AI and Quantum
Vodafone Group Plc

Abstract

Dynamic pricing under inventory constraints is a critical problem in revenue management, especially when customer demand functions are unknown and heterogeneous. This project implements and tests the Primal-Dual Learning Algorithm for Personalized Dynamic Pricing with an Inventory Constraint, as proposed by Chen and Gallego (2018). The algorithm leverages the estimation of a dual variable representing the shadow price of inventory to decouple a multi-type pricing problem into independent single-type learning tasks, thereby avoiding the curse of dimensionality.

The implementation was developed in Python within a modular Jupyter Notebook framework, incorporating both the exploration phase (price discretization, demand observation, and dual estimation) and the exploitation phase (price interval refinement and inventory-aware allocation). Simulated demand data for multiple customer types were generated using logistic models and extended with machine learning estimators to capture unknown demand relationships. Key performance metrics including optimal prices, estimated demand functions, actual revenue, oracle benchmark revenue, and regret were computed and analyzed.

Experimental results demonstrate that the implemented algorithm consistently achieves low regret (all below 10, with a minimum near 0) and produces revenue close to the oracle benchmark across varying inventory scenarios. The findings validate the theoretical guarantees of the primal-dual approach and highlight its potential for scalable, personalized pricing strategies in practice.

Acknowledgements

We would like to express our sincere gratitude to Dr. Guenter Klas, Dr. Simone Mangiante, and Ana Ccarita from Vodafone for providing the opportunity to undertake this project and for their continuous support, valuable insights, and encouragement throughout its development. We also wish to extend our deep appreciation to professor Dr. Sven Müller for facilitating this collaboration and for offering consistent guidance, constructive feedback, and expertise during all stages of the project.

Contents

List of figures:

List of flowcharts:

List of codes:

1. Introduction

1.1 Motivation and Industry Context

- **Industry Challenges**: Telecom operators face shrinking margins due to high operating costs, ongoing investments in next-gen networks, and intense competition in the consumer market.
- **OTT Disruption**: Applications like Netflix, YouTube, and WhatsApp use mobile networks heavily without compensating operators, shifting value away from network providers.
- **Strategic Response**: Operators are exploring new revenue models—such as value-added services, private 5G, and Network as a Service (NaaS)—though financial returns remain uncertain.
- **Emerging Opportunity**: A promising idea is to monetize underutilized compute capacity in the Radio Access Network (RAN), especially GPU resources that are idle during off-peak hours.
- **AI Integration**: As AI becomes central to 5G/6G RAN operations (e.g., energy optimization, beam steering), GPU acceleration is increasingly valuable internally.
- **Two-Sided Potential**: This creates a dual opportunity—use GPUs for internal AI tasks during peak periods and offer idle capacity to external enterprise customers during off-peak windows.
- **Operator Caution**: Despite vendor interest, operators are wary due to risks like operational complexity, balancing internal vs. external workloads, uncertain demand, and pricing challenges.
- **Revenue Management Need**: Designing effective pricing and allocation mechanisms is a key unresolved issue in making RAN compute monetization commercially viable.

1.1.1 Research Gap and Rationale

- **Current Focus**: Industry discussions center on technical feasibility—AI-ready hardware, virtualization, and cloud-native RAN.
- **Missing Piece**: There's limited analysis on how to price and allocate time-sensitive, limited GPU inventory across different products and unpredictable demand.
- **Study Contribution**: This work addresses that gap by developing a revenue management framework tailored to the telco RAN setting, where AI-accelerated compute is the core constrained resource.

1.2 Problem statement

We study a mobile network operator that deploys GPU resources to support AI-enhanced 5G/6G RAN operations. GPU utilization is time-varying: it is high during daytime and telco peak hours and lower during off-peak periods, creating windows where excess capacity can be offered to enterprise customers. Compute is cloudified and virtualized, enabling productization and time-windowed access.

- The operator sells GPU capacity in discrete units up to a capacity limit (c) (resource constraint).
- Sales occur over a finite booking horizon (t in [0, T]), with purchased capacity scheduled for use in a subsequent service window (t in [T, 2T]); revenue is realized over the booking horizon.
- Two product types are offered:
    - **Reserved instances**: exclusive, non-interruptible access during operator-specified hours in the service window.
    - **Preemptable instances**: lower-priced access that may be interrupted intermittently, suitable for non-critical workloads.
- All instances share the same GPU hardware type; GPUs are the binding resource and are economically more valuable than CPUs.

- The operator observes its internal time-varying GPU utilization and knows the windows in which capacity can be safely offered to third parties.

The operator's objective is to design a pricing and allocation policy that maximizes expected revenue from selling idle GPU units during the booking horizon, subject to stochastic demand and inventory constraints across products and time windows. The policy must determine product-specific prices and acceptance decisions over time while ensuring that allocations do not compromise internal network performance in the service window.

Formally, this is a finite-horizon revenue management problem with:

- **Inventory constraint** (c) on GPU units available for external sale;
- **Time coupling** between booking and service windows ([0, T]) and ([T, 2T]);
- **Product differentiation** via reserved versus preemptable instances;
- **Uncertain, potentially nonstationary demand** for each product type as a function of price;
- **Operational feasibility constraints** reflecting the primacy of internal telco workloads.

1.2.1 Scope and assumptions

- **Cloudified compute**: RAN-adjacent AI workloads run on virtualized cloud infrastructure; GPU resources are fungible across instances.
- **Internal priority**: Internal AI workloads have priority during peak hours; only off-peak windows are commercialized for third-party use.
- **Capacity granularity**: External capacity is sold in discrete GPU units up to (c).
- **Booking–service separation**: Sales occur in ([0, T]); service is delivered in ([T, 2T]) with revenue attributed to ([0, T]).
- **Product catalogue**: Only two products—reserved and preemptable—powered by identical GPU hardware; preemptable instances may be interrupted for short periods.

- **Information**: The operator knows internal utilization profiles sufficiently to publish time windows and quantities for external sale.

1.2.2 Objectives and research questions

- **Primary objective**: Maximize expected revenue from idle GPU capacity while preserving internal network performance guarantees.
- **Research questions**:
    1. How should prices for reserved and preemptable instances be set over the booking horizon under demand uncertainty.
    2. How should limited GPU inventory be allocated across products and time windows to balance yield and risk.
    3. What is the revenue impact of pre-emption policies and internal demand variability on feasible external offerings.
    4. Which revenue management approach (e.g., dynamic pricing with inventory shadow values) achieves robust performance with tractable complexity for operators.

1.2.3 Expected contribution

- A formal revenue management formulation for RAN compute monetization with booking–service separation and product differentiation.
- A pricing and allocation framework that incorporates inventory constraints, demand uncertainty, and pre-emption risk.
- Quantitative evaluation via simulation under realistic utilization patterns, demonstrating trade-offs between reserved and preemptable mix, pricing levels, and capacity limits (c).

1.3 Proposed solution: primal–dual learning framework as the core approach

This report proposes adopting the **Primal–Dual Learning Algorithm for Personalized Dynamic Pricing with an Inventory Constraint**, introduced by Chen and Gallego (2018), as the central decision-making and optimisation engine for addressing the project's challenges. This algorithm directly targets the two most pressing issues in our context:

1. **Limited, high-value capacity** that must be allocated intelligently across offerings.
2. **Unknown and variable demand** patterns that need to be learned and acted upon quickly.

1.3.1 Why primal dual learning method is a strong fit

- **Strategic simplicity with computational efficiency**
  Instead of wrestling with the complexity of managing multiple customer segments and pricing strategies all at once, the primal–dual approach reframes the problem so that each segment's pricing decision can be optimised independently. The segments are coordinated through a single unifying factor — the estimated economic value of the constrained resource. This design keeps the computational requirements low, even when the number of product types is large.

- **Adaptive learning for uncertain environments**
  Demand behaviour for different customer types is not assumed to be known in advance. The algorithm methodically tests different prices, observes customer responses, and adjusts its estimates of both demand and the "shadow price" of capacity. This allows it to adapt in real time to emerging market conditions.

- **Scalable to multiple offerings**
  Whether there are two distinct product tiers or a dozen, the algorithm's performance does not deteriorate with added complexity. This scalability is essential for environments where the product mix may evolve or expand.

- **Proven performance and theoretical guarantees**
  Chen and Gallego demonstrated that their approach maintains low performance loss compared to an all-knowing "oracle" policy, and critically, that this loss does not grow with the number of customer types. That means reliable results without runaway complexity.

1.3.2 How primal dual learning works in practice

The method unfolds in two clear stages:

1. **Exploration** – Systematically vary prices for each customer type to collect demand data. This is a structured process, not random trial and error, ensuring that the right information is gathered quickly.
2. **Exploitation** – With the data in hand, focus on prices that maximise revenue, while staying within the capacity limit. Adjustments are made if the inventory runs low faster than expected, helping avoid stock-outs or suboptimal allocation.

1.3.3 Relevance to the project scenario

In our case — managing and monetising a finite GPU pool across multiple offerings — the parallels are exact:

- The **capacity limit** is the fixed number of GPUs available for sale at any given time.
- The **multiple customer types** are the different instance types or market segments with distinct demand curves.

- The **unknown demand** is the varying willingness to pay for GPU time across these segments.
- The **unifying variable** is the estimated marginal value of the GPUs, which guides pricing across all segments at once.

By using Chen & Gallego's primal–dual learning framework, the project gains a proven, academically grounded method that can guide decisions from day one, adapt to real-world demand behaviour, and ensure every GPU hour is sold to the right customer at the right price.

1.4 Project Objectives

This project is motivated by the need to develop a scalable and adaptive pricing strategy for monetizing excess GPU capacity in a resource-constrained telecommunications environment. Drawing on the primal–dual learning framework proposed by Chen and Gallego (2018), the project sets out to achieve the following core objectives:

- **Implement the Algorithm**: Develop a working version of the primal–dual learning algorithm in Python, using a two-phase structure (exploration and exploitation) to estimate demand and guide pricing under inventory constraints.

- **Test with Simulated Data:** Use multi-type customer simulations to assess adaptability, personalization, and dynamic pricing under realistic conditions.

- **Ensure Robustness:** Validate the algorithm's ability to handle noisy, non-stationary demand and tight inventory scenarios.

- **Analyze Dual Variable:** Study the behavior and economic meaning of the dual variable as a signal for inventory value and pricing coordination.

- **Minimize Regret:** Evaluate the algorithm's performance by comparing actual revenue to oracle benchmarks, aiming for low regret across customer types.

- **Assess Scalability:** Examine computational efficiency as problem size grows, and identify areas for optimization.

- **Explore Real-World Use:** Outline steps for integrating the algorithm into practical pricing systems, including data and deployment considerations.

2. Literature Review

Dynamic pricing, also referred to as revenue management pricing, is a strategic approach in which prices are adjusted over time in response to changes in demand, supply, and market conditions. Originating in the airline and hospitality industries (Talluri & van Ryzin, 2004), dynamic pricing has since been widely adopted across sectors such as e-commerce, ride-hailing, and telecommunications, where heterogeneous customer preferences and uncertain demand present significant challenges. Traditional models often assume prior knowledge of demand functions, enabling optimal prices to be computed directly. However, in many real-world settings, demand is initially unknown and must be learned from observed data while simultaneously managing operational constraints such as limited inventory.

A key challenge in such problems is the **exploration–exploitation trade-off**, where the decision-maker must balance learning about demand (exploration) with applying current knowledge to maximize revenue (exploitation). Classical approaches to this problem include stochastic approximation methods, multi-armed bandit algorithms (Bubeck & Cesa-Bianchi, 2012), and reinforcement learning-based strategies (Sutton & Barto, 2018). While these methods have achieved success in unconstrained or low-dimensional settings, their performance can degrade significantly when the number of customer types or pricing dimensions increases—a phenomenon often referred to as the "curse of dimensionality."

The **Primal-Dual Learning Algorithm** proposed by Chen and Gallego (2018) addresses this scalability issue by leveraging the **dual formulation** of the constrained optimization problem. The algorithm estimates the **dual optimal variable**, also known as the shadow price of inventory, which represents the marginal value of an additional unit of stock. Once this dual variable is learned, the complex joint optimization problem over multiple customer types can be decomposed into independent single-type pricing subproblems. This decomposition not only reduces computational complexity but also ensures that the regret bound is independent of the number of customer types, a significant improvement over prior methods where regret typically scales with problem dimensionality.

In the primal formulation, the objective is to maximize expected revenue subject to inventory constraints, while the dual formulation introduces a Lagrange multiplier for the inventory constraint. The Chen & Gallego approach integrates learning into this framework by first running an **exploration phase** with discretized prices to estimate demand functions and the shadow price, followed by an **exploitation phase** where prices are refined based on these estimates. Their analysis demonstrates that the algorithm achieves a regret rate of O(T^{-1/2}) in the scaling regime, independent of the number of customer types, where T is the time horizon.

Subsequent studies have extended primal-dual and bandit-based methods to various domains, including online advertising (Balseiro et al., 2015), cloud computing resource allocation (Wang et al., 2020), and large-scale retail pricing (Ferreira et al., 2018). These works reinforce the value of dual-based approaches in managing resource-constrained optimization problems under uncertainty. Furthermore, machine learning models such as logistic regression, random forests, and gradient boosting have been incorporated into demand estimation, enabling more flexible representations of customer behavior beyond parametric demand functions.

This project builds directly on the Chen & Gallego framework, implementing the full primal-dual learning algorithm for a multi-type customer setting with simulated demand. The implementation incorporates both logistic demand models and machine learning estimators, allowing a comparison between theoretical and data-driven demand estimation techniques. By integrating these approaches, the work demonstrates the adaptability of the primal-dual learning methodology to practical scenarios where demand patterns are complex, dynamic, and initially unknown.

3. Methodology

3.1 Problem formulation:

3.1.1 Primal Problem

The primal problem focuses on helping the provider maximize revenue from selling GPU-powered compute instances over a fixed booking horizon.

Objective

- Maximize expected revenue by offering prices that balance profitability and customer acceptance.
- Expected revenue is calculated as:
  Price × Probability of Purchase, summed across all time steps.

Inventory Constraint

- GPU inventory is limited and fixed for the entire horizon.
- This creates a trade-off:
  - Higher prices may reduce demand but increase per-unit revenue.
  - Lower prices may increase demand but risk exhausting inventory prematurely.

