DBA5113 Online Marketplace

Assignment 1

Ying WANG
Tanya TOOLEY

National University of Singapore

Submission Date: February 5, 2026

Link to code repository

Exercise 1: Ride-Sharing Platform Simulation

1. Generate Data

Note: The complete Python implementation for Exercise 1 is available in Appendix A.

We generate and simulate the positions of n riders and n drivers as random points with $n \in \{10, 30, 50\}$ respectively and the pairwise euclidean distances between all riders and drivers are calculated. The data generation is managed by function *generate_sample_data(n, rng)*.

2. Implement Matching Methods

We implement random matching, greedy matching and optimal matching methods (Hungarian algorithm) to match riders and drivers based on the generated distance data.

The three matching methods are implemented in functions:

- *random_matching(distance_matrix)*
- *greedy_matching(distance_matrix)*
- *optimal_matching(distance_matrix)*

3. Simulation

We run simulations for n = 10, 30, and 50 riders and drivers. For each n, we generate 1000 random instances of rider and driver locations, compute the total distance of each matching algorithm, and visualize the distribution of total distances.

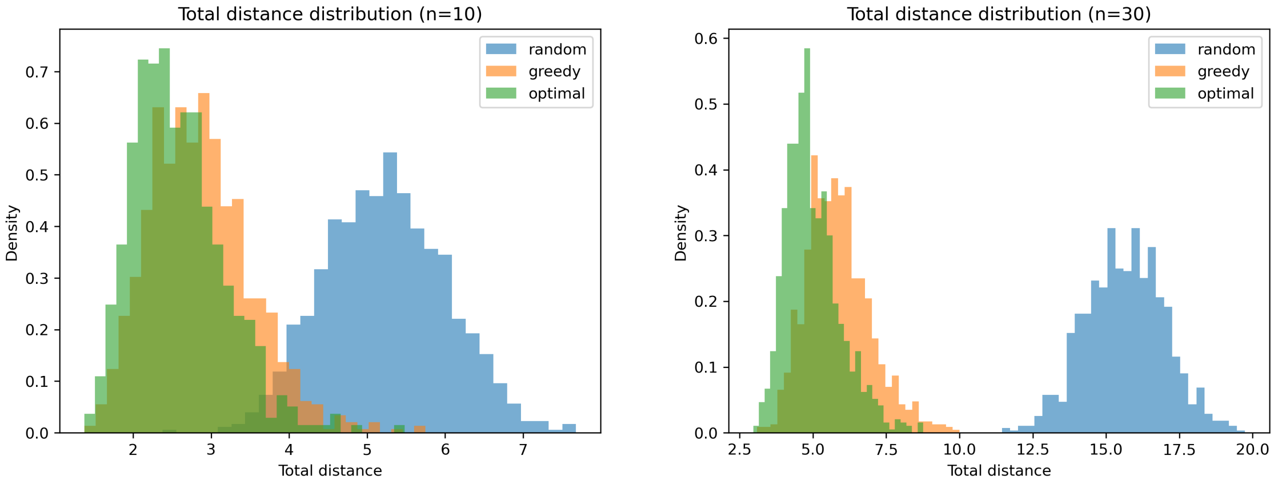

Figure 1: Total Distance Distribution for n=10 (left) and n=30 (right)

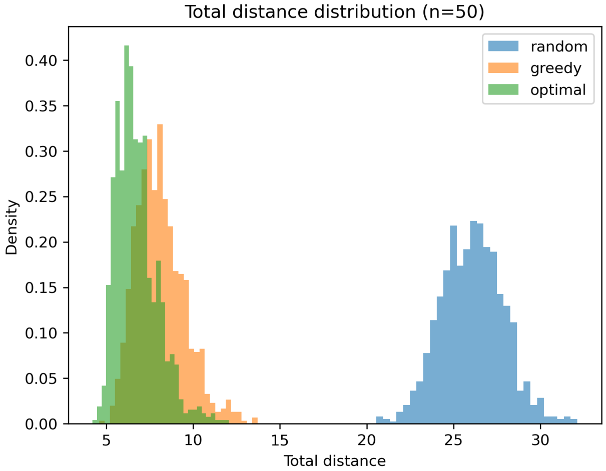

Figure 2: Total Distance Distribution for n=50

Method	**n=10**	**n=30**	**n=50**
Random	5.23	15.65	26.08
Greedy	2.86	5.86	8.11
Optimal	2.59	4.99	6.75

Table 1: Average Total Distance

4. Network Effects

When n scales by a factor of 3 (from 10 to 30), the average total distance for random matching increases by approximately 3 times, while the average total distance for greedy and optimal matching increases by approximately $\sqrt{3} \approx 1.73$ times (here we see 1.9–2.0 times).

This scaling behavior can be explained as follows: With n points uniformly distributed in a unit square, each point effectively "owns" an area of $1/n$. For optimized/heuristic matching algorithms (greedy and optimal), each rider is matched to a nearby driver within a region of side length $\sim 1/\sqrt{n}$. Thus, the expected distance per matched pair is proportional to $1/\sqrt{n}$, and the total distance across all n pairs is $n \times (1/\sqrt{n}) = \sqrt{n}$. When n increases by a factor of 3, the total distance increases by $\sqrt{3} \approx 1.73$, which aligns with the observed ratios of approximately 1.9–2.0 in our simulation results.

For random matching, since pairs are assigned without optimization, the expected distance between any randomly matched rider-driver pair is constant (approximately 0.5 for a unit square). Therefore, the total distance scales linearly as $O(n)$, explaining why tripling n roughly triples the total distance.

5. Endogeneity of n

The trade-off from increasing n is between **matching quality** and **waiting time**. As shown, when n is larger, the network effect is stronger, leading to shorter average pickup distances. However, increasing n also means customers must wait longer for n riders and n drivers to be batched together, especially during periods when demand or supply is sparse. This waiting time can lead to customer dissatisfaction and potential loss of business. Additionally, the optimal matching algorithm has $O(n^3)$ complexity, making computation more expensive as n grows.

Modeling the Trade-off

Let $W(n)$ denote the expected waiting time to batch n riders and n drivers, and let $D(n)$ denote the expected average pickup distance. From our analysis:

- $D(n) \propto \frac{1}{\sqrt{n}}$ (average distance per pair decreases with n)
- $W(n) \propto n$ (waiting time increases linearly with batch size)

A platform objective that accounts for this trade-off could be:

$$\min_n \quad \alpha \cdot D(n) + \beta \cdot W(n) = \alpha \cdot \frac{c_1}{\sqrt{n}} + \beta \cdot c_2 \cdot n$$

where α represents the cost per unit distance (fuel, driver time) and β represents the cost of customer waiting (dissatisfaction, churn risk). Taking the derivative and setting to zero:

$$\frac{d}{dn}\left(\frac{\alpha c_1}{\sqrt{n}} + \beta c_2 n\right) = -\frac{\alpha c_1}{2n^{3/2}} + \beta c_2 = 0$$

Solving for optimal n^*:

$$n^* = \left(\frac{\alpha c_1}{2\beta c_2}\right)^{2/3}$$

This shows that the optimal batch size depends on the relative importance of distance costs versus waiting costs. When distance costs dominate (α large), the platform should use larger batches. When waiting costs dominate (β large), smaller batches are preferred.

