Pricing Optimization Using MNL Model

Wenbin Wan, Rachel Lin, Serena Cheng, Zikai Liu

December 9, 2024

Executive Summary

Airbnb aims to maximize expected revenue as a rental home listing platform by meticulously selecting which properties to list on top to attract customers. Airbnb charges a certain percentage for each booking, thus having the most appealing houses that match the customer demand listed as the top options gives a higher chance of successful booking.

This project addresses the decision problem of selecting and pricing a subset of Airbnb listings to maximize expected platform revenue. The methodological approach integrates three key stages:

- **Demand Prediction**: We first estimate how listing attributes (e.g., superhost status, host response rate, amenities, price, and quality signals from reviews) correlate with demand using Maximum Likelihood Estimation through linear regression.
- **Multinomial Logit (MNL) Choice Modeling**: Using the theory of discrete choice [Train, 2009], we assume that customers belong to distinct segments based on desired room size and select listings according to an MNL model incorporating utility parameters influenced by observed listing attributes.
- **Constrained Optimization**: With estimated utilities and predicted choice probabilities, we solve a Mixed-Integer Linear Program (MILP) to determine an optimal subset of listings and their prices to maximize expected revenue.

We compare this MNL-based optimization approach against a simplified approximation (referred to as "Simple 1-m Approximation") that selects properties solely based on sorting by potential revenue (price). The proposed MNL-based model captures the complexity of customer substitution, providing more nuanced and potentially more profitable recommendations than the simplistic approach.

Background and Rationale

The short-term rental industry, represented by Airbnb, exemplifies a marketplace that dynamically matches diverse options of accommodation supply to a continuously evolving demand. Customer behavior in this setting is influenced by different factors, including the timing of travel (seasonality), property characteristics (location, amenities), host reputation (reviews, response rate), and most importantly, price. At the same time, hosts have diverse listing attributes, and individual guests have varied willingness-to-pay thresholds and preferences for certain property types.

Existing research on pricing in two-sided marketplaces often leverages predictive modeling of demand. Techniques usually involve machine learning methods, such as linear regression and random forests [Rezazadeh Kalehbasti et al., 2021]. However, demand prediction alone does not solve the strategic problem of how a platform should select and price properties to maximize overall revenue. Complexities arise because listings are not chosen in isolation: the inclusion of one property at a particular price point alters the relative attractiveness of others, affecting choice probabilities and, ultimately, expected revenue. Previous work in revenue management has shown that discrete choice models, particularly the Multinomial Logit (MNL) model, effectively capture these substitution effects. The MNL model posits that the probability of choosing a particular listing is determined by the exponential of its utility (derived from estimated parameters) divided by the sum of exponentials of utilities across all available options in that segment. This choice model can be incorporated into a mixed-integer linear programming framework to optimize the selection of properties and their price points, which can be efficiently solved using commercial solvers such as Gurobi.

Data Collection

We found Airbnb listing data for Albany, NY on Inside Airbnb, which provides detailed listing information, calendar data, geographical mapping, and review data for the properties. For the purposes of our modeling approach, we extract key variables that influence customer utility: host response rate, superhost status, identity verified status, normalized review score, and price of the property. We use the number of reviews as a proxy for realized demand (sales). Future work could integrate actual booking data or external demand signals.

Properties are segmented into three categories (small, medium, large) based on accommodation capacity. Given a total market demand, we assume fixed proportions of customers preferring each room size (e.g., 30% small, 50% medium, 20% large). Within each segment, customers choose among available listings according to the MNL model. We further assume a base price of $50 per person per night to define reasonable price ranges for the optimization framework.

Method

Utility Parameters Estimation

We begin by approximating utility parameters for the MNL model. Ideally, we would estimate these parameters directly using Maximum Likelihood Estimation (MLE) on individual-level choice data. However, due to data constraints and the absence of fully observed choice sets and actual bookings, we adopt a practical approximation. We employ a linear regression model using listing-level features such as host response rate and price, the number of reviews—as a noisy proxy for demand (sales). This step does not yield true MNL parameters; instead, it provides a set of coefficients that we later rescale to form a heuristic approximation of utilities. The resulting "utility parameters" must be interpreted cautiously as indicative preference weights rather than statistically rigorous MNL coefficients. For a comprehensive discussion on this limitation and its implications, refer to the Discussion.

MNL Choice Modeling

The MNL model posits that customers choose among a set of N alternatives by comparing their utilities, for property i at price P_i:

$$U_i = \beta_0 + \beta_1 R_i + \beta_2 S_i + \beta_3 I_i + \beta_4 Q_i + \beta_5 P_i,$$

where R_i is host response rate, S_i is a binary indicator for superhost status, I_i is an identity verification indicator, Q_i is a normalized review score, and P_i is the price. The parameters $\{\beta_k\}$ are derived from our regression-based approximation.

	Raw Coefficient (Regression)	Normalized Parameter
host_response_rate	3.764	0.37645
is_superhost	5.918	0.59176
identity_verified	7.682	0.76818
review_score	21.931	2.19314
price	-4.748	-0.00475

Table 1: Raw Coefficients and Normalized Utility Parameters
The parameters presented here are not directly interpretable as MNL utility coefficients

For each room-size segment, we assume a fixed proportion of total demand and model the probability that a customer in that segment selects property i at price P_i as:

$$\phi_i = \frac{\exp(U_i)}{\sum_{k \in \text{Size}(i)} \exp(U_k)},$$

where Size(i) represents the set of properties available in the same size category as property i. We abstract away the outside option here for simplicity, noting that in practice customers may choose not to book at all.