Broader Relevance

This formulation is foundational in revenue management and operations research, especially in environments with:

- Scarce resources
- Uncertain customer behavior
- Real-time decision requirements

Primal problem formulation:

Primal Problem:

Objective function: **maximize revenue**

Normally, revenue = price x units sold

Revenue in implementation = price x demand x total customer approached

$$\max_{\{p_t\}} \sum_{t=1}^{T} \mathbb{E}[p_t \cdot D(x_t, p_t)]$$

Constraint: **Inventory**, total demand should not exceed inventory.

$$\sum_{t=1}^{T} \mathbb{E}[D(x_t, p_t)] \leq c$$

Here,

p_t: price set at time t

x_t: type of customer at time t (could be Reserved or Preemptive)

$D(x_t, p_t)$: probability that a type-x_t customer buys at price p_t

c: total inventory (e.g., 100 GPUs)

3.2.2 Lagrangian formulation for primal problem relaxation

The Lagrangian is a key technique in constrained optimization, used to reformulate problems that involve maximizing or minimizing an objective function subject to

constraints. Instead of solving the constrained problem directly, the method embeds the constraints into the objective function using **dual variables** (also known as Lagrange multipliers), which act as penalties for constraint violations.
This approach transforms the original problem into a form that can be addressed using unconstrained optimization techniques. The solution involves finding a **saddle point**—minimizing the Lagrangian with respect to the decision variables and maximizing it with respect to the dual variables.

Application in Inventory-Constrained Revenue Maximization
In the context of this study, the primal problem involves maximizing expected revenue from selling GPU-powered compute instances, subject to a fixed inventory constraint.

The Lagrangian incorporates this constraint into the revenue objective, with the **dual variable representing the shadow price of inventory**—the marginal value of an additional GPU unit.

- When inventory is scarce, the dual variable is high, discouraging low-price sales.
- When inventory is abundant, the penalty is low, allowing more aggressive pricing.

This formulation enables the use of **primal–dual algorithms**, which iteratively update both pricing decisions and inventory valuation. At each step, the algorithm:

- Selects a price that maximizes the Lagrangian given the current dual estimate.
- Adjusts the dual variable based on observed demand and remaining inventory.

This dynamic process allows the algorithm to learn both demand behavior and optimal pricing strategies over time, without requiring full prior knowledge of customer preferences.

Lagrangian Formulation example:

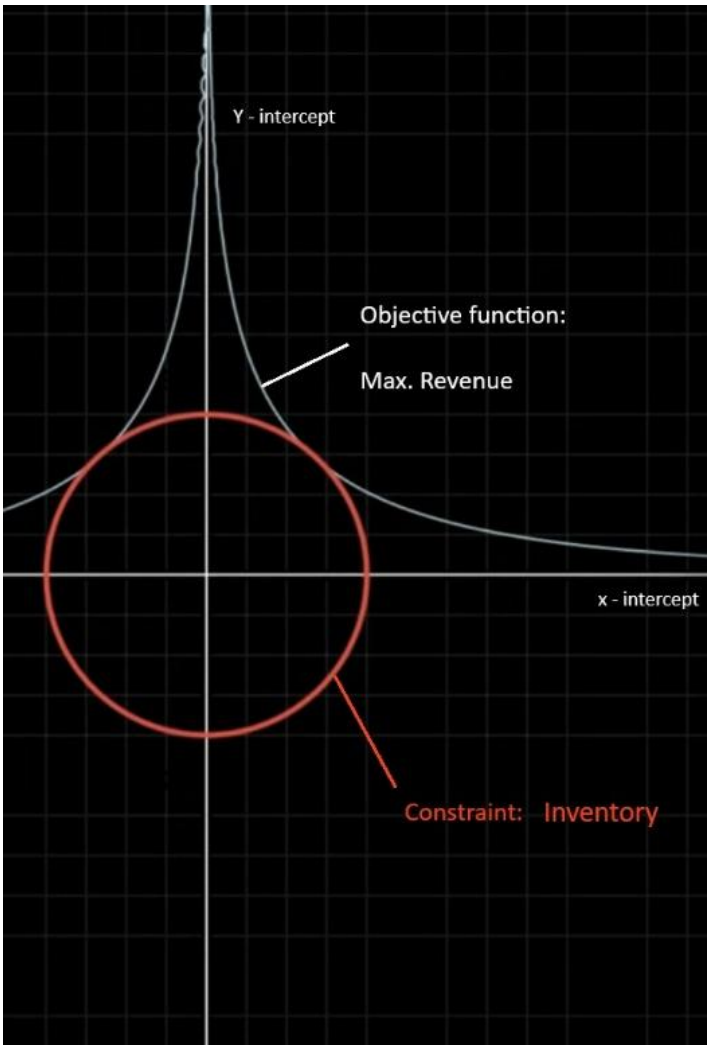


Figure .01: Representation of lagrangian formulation, hyperbola represents objective function (maximize revenue) and circle represents constraint (limited inventory).

We are given an objective function (white colour Hyperbolic Curve in above figure – **maximize revenue**):

$$R(x, y) = x^2 e^y$$

subject to the constraint (red circle in above figure – **limited inventory**):

$$B(x, y) = x^2 + y^2 = b$$

The necessary condition for optimality is that the gradients of the objective and constraint functions are proportional:

$$\nabla R = \lambda \nabla B$$

This condition ensures that the gradients (directions of steepest ascent) of the revenue and constraint functions are aligned—meaning, at a point of optimal trade-off.

To solve this constrained optimization problem using the Lagrangian method, we define the Lagrangian function:

$$L(x, y, \lambda) = R(x, y) - \lambda(B(x, y) - b)$$

Here,

λ: The Lagrange multiplier, representing how much the objective function would improve if the constraint were relaxed.

From Primal to Lagrangian Formulation:

The transition from the primal problem to the Lagrangian formulation involves the following conceptual steps:

1. **Start with the primal problem**: Maximize expected revenue subject to an inventory constraint.
2. **Introduce a dual variable**: This variable represents the cost of violating the inventory constraint.
3. **Construct the Lagrangian**: Combine the revenue objective and the constraint, weighted by the dual variable.
4. **Optimize the Lagrangian**: Use iterative methods to find the best pricing decisions and update the dual variable based on feedback.

$$\mathcal{L}(\{p_t\}, z) = \sum_{t=1}^{T} \mathbb{E}[(p_t - z) \cdot D(x_t, p_t)] + z \cdot c$$

p_t: price set at time t

x_t: type of customer at time t (could be Reserved or Preemptive)

$D(x_t, p_t)$: probability that a type-x_t customer buys at price p_t

c: total inventory (e.g., 100 GPUs)

z: unit value of inventory.

This approach transforms a constrained optimization problem into a structure that is more amenable to learning and adaptation. It also provides interpretability: the dual variable gives insight into how valuable the inventory is at any point in time, guiding pricing decisions in a principled way.

3.2.3 Dual Problem Formulation

The dual problem offers a complementary view of constrained optimization by introducing a dual variable (z), representing the marginal value of inventory. While the primal problem maximizes revenue under a capacity constraint, the dual embeds this constraint into the objective via the Lagrangian.

Motivation

Directly solving the primal is complex under uncertain or heterogeneous demand. The dual formulation simplifies this by:

1. Maximizing the Lagrangian over prices for each customer type and time.
2. Minimizing over (z >= 0), capturing inventory valuation.

This structure enables simultaneous learning of demand patterns and inventory value.

Dual Objective Function

The dual problem is defined by the function:

$$g(z) = \sum_{t=1}^{T} \max_{p \in \mathcal{P}} (p - z) \cdot D(x_t, p) + z \cdot c$$

p_t: price set at time t

x_t: type of customer at time t (could be Reserved or Preemptive)

$D(x_t, p_t)$: probability that a type-x_t customer buys at price p_t

c: total inventory (e.g., 100 GPUs)

z: unit value of inventory.

This function captures the trade-off between pricing decisions and inventory valuation. The first term computes the maximum expected profit at each time step, adjusted by the dual variable. The second term adds the total value of the inventory, scaled by (z).

Practical Role in Implementation

In the Jupyter Notebook implementation, this dual formulation is central to the algorithm's logic.

- Compute the optimal price for each customer type by maximizing the **adjusted revenue** term.
- Track and update the dual variable (z) based on observed demand and inventory usage.
- Use (z) to guide future pricing decisions, ensuring that inventory is allocated efficiently across the booking horizon.

This dual perspective not only simplifies the optimization but also provides interpretability: the dual variable acts as a dynamic signal of resource scarcity, shaping pricing behavior in real time.

3.2.4 Dual variable and optimal pricing:

Dual Variable (z*):

The optimal dual variable (z*) is the value of (z) that minimizes the dual objective function. It represents the **true marginal value of inventory** under the current demand environment and pricing structure. Once computed, (z*) becomes the central coordinating signal for pricing decisions across all customer types and time periods.

Mathematical Definition:

$$z^* = \arg\min_{z \geq 0} \left\{ \sum_{t=1}^{T} \max_{p \in \mathcal{P}_{x_t}} (p - z) \cdot \hat{D}_{x_t}(p) + z \cdot c \right\}$$

p_t: price set at time t

x_t: type of customer at time t (could be Reserved or Preemptive)

$D(x_t, p_t)$: probability that a type-x_t customer buys at price p_t

c: total inventory (e.g., 100 GPUs)

z: unit value of inventory.

- **Revenue generation** from each customer, adjusted by the penalty (z).
- **Inventory valuation**, represented by the term (z*c).

The optimal (z*) is the point where this trade-off is minimized — where the pricing strategy yields the highest adjusted revenue without over-consuming inventory.

Optimal pricing (p*) via adjusted revenue:

Once (z*) is computed, it guides the pricing strategy as follows:

- For each arriving customer, the algorithm selects the price (p) that maximizes adjusted revenue.

- This ensures that prices are chosen not just for immediate profit, but also in consideration of inventory scarcity.
- As demand patterns evolve, (z*) is periodically re-estimated to reflect updated beliefs about customer behavior and remaining inventory.

Mathematical Definition:

$$p^*(x) = \arg\max_{p \in \mathcal{P}} (p - z^*) \cdot \hat{D}_x(p)$$

p(x)*: Optimal price for customer type x

p: Price from the available price list

z*: Dual variable, representing the optimal unit value of inventory

$\hat{D}_x(p)$: Estimated demand at price p for customer type x

x = customer type

Dynamic Behavior

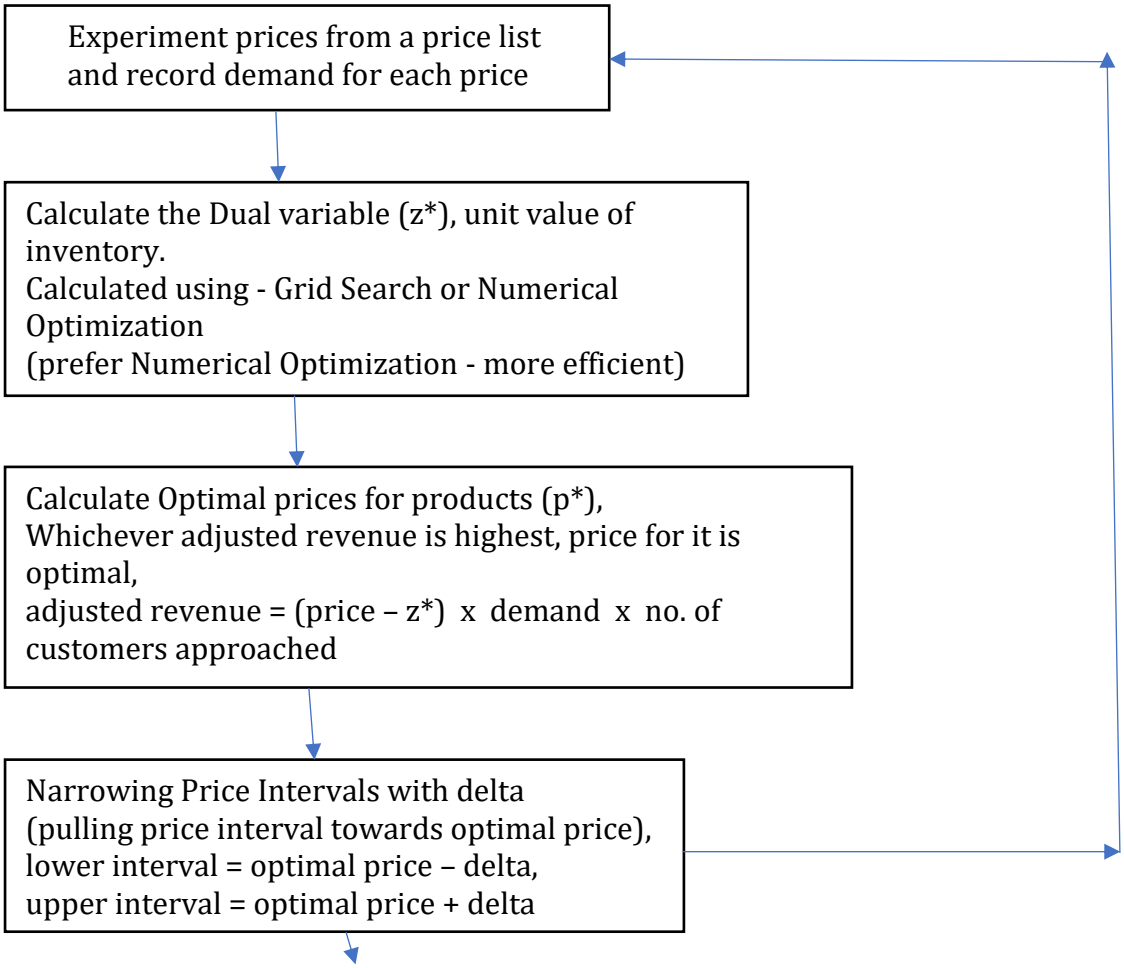
- Early in the booking horizon, when inventory is plentiful, (z*) tends to be low. This encourages exploration and demand learning.
- Later, as inventory depletes, (z*) rises, shifting the algorithm toward exploitation and more conservative pricing.

This dynamic adjustment makes (z*) a powerful control signal — it adapts to both temporal constraints and demand uncertainty, enabling robust revenue management.

3.2 Primal dual learning algorithm:

The **Primal–Dual Learning Algorithm**, as introduced by Chen and Gallego (2018), provides a robust framework for dynamic pricing under inventory constraints and demand uncertainty. Its relevance to the monetization of idle GPU capacity in telco RAN environments stems from its ability to simultaneously learn demand behavior and optimize pricing decisions in real time, while respecting resource limitations and operational priorities.

This section presents a detailed exposition of the algorithm's two-phase structure—**Phase I: Exploration** and **Phase II: Exploitation**—with emphasis on its mechanics, decision logic, and application to differentiated product offerings (reserved and preemptable GPU instances).