Note: Here waiting time is modeled as linear in n for tractability. In practice, it depends on demand/supply arrival rates and could be analyzed more precisely using queuing theory. But even with queuing theory, the waiting time is still proportional to n, so the equation should still be correct.

Exercise 2: Quality Selection

Note: The complete Python implementation for Exercise 2 is available in Appendix B.

1. Compute Demand

The plot for the two different density function is shown in Figure 3. We can see the two density functions have quite different tail behavior. $f_1(v) = 0.5v^{-1.5}$ has a heavier tail (slower decay), meaning more buyers have high valuations ($v \geq 2$). In contrast, $f_2(v) = 3v^{-4}$ decays much faster, concentrating most buyer mass near $v = 1$ (low valuations).

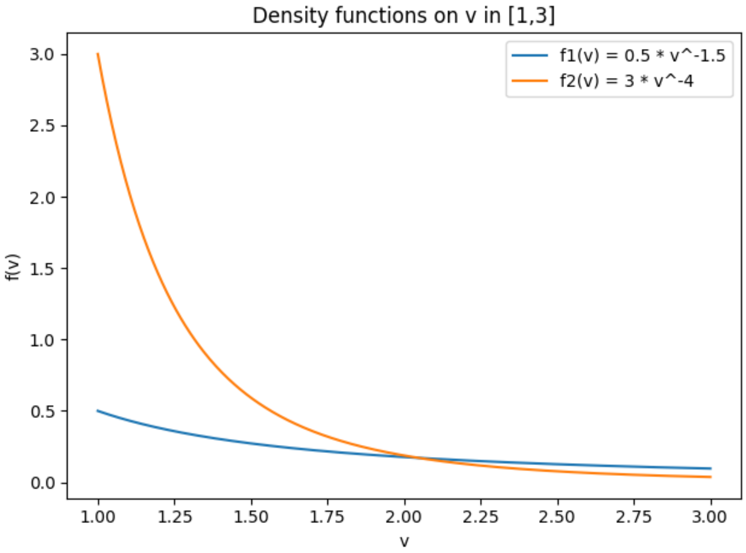

Figure 3: Density Functions $f_1(v)$ and $f_2(v)$

The demand for each menu and density function is computed using the provided code. The results are summarized below:

Density	**Menu**	D_L	D_H
$f_1(v) = 0.5v^{-1.5}$	Menu 1 (L+H)	0.1840	0.6325
	Menu 2 (H only)	—	0.7071
$f_2(v) = 3v^{-4}$	Menu 1 (L+H)	0.2323	0.0640
	Menu 2 (H only)	—	0.1250

Table 2: Demand Comparison for Different Density Functions and Menus

We can see for both functions, when we apply only high quality menu, some low quality menu buyers moved towards high quality menu, leading to high quality menu demand increase. This aligns with class content.

2. Revenue

The revenue for each menu and density function is computed using the provided code. The results are summarized below:

Density	**Menu**	**Revenue**
$f_1(v) = 0.5v^{-1.5}$	Menu 1 (L+H)	\$2.81
	Menu 2 (H only)	\$2.83
$f_2(v) = 3v^{-4}$	Menu 1 (L+H)	\$0.60
	Menu 2 (H only)	\$0.50

Table 3: Revenue Comparison for Different Density Functions and Menus

We can see that for f_1, the revenue is slightly higher when only offering the high quality menu, while for f_2, the revenue is lower. This relates to **price elasticity**: in this model, a buyer's valuation parameter v determines their willingness to pay for quality. High-v buyers are less price sensitive (inelastic), while low-v buyers are more price sensitive (elastic).

For f_1 (heavy tail), 70.7% of buyers have $v \geq 2.0$, meaning the population is predominantly **less price elastic**. When the L option is removed, buyers in $[2.0, 2.5)$ who previously bought L now upgrade to H at a higher price, and the platform doesn't lose much revenue.

For f_2 (light tail), 87.5% of buyers have $v < 2.0$, meaning the population is predominantly **more price elastic**. When the L option is removed, these low-v buyers cannot afford H and exit the market entirely. The revenue lost from these departing customers exceeds the gain from upgrading buyers, resulting in lower total revenue.

3. Different Demand Model

Our analysis uses a multiplicative utility model: $U(v, q, p) = v \cdot q - p$, where the quality benefit scales with the buyer's valuation parameter v. This model is well-suited for ridesharing platforms like Grab, but may be less appropriate for freelance marketplaces like Upwork.

Why the multiplicative model fits ridesharing:

- Standardized, easy-to-judge service

  Rideshare services are uniform and easily comparable: a ride from A to B is the same core service across all drivers. Quality dimensions (vehicle condition, cleanliness, driver courtesy) are straightforward to evaluate.

- Wide quality range

  The quality difference between JustGrab and GrabCar Premium is substantial and experiential, while the vehicle comfort, driver professionalism, and overall ride experience vary significantly.

- Heterogeneous valuations

Riders have diverse willingness to pay for comfort. A business traveler values premium features much more than a budget-conscious student, making the $v \cdot q$ interaction meaningful.

Why Upwork may require a different model:

- Diverse, non-standard services

  Unlike ridesharing, freelance work on Upwork spans vastly different domains (writing, coding, design, consulting) with no uniform quality metric. What constitutes "quality" varies entirely by project type, making it difficult for buyers to compare across sellers.

- Pre-screening and verification

  As discussed in class, platforms like Upwork pre-screen and verify sellers to ensure a minimum quality level. This creates a **quality floor** that compresses the effective quality range.

- Narrow quality range

  With pre-screening, the difference between "Top Rated" and "Standard" sellers is smaller than the difference between premium and basic rideshare services. Both tiers deliver professional-quality work.

- Additive model may be more appropriate

  When quality differences are small, an additive utility model $U = v + q - p$ (where quality provides a fixed bonus rather than scaling with v) may better capture buyer behavior. In this model, the premium for higher quality is relatively constant across buyers, rather than scaling with their valuation.