After all, subphases of phase 1 are finished
Return optimal price, other values for exploitation (Dual variable, remaining booking period and inventory)

Flowchart 1: Phase 1 of algorithm (Exploration).

3.2.1 Phase I: Exploration – Demand Learning and Shadow Price Estimation

This phase focuses on understanding how customers respond to different prices and estimating the value of inventory.

Step 1: Price Experimentation

For each customer type (reserved and preemptable), the operator selects a different prices within a defined interval. These prices are offered to customers over multiple booking instances.

Step 2: Demand Observation

At each price, the operator records how many customers were offered the product and how many accepted. This helps estimate the demand function — essentially, how demand varies with price.

Step 3: Adjusted Revenue Calculation

For each price, the operator calculates adjusted revenue. This is the actual revenue minus the cost of consuming inventory, which is estimated using a variable called "dual variable." The price that yields the highest adjusted revenue is considered the best price for that subphase.

Step 4: Interval Refinement

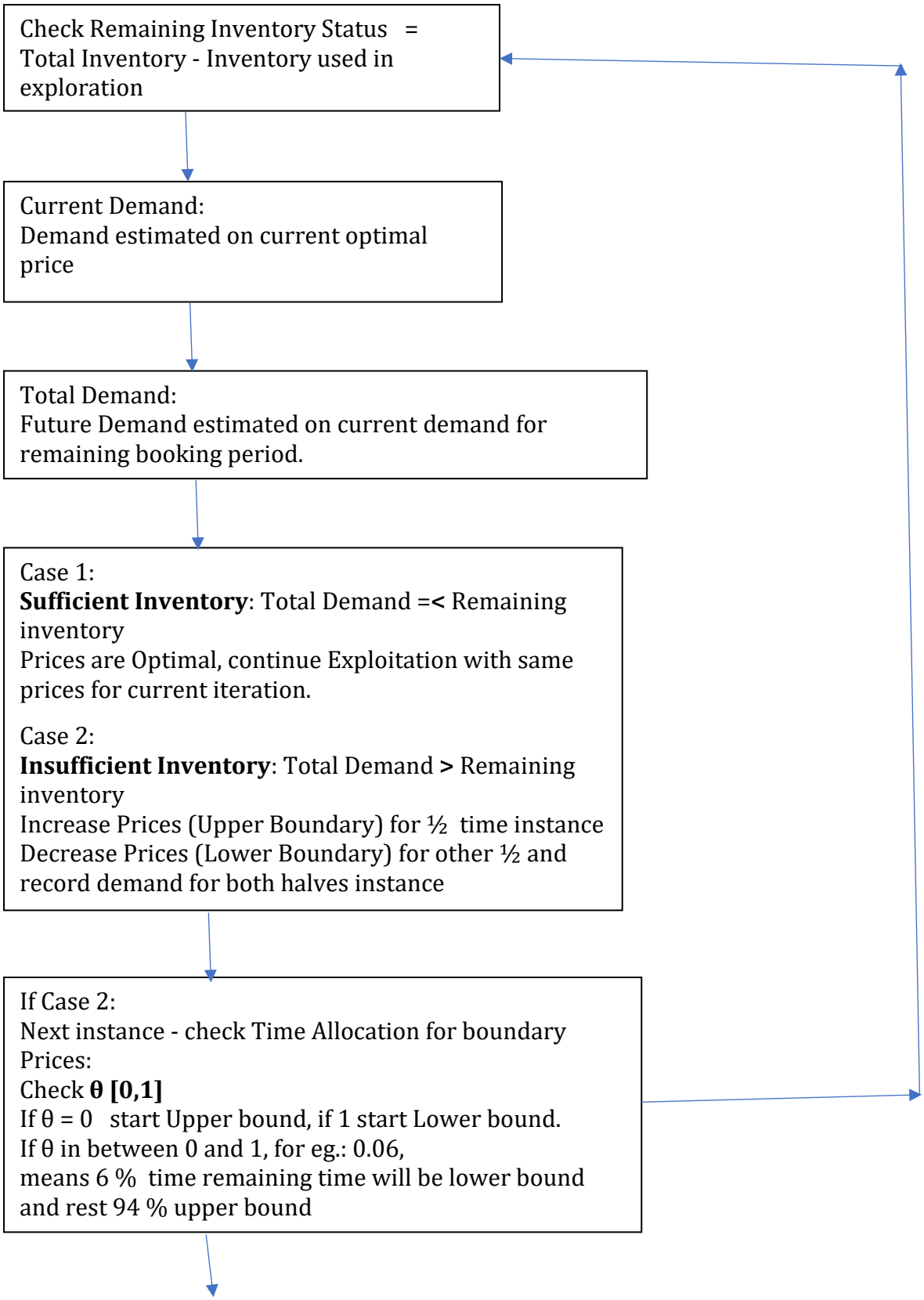
Once the best price is identified, the operator defines a narrower interval around it. For example, if the best price is p*, the new interval is from p* - delta to p* + delta. This refined interval is used in the next subphase to test prices more precisely.

Step 5: Iteration Across Subphases

This process — testing prices, observing demand, calculating dual variable and optimal prices via adjusted revenue, and refining the interval, is repeated till all subphases in phase 1 are completed. Each iteration improves the valuation of inventory and optimal price.

Step 6: Transition to Exploitation

At the end of exploration, the operator passes the following to the next phase:


Check Remaining Inventory Status = Total Inventory - Inventory used in exploration
Current Demand: Demand estimated on current optimal price
Total Demand: Future Demand estimated on current demand for remaining booking period.
Case 1: Sufficient Inventory: Total Demand =< Remaining inventory Prices are Optimal, continue Exploitation with same prices for current iteration.
Case 2: Insufficient Inventory: Total Demand > Remaining inventory Increase Prices (Upper Boundary) for ½ time instance Decrease Prices (Lower Boundary) for other ½ and record demand for both halves instance
If Case 2: Next instance - check Time Allocation for boundary Prices: Check θ [0,1] If θ = 0 start Upper bound, if 1 start Lower bound. If θ in between 0 and 1, for eg.: 0.06, means 6 % time remaining time will be lower bound and rest 94 % upper bound

When Inventory or Booking Period Finish,
Start Calculating Regret - (Performance of Algorithm).

Flowchart 2: Phase 2 of algorithm (Exploitation).

3.2.2 Phase II: Exploitation — Adaptive Pricing and Inventory Management

This phase uses the insights from exploration to make real-time pricing decisions that balance revenue and inventory.

Step 1: Demand Estimation

Using the optimal price from exploration (p*), the operator estimates current demand. This is multiplied by the number of days left in booking period to project total future demand.

Step 2: Inventory Comparison

The projected demand is compared to the remaining inventory. If inventory is sufficient, the operator continues offering the product at p* (Case 1: Sufficient Inventory). If not, prices are adjusted to slow down sales.

Step 3: Price Adjustment

To adjust pricing, the operator defines a new interval around p*. For example:

- p_upper = p* + delta
- p_lower = p* - delta

In the next booking instance, the algorithm offers p_upper for the first half and p_lower for the second half.

Step 4: Test new prices (upper and lower bound on 1 instance):

Here, the algorithm experiments the upper and lower bound prices for 1 instance total, meaning half instance is for upper bound and other half is for lower bound. Again, the demand is recorded for both bounds of prices. This demand is used in next step for deciding which bound of price is more important.

Step 6: Time Allocation

Based on demand recorded in last step for halves of instances, first to upper bound of price and second to lower, the algorithm calculates theta, which tells us what percent of next two instances combine time should be allocated to upper and lower bound. If theta

*Note - *In our implementation, time allocation is performed for the subsequent two instances only. In contrast, the algorithm presented in the referenced paper allocates time across the entire remaining booking period. Our approach involves allocating time to the next two instances, updating the optimal prices accordingly, and then proceeding to the next iteration. This iterative mechanism ensures safety, as the algorithm evaluates at the beginning of each iteration whether a price adjustment is warranted solely due to the transition into a new iteration.*

Step 7: Iterative Refinement

This cycle — projecting demand, comparing inventory, adjusting prices, and reallocating time, continues until the booking horizon ends or inventory is depleted.

After completion of all phases, there is regret calculation which tells us about the different between actual revenue generated and optimal revenue. According to paper, if the regret is between 0 to 15%, it's assumed as a good regret.

4. Implementation:

4.1 Code Appendix or Variable description:

Main loop 1 and User inputs - Code Appendix or Variable description:

c = Current total inventory remaining.
c0 = Initial inventory, before or at start of booking period
customer_data = List to store customer data
customer_id = 1 = Serial number given to each customer as they approach irrespective of bought or not. This is new even if a customer approaches more than one time.
customers_per_price = number of customers approaching per price
demand_history = list to store demand values with respect to price for plotting price vs demand for Phase 1.
delta = (δ) parameter used in exploration for narrowing price intervals near optimal price.
df = data frame for demand generated in 1 subphase in Exploration (Phase 1).
df_1 = data frame for complete Exploration (Phase 1).
df_1 = dataframe to store Phase 1 demand simulation data
df_1_type_1 = dataframe to store Phase 1 demand simulation data only for customer type 1.
df_1_type_2 = dataframe to store Phase 1 demand simulation data only for customer type 2.
K = Total subphases in phase 1 (K x t = P1)
ki = 1 Subphase in Phase 1 or 1 element in K range
num_steps = Total number of Price points wanted
p = Current price list for type 1 (Reserve Customers), initially created after inputs of Highest and Lowest price taken form user.
p_1_initial = Initial price list for customer type 1, this does not change.
p_2_initial = Initial price list for customer type 2, this does not change.
p2 = Current price list for type 2 (Preemptive Customers), initially created after inputs of Highest and Lowest price taken form user.

p_low = lower price limit when we narrow price intervals towards optimal price.
p_opt_1 = p*, optimal price for 1st type (Reserve Customers) in Exploration.*
p_opt_2 *= p*, optimal price for 2nd type (Preemptive Customers) in Exploration.
p_opt_type1 = list for collecting optimal prices of type 1 customer in phase 1
p_opt_type2 = list for collecting optimal prices of type 2 customer in phase 1
p_upp = upper price limit when we narrow price intervals towards optimal price.
pt = price for a particular instance, pt is in range p.
P1 = total booking time of phase 1.
P2 = total percbooking time of phase 2, P1+P2 = T, a phase can have many subphases.
T = current total booking tim remaining
T0 = Initial booking peroid
t = time instances to divide booking time.
total_customers = 2 x customers_per_price x num_steps (total price points), total customers approaching for a subphase ki in Phase 1. 2 is for 2 customer types.
unit_T = unit of booking peroid (Usually days)
xt = Unit of Booking time T one instance has.
z = dual variable at beginning, can be zero at 1st instance.
z_opt = z*, optimal dual variable for both types (Reserved and Preemptive) customers.

Exploration function (Phase 1) - Code Appendix or Variables description:

customer_data = List to store demand simulation data as customer approaches
customer_id = an serial number order number given to all customers starting from 1 irrespective of bought or not.
demand_probability_1 = demand probability calculation for customer type 1 after demand simulation and recording.
demand_probability_2 = demand probability calculation for customer type 2 after demand simulation and recording.
demand_history = list to store demand values with respect to price for plotting price vs demand for Phase 1.

df = dataframe inside exploration funciton consisting data for demand simulation for that iterating subphase in phase 1 (exploration)
new_price_list_1 = New price list for customer type 1 found after narrowing price intervals via delta
new_price_list_2 = New price list for customer type 2 found after narrowing price intervals via delta
p_low_1 = new lower price interval found after narrowing with respect to p_opt_1
p_low_2 = new lower price interval found after narrowing with respect to p_opt_2
p_opt_1 = Optimal price for customer type 1
p_opt_2 = Optimal price for customer type 2
p_upp_1 = new upper price interval found after narrowing with respect to p_opt_1
p_upp_2 = new upper price interval found after narrowing with respect to p_opt_2
revenue_per_price_1 = revenue per price for customer type 1
revenue_per_price_2 = revenue per price for customer type 2
total_revenue = revenue for both prices. here, revenue is normal revenue i.e. price x units sold.
total_revenue_1 = total revenue for customer type 1, here, revenue is normal revenue i.e. price x units sold.
total_revenue_2 = total revenue for customer type 2, here, revenue is normal revenue i.e. price x units sold.
units_sold = inventory sold in one subphase of phase one.
z_opt = Combined dual variable value for both customers types
z_opt_1 = Dual variable, Optimal unit value of inventory obtained from demand data of customer type 1
z_opt_2 = Dual variable, Optimal unit value of inventory obtained from demand data of customer type 2

Main loop 2 - Code Appendix or Variable description:

df_2 = dataframe to store Phase 2 demand simulation data.
df_2_type_1 = dataframe to store Phase 2 demand simulation data for customer type 1.
df_2_type_2 = dataframe to store Phase 2 demand simulation data for customer type 2.

safe_T = Safe booking time limit, T > safe_T ensures booking peroid does not become negative while conducting exploitation's last iteration
safe_c = Safe Inventory, to avoid inventory going (-ve) negative
ti = iteration variale which will be used to see what iteration we are in while doing exploitation phase.

Exploitation function (Phase 2) - Code Appendix or Variables description:

alpha_1 = Alpha (bound length for a optimal price) for customer type 1
alpha_2 = Alpha (bound length for a optimal price) for customer type 2
D_lower_1 = demand probability calculated from df_temp_lower for customer type 1
D_lower_2 = demand probability calculated from df_temp_lower for customer type 2
D_upper_1 = demand probability calculated from df_temp_upper for customer type 1
D_upper_2 = demand probability calculated from df_temp_upper for customer type 2
demand_estimate_1 = Estimated demand for optimal price for customer type 1 via *llightgbm_buying_probability* function.
demand_estimate_2 = Estimated demand for optimal price for customer type 2 via *llightgbm_buying_probability* function.
df_exploitation = data frame to store demand simulation for complete 1 iteration of exploitation.
df_temp_lower = = temporary data frame in exploitation function for storing demand simulation for half instance of lower bound on both customer types.
df_temp_upper = temporary data frame in exploitation function for storing demand simulation for half instance of upper bound on both customer types.
p_opt_1_high = higher bound for Optimal price for customer type 1, p_opt_1 + alpha_1
p_opt_1_low = Lower bound for Optimal price for customer type 1, p_opt_1 - alpha_1
p_opt_2_high = higher bound for Optimal price for customer type 2, p_opt_2 + alpha_2
p_opt_2_low = Lower bound for Optimal price for customer type 2, p_opt_2 - alpha_2
theta_1 = Time allocation metric for price bounds for customer type 1.
theta_1_lower_time = time allocated to lower bound of price for customer type 1

theta_1_upper_time = time allocated to upper bound of price for customer type 1
theta_2 = = Time allocation metric for price bounds for customer type 2.
theta_2_lower_time = time allocated to lower bound of price for customer type 2
theta_2_upper_time = time allocated to upper bound of price for customer type 2
total_estimated_demand = total of demand_estimate_1 and demand_estimate_2
total_future_demand = Total future demand estimated based on current demand according to current optimal prices.