In summary, the multiplicative model captures markets where quality differences are large and experiential, while additive models may better fit markets where quality is more standardized due to platform curation.

Exercise 3: Surge Pricing Schemas

1. Trip Length Distribution

The Exponential Distribution Model assumes that trip lengths follow an exponential distribution $\lambda e^{-\lambda x}$, in the given example parameter $\lambda = 0.1$, meaning the mean trip length is $\mu = 1/\lambda = 10$ miles.

Note: The complete Python implementation for Exercise 3 is available in Appendix C.

1.1 Simulation Results

We generated 10,000 random trip lengths in order to have a significant representation from an exponential distribution with $\lambda = 0.1$. Figure 4 shows the histogram of simulated trip lengths overlaid with the theoretical probability density function (PDF):

$$f(x) = \lambda e^{-\lambda x} = 0.1e^{-0.1x}$$

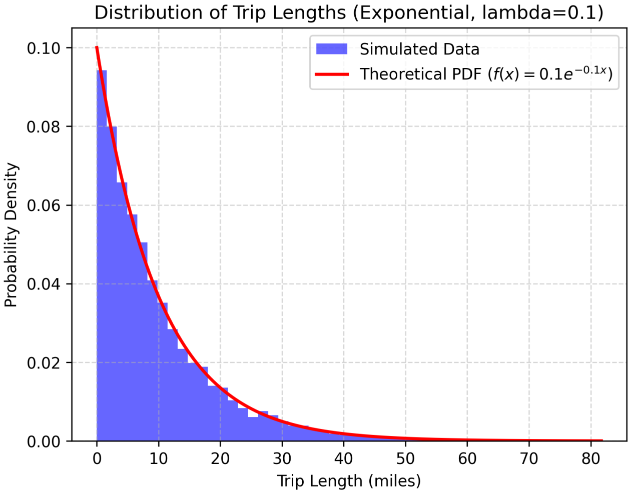

Figure 4: Distribution of Trip Lengths (Exponential, $\lambda = 0.1$)

The simulation confirms that the data follows the expected exponential distribution with sample mean = 9.77 which is close to the theoretical mean of 10 miles, matching the theoretical mean.

1.2 Results Analysis

a. Assumptions and Realism

The exponential distribution implies that short trips are most common, and the probability of a trip length decreases exponentially as the length increases. In ride-sharing, this assumption aligns with real-world data where the vast majority of rides are short commutes or last-mile connections (e.g., 2-5 miles), while very long trips (e.g., 20+ miles) are rare events.

b. Support from Literature

While the exponential distribution is widely used in modeling and simulation, the choice of distribution depends on spatial scale. Barbosa et al. (2018) [1] provide a comprehensive review showing that for short distances within a city, trip distributions can appear exponential (driven by cost and time constraints), but as the spatial scale increases to include suburbs or regional travel, Power Law or Log-Normal distributions provide superior fits.

For ride-sharing applications focused on intra-city urban trips (typically under 10 miles), the exponential distribution remains a reasonable and practical simplification, capturing the essential characteristic that shorter trips dominate the distribution.

c. Memoryless Property

While less physically intuitive for trips, the memoryless property simplifies the mathematics. It implies that the probability of a trip continuing for another mile doesn't depend on how far it has already gone (though in reality, trips do have destinations). Ideally, a Log-Normal distribution might fit better, but Exponential is a standard simplifying assumption for these problems.

2. Pricing Schemes Comparison

We compare two surge pricing mechanisms:

Multiplicative Surge Pricing:

$$\text{Payout} = \text{Base Fare} \times \text{Surge Multiplier}$$

Additive Surge Pricing:

$$\text{Payout} = \text{Base Fare} + (\text{Surge Multiplier} - 1) \times \text{Surge Bonus}$$

where Base Fare $= c + r \cdot x$, with c is a fixed constant fee, r is the per-mile rate, and x is the trip length. The Surge Bonus is a fixed amount added during surge periods.

2.1 Simulation Results

We generated 10,000 trips with:

- Trip lengths: Exponential distribution ($\lambda = 0.1$)
- Surge multipliers: $\{1.0, 1.5, 2.0, 2.5\}$ with probabilities $\{0.5, 0.3, 0.15, 0.05\}$
- Parameters: $c = 2.50$, $r = 0.75$, Surge Bonus $= 5.00$

Pricing Scheme	Mean Payout ($)	Variance
Multiplicative Surge	13.58	132.68
Additive Surge	11.74	58.37

Table 4: Pricing Scheme Statistics

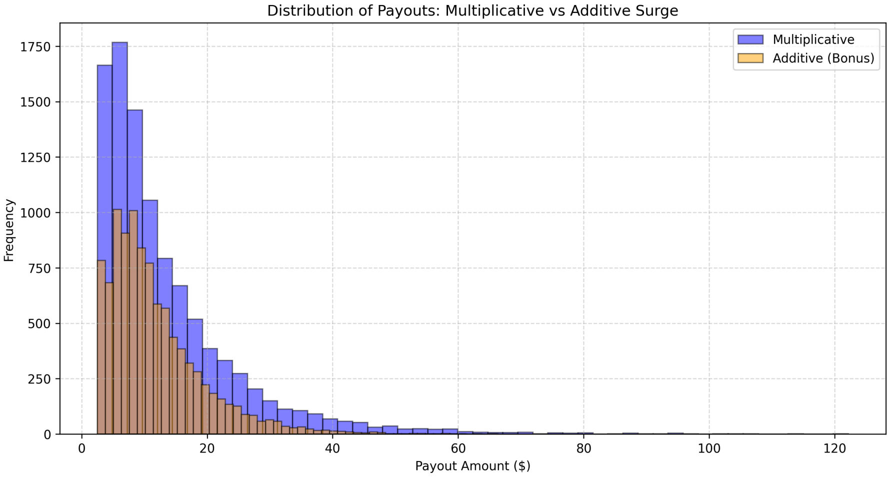

Figure 5: Distribution of Payouts: Multiplicative vs Additive Surge

2.2 Results Analysis

The results confirm that multiplicative surge pricing introduces significantly higher variability into the driver's earnings (or passenger's cost) compared to a flat bonus scheme.

Mathematical Reasoning:

- **Additive:** The payout is $\text{BaseFare} + (K - 1) \times \$5$, where K is the random surge multiplier. For independent variables, $\text{Var}(X + cY) = \text{Var}(X) + c^2\text{Var}(Y)$. Here, $\text{Var}(\text{Payout}) = \text{Var}(\text{BaseFare}) + 25 \cdot \text{Var}(K)$. Since the multiplier coefficient ($5) is fixed and relatively small, the surge component adds limited variance.
- **Multiplicative:** The payout is $K \times \text{BaseFare}$, the product of two random variables. For independent X and Y: $\text{Var}(XY) = E[X]^2\text{Var}(Y) + E[Y]^2\text{Var}(X) + \text{Var}(X)\text{Var}(Y)$. This formula shows that the variability of both components combines multiplicatively, leading to much higher total variance than the additive case.

This higher variance means that under multiplicative pricing, drivers face a wider range of possible earnings for the same trip length, leading to uncertainty and potential dissatisfaction.

3. Driver Strategic Behavior

(a) Impact of Multiplicative Surge Pricing:

In multiplicative surge pricing, drivers are more likely to cherry-pick rides. Since the variance is much higher, the difference in payout between long trips and short trips is significant. Drivers will ignore short rides to wait for a long one, especially during surge periods when the multiplier amplifies the base fare difference.

(b) Uber's Transition to Additive Pricing:

Moving to additive (flat bonus) pricing fixes this issue. The difference between long and short trips becomes relatively smaller. A +$5 bonus is proportionally huge for a short trip (making it worth taking) but just a nice extra for a long trip. This levels the playing field and ensures short trips don't get rejected during busy times.

By reducing variability and making short trips more attractive relative to their effort, the additive scheme encourages drivers to accept all rides rather than waiting for high-value trips, improving overall platform efficiency and customer satisfaction.

From a utility-maximization perspective, drivers accept trips when the utility (payout relative to effort) exceeds their reservation threshold. Under multiplicative pricing, the utility gap between short and long trips is substantial, making it rational for drivers to incur the opportunity cost of waiting for high-utility trips [2]. The additive scheme narrows this utility gap: a flat +$5 bonus represents a larger proportional utility gain for short trips than for long trips. When utility differences become sufficiently small, the marginal benefit of cherry-picking no longer justifies the waiting cost [3]. This aligns with rational inattention principles [4], where agents economize on cognitive effort when differences are small, leading drivers to accept whichever trip arrives first rather than strategically waiting.

Exercise 4: Spatial Pricing Optimization

Note: The complete Python implementation for Exercise 4 is available in Appendix D.

1. Optimize Prices: Spatial vs Uniform Pricing

We consider a simplified ride-sharing platform with two zones (Zone A and Zone B) with the following characteristics:

Demand Functions:

$$D_A(p) = \max(0, 100 - 10p)$$
$$D_B(p) = \max(0, 50 - 8p)$$

Supply Constraints:

$$S_A = 45 \text{ rides}$$
$$S_B = 35 \text{ rides}$$

Profit Function:

For each zone, the profit is:

$$\Pi_i(p_i) = p_i \cdot \min(D_i(p_i), S_i)$$

Results Comparison: The price optimization results for both uniform and spatial pricing strategies are summarized in Table 5.

Strategy	**Zone**	**Optimal Price ($)**	**Sales**	**Revenue ($)**
Spatial Pricing	A	5.50	45	247.50
	B	3.12	25	78.12
	Total	—	**70**	**325.62**
Uniform Pricing	Both	5.50	51	280.50
Profit Increase:	Spatial pricing yields $45.12 (16% increase)			

Table 5: Pricing Strategy Comparison

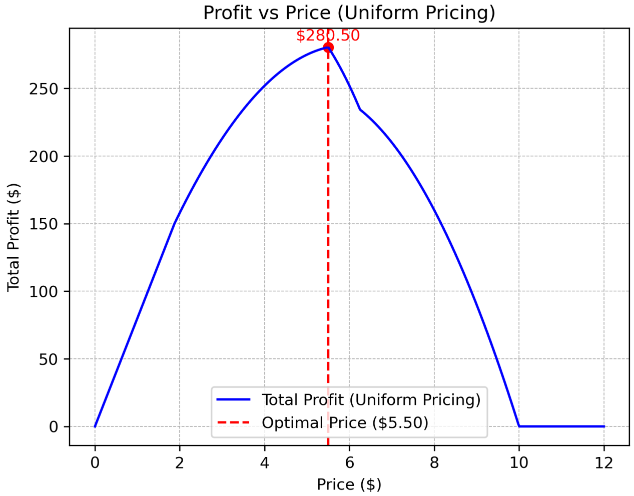

Figure 6: Profit vs Price (Uniform Pricing Strategy)

The analysis shows that spatial pricing can be beneficial when different zones have different demand elasticities. Zone A has higher demand and lower price sensitivity, while Zone B has lower demand but requires a lower price to maximize profit.

2. Estimate Demand and Optimize Prices

So instead of using known demand functions, we estimate demand using historical data and Random Forest regression. Since this is a non-linear method, we cannot use calculus to find optimal prices. Instead, we perform a grid search over possible prices to find the one that maximizes profit.

2.1 Explanation of the Estimation Step

This step uses Random Forest to **estimate the demand function** from historical data, rather than assuming a known mathematical formula or a linear model. Random Forest can capture non-linearities (e.g., demand dropping to zero sharply) and interactions (e.g., price sensitivity increasing during bad sentiment) automatically without manually specifying interaction terms like `price` × `sentiment`.

The Goal

We aim to understand how demand (D) responds to price (p) and other market factors. Since the true functional form is unknown (unlike the analytical case where $D = 100 - 10p$), we train a model to learn this relationship from data. While linear regression is an alternative approach, it assumes a linear relationship between features and demand, which may be too simplified to capture threshold effects, interaction terms, and other non-linear patterns commonly observed in real-world demand behavior.

The Method

The code uses a **Random Forest Regressor**, which is a non-linear supervised learning algorithm.

- Input (X): Price, Market Sentiment, Zone-Specific Features, Competitor Prices
- Output (y): Observed Demand

By running `.fit(X_train, y_train)`, the model analyzes past transactions to find patterns. It effectively creates a mathematical function $f(X)$ that can predict demand for any given price.

Validation

The code splits the data into a Training Set (80%) and a Test Set (20%).

- The model learns from the Training Set
- We test it on the Test Set to ensure it actually understands the general rule and didn't just memorize specific examples (overfitting)

Summary

"Estimation" here means using an algorithm to approximate the unknown relationship between price and demand so that we can use it for optimization later.

2.2 Price Optimization Using Estimated Demand

Using the trained Random Forest models, we optimize prices for both spatial and uniform pricing strategies.

Current Market Context:

- Market Sentiment: 1.0
- Zone A Specific: 2.1, Zone B Specific: 1.0
- Competitor Prices: Zone A = \$4.00, Zone B = \$6.00
- Supply: $S_A = 45$, $S_B = 45$

Optimization Results:

The Random Forest-based optimization results are summarized in Table 6.