* Preemptive means Preemptable but since it is used from beginning and is throught the code, it is kept as Preemptive.

4.2 Assumptions:

Customer Arrival: Customers are assumed to arrive continuously across all instances.

Price Duration: Each instance is associated with a single posted price, which remains valid for the full duration of that instance (e.g., if an instance spans two days, the price applies throughout).

Time Simulation: The implementation does not simulate real-time progression. Instead, time is abstracted through a fixed number of customer arrivals per price (customers_per_price). Partial instances are modeled by proportionally adjusting the number of customers rather than actual time.

Customer Sensitivity: Reserved customers exhibit lower price sensitivity, while preemptable customers are more responsive to price changes.

4.3 Code structure, implementation flow and functions in detail:

The algorithm is structured around two primary loops: one for the **exploration phase** and one for the **exploitation phase**. The first loop executes the exploration function across subphases until Phase 1 is complete. The second loop initiates the exploitation function, which runs Phase 2 over the remaining booking horizon. Within both phases, supporting functions are invoked to perform specific tasks relevant to each phase. The detailed structure of the implementation is outlined below.

Note:

Serial numbering in the implementation section corresponds directly to the Jupyter Notebook for ease of cross-referencing. While the numbering may appear non-sequential or inconsistent in implementation section especially in Code structure in the report, it has been intentionally preserved to align with the notebook structure. This approach facilitates straightforward comparison between the report and the code, enhancing clarity for the reader.

0. Importing Libraries:

All necessary libraries required for all functions are called here.

Code 01: Libraries used in implementation

```
# importing libraries
import numpy as np
import pandas as pd
from scipy.optimize import minimize_scalar
from scipy.optimize import minimize
import math
import matplotlib.pyplot as plt
import lightgbm as lgb
from sklearn.model_selection import train_test_split
```

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split, GridSearchCV
```

Each library's purpose:

- **numpy**: Numerical operations and array handling
- **pandas**: Data manipulation and tabular structures
- **scipy.optimize**: Numerical optimization (for dual variable estimation)
- **math**: Basic mathematical functions
- **matplotlib.pyplot**: Plotting and visualization
- **lightgbm**: Gradient boosting for demand prediction
- **sklearn.model_selection**: Data splitting and hyperparameter tuning
- **sklearn.ensemble**: Random Forest model for demand estimation
- **sklearn.metrics**: Model evaluation (e.g., accuracy)

1. Phase 1 - Exploration:

Consists of all functions for phase 1 and also the main loop 1.

1.1 Demand Function:

Returns base probability of purchase at a given price among the price list.

Demand Generation

Demand is modeled using a logistic acceptance function to reflect realistic market behavior: lower prices increase purchase probability, outcomes are stochastic, and customer types respond differently. The curve is controlled by three parameters—maximum probability, slope (price sensitivity), and midpoint (typical willingness-to-pay). Reserved and preemptable customers are differentiated by these parameters to simulate heterogeneity.

Code 02: Logistic demand function

```
def logistic_demand_reserved (pt, p, p0=0,L=1, k=0.001 ):
        p0= (p.min()+p.max())/2
        return L / (1 + np.exp(k * (pt - p0)))
```

here:
L = max demand, (here, 1)
k = steepness,
pt = a price in price interval
p0 = average price
p.min() = minimum price in price list
p.max() = maximum price in price list

This function is built according to **logistic function:**

$$d(p) = \frac{L}{1 + e^{-k(p-p_0)}}$$

here:
D (p) = demand probability, L = max demand, (here, 1), k = steepness, pt = a price in price interval, p0 = average price

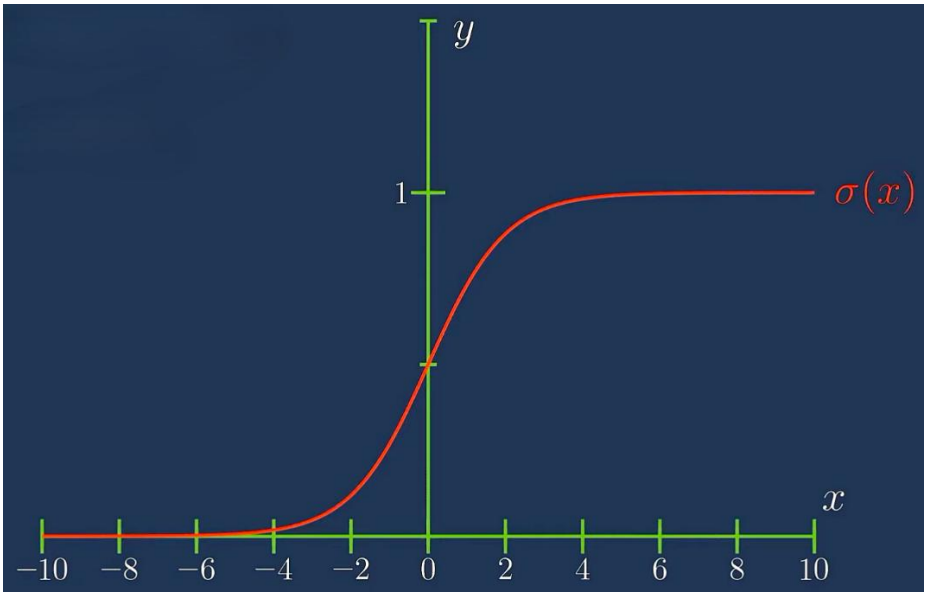


Figure 2: logistic S shaped curve, for representation of logistic function.

Logistic Function:

The logistic function is a smooth, S-shaped curve that maps price to purchase probability, bounded between 0 and 1. It captures key demand characteristics: high likelihood of purchase at low prices, diminishing sensitivity at extremes, and steep variation near a central midpoint. The slope parameter governs price sensitivity, enabling flexible modeling of customer behavior. Its interpretability and calibration ease make it well-suited for simulating realistic, price-biased demand.

1.2 Demand Probability function:

Code 03: Demand probability function

```
def compute_demand_probability(df, p):

    # List to store demand for different prices
    demand_probability = {}

    for pt in p:
        # Use tolerance to avoid floating point mismatch
        subset = df[np.isclose(df['price'], pt, atol=0.01)]
        if not subset.empty:
            demand_prob = subset['buy'].sum() / subset['buy'].count()
            demand_probability[pt] = demand_prob
        else:
            demand_probability[pt] = 0.0  # fallback if no data at this price

    # Print demand probabilities for debugging
    for pt in p:
        print(f"Price: {pt:.2f}, Demand Probability: {demand_probability[pt]:.2f}")

    return demand_probability
```

The demand probability function computes the likelihood of purchase based on price, representing simulated demand derived from the demand simulation dataframe. It takes the list of prices explored in the simulation function and, for each price, calculates the ratio of buyers to individuals approached, yielding the estimated demand at each price point.

1.3 Revenue calculation function:

Code 04: Revenue calculation function

```
def calculate_revenue(df):

    # Calculate revenue for each transaction
    df['revenue'] = df['price'] * df['buy']

    # Calculate total revenue
    total_revenue = df['revenue'].sum()

    # Calculate revenue per price point
    revenue_per_price = df.groupby('price')['revenue'].sum().to_dict()

    return total_revenue, revenue_per_price
```

The revenue calculation function computes standard revenue as the product of price and units sold. It uses the price list from the demand simulation dataframe (*df*) generated during the exploration subphase and calculates revenue for each price point.

1.4 Dual Variable: '*z_opt*' calculation function:

Code 05: Dual variable (z_opt) calculation function

```
"""For bounds from 0 to maximum price"""
""" Minimize_scalar used to optimize"""

def dual_objective(z, demand_probability, p, c):

    magnitude_p_opt = len(str(int(abs(min(p)))))
    magnitude_order_p_opt=(10**(magnitude_p_opt-1))

    def objective(z):
        return max((pt - z) * demand_probability[pt] for pt in p) + z * c
    z_opt = minimize_scalar(objective, bounds=(0, max(p)), method='bounded')
    return z+((z_opt.x) *(10**5)*(magnitude_order_p_opt))
```

Dual variable (z* in paper), **z_opt** in code:

$$z^* = \arg\min_{z \geq 0} \left\{ \sum_{t=1}^{T} \max_{p \in \mathcal{P}_{x_t}} (p - z) \cdot \hat{D}_{x_t}(p) + z \cdot c \right\}$$

- p is the price offered at time t.
- x_t is the customer type at time t.
- c is the total inventory capacity.
- z is the dual variable.
- $z*$ is the optimal value of the dual variable.
- D is the demand for particular customer type at a price

In constrained optimization, the dual variable z represents the marginal value of GPU inventory—crucial for pricing and allocation under uncertainty. It arises from the Lagrangian formulation, balancing immediate revenue against long-term resource preservation. Low z implies inventory abundance and supports aggressive pricing; high z signals scarcity, encouraging conservation.

1.4.1 Scaling the Dual Variable (*z_opt*) for Interpretability:

In implementation, the dual variable **z_opt** is scaled by the order of magnitude of the optimal price to improve interpretability. Initially, z_opt tends to be numerically very small and remains nearly constant across different price scales. This means that whether the price is 100, 1000, or 10000, the value of z_opt does not change significantly, making it difficult to draw meaningful insights from its raw value. To resolve this, z_opt is first normalized by multiplying it with a suitable power of ten to bring it into a more interpretable range. This normalized value is then multiplied by the order of magnitude of the price, resulting in a scaled dual variable that better reflects the pricing context.

Example:

Consider a price of 4000. The order of magnitude for this price is 1000. Suppose the actual value of **z_opt** is 0.000005.

First, we normalize it by multiplying with (10^5):

z_opt × 10^5 = 0.000005 × 10^5 = 0.5

Next, we scale this normalized value by the price order magnitude:

0.5 × 1000 = 500

This final value of 500 is more interpretable and aligns with the pricing context—especially when the minimum price in the list is 1000 (e.g., price list = 1000, 2000, 3000, 4000, 5000).

Scaling in Implementation in Jupyter notebook:

Magnitude Order Calculation:

The scaling factor is derived from the order of magnitude of the optimal price

$$\text{Magnitude Order}_z = 10^{(\text{digits in } |p_{\text{opt}}| - 1)}$$

Dual Variable Optimization:

$$z_{\text{opt}} = \underset{z}{\text{argmin}} \left[\max_{p_t \in P} \left((p_t - z) \cdot \text{DemandProb}(p_t) \right) + z \cdot c \right]$$

Final Scaled Dual Variable:

$$z_{\text{opt}}^{\text{scaled}} = z + z_{\text{opt}} \cdot 10^5 \cdot \text{Magnitude Order}_{p_{\text{opt}}}$$

Price order of magnitude is calculated in code by:

Code 06: Magnitude order for dual variable (z_opt) scaling

```
magnitude_p_opt = len(str(int(abs(min(p)))))

magnitude_order_p_opt=(10**(magnitude_p_opt-1))
```

1. **abs(min(p))**: Finds the absolute value of the minimum price in the list p.
2. **int(...)**: Converts it to an integer to remove any decimals.

3. **str(...) and len(...)**: Converts the integer to a string and counts the number of digits — this gives the magnitude (e.g., 6000 → 4 digits).
4. **10**(magnitude_p_opt - 1)**: Calculates the order of magnitude (e.g., 4 digits → (10^{3} = 1000)).

So, the final result is the base unit representing the scale of the price, used to adjust **z_opt** accordingly.

1.5 Optimal Price calculation function:

Code 07: Optimal Price calculation function

```
def exploration_optimal_price (z_opt,
       demand_probability, p,customers_per_price):

    optimal_price = None
    best_value = float('-inf')

    for pt in p:
        demand = demand_probability.get(pt, 0)
        value = (pt - float(z_opt)) * demand
        if value > best_value:
            best_value = value
            optimal_price = pt

    #print(f"*** Optimal Price Calculation ***")
    print(f"Optimal Price (p*): {optimal_price:.2f},\nFor Highest Revenue Value:
{best_value*customers_per_price:.2f}")
    return optimal_price
```

Optimal Price Calculation Function:

The optimal price calculation function determines the price point that maximizes adjusted profit revenue, considering both the unit value of inventory and customer demand. It identifies the price at which the highest adjusted revenue is achieved.

Adjusted Revenue is computed as:

(price – z_opt) × demand

Here, **z_opt** represents the unit value of inventory, and **demand** denotes the probability of purchase among approached customers, or equivalently, the expected number of units sold at that price.

1.6 Narrow Price Interval Calculation function:

Code 08: Narrow price interval calculation function

```
def narrow_price_interval(p_opt, n, k, epsilon, p, num_steps,
       global_min,global_max ):

    magnitude_p_opt = len(str(int(abs(min(p)))))
    magnitude_order_p_opt=(10**(magnitude_p_opt-1))


    # Compute interval shrinking factor - parameter δ - Delta
    log_n = np.log(n)
    delta = ((n ** ((-0.25) * (1 - ((3 / 5) ** k)))) * (log_n ** (-3 * epsilon)))  *
(magnitude_order_p_opt)


    # Generate new interval around p_opt
    p_low = max(min(p), p_opt - delta)
    p_upp = min(max(p), p_opt + delta)

    # Ensure Interval does not Collapse*
    if p_low >= p_upp:
        # Fallback: Use 10% of current interval width
        width = (max(p) - min(p)) * 0.1
        p_low = max(global_min, p_opt - width)
        p_upp = min(global_max, p_opt + width)

    # New refined price grid
    new_price_list = np.linspace(p_low, p_upp, num_steps)

    #print(f"*** Subphase {k+1} Price Interval Update ***")
    print(f"delta: {delta:.5f}")
    print(f"New Interval: [{p_low:.2f}, {p_upp:.2f}]")
    #print(f"New Price List: {new_price_list}")
    #or
```