Strategy	Zone	Optimal Price (\$)	Demand	Revenue (\$)
Spatial Pricing	A	5.30	40.26	213.37
	B	3.40	13.47	45.80
	Total	—	**53.73**	**259.17**
Uniform Pricing	Both	5.30	41.92	**222.19**
Profit Increase:	Spatial pricing yields \$36.98 (17% increase)			

Table 6: Random Forest-Based Pricing Strategy Comparison

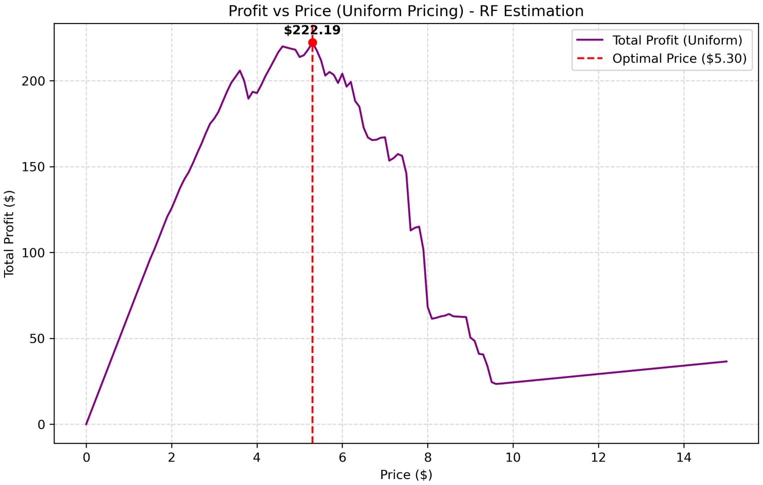

Figure 7: Profit vs Price (Uniform Pricing) - Random Forest Estimation

The Random Forest approach accounts for market conditions and competitor pricing, providing more realistic demand estimates than simple linear models.

3. Challenges in Scaling to Multiple Zones

When scaling this independent estimation + optimization approach to many zones, three main challenges arise:

a. **Optimization Computation Complexity (The "Curse of Dimensionality")**

In the current approach, we used grid search to check prices. This works fine for 1 or 2 zones. However, if you have N zones, the search space grows exponentially (PriceGridSize^N). This becomes computationally intractable for large-scale systems.

For example, with 100 zones and 151 price points each, we would need to evaluate 151^{100} combinations, which is impossible with brute force or simple grid search. We would need sophisticated optimization solvers (e.g., gradient-based methods, evolutionary algorithms) that can handle high-dimensional search spaces efficiently.

b. **Modeling Cross-Zone Spillover Effects**

The current code trains `rf_A` and `rf_B` independently, assuming Zone A's price doesn't affect Zone B. In a dense city with many zones, if Zone A is expensive, users might walk to Zone B or choose an alternative.

Demand in Zone i depends on prices in neighboring zones $j, k, l, \ldots$. Modeling these interactions rigorously requires estimating $N \times N$ cross-elasticities, which is:

- Data-intensive: Requires sufficient observations of all price combinations
- Computationally expensive: Training complex models with interaction terms
- Model complexity: A simple "one model per zone" approach fails to capture these substitution patterns

c. **Model Overfitting**

The rule of thumb says to train a Random Forest for every single zone, we need sufficient historical data (often exponential in size compare to feature numbers) for **each** zone covering various price points. In a system with thousands of zones:

- Many zones (e.g., quiet suburbs) will have very few data (few rides)
- Machine learning models like Random Forest perform poorly on little data
- We might end up with unreliable demand curves for low-traffic zones
- This leads to overfitting incorrect pricing decisions

Potential solutions include:

- **Zone-Specific Clustering:** Share information across similar zones, borrowing the clustering idea we used in class for A/B testing, before running demand estimation, we merge zones with similar characteristics to increase data size per model, and introduce more generation ability. This also bring an extra advantages of reducing the number of models to train, alleviating the computational burden.
- **Transfer learning:** Use data from high-traffic zones to inform low-traffic zones. So that we "increase" the data size for zones with little data.
- **Bayesian approaches:** For low data zone, incorporate prior knowledge and uncertainty quantification, instead of training from scratch.

References

[1] Barbosa, H., Barthelemy, M., Ghoshal, G., James, C. R., Lenormand, M., Louail, T., Menezes, R., Ramasco, J. J., Simini, F., and Tomasini, M. (2018). *Human Mobility: Models and Applications*. Physics Reports, 734, 1-74.

[2] Chen, M. K., and Sheldon, M. (2016). *Dynamic Pricing in a Labor Market: Surge Pricing and Flexible Work on the Uber Platform*. EC '16: Proceedings of the 2016 ACM Conference on Economics and Computation, 455-455.

[3] McFadden, D. (1974). *Conditional Logit Analysis of Qualitative Choice Behavior*. In P. Zarembka (Ed.), Frontiers in Econometrics (pp. 105-142). Academic Press.

[4] Sims, C. A. (2003). *Implications of Rational Inattention*. Journal of Monetary Economics, 50(3), 665-690.

A Exercise 1 Complete Code

Below is the complete Python code for Exercise 1, which simulates a ride-sharing platform with varying numbers of drivers and customers.

Listing 1: exercise1.py - Complete Implementation

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.spatial.distance import cdist
from scipy.optimize import linear_sum_assignment

def generate_positions(n, rng):
    return rng.random((n, 2))

def generate_sample_data(n, rng):
    """Generate a single sample of positions and distance matrix"""
    riders = generate_positions(n, rng)
    drivers = generate_positions(n, rng)
    dist = cdist(riders, drivers, metric="euclidean")
    return riders, drivers, dist

def random_matching(dist, rng):
    n = dist.shape[0]
    perm = rng.permutation(n)
    return dist[np.arange(n), perm].sum()

def greedy_matching(dist):
    n = dist.shape[0]
    remaining_rows = set(range(n))
    remaining_cols = set(range(n))
    total = 0.0

    while remaining_rows:
        best = (None, None, float("inf"))
        for i in remaining_rows:
            for j in remaining_cols:
                d = dist[i, j]
                if d < best[2]:
                    best = (i, j, d)

        i, j, d = best
        total += d
        remaining_rows.remove(i)
        remaining_cols.remove(j)

    return total

def optimal_matching(dist):
    r, c = linear_sum_assignment(dist)  #Using the Hungarian algorithm
    return dist[r, c].sum()

def run_simulations(n, sims=1000, seed=1):
    rng = np.random.default_rng(seed)
    out = {"random": [], "greedy": [], "optimal": []}

    for _ in range(sims):
        _, _, dist = generate_sample_data(n, rng)
        out["random"].append(random_matching(dist, rng))
        out["greedy"].append(greedy_matching(dist))
        out["optimal"].append(optimal_matching(dist))