```
    print(f"New Price List: {[int(x) for x in new_price_list]}")


    return p_low, p_upp, new_price_list
```

Narrow Price Interval Calculation Function:

This function refines the price interval around the optimal price based on a calculated delta value. As described in the referenced paper, delta is a function of **n** (the number of customers approached at that price) and **k** (the subphase index). By adding and subtracting delta from the current optimal price, new price bounds are established.

This mechanism progressively narrows the price interval toward the optimal price. As the subphase **k** increases, the value of delta decreases, resulting in a tighter interval. This shrinking effect enables more precise convergence toward the optimal pricing point.

Delta formula according to paper:

$$\delta_m^{(k)} \leq \bar{\Delta}^{(k)} / N^{(k)} = n^{-(1/4)(1-(3/5)^k)} (\log n)^{-3\epsilon}$$

Here,
k = subphase
Epsilon = 0.01
n = customers_per_price (in code, number of customers approaching per price)

New Price interval:
Upper interval limit: Optimal price + delta
Lower price limit: Optimal price – delta

1.6.1 Scaling of Delta in Price Interval Refinement:

Delta Scaling Based on Price Magnitude

To ensure that **δ (delta)** remains proportionate to the pricing domain, we apply a **magnitude-based scaling** using the order of magnitude of the **minimum price** in the dataset.

Magnitude Order Calculation

We first compute the number of digits in the absolute value of the minimum price min(p), then derive the magnitude order:

$$\text{Magnitude Order}_{\delta} = 10^{(\text{digits in } |\min(p)| - 1)}$$

Delta Formula

Delta is then scaled as follows:

$$\delta = \left(n^{-0.25(1-(3/5)^{k})} \right) \cdot \log(n)^{-3\epsilon} \cdot \text{Magnitude Order}_{\delta}$$

This scaling ensures that δ adapts to the **scale of the price domain**, maintaining its effectiveness across different pricing ranges and preventing it from being disproportionately small or large.

Code 09: Magnitude order scaling for delta

```
magnitude_p_opt = len(str(int(abs(min(p)))))

magnitude_order_p_opt=(10**(magnitude_p_opt-1))

delta = ((n ** ((-0.25) * (1 - ((3 / 5) ** k)))) * (log_n ** (-3 * epsilon)))
       * (magnitude_order_p_opt)
```

1.7 Plotting Price vs Demand function:

Plots price vs demand recorded in phase 1 (exploration)

The history attribute is passed from exploration function which records all demand history. In implementation it is named as: **demand_history**

Code 10: Plotting price vs demand - Phase 1 function

```
def plot_exploration_demand_curves(history):

    plt.figure(figsize=(24, 6))
```

```
    colors = plt.cm.viridis(np.linspace(0, 1, len(history)))  # Color gradient

    # Type 1 Plot
    plt.subplot(1, 2, 1)
    for i, data in enumerate(history):
        plt.plot(data['type1'][0], data['type1'][1],
                 'o-', color=colors[i],
                 label=f'Subphase {data["subphase"]}',
                 alpha=0.8)
    plt.title('Exploration: Reserved Customers (Type 1)\nPrice vs. Demand
Probability')
    plt.xlabel('Price ($)')
    plt.ylabel('Demand')
    plt.grid(alpha=0.2)
    plt.legend()
```

1.8 Exploration function:

Note: The full function code is excluded from this report as it is extensive and would unnecessarily increase the overall length.

The exploration function orchestrates a complete iteration of **Phase 1** in the Primal-Dual algorithm for dynamic pricing and inventory allocation. Each invocation of this function corresponds to one **subphase**, during which simulated customer interactions are used to refine pricing decisions and update inventory and time parameters.

Sequential Steps in the Exploration Subphase

1. Customer Simulation

A fixed number of customers (total_customers) are simulated.

- **Type Assignment**: Each customer is randomly assigned as either **Type 1 (Reserved)** or **Type 2 (Preemptive)**.
- **Price Selection**: A price is randomly chosen from the respective price list (p or p2).

- **Demand Probability**: Purchase probability is computed using a logistic demand function, which models price sensitivity.
- **Noise Injection for Realism**

  To simulate realistic behavior, N**oise** is added to the base probability:
  - This introduces randomness, reflecting that real customers may not always follow predictable price-response patterns.
  - It prevents overly deterministic outcomes, allowing for occasional unexpected purchases or refusals.
  - Different noise levels are used for each type to reflect behavioral volatility.
- **Purchase Decision**: A binary decision is simulated based on the noisy probability.

2. Demand Recording

- Customer data—including ID, offered price, purchase decision, and type—is stored in a structured format (DataFrame).
- This data serves as the empirical basis for demand estimation.

3. Demand Probability Estimation

- Demand probabilities are computed separately for each customer type using observed purchase behavior across the offered price points.
- These probabilities reflect the empirical likelihood of purchase at each price.

4. Revenue Calculation

- Revenue is computed per price point and aggregated for each customer type.
- Total revenue across both types is also calculated to assess overall performance.

5. Dual Variable Optimization

- The **dual variable** (z^*), representing the unit value of inventory, is computed using a dual objective function.

- Separate computations are performed for each customer type, and a combined value is derived via averaging.
- This variable plays a central role in balancing revenue maximization against inventory constraints.

6. Optimal Price Determination

- Using the optimized dual variable and demand probabilities, the **optimal price** is computed for each customer type.
- This price maximizes expected revenue while respecting inventory valuation.

7. Price Interval Narrowing

- The price intervals are refined around the optimal prices to focus subsequent exploration on promising regions.
- This narrowing is performed using a controlled epsilon-based strategy, ensuring convergence toward optimal pricing.
- **Delta Scaling** is applied here to ensure that the narrowing step is proportional to the magnitude of the prices:
- magnitude_p_opt = len(str(int(abs(min(p)))))
- magnitude_order_p_opt = 10**(magnitude_p_opt - 1)

The delta is scaled by magnitude_order_p_opt to maintain consistency across different pricing scales.

8. Inventory and Time Update

- Units sold during the subphase are subtracted from the available inventory c.
- The booking period T is updated based on the number of steps and time per step (xt).
- These updates ensure that the algorithm respects temporal and resource constraints.

9. Demand History Logging

- Demand probabilities and price lists for both customer types are recorded in demand_history.
- This data is used for visualization and analysis after Phase 1 concludes.

Finalization of Phase 1

Upon completion of the final subphase (ki == K - 1), the function:

- Prints summary statistics including:
    - Final optimal prices
    - Inventory usage
    - Remaining booking period
- Plots demand curves for both customer types using the accumulated demand history.

Return Values

The function returns a comprehensive set of outputs:

- Simulated customer data (df)
- Optimal dual variable (z_opt)
- Optimal prices for both customer types
- Updated inventory and booking period
- Refined price intervals and new price lists
- Total revenue and updated customer ID

1.9 Price List generation function:

This function generates a price list by evenly dividing the user-defined interval between a minimum and maximum price (e.g., 3000 to 7000) into a specified number of steps

(num_steps). It returns a sequence of price points used for demand simulation and pricing decisions.

Code 11: Price list generation function

```
# Generate price ranges step wise as mentioned by user for custoemr type 1
def generate_price_list_1(num_steps):
    min_price_1 = float(3000)  #float(input("Enter the minimum price: "))
    max_price_1 = float(7000)  #float(input("Enter the maximum price: "))
    num_steps = num_steps
    price_list = (np.linspace(min_price_1, max_price_1, num_steps))
    return price_list
```

***min_price and max_price values are hardcoded which can be taken from user as input after uncommenting the later section in command.

1.10 Time allocation function to Phase 1 i.e. Exploration:

Code 12: Time allocation to Phase 1 function

```
def phase_one_days_calc(num_steps,xt,T):
    xp=int(33)    #int(input("Enter % of Phase 1, between 0 to 100:"))

    if (T*(xp/100)) % (num_steps*xt) == 0:
        phase = T*(xp/100)
    else:
        phase = (T*(xp/100)) - ((T*(xp/100))% (num_steps*xt))
    return phase
```

Phase 1 Duration Allocation

This function calculates the duration of Phase 1 based on the total time T, the number of price points (num_steps), and the time per step (xt). It uses a user-defined percentage (xp) to allocate time to Phase 1. To maintain consistency in step-wise progression, the function adjusts the phase duration to be divisible by the total time required for all price

steps. Improper user input may lead to irregular phase durations, potentially disrupting program flow.

1.10.1 Subphase (ki) Structure in Price Exploration

A **subphase** represents a discrete segment within a broader **exploration phase**, designed to test multiple price points within a defined interval. Each subphase consists of a fixed number of **instances**, where each instance corresponds to a unique price point being evaluated.

For example, if the price interval contains five distinct price points, the subphase will span **five instances**, each dedicated to experimenting with one of those prices.

Abbreviations in code:
K = Total Subphases
ki = one subphase in K subphases
total instances in Phase 1:
P1 = Subphases (K) x Price points in price interval (num_steps)

1.10.2 Phase Composition in Price Exploration

A **phase** encompasses all subphases executed during **Phase 1** of the pricing strategy. It represents the complete cycle of price experimentation across the defined interval.

Each **phase** has **total instances** of:

- **Phase = Subphases (K) × Price Points (num_steps)**

This means that for every subphase, the algorithm iterates through all available price points.

This structure enables comprehensive exploration by:

- Repeating price trials across multiple subphases
- Capturing variability in demand across time and conditions
- Building a robust dataset for estimating optimal pricing behavior

1.11 User Input:

Code 13: User Input code block

```
c      = float(10000)
#float(input("Total Inventory 'c' :   "))

c0     = c
# Initial Inventory declared before or at start of booking period,
# variable defined as original variable c is changing as we run code

T      = float(100)
#float(input("Total booking peroid 'T' :  "))

T0     = T
# Booking peroid set by user,
# again variable made as T original variable is going
# to change while running algorithm

unit_T= "days"
#input("Enter unit of booking peroid (Usually days):")

t      = float(100)
#float(input("Instances 't' to divide Time peroid T into :  "))

xt     = T/t

num_steps = int(5)
#int(input("Enter how many price points you want: "))

p      = generate_price_list_1(num_steps)
# price list for customer type 2

p_1_initial = p
# saving initial price list as p pricelist will change

p2     = generate_price_list_2(num_steps)
# price list for customer type 2

p_2_initial = p2
# saving initial price list as p2 pricelist will change

P1     = phase_one_days_calc(num_steps,xt,T)
# here inside fucntion: "phase_one_days_calc",
# input phase percent value is hardcoded as 33 or 35, user can give user input

K      = int (P1/(num_steps*xt))
```

```
z     = 00
#input("Enter imitial value of 1 unit of inventory (Can be zero): ")

customers_per_price = 100
#int(input("Enter number of customer approaching per price (e.g. 10):"))

total_customers = 2*customers_per_price*num_steps
#2 for 2 customer types, and num_steps is price points,
# so customer approaching for each price and in that each type.


customer_data = []
# list to store customer data into data frame df_1 from exploration phase

df_1 = pd.DataFrame(customer_data, columns=
                    ["customer_id", "price", "buy", "customer_type"])
# dataframe to store Phase 1 data

demand_history = []
# list to store demand values with respect to price
#  for plotting price vs demand for Phase 1.

p_opt_type1 = [] #list for collecting optimal prices of type 1 customer
p_opt_type2 = [] #list for collecting optimal prices of type 2 customer

# Serial number given to each customer as they approach
customer_id = 1
```

User Input Initialization Block

This section sets up the initial parameters required for the simulation. Although user inputs are currently hardcoded for demonstration purposes, the commented lines allow for dynamic input collection. These inputs define key variables such as total inventory (c), booking period (T), time granularity (t), and the number of price points (num_steps). The values chosen here align with a model example discussed later in the report.

The primary role of this block is to supply the algorithm with necessary information to simulate customer behavior and optimize pricing. It also initializes data structures for tracking customer interactions, demand patterns, and optimal prices. The main

simulation loop follows immediately after this setup, using these inputs to execute the pricing strategy.

1.12 Main Loop 1: Conduct Exploration

Code 14: Main Loop 1 – Conduct Exploration code block

```
# Conduct Phase 1   i.e.  ***Exploration***:

print("\n","***Expoloration Start - Detailed Data:***")


# initializing variable ki to loop in total subphases in Phase 1
ki = 0

for ki in range(K):

    if c>0: # Inventory should exist  before sale

        print("\n","Phase 1 (P1), Subphase (ki): ", ki+1,"\n")

        df, z_opt, p_opt_tuple,
        c, T, t, p_low_tuple,
        p_upp_tuple, updated_p_tuple,
        total_revenue,
        customer_id = exploration(c,
                                  c0,T,
                                    unit_T,
                                    t, xt,
                                    ki, K, P1, p, p2, z,
                                    customers_per_price,
                                    total_customers,
                                    num_steps, demand_history,
                                    customer_id,T0, p_1_initial,
                                    p_2_initial)


        # Dataframe update for exploration - Phase 1
        df_1 = pd.concat([df_1, df], ignore_index=True)
        df_1["price"] = df_1["price"].round(2) # rounding price


        # Update price lists for next iteration
        p = updated_p_tuple[0]
```

```
        p2 = updated_p_tuple[1]

        # Seprating and storing Optimal prices in a list
        p_opt_type1.append(p_opt_tuple[0])
        p_opt_type2.append(p_opt_tuple[1])

    else:
        print("*** INSUFFICIENT INVENTORY ***")



# Save the Phase 1 dataframe to a CSV file
df_1.to_csv("all_output.csv", index=False)
print("Data from df_1 (Phase 1 - Exploration) appended and saved to all_output.csv")


# Seprate Data frames for different customer types for phase 1
df_1_type_1 = df_1[df_1['customer_type'] == 1].copy()
df_1_type_2 = df_1[df_1['customer_type'] == 2].copy()