    # converting to arrays
    return {k: np.array(v) for k, v in out.items()}

def plot_distributions(results_by_n, display=False):
    for n, res in results_by_n.items():
        plt.figure()
        for method in ["random", "greedy", "optimal"]:
            plt.hist(res[method], bins=30, density=True, alpha=0.6, label=method)
        plt.title(f"Total distance distribution (n={n})")
        plt.xlabel("Total distance")
        plt.ylabel("Density")
        plt.legend()
        if display:
            plt.show()
        else:
            plt.savefig(f"figures/total_distance_distribution_n_{n}.png", dpi=300, bbox_inches='tight')
            plt.close()

def main():
    # Generate sample distance matrix for n=10
    print("=== Sample Distance Matrix (n=10) ===")
    rng = np.random.default_rng(1)
    _, _, dist = generate_sample_data(10, rng)
    print(f"Sample distance matrix (first 5x5):\n{dist[:5, :5]}\n")

    # Run simulations
    results = {
        10: run_simulations(10, sims=1000, seed=1),
        30: run_simulations(30, sims=1000, seed=2),
        50: run_simulations(50, sims=1000, seed=3),
    }

    # Printing averages and graph
    for n in [10, 30, 50]:
        print(f"\n=== n={n} ===")
        for method in ["random", "greedy", "optimal"]:
            print(f"{method:>7}: mean={results[n][method].mean():.4f}")

    plot_distributions(results)

if __name__ == "__main__":
    main()
```

B Exercise 2 Complete Code

Below is the complete Python code for Exercise 2, which computes demand based on different density functions.

Listing 2: exercise2.py - Complete Implementation

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import quad

def demand_menu1(f, v_L, v_H, v_LH):
    # Demand for L: buyers in [v_L, v_LH) who prefer L
    if v_L < v_LH:
        DL = quad(f, v_L, v_LH)[0]
    else:
        DL = 0.0  # L is never preferred
    # Demand for H: buyers >= v_LH who can afford H
    # They must also have v >= v_H for positive utility from H
    lower_H = max(v_LH, v_H)
    DH = quad(f, lower_H, np.inf)[0]
    return DL, DH

def demand_menu2(f, v_H):
    DH = quad(f, v_H, np.inf)[0]
    return DH

def revenue_menu1(pL, DL, pH, DH):
    return pL * DL + pH * DH

def revenue_menu2(pH, DH):
    return pH * DH

# The density functions
def f1(v):
    return 0.5 * v ** (-1.5)

def f2(v):
    return 3.0 * v ** (-4.0)

def plot_density_functions(display=False):
    # Compute Demand, and plotting the density functions on [1,3]
    vs = np.linspace(1.0, 3.0, 400)
    plt.figure()
    plt.plot(vs, [f1(v) for v in vs], label="f1(v) = 0.5 * v^-1.5")
    plt.plot(vs, [f2(v) for v in vs], label="f2(v) = 3 * v^-4")
    plt.title("Density functions on v in [1,3]")
    plt.xlabel("v")
    plt.ylabel("f(v)")
    plt.legend()
    plt.tight_layout()
    if display:
        plt.show()
    else:
        plt.savefig("figures/density_functions.png")
        plt.close()

def find_thresholds(pL, qL, pH, qH):
    # v_L: threshold where buyer is indifferent between L and not buying
    v_L = pL / qL   # = 1.5/1 = 1.5
    if v_L < 1.0:
        v_L = 1.0
    # v_H: threshold where buyer is indifferent between H and not buying
    v_H = pH / qH   # = 4/2 = 2.0
    if v_H < 1.0:
        v_H = 1.0
    # v_LH: threshold where buyer is indifferent between L and H
    v_LH = (pH - pL) / (qH - qL)   # = (4-1.5)/(2-1) = 2.5
    if v_LH < 1.0:
        v_LH = 1.0
    return v_L, v_H, v_LH

def main():
    plot_density_functions()

    # find the threshold v where user would chose between menus
    pL, qL, pH, qH = 1.5, 1.0, 4.0, 2.0
    v_L, v_H, v_LH = find_thresholds(pL, qL, pH, qH)

    demand_menu1_f1 = demand_menu1(f1, v_L, v_H, v_LH)
    demand_menu2_f1 = demand_menu2(f1, v_H)

    demand_menu1_f2 = demand_menu1(f2, v_L, v_H, v_LH)
    demand_menu2_f2 = demand_menu2(f2, v_H)

    print(f"\nThe demand results for \nf1 with menu 1 is: DL = {
       demand_menu1_f1[0]:.4f}, DH = {demand_menu1_f1[1]:.4f}. \nf1
       with menu 2 is: DH = {demand_menu2_f1:.4f}. \nf2 with menu 1 is
       : DL = {demand_menu1_f2[0]:.4f}, DH = {demand_menu1_f2[1]:.4f}.
        \nf2 with menu 2 is: DH = {demand_menu2_f2:.4f}.\n")

    revenue_f1_menu1 = revenue_menu1(pL, demand_menu1_f1[0], pH,
       demand_menu1_f1[1])
    revenue_f1_menu2 = revenue_menu2(pH, demand_menu2_f1)

    revenue_f2_menu1 = revenue_menu1(pL, demand_menu1_f2[0], pH,
       demand_menu1_f2[1])
    revenue_f2_menu2 = revenue_menu2(pH, demand_menu2_f2)

    print(f"The revenue results for \nf1 with menu 1 is: R = {
       revenue_f1_menu1:.4f}. \nf1 with menu 2 is: R = {
       revenue_f1_menu2:.4f}. \nf2 with menu 1 is: R = {
       revenue_f2_menu1:.4f}. \nf2 with menu 2 is: R = {
       revenue_f2_menu2:.4f}.\n")

if __name__ == "__main__":
    main()
```

C Exercise 3 Complete Code

Below is the complete Python code for Exercise 3, which generates the trip length distribution and pricing scheme comparison visualizations:

Listing 3: exercise3.py - Complete Implementation

```
import numpy as np
import matplotlib.pyplot as plt
import os

np.random.seed(42)
# create figures directory if it doesn't exist
os.makedirs('figures', exist_ok=True)

def get_trip_length(lambd=0.1, n_samples=10000, display_plot=False):
    mean_trip = 1 / lambd
    trip_lengths = np.random.exponential(scale=mean_trip, size=
       n_samples)
    _, _, _ = plt.hist(trip_lengths, bins=50, density=True, alpha=0.6,
        color='b', label='Simulated Data')

    # Plot the theoretical PDF as baseline f(x) = lambda * exp(-lambda
        * x)
    x = np.linspace(0, max(trip_lengths), 1000)
    pdf = lambd * np.exp(-lambd * x)
    plt.plot(x, pdf, linewidth=2, color='r', label='Theoretical PDF ($f(x) = 0.1 e^{-0.1x}$)')

    plt.title('Distribution of Trip Lengths (Exponential, lambda=0.1)')
    plt.xlabel('Trip Length (miles)')
    plt.ylabel('Probability Density')
    plt.legend()
    plt.grid(True, linestyle='--', alpha=0.5)

    # Sanity check
    print(f"Sample Mean: {np.mean(trip_lengths):.2f} (Target: 10.00)")
    if display_plot:
        plt.show()
    else:
        plt.savefig('figures/trip_length_distribution.png', dpi=300, bbox_inches='tight')
        plt.close()

    return trip_lengths

def compare_pricing_schemes(n_trips=10000, lambd=0.1, c=2.50, r=0.75, surge_bonus=5.00, display_plot=False):
    # 1. Generate Trip Lengths
    trip_lengths = get_trip_length(lambd, n_trips, display_plot)

    # 2. Generate Surge Multipliers with probability given
    # Values: {1.0, 1.5, 2.0, 2.5}
    # Probs:  {0.5, 0.3, 0.15, 0.05}
    multipliers = np.random.choice(
        [1.0, 1.5, 2.0, 2.5],
        size=n_trips,
        p=[0.5, 0.3, 0.15, 0.05]
    )

    # Base Fare (Cost)
    base_fares = c + r * trip_lengths

    # Scheme A: Multiplicative Surge
    payouts_mult = multipliers * base_fares

    # Scheme B: Additive Surge (Bonus)
    payouts_add = base_fares + (multipliers - 1) * surge_bonus

    # Analysis mean and variance of each Schema
    mean_mult = np.mean(payouts_mult)
    var_mult = np.var(payouts_mult)

```

```python
    mean_add = np.mean(payouts_add)
    var_add = np.var(payouts_add)

    print(f"For Mulplicative Surge Pricing: Mean Payout: ${mean_mult:.2f}, Variance: {var_mult:.2f}")
    print(f"For Additive Surge Pricing: Mean Payout: ${mean_add:.2f}, Variance: {var_add:.2f}")

    # Plotting
    plt.figure(figsize=(12, 6))
    plt.hist(payouts_mult, bins=50, alpha=0.5, label='Multiplicative', color='blue', edgecolor='black')
    plt.hist(payouts_add, bins=50, alpha=0.5, label='Additive (Bonus)', color='orange', edgecolor='black')

    plt.title('Distribution of Payouts: Multiplicative vs Additive Surge')
    plt.xlabel('Payout Amount ($)')
    plt.ylabel('Frequency')
    plt.legend()
    plt.grid(True, linestyle='--', alpha=0.5)
    if display_plot:
        plt.show()
    else:
        plt.savefig('figures/pricing_scheme_comparison.png', dpi=300, bbox_inches='tight')
        plt.close()
    return payouts_mult, payouts_add

if __name__ == "__main__":
    # Compare pricing schemes and save the figure
    compare_pricing_schemes(display_plot=False)
```

D Exercise 4 Complete Code

Below is the complete Python code for Exercise 4, which implements spatial vs uniform pricing optimization using both analytical demand functions and Random Forest demand estimation:

Listing 4: exercise4.py - Complete Implementation

```python
import numpy as np
from scipy.optimize import minimize_scalar
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split

def demand_A(p):
    return max(0, 100 - 10 * p)

def demand_B(p):
    return max(0, 50 - 8 * p)

# Objective functions for minimization (negative profit)
def neg_profit_A(p, supply_A):
    d = demand_A(p)
    q = min(d, supply_A)
    return -(p * q)

def neg_profit_B(p, supply_B):
    d = demand_B(p)
    q = min(d, supply_B)
    return -(p * q)

def neg_total_profit_uniform(p, supply_A=45, supply_B=35):
    return neg_profit_A(p, supply_A) + neg_profit_B(p, supply_B)

def optimize_spatial(supply_A=45, supply_B=35):
    # Optimize Zone A (Demand A becomes 0 at p=10)
    res_A = minimize_scalar(neg_profit_A, bounds=(0, 10), method='
        bounded', args=(supply_A,))
    opt_p_A = res_A.x
    opt_rev_A = -res_A.fun

    # Optimize Zone B (Demand B becomes 0 at p=6.25)
    res_B = minimize_scalar(neg_profit_B, bounds=(0, 6.25), method='
        bounded', args=(supply_B,))
    opt_p_B = res_B.x
    opt_rev_B = -res_B.fun

    total_profit = opt_rev_A + opt_rev_B
    sales_A = min(demand_A(opt_p_A), supply_A)
    sales_B = min(demand_B(opt_p_B), supply_B)
    print(f"Optimal Price Zone A: ${opt_p_A:.2f}, Sales Zone A: {sales_A:.2f}, Optimal Revenue Zone A: ${opt_rev_A:.2f}")
    print(f"Optimal Price Zone B: ${opt_p_B:.2f}, Sales Zone B: {sales_B:.2f}, Optimal Revenue Zone B: ${opt_rev_B:.2f}")
    print(f"Total Profit: ${total_profit:.2f}")

def optimize_uniform(display=False):
    res = minimize_scalar(neg_total_profit_uniform, bounds=(0, 10), method='bounded')
    opt_p = res.x
    opt_rev = -res.fun
    sales = min(demand_A(opt_p), 45) + min(demand_B(opt_p), 35)
    print(f"Optimal Uniform Price: ${opt_p:.2f}, Sales at Uniform price: {sales:.2f}, Total Profit: ${opt_rev:.2f}")
    prices = np.linspace(0, 12, 500)
    profits = [-neg_total_profit_uniform(p) for p in prices]

    plt.plot(prices, profits, label='Total Profit (Uniform Pricing)', color='blue')
    plt.title('Profit vs Price (Uniform Pricing)')
    plt.xlabel('Price ($)')
    plt.ylabel('Total Profit ($)')
    plt.grid(True, which='both', linestyle='--', linewidth=0.5)

    # Highlight optimal price
    opt_profit = -neg_total_profit_uniform(opt_p)
    plt.axvline(x=opt_p, color='red', linestyle='--', label=f'Optimal Price (${opt_p:.2f})')
    plt.scatter([opt_p], [opt_profit], color='red')
    plt.text(opt_p, opt_profit + 5, f'${opt_profit:.2f}', ha='center', color='red')

    plt.legend()
    if display:
        plt.show()
    else:
        plt.savefig('figures/uniform_pricing_profit.png', dpi=300, bbox_inches='tight')
        plt.close()

def train_demand_models_with_rf():
    data_sample = pd.read_csv("./historical_data_with_demand_v2.csv")
    features_A = ["price_A", "market_sentiment", "zone_specific_A", "competitor_prices_A"]
    features_B = ["price_B", "market_sentiment", "zone_specific_B", "competitor_prices_B"]
    X_A = data_sample[features_A]
    y_A = data_sample["demand_A"]
    X_B = data_sample[features_B]
    y_B = data_sample["demand_B"]
    # Train-test split
    X_train_A, X_test_A, y_train_A, y_test_A = train_test_split(X_A, y_A, test_size=0.2, random_state=42)
    X_train_B, X_test_B, y_train_B, y_test_B = train_test_split(X_B, y_B, test_size=0.2, random_state=42)
    # Random Forest Regressor
    rf_A = RandomForestRegressor(n_estimators=100, random_state=42)
    rf_B = RandomForestRegressor(n_estimators=100, random_state=42)
    rf_A.fit(X_train_A, y_train_A)
    rf_B.fit(X_train_B, y_train_B)

    score_A = rf_A.score(X_test_A, y_test_A)
    score_B = rf_B.score(X_test_B, y_test_B)
    print(f"\nRandom Forest Model Performance: Zone A R^2 Score: {score_A:.3f}, Zone B R^2 Score: {score_B:.3f}.\n")
    return rf_A, rf_B

def get_demand_A_rf(rf_A, price, market_context):
    # Create a DataFrame with the exact feature names used in training
    # features_A = ["price_A", "market_sentiment", "zone_specific_A", "competitor_prices_A"]
    X_pred = pd.DataFrame({
        "price_A": [price],
        "market_sentiment": [market_context["market_sentiment"]],
        "zone_specific_A": [market_context["zone_specific_A"]],
        "competitor_prices_A": [market_context["competitor_prices_A"]]
    })
    # rf_A is the trained model from previous cell
    predicted_demand = rf_A.predict(X_pred)[0]
    return max(0, predicted_demand)

def get_demand_B_rf(rf_B, price, market_context):
    # features_B = ["price_B", "market_sentiment", "zone_specific_B", "competitor_prices_B"]
    X_pred = pd.DataFrame({
        "price_B": [price],
        "market_sentiment": [market_context["market_sentiment"]],
        "zone_specific_B": [market_context["zone_specific_B"]],
        "competitor_prices_B": [market_context["competitor_prices_B"]]
    })
    predicted_demand = rf_B.predict(X_pred)[0]
    return max(0, predicted_demand)

def optimize_spatial_rf(rf_A, rf_B, price_grid, market_context, S_A=45, S_B=45):
    # ---------------- Optimization (Grid Search) ----------------
    # 1. Spatial Pricing (Optimize independently)
    best_pA = 0
    max_rev_A = 0
    max_demand_A = 0
    for p in price_grid:
        d = get_demand_A_rf(rf_A, p, market_context)
        rev = p * min(d, S_A)
        if rev > max_rev_A:
            max_rev_A = rev
            best_pA = p
            max_demand_A = d

    best_pB = 0
    max_rev_B = 0
    max_demand_B = 0
    for p in price_grid:
        d = get_demand_B_rf(rf_B, p, market_context)
        rev = p * min(d, S_B)
        if rev > max_rev_B:
            max_rev_B = rev
            best_pB = p
            max_demand_B = d

    print(f"Optimal Price Zone A: ${best_pA:.2f}, Demand A at optimal price: {max_demand_A:.2f}, Optimal Revenue Zone A: ${max_rev_A:.2f}")
    print(f"Optimal Price Zone B: ${best_pB:.2f}, Demand B at optimal price: {max_demand_B:.2f}, Optimal Revenue Zone B: ${max_rev_B:.2f}")
    print(f"Total Profit: ${max_rev_A + max_rev_B:.2f}")

def optimize_uniform_rf(rf_A, rf_B, price_grid, market_context, S_A=45, S_B=45, display=False):
    # Uniform Pricing (Optimize pA = pB = p)
    best_p_uniform = 0
    max_rev_uniform = 0
    max_sales_A_uniform = 0
    max_sales_B_uniform = 0
    uniform_profits = []

    for p in price_grid:
        a = min(get_demand_A_rf(rf_A, p, market_context), S_A)
        b = min(get_demand_B_rf(rf_B, p, market_context), S_B)
        prof = p * a + p * b
        uniform_profits.append(prof)
        if prof > max_rev_uniform:
            max_rev_uniform = prof
            best_p_uniform = p
            max_sales_A_uniform = a
            max_sales_B_uniform = b
```

```python

    print(f"Optimal Price: ${best_p_uniform:.2f}, Sales at optimal price: {max_sales_A_uniform + max_sales_B_uniform:.2f}, Total Profit: ${max_rev_uniform:.2f}")

    plt.figure(figsize=(10, 6))
    plt.plot(price_grid, uniform_profits, label=’Total Profit (Uniform)’, color=’purple’)
    plt.axvline(x=best_p_uniform, color=’red’, linestyle=’--’, label=f’Optimal Price (${best_p_uniform:.2f})’)
    plt.scatter([best_p_uniform], [max_rev_uniform], color=’red’, zorder=5)
    plt.text(best_p_uniform, max_rev_uniform + 5, f"${max_rev_uniform:.2f}", ha=’center’, color=’black’, fontdict={’weight’: ’bold’})

    plt.title(’Profit vs Price (Uniform Pricing) - RF Estimation’)
    plt.xlabel(’Price ($)’)
    plt.ylabel(’Total Profit ($)’)
    plt.legend()
    plt.grid(True, linestyle=’--’, alpha=0.5)
    if display:
        plt.show()
    else:
        plt.savefig(’figures/uniform_pricing_rf_profit.png’, dpi=300, bbox_inches=’tight’)
        plt.close()

def main():
    # Define market context
    market_context = {
        "market_sentiment": 1.0,
        "zone_specific_A": 2.1,
        "zone_specific_B": 1.0,
        "competitor_prices_A": 4.0,
        "competitor_prices_B": 6.0
    }

    # Profit optimization with known demand functions
    optimize_spatial()
    optimize_uniform()

    # Profit optimization with Random Forest demand estimation
    rf_A, rf_B = train_demand_models_with_rf()
    price_grid = np.linspace(0, 15, 151)  # Check every $0.10 from $0 to $15
    optimize_spatial_rf(rf_A, rf_B, price_grid, market_context)
    optimize_uniform_rf(rf_A, rf_B, price_grid, market_context, display=False)
```

```python

if __name__ == "__main__":
    main()
```