# Plotting Price vs demand for Phase 1 (Exploration) for both customer types
plot_demand_curves_from_df(df_1_type_1, df_1_type_2)
```

Main Loop 1: Phase 1 – Exploration

This loop executes the **exploration phase** of the Primal-Dual algorithm by repeatedly calling the exploration function for each subphase (ki) until all subphases in Phase 1 are completed (K iterations).

- **Inventory Check**: The loop proceeds only if sufficient inventory ($c > 0$) is available, ensuring feasibility of simulated sales.
- **Function Execution**: In each iteration, the exploration function simulates customer behavior, computes demand, revenue, dual variables, and optimal prices, and refines price intervals.
- **Data Aggregation**: After each subphase, the resulting demand simulation data (df) is appended to the main Phase 1 dataframe (df_1), which accumulates all customer interactions.

- **Price Update**: Refined price lists are updated for the next iteration to focus exploration around optimal values.
- **Optimal Price Tracking**: Optimal prices for each customer type are stored for analysis across subphases.

Upon completion, the full Phase 1 dataset is saved to a CSV file (all_output.csv) and separated by customer type for further analysis and demand curve plotting.

2. Phase 2 - Exploitation:

2.1 Demand Estimation Function:

Accurate demand estimation is central to dynamic pricing, particularly in environments with uncertain customer behavior and constrained inventory. In this implementation, the demand estimation function predicts the probability of purchase at a given price, enabling the exploitation phase to align pricing decisions with projected demand and available capacity.

2.1.1 Purpose and Integration

The function estimates demand for a given price input and is invoked during the exploitation phase to:

- Project future demand over the remaining booking horizon
- Compare projected demand with remaining inventory
- Decide whether to retain or adjust the current optimal price

This ensures pricing decisions are both data-driven and inventory-aware.

2.1.2 Model Selection

Two supervised learning models are implemented:

- **Random Forest Classifier**: An ensemble of decision trees trained via bootstrap aggregation. It is non-parametric and well-suited for capturing nonlinear relationships without assuming a specific functional form.
- **LightGBM (Gradient Boosting Machine)**: A tree-based boosting algorithm optimized for speed and efficiency. It uses histogram-based decision rules and leaf-wise tree growth, making it computationally superior for large datasets.

Both models are capable of estimating demand even when the underlying price-demand relationship is unknown or noisy—unlike traditional models (e.g., linear, polynomial, logistic) that require predefined functional assumptions.

2.1.3 Model Training and Evaluation

Each model follows a structured training pipeline:

- **Data Preparation**:
  The input DataFrame is filtered to extract relevant features (e.g., price, purchase outcome). Additional features such as remaining time and inventory can be incorporated to improve predictive accuracy.
- **Train-Test Split**:
  Data is split into training and test sets (typically 70/30) to evaluate generalization performance.
- **Hyperparameter Tuning**:
  - *Random Forest*: Uses GridSearchCV with 5-fold cross-validation to optimize parameters such as tree depth, number of estimators, leaf size, and feature selection strategy.
  - *LightGBM*: Trained with early stopping and predefined parameters (e.g., learning rate, number of leaves, max depth), using validation loss to halt training when performance plateaus.
- **Model Evaluation**:
  Accuracy is computed on the test set to assess predictive performance. Both

models currently yield ~80% accuracy, with scope for improvement via feature engineering.

- **Probability Prediction**:
  The trained model outputs a probability score indicating the likelihood of purchase at the input price. This score is bounded between 0 and 1 and serves as the estimated demand rate.

2.1.4 Computational Efficiency

- **Random Forest**: Offers interpretability and robustness but requires ~10–12 minutes per run due to exhaustive grid search.
- **LightGBM**: Executes in under a minute, making it suitable for real-time or iterative pricing decisions.

Given its efficiency, **LightGBM is currently used** in the exploitation phase, though both models are retained for comparative analysis and future tuning.

2.1.5 Feature Expansion Strategy

To improve model accuracy, additional features can be introduced:

- Remaining time period
- Remaining inventory
- Interaction terms (e.g., price × customer type, $price^2$)

This enhances the model's ability to capture complex demand dynamics and supports more granular pricing decisions.

2.1.6 Model Comparison: Random Forests vs. LightGBM

Random Forests:

- **Model Type**: Ensemble of decision trees trained via bootstrap aggregation (bagging).
- **Functionality**: Aggregates predictions from multiple trees to reduce variance and improve generalization.
- **Flexibility**: Handles nonlinear and complex relationships without requiring a predefined functional form.
- **Robustness**: Performs well with noisy or biased data; less sensitive to outliers.
- **Training Overhead**: Requires grid search for hyperparameter tuning (e.g., number of trees, depth, leaf size), leading to longer runtimes (~10–12 minutes).
- **Use Case**: Suitable for exploratory modeling and scenarios where interpretability and robustness are prioritized over speed.

LightGBM (Light Gradient Boosting Machine):

- **Model Type**: Gradient boosting framework using decision trees with leaf-wise growth strategy.
- **Efficiency**: Highly optimized for speed and memory usage; uses histogram-based binning for faster computation.
- **Accuracy**: Maintains strong predictive performance while being computationally lightweight.
- **Scalability**: Ideal for large datasets and iterative learning environments.
- **Training Speed**: Executes in seconds (typically <1 minute), enabling rapid experimentation and real-time decision-making.
- **Use Case**: Preferred for production-level deployment and time-sensitive applications where speed is critical.

Code 15: **LightGBM** model for demand estimation

```
### This demand estimation is done by lightgbm / xgboost
def lightgbm_buying_probability(df, price_value):
    df_clean = df.copy()
    X = df_clean[['price']]
    y = df_clean['buy'].astype(int)

    # Split data
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)

    # Create LightGBM datasets
    train_data = lgb.Dataset(X_train, label=y_train)
    test_data = lgb.Dataset(X_test, label=y_test, reference=train_data)

    # Set parameters
    params = {
        'objective': 'binary',
        'metric': 'binary_logloss',
        'verbosity': -1,
        'boosting_type': 'gbdt',
        'learning_rate': 0.1,
        'num_leaves': 15,
        'max_depth': 5,
        'seed': 42
    }

    # Train with early stopping
    model = lgb.train(
        params,
        train_data,
        num_boost_round=100,
        valid_sets=[test_data],
        valid_names=['valid'],
        callbacks=[lgb.early_stopping(stopping_rounds=10)]
    )

    # Predict on test set for accuracy
    test_preds = model.predict(X_test)
    test_preds_binary = [1 if p > 0.5 else 0 for p in test_preds]
    test_acc = accuracy_score(y_test, test_preds_binary)
    print(f"Tuned Light Gmb test accuracy: {round(test_acc, 2)}")

    # Predict probability for input price
    predicted_prob = model.predict([[price_value]])[0]

    #safeguard to ensure demand probability stays between 0 and 1
```

```
    predicted_prob = max(0, min(1, predicted_prob))

    return round(predicted_prob, 2)
```

Code 16: **Random Forests** model for demand estimation.

```
### This demand estimation is done by random forest
def random_forest_buying_probability(df, price_value):

    df_clean = df.copy()
    X = df_clean[['price']]
    y = df_clean['buy'].astype(int)

    # Split your data
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
       random_state=42)

    # Define hyperparameter grid
    param_grid = {
        'n_estimators': [100, 200, 500],
        'max_depth': [None, 5, 10],
        'min_samples_leaf': [1, 2, 4],
        'max_features': ['sqrt', 'log2']
    }

    # Set up GridSearchCV
    grid_search = GridSearchCV(
        RandomForestClassifier(random_state=42),
        param_grid,
        cv=5,                    # 5-fold cross-validation
        scoring='accuracy',
        n_jobs=-1
    )

    grid_search.fit(X_train, y_train)
    best_model = grid_search.best_estimator_

    # Evaluate accuracy on test set
    test_predictions = best_model.predict(X_test)
    test_acc = accuracy_score(y_test, test_predictions)
    print(f"Tuned Random Forest test accuracy: {round(test_acc, 2)}")
    # print("Best parameters found:", grid_search.best_params_)

    # Predict probability for input price
    predicted_prob = best_model.predict_proba([[price_value]])[0][1]

    #safeguard to ensure demand probability stays between 0 and 1
```

```
    predicted_prob = max(0, min(1, predicted_prob))

    return round(predicted_prob, 2)
```

2.2 Alpha (α) and Price bounds calculation function:

In our implementation, α (alpha) is used to calculate the bounds around the optimal price. Specifically, the upper bound is defined as optimal price + α, and the lower bound as optimal price − α. The value of α is dynamically determined based on a scaling factor, denoted as n, which in our case corresponds to customers_per_price.

Alpha formula according to paper:

$$\alpha = (\log n)^{1+9\epsilon} n^{-1/4}$$

Here, n = scaling factor = customers_per_price (in implementation), epsilon = 0.01

When the algorithm enters Case 2: Insufficient Inventory, it triggers the computation of α and subsequently derives the price bounds. Following this, a demand simulation is conducted: half of the customer instances are exposed to the lower bound price, and the other half to the upper bound price. This simulation helps record the demand response at both ends of the pricing spectrum, enabling more informed inventory allocation and pricing decisions.

Code 17: Alpha and price bound calculation function

```
def calculate_alpha_and_bounds (p_opt,p):

        # defying variables for alpha, according to which prices are changed to upper
or lower bound
        n = customers_per_price
        log_n = np.log(n)
        epsilon = 0.01



        # Calculating alpha according to price order of magnitude to make sense -
Cust. type 1
        magnitude_p_opt = len(str(int(abs(p_opt))))
        magnitude_order_p_opt = (10**(magnitude_p_opt-2))
```

```
        alpha = (log_n**(1+(9*epsilon)))*(n**(-1/4))*magnitude_order_p_opt
        alpha = round(alpha, 2)


        # Calculate Lower and upper bounds of prices for Cust type 1. according to
alpha (ensure prices don't cross main price interval)
        p_opt_low =   max(min(p), p_opt - alpha)
        p_opt_high =  min(max(p), p_opt + alpha)

        return  alpha, p_opt_low, p_opt_high
```

2.2.1 Alpha Scaling:

In order to adapt α (alpha) to the scale of the optimal price p_opt, we multiply α by a factor derived from the order of magnitude of p_opt. This ensures that α scales appropriately across different pricing ranges.

Specifically, we compute:

$$\text{Magnitude Order} = 10^{(\text{number of digits in } |p_{\text{opt}}| - 2)}$$

This factor is then used to scale α:

$$\alpha = \left(\log(n)^{1+9\epsilon}\right) \cdot n^{-1/4} \cdot \text{Magnitude Order}$$

This scaling maintains the relative impact of α in the pricing model, ensuring consistency whether optimal price p_opt is in the tens, hundreds, or thousands.

Code 18: Magnitude order scaling for Alpha

```
      magnitude_p_opt = len(str(int(abs(p_opt))))
      magnitude_order_p_opt = (10**(magnitude_p_opt-2))

      alpha = (log_n**(1+(9*epsilon)))*(n**(-1/4))*magnitude_order_p_opt
```

2.3 Time Allocation, Theta (θ) function:

Theta is calculated by solving, Total Sales = Remaining inventory

$$(T - t_K)(\theta D_l^{(K)} + (1 - \theta) D_u^{(K)}) = nc - S(t_K)$$

Here, (according to the main paper)

T-tk = Remaining Booking period

nc – S(tk) = Remaining Inventory

Du = Demand for Upper bound

Dl = Demand for lower bound

K = beginning of exploitation, Phase K in paper

Simplified as per variables in implementation:

$$T\left(\theta \cdot D_l + (1 - \theta) \cdot D_u\right) = c$$

In implementation,

Dl = D_lower_1 or D_lower_2 (1 for customer type 1 and 2 for customer type 2)

Du = D_upper_1 or D_upper_2 (1 for customer type 1 and 2 for customer type 2)

Θ = theta_1 or theta_2 (1 for customer type 1 and 2 for customer type 2)

2.3.1 Theta (θ):

Theta is computed during **Phase 2** (Exploitation), specifically under **Case 2**—when **inventory** is **insufficient**. It is derived after simulating demand at both the lower and upper bounds of the optimal price, each tested over half of the first instance when entered into case 2.

The variable **theta** governs how pricing is distributed between the lower and upper bounds of the optimal price when inventory is insufficient to meet projected demand. Theta is calculated after obtaining demand on lower and upper bound of prices both

simulated on half instances when the algorithm gets in case 2 i.e. insufficient inventory of phase 2 i.e. exploitation. It is a dynamic allocation factor ranging from 0 to 1, calculated based on:

- Demand at the lower and upper price bounds
- Remaining inventory (c)
- Remaining booking period (T)

A **theta** value of 0.3, for example, implies that 30% of the next two pricing instances will use the lower bound, while 70% will use the upper bound. This allows the algorithm to adaptively shift pricing to manage inventory consumption.

Function Overview: compute_theta

- **Demand Comparison**: If demand at both bounds is nearly equal, theta defaults to 0.5.
- **Scaling**: Theta is scaled by the order of magnitude of inventory to maintain interpretability across different inventory levels.
- **Robustness**: The function includes safeguards against division by zero, infinity, and NaN values.
- **Clamping**: Final theta is constrained between 0 and 1 to ensure valid allocation proportions.

This mechanism enables controlled pricing adjustments under inventory pressure, balancing revenue optimization with resource preservation.

Code 19: Theta calculation function

```
def compute_theta(c, T, D_lower, D_upper, p_opt):

    # Avoid division by zero with a small epsilon
    epsilon = 1e-8
    denominator = D_lower - D_upper

    # If demand for upper and lower bounds is nearly equal, assume theta as 0.5
(midpoint)
    if abs(denominator) < epsilon:
        return 0.5
```

```
    # we multiply theta by price magnitude, this is done after thorough observation of
output
    magnitude_c = len(str(int(abs(c))))
    magnitude_order_c = (10**(magnitude_c))

    raw_theta = (((c/T) - D_upper)/(D_lower - D_upper))/magnitude_order_c
    print(f"raw_theta: {raw_theta:.2f}")

    # If theta is Infinity:
    if math.isinf(raw_theta):
        if raw_theta > 0:
            theta = 1.0
        else:
            theta = 0.0

    # If theta is Nan:
    elif math.isnan(raw_theta):
        theta = 0.0

    # Safe guard Clamp and abs, ensures theta styas between 0 and 1.
    theta = max(0.0, min(1.0, abs(raw_theta)))


    return round(theta, 2)
```

2.3.2 Theta Scaling:

To ensure numerical stability and interpretability, theta is scaled relative to the magnitude of the target cost (c). The unscaled theta is computed as:

$$\theta = \frac{\left(\frac{c}{T} - D_u\right)}{D_l - D_u}$$

Given that large values of (c) can produce theta values outside the expected range ([0, 1]), a normalization step is applied:

$$\theta = \frac{\left(\frac{c}{T} - D_u\right)}{D_l - D_u} \div 10^{\text{order}(c)}$$

where order (c) denotes the number of digits in (c). This scaling ensures theta remains within a reasonable range. Additional safeguards are implemented to clamp theta between 0 and 1 and to handle undefined or infinite values gracefully.

2.4 Exploitation Function:

Note: The full function code is excluded from this report as it is extensive and would unnecessarily increase the overall length.

The exploitation function governs **Phase 2** of the Primal–Dual algorithm, where pricing decisions are refined based on estimated demand and remaining inventory. It operates iteratively, simulating customer behavior and updating pricing strategies in response to inventory constraints.

2.4.1 Demand Estimation and Case Selection

- The function begins by estimating demand at the current optimal prices using the **LightGBM demand estimation model**.
- Future demand is projected by scaling these estimates over the remaining booking period and customer volume.
- Based on this projection, the function selects one of two cases:
    - **Case 1: Sufficient Inventory** — If projected demand is less than or equal to available inventory, the function proceeds with the current optimal prices for one iteration.
    - **Case 2: Insufficient Inventory** — If projected demand exceeds inventory, the function initiates a price adjustment mechanism.

2.4.2 Case 1: Sufficient Inventory

- The current optimal prices are retained.
- Demand is simulated for both customer types using logistic functions with noise.

- Results are recorded, inventory is updated, and the booking period is reduced by one instance.

2.4.3 Case 2: Insufficient Inventory

- The function calls the **alpha and bound calculation** module to determine upper and lower price bounds.
- Demand is simulated separately for each bound over half an instance.
- These simulations yield demand estimates used to compute **theta**, which allocates time between the bounds (upper and lower) for the next two instances.
- Based on theta values, the function simulates sales at each bound and updates inventory and time accordingly.
- The optimal price is then updated depending on which bound received more time allocation.

- At the end the booking period is reduced by 3 instances.

2.4.4 Data Recording and Iteration

- All customer interactions from the current iteration are stored in df_exploitation.
- Inventory and booking period are updated.
- The function returns updated values to continue the main loop until either time or inventory is exhausted.

2.5 Main Loop 2 – Phase 2 i.e. Exploitation:

Main Loop 2 orchestrates the iterative execution of the exploitation phase, leveraging optimal prices derived from the preceding exploration phase. Its primary objective is to simulate customer interactions, update inventory and pricing decisions, and record transactional data until either inventory or booking time is exhausted.

2.5.1 Initialization

- Optimal prices for both customer types are retrieved and rounded.
- A safety threshold for inventory (safe_c) and booking time (safe_T) is defined to prevent negative values during the final iterations.
- An empty DataFrame df_2 is initialized to store all customer interactions during exploitation.
- The iteration counter ti is set to zero.

Safe Inventory Threshold

To prevent negative inventory during exploitation, a safety threshold safe_c is defined:

$$safe\_c = 3 \times 2 \times customers\_per\_price \times 0.6$$

- **3**: Max instances used in Case 2 (Insufficient Inventory)
- **2**: Customer types (Reserved, Preemptive)
- **0.6**: Estimated purchase probability (decided after thorough observation)
- **customers_per_price**: Incoming customers per price per type

This threshold ensures the loop only runs when inventory is sufficient to handle worst-case demand, maintaining feasibility and avoiding negative stock.

2.5.2 Iterative Execution

- The loop continues as long as the remaining booking period exceeds safe_T.
- Within each iteration, the exploitation function is invoked only if inventory exceeds **safe_c**, ensuring feasibility of simulated sales.
- The exploitation function returns updated values for inventory, booking time, optimal prices, and customer data.
- The resulting customer data (df_exploitation) is concatenated to the cumulative DataFrame df_2.

2.5.3 Termination and Output

- If inventory falls below the **safety threshold**, the loop terminates early to avoid infeasible operations.

- Upon completion, the full dataset df_2 is saved to a CSV file (all_output.csv) for further analysis.
- Summary statistics are printed, including:
    - Final inventory and utilization percentage
    - Booking period consumed
    - Final optimal prices for both customer types

2.5.4 Post-Processing

- Separate DataFrames are created for each customer type (df_2_type_1 and df_2_type_2) to facilitate downstream tasks such as visualization or targeted analysis.

Code 20: Main loop 2 – Conduct Phase 2 i.e. Exploitation

```
# Conduct Phase 2   i.e.  ***Exploitation***:

# extracting optimal prices from tuple
p_opt_1 = p_opt_tuple[0]
p_opt_1 = round(p_opt_1, 2)

p_opt_2 = p_opt_tuple[1]
p_opt_2 = round(p_opt_2, 2)

# list to store customer data into data frame df_2 from exploration phase
customer_data = []

# dataframe to store Phase 2 data
df_2 = pd.DataFrame(customer_data, columns=["customer_id",
                                            "price", "buy",
                                            "customer_type"])


T=int(T)

ti = 0

safe_c = 3*2*customers_per_price*0.6


safe_T = xt*3


while  T > safe_T :
```

```
    if c > safe_c:
        T, c, p_opt_1,
        p_opt_2,
        customer_id,
        df_exploitation,
        p_opt_1,
        p_opt_2, ti = exploitation(c,T, p_opt_1,
                                   p_opt_2, z_opt,
                                   p,p2, customer_id,
                                   xt,
                                    p_1_initial, p_2_initial, T0,
                                    unit_T, df_1_type_1,
                                    df_1_type_2, ti,
                                    customers_per_price)


        # Dataframe update for exploitation - Phase 2
        # Complete data of both phases in df_2
        df_2 = pd.concat([df_2, df_exploitation], ignore_index=True)


    else:
        print('Insufficient Inventory')
        break


# Save the final dataframe to a CSV file
df_2['price'] = df_2['price'].round(2)  # rounds to 2 decimal places
df_2.to_csv("all_output.csv", mode='a', header=False, index=False)
print("\nData from df_2 appended to all_output.csv")


# Separate Data frames for different customer types for phase 2
df_2_type_1 = df_2[df_2['customer_type'] == 1].copy()
df_2_type_2 = df_2[df_2['customer_type'] == 2].copy()
```

3.0 Data Management and Preparation

This section outlines the post-processing steps applied to customer interaction data from both phases—**Exploration** and **Exploitation**—to compute key performance metrics such as **revenue**, **demand**, and **regret**, and to prepare data for visualization and storage.

3.1 Revenue and Demand Calculation

Revenue and demand are computed for each price point and customer type using two distinct approaches:

3.1.1 With Dual Variable (z_opt) Consideration

- The dual variable (z* in main paper or z_opt in code implementation) represents the **unit value of inventory**, derived from the Primal–Dual algorithm.
- Revenue is calculated as:

  Revenue = (Price - z*) x Units Sold

- This approach reflects opportunity cost and is aligned with algorithmic optimization.

3.1.2 Without Dual Variable (z_opt = 0)

- This method reflects **natural business revenue**, ignoring algorithmic cost considerations.
- Revenue is calculated as:

  Revenue = Price x Units Sold

- Useful for real-world benchmarking and regret analysis.

Price Ordering

- Two functions are used:
  - compute_final_revenue_demand: Sorts prices in ascending order.
  - compute_final_revenue_demand_natural_order: Retains the order of last price occurrence, useful for regret and behavioral analysis.

3.2 Data Preparation for Analysis and Plotting

3.2.1 Data Consolidation

- Data from both phases is merged into a unified DataFrame df_final.
- Separate DataFrames are created for each customer type:
    - df_final_type_1 for Reserved customers
    - df_final_type_2 for Preemptive customers

3.2.2 Metric Calculation and CSV Storage

- Revenue and demand are computed for:
    - Combined customer types
    - Each customer type individually
- Results are stored in separate CSV files for both **with** and **without** z_opt consideration:
    - results_with_z_opt.csv, results.csv
    - results_type_1_with_z_opt.csv, results_type_1.csv
    - results_type_2_with_z_opt.csv, results_type_2.csv

3.2.3 Sorted DataFrames for Visualization

- To ensure coherent plotting, sorted versions of the results are created:
    - Ascending price order improves interpretability of demand and revenue curves.
    - Both sorted and unsorted versions are retained for comparative analysis.

Sections 4.0 and 5.0 of jupyter notebook implementation present the results and plots, which are further explained in the subsequent Results section.

4.3 Challenges faced in Implementation:

Implementation Challenges and Resolutions

During the development of the Primal–Dual pricing algorithm, several implementation challenges were encountered. These issues primarily stemmed from scale inconsistencies, unrealistic demand behavior, and unstable parameter values. Below is a summary of key challenges and the corresponding resolutions:

4.3.1 Realism in Demand Simulation

Initial demand generation was overly deterministic and lacked randomness, resulting in artificial and predictable behavior. To simulate more realistic market conditions, the logistic demand function was tuned by adjusting slope parameters and introducing stochastic noise. This combination produced demand patterns that were less obvious and more representative of real-world variability.

4.3.2 Dual Variable Scaling

The dual variable (z* or z_opt in implementation), representing the unit value of inventory, initially exhibited negligible changes (on the order of (10^{-7})), rendering it ineffective relative to the price scale. To address this, a scaling mechanism was introduced, aligning (z* or z_opt in implementation) with the magnitude of prices in the price list. Post-scaling, the dual variable became meaningful and contributed effectively to revenue calculations.

4.3.3 Delta Adjustment in Price Interval Narrowing

The delta parameter used to refine price intervals in Phase 1 did not adapt to varying price magnitudes (e.g., 100, 1000, 10000). This led to ineffective narrowing around the optimal price. By scaling delta proportionally to the price magnitude, the algorithm achieved consistent interval refinement across different pricing scales.

4.3.4 Alpha Scaling

The alpha parameter, used in bounding price intervals, was insensitive to changes in price magnitude. This limited its utility in guiding interval adjustments. Scaling alpha relative to the optimal price resolved this issue, yielding more responsive and effective bounds.

4.3.5 Theta Stability and Scaling

Theta, which allocates time between upper and lower price bounds in Phase 2, was highly sensitive to inventory levels. In cases of large inventory (e.g., 10,000 units over 100 days), theta values exceeded the valid range ([0, 1]). To stabilize theta:

- It was scaled based on the order of magnitude of inventory.
- Clamping was applied to restrict values within ([0, 1]).
- Special handling was added for undefined or infinite values (e.g., when upper and lower bound demands are equal), assigning a default value of 0.5.

These adjustments ensured theta remained interpretable and operationally valid throughout the exploitation phase.

4.4 Model Configuration:

To evaluate the algorithm's performance, the following model example was defined:

- **Total Inventory Capacity**: c0 = 10,000 units
- **Total Booking Period**: T0 = 100 days
- **Total Instances**: t = 100 decision points
- **Price Discretization**: num_steps = 5 price points
- **Customer Type 1 Price Range**: [3,000; 7,000]
- **Customer Type 2 Price Range**: [1,000; 5,000]
- **Phase 1 Duration**: 30% of booking period
- **Inventory Unit Value**: z = 0
- **Customer Volume per Price Point**: customers_per_price = 100 customers

This configuration enabled controlled experimentation across customer segments, pricing strategies, and inventory dynamics within a finite booking horizon.

5. Results

The performance of the pricing algorithm was evaluated across multiple dimensions, including price interval convergence, regret analysis, and the relationship between price, demand, and revenue. These observations provide critical insights into the behavior of the algorithm and its effectiveness in learning optimal pricing strategies under uncertainty.

5.1 Price Tuning Across all Phases

The algorithm exhibits distinct tuning behavior across the two phases:

- **Phase 1 (Exploration)**:
  Price tuning is **coarse**, characterized by broader intervals that progressively narrow with each iteration. This allows the algorithm to explore a wide range of price points and converge toward promising regions. By the end of Phase 1, the intervals become significantly tighter, reflecting increased confidence in the estimated optimal prices.

Figure 3: Price trend, Price vs customer_id in phase 1 for customer type 1

- **Phase 2 (Exploitation)**:

  Price tuning becomes **fine-grained**, operating within a narrow interval around the optimal price. Adjustments occur primarily when inventory is insufficient relative to projected demand. These changes are subtle compared to Phase 1, preserving pricing stability while adapting to inventory constraints.

Figure 4: Price trend, Price vs customer_id in phase 1 for customer type 2

This transition from coarse to fine tuning is visually evident in the corresponding figure (3 and 4), illustrating the algorithm's adaptive refinement strategy across phases.

5.2 Price Interval Narrowing in Phase 1

Distinct patterns were observed in how price intervals narrowed during the exploration phase:

- **Customer Type 1**: Price intervals typically narrowed toward the lower or midpoint of the original range, reflecting balanced valuation and moderate price sensitivity.

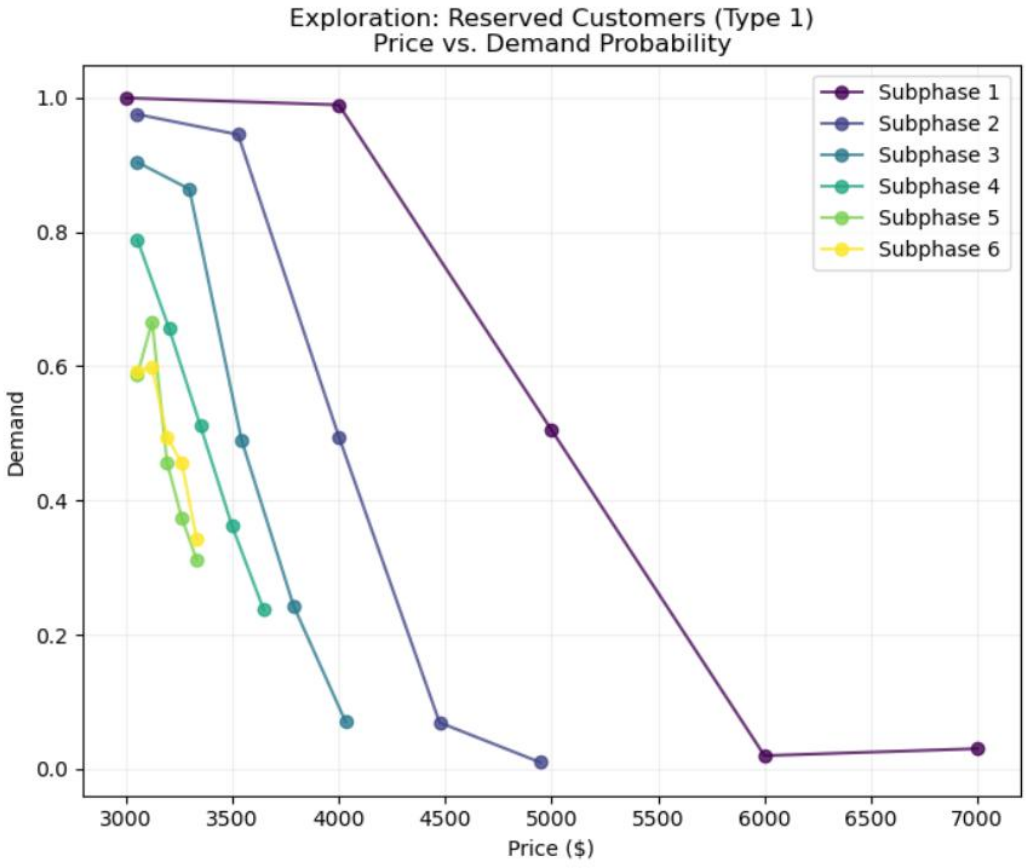


Figure 5: Price vs demand in phase 1 for customer type 1

- **Customer Type 2**: Intervals shifted closer to the midpoint, avoiding both extremes. This behavior aligns with their higher price sensitivity and the influence of adjusted revenue.

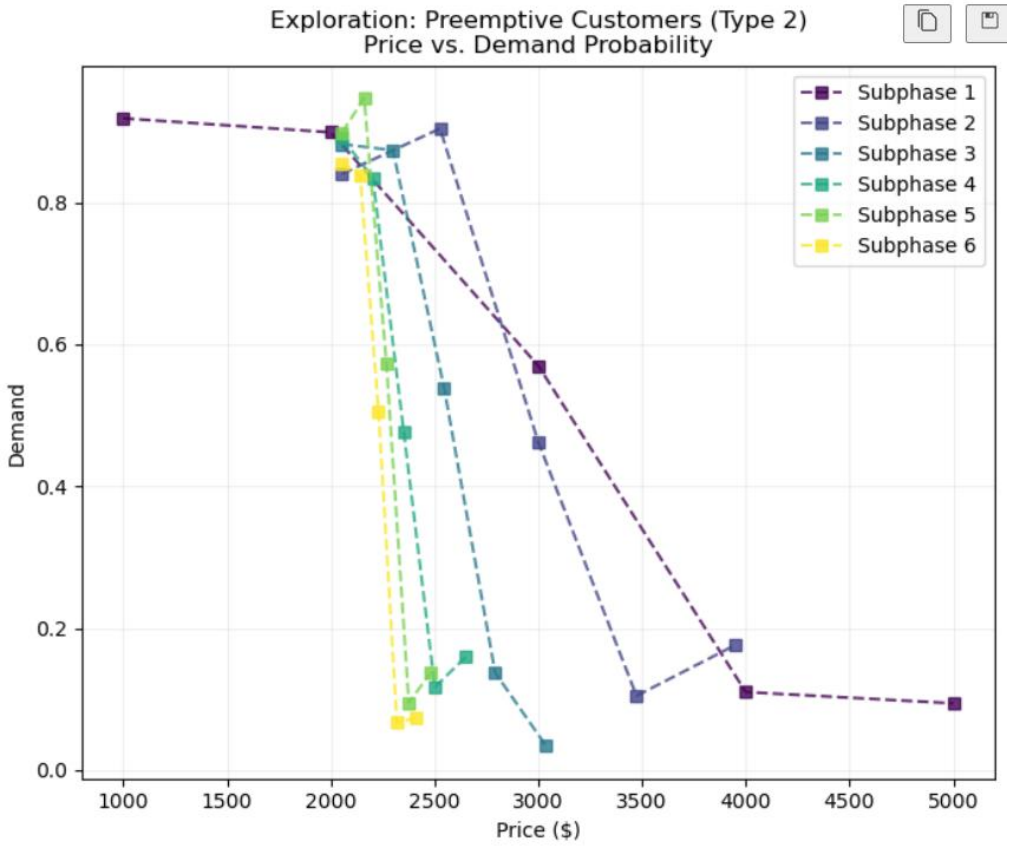


Figure 6: Price vs demand in phase 1 for customer type 2

The subtraction of the dual variable (z*) from price during revenue calculation reduces the impact of low prices, even when demand is high. For Type 2 customers, lower prices yield minimal adjusted revenue, while higher prices suffer from low demand. As a result, mid-range prices become the most effective for revenue generation.

5.3 Regret Analysis

According to the theoretical framework, the algorithm's regret should remain below 15%, meaning actual revenue should not fall more than 15% short of the optimal benchmark. A **negative regret** indicates that the algorithm outperforms the ideal solution.

- **Customer Type 1 (Reserved)**: Regret ranged from **0% to 5%**, indicating consistent performance within acceptable bounds.

- **Customer Type 2 (Preemptive)**: Regret ranged from **5% to 10%**, occasionally exceeding the threshold. This is attributed to higher price sensitivity and biased purchasing behavior in this segment.

The elevated regret in Type 2 customers suggests that their responsiveness to price fluctuations introduces greater variability, making revenue optimization more challenging.

5.4 Price–Demand and Price–Revenue Relationships

Empirical analysis reveals a nonlinear relationship between price, demand, and revenue. While demand tends to increase at favorable price points, this does not always correlate with higher revenue.

- **Price vs. Demand**:
  Demand peaks at multiple price levels, indicating that customers are willing to purchase at both low and moderately high prices. This behavior reflects varying valuations and urgency across customer segments.

- **Price vs. Revenue**:
  Revenue does not necessarily peak where demand is highest. Since revenue is the product of price and demand, high demand at low prices may yield suboptimal revenue. Conversely, moderate demand at higher prices can result in greater profitability.

This underscores the importance of **adjusted revenue** as the guiding metric in the algorithm. By incorporating the dual variable (z*), the algorithm accounts for the opportunity cost of inventory consumption. This prevents over-prioritization of high-demand, low-revenue price points and promotes long-term revenue optimization.

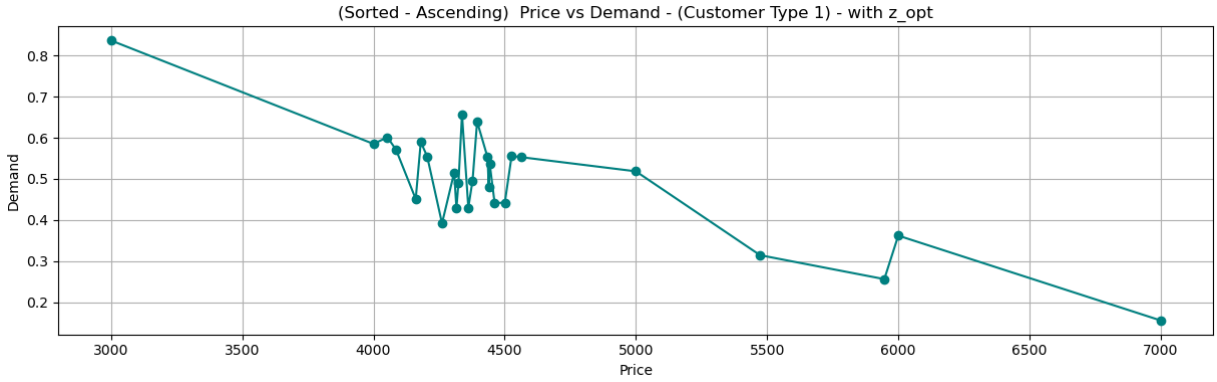


Figure 7: Price vs Demand after all phases for customer type 1 – considering z_opt means considering absolute revenue

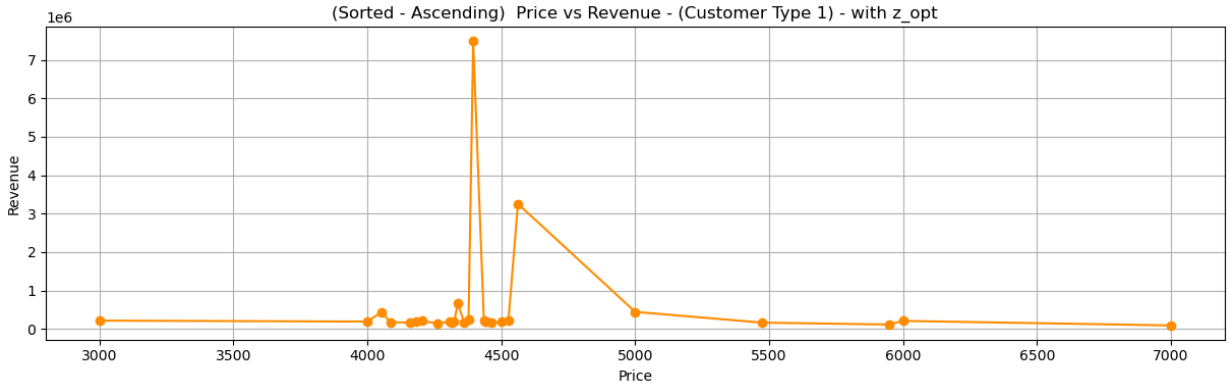


Figure 8: Price vs Revenue after all phases for customer type 1 – considering z_opt means considering absolute revenue

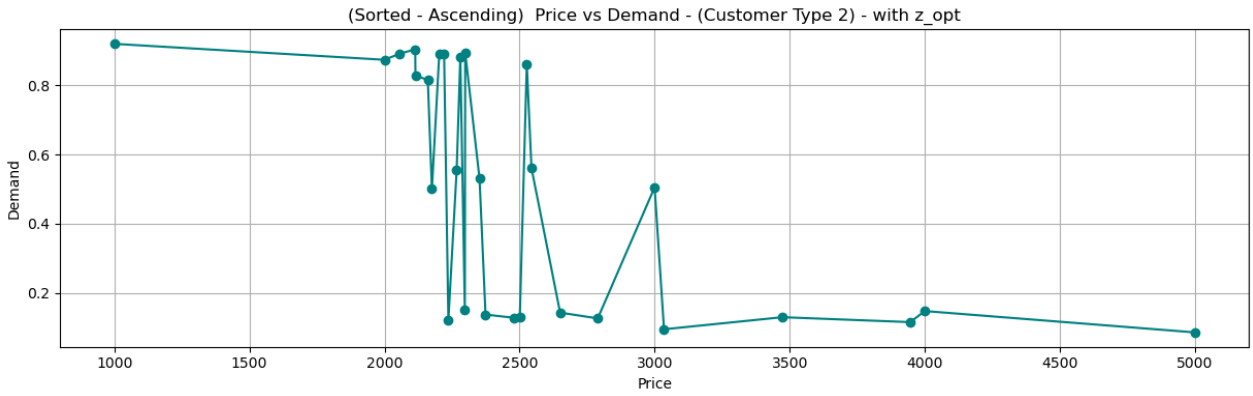


Figure 9: Price vs Demand after all phases for customer type 2 – considering z_opt means considering absolute revenue

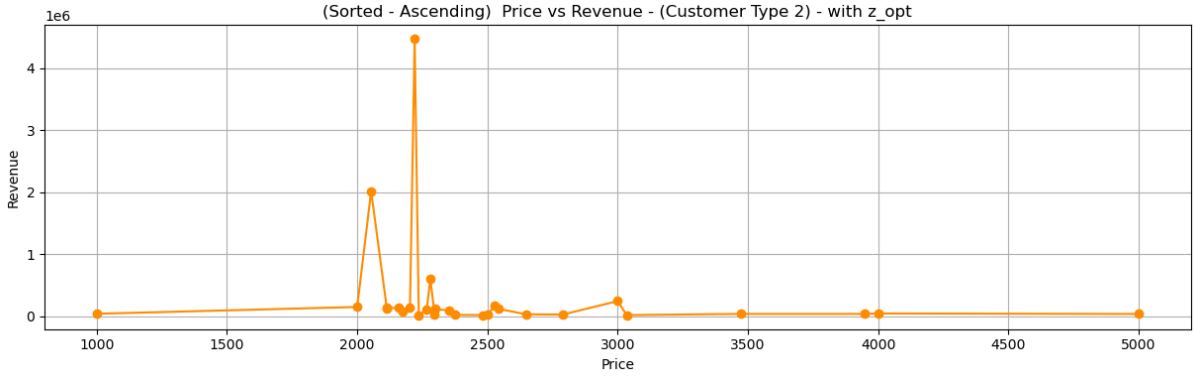


Figure 10: Price vs Revenue after all phases for customer type 2 – considering z_opt means considering absolute revenue

6.0 Future Work and Alternative Approach

Several avenues remain open for extending and refining the current implementation. These directions aim to enhance model robustness, improve predictive accuracy, and explore hybrid methodologies for dynamic pricing:

6.1 Scalability Testing

The algorithm should be evaluated on larger and more diverse datasets to assess its scalability and consistency. This includes verifying whether the current flow and decision logic remain stable under varying demand patterns and inventory scales.

6.2 Enhanced Demand Estimation

Improving the accuracy of the LightGBM-based demand estimation model is critical. Targeting an accuracy threshold of 90% or higher may be achievable by incorporating additional features such as:

- Remaining booking period
- Remaining inventory
- Interaction terms (e.g., price × customer type, price^2)

These enhancements would allow the model to capture more nuanced demand behavior and improve pricing precision.

6.3 Exploration Phase Tuning

Extending the duration of Phase 1 (Exploration) could yield insights into the trade-off between coarse and fine-grained price optimization. This adjustment may also influence inventory consumption rates, necessitating close monitoring of depletion dynamics.

6.4 Reinforcement Learning Integration

Exploring the integration of reinforcement learning (RL) into Phase 2 (Exploitation) presents a promising hybrid approach. RL could offer adaptive decision-making capabilities, allowing the algorithm to respond dynamically to evolving demand and inventory conditions. Practical implementation and performance benchmarking would determine whether this hybrid framework enhances flexibility and generalization across varied problem scenarios.

7. References

Main Reference Paper

- Chen, N., & Gallego, G. (2022). *A primal–dual learning algorithm for personalized dynamic pricing with an inventory constraint*. *Mathematics of Operations Research*, 47(4), 2585–2613. https://doi.org/10.48550/arXiv.1812.09234

Other References

- Gallego, G., & van Ryzin, G. (1994). *Optimal dynamic pricing of inventories with stochastic demand over finite horizons*. *Management Science*, 40(8), 999–1020.

- Besbes, O., & Zeevi, A. (2009). *Dynamic pricing without knowing the demand function: Risk bounds and near-optimal algorithms*. *Operations Research*, 57(6), 1407–1420. https://business.columbia.edu/faculty/research/dynamic-pricing-without-knowing-demand-function-risk-bounds-and-near-optimal

- Ferreira, K. J., Lee, B. H. A., & Simchi-Levi, D. (2016). *Analytics for an online retailer: Demand forecasting and price optimization*. *Manufacturing & Service Operations Management*, 18(1), 69–88. https://www.hbs.edu/ris/Publication%20Files/kris%20Analytics%20for%20an%20Online%20Retailer_6ef5f3e6-48e7-4923-a2d4-607d3a3d943c.pdf

- Javanmard, A., & Nazerzadeh, H. (2019). *Dynamic pricing in high-dimensions*. *Journal of Machine Learning Research*, 20(1), 1–49. https://jmlr.org/papers/volume20/17-357/17-357.pdf