Linearization of Choice Probabilities: To maintain linearity in the objective function, we approximate the denominator (the sum of exponential utilities $\exp(U_k)$ within each room-size category) as constant. This is accomplished through:

1. Pre-computing all utilities U_i using the estimated regression coefficients
2. Normalizing the probabilities within each size group

This approximation allows the objective function in the optimization formulation to remain linear. The Independence of Irrelevant Alternatives (IIA) assumption is applied across groups such that the relative odds of choosing between any two rooms of different sizes are independent. Defining new continuous variables to represent the probabilities and handle the nonlinearity by evaluating every possible assortment combination can enhance the accuracy of the optimization model.

For each property i, we establish a feasible price range based on its accommodation capacity and a base price of \$50 per person per night. The ceiling and floor prices are calculated as $\text{Ceiling}_i = 50 \cdot A_i \cdot (1 + r)$ and $\text{Floor}_i = 50 \cdot A_i \cdot (1 - r)$ respectively, where A_i is the property's accommodation capacity, and r is a fixed ratio (e.g., 0.2) determining the allowable price deviation. Within this range, we generate s equally spaced price points $P_{i,j}$. For instance, a two-person property with $B = \$50$ and $r = 0.2$ would have a price range of [\$80, \$120], which could be divided into discrete steps (e.g., \$80, \$84, ..., \$120).

For property i and price step j, the utility for each property-price combination is then adjusted as

$$U_{i,j} = U_i + \beta_5 \cdot P_{i,j}$$

where β_5 is the price sensitivity coefficient.

Optimization Formulation

The optimization formulates a Mixed-Integer Linear Programming (MILP) model and uses Gurobi to select [Udwani, 2023]:

- A subset of properties within each room-size category, subject to upper-bound constraints on the number of properties per category.

- A price point for each selected property (from a discrete set of candidate price points).

Decision Variables Let $y_{i,j} \in \{0, 1\}$ denote whether house i is selected at price p_j

Objective Function The objective is to maximize the expected revenue generated from bookings across all selected properties and their assigned price points defined as:

$$\max \sum_{\text{size}\in\{\text{small},\text{medium},\text{large}\}} \alpha_{\text{size}} \cdot D \sum_{i\in\text{Size(size)}} \sum_{j} p_{i,j} \cdot y_{i,j} \cdot \phi_{i,j}$$

Where:

- α_{size} is the probability of a customer belonging to the room-size segment $size$
- D represents the total demand (number of bookings)
- $p_{i,j}$ is the price point j assigned to property i
- $\phi_{i,j}$ is the MNL choice probability of property i being selected at price at price $p_{i,j}$

The optimization model incorporates the following constraints:

1. Room Size Constraints

$$\sum_{i\in\text{Size(size)}} \sum_{j} y_{i,j} \leq k_{\text{size}}, \quad \forall \text{size} \in \{\text{small}, \text{medium}, \text{large}\}$$

Where $k_{\text{size}} = \alpha_{\text{size}} \cdot D$ ensures that the number of selected properties in each room-size category does not exceed the maximum allowable based on demand allocation (cardinality constraint). For our model we assume $\alpha_{small} = 0.5$, $\alpha_{medium} = 0.3$ and $\alpha_{large} = 0.2$.

2. Price Selection Constraints

$$\sum_{j} y_{i,j} \leq 1, \quad \forall i, j$$

This constraint ensures that each selected property is assigned exactly one price point from a predefined set of candidate prices.

3. Binary Variable Constraints

$$y_{i,j} \in \{0, 1\}, \quad \forall i, j$$

Discussions

Model Selection and Estimation Challenges

Alternatively, we considered employing a Maximum Likelihood Estimation (MLE) approach to directly estimate the Multinomial Logit (MNL) model parameters using simulated discrete choice data. In this simulation, we assumed multiple price points for each listing and predefined choice probabilities to reflect customer booking probability under varying price scenarios. However, this approach encountered significant challenges. The primary limitation arises from the lack of transactional choice data, as we only have listing-level data without detailed information on individual customer choices. Additionally, the limited variation in attributes other than price within each listing's alternatives hinders the model's ability to accurately identify the influence of features such as host response rate, superhost status, and review scores on utility. Consequently, the MLE-based MNL model was unable to reliably recover meaningful utility parameters, often resulting in negligible estimates for non-price attributes in our experiment. On the other hand, the linear regression approach we used does not explicitly model the choice context. It treats the problem as a continuous-response regression rather than a discrete-choice estimation, thereby preventing the interpretation of the parameters as true MNL utility coefficients.

Results and Future Work

In this study, we aimed to optimize Airbnb's listing selection and pricing to maximize expected revenue. We adopted a two-step approach to address the absence of customer choice data. First, we estimated utility parameters through regression. The resulting estimated parameters serve as proxies for customer preferences, providing the foundation for implementing the subsequent choice model. Second, we applied a Multinomial Logit (MNL) choice model to compute the probability of a property being selected at different price points. We assumed that the denominator of the choice probabilities—the sum of exponentiated utilities—was constant within each room size category, simplifying the optimization problem.

Our optimization results demonstrate that integrating property features into pricing decisions can significantly enhance revenue generation. The model identified optimal average prices of \$138.00 for small properties, \$208.00 for medium properties, and \$417.60 for large properties, achieving a total revenue of \$21,492 under a demand scenario of 100 bookings. This outcome is comparable to the revenue generated by a simpler strategy of selecting the most expensive property in each category, which achieved \$22,794.

However, the methodology has limitations. Using review counts as a proxy for demand may introduce bias, as not all guests leave reviews and longer-listed properties may accumulate more reviews. The fixed denominator assumption in the MNL model might not hold if supply significantly exceeds demand, affecting the accuracy of choice probabilities. Furthermore, the inherent Independence of Irrelevant Alternatives (IIA) assumption [Udwani, 2024] in the MNL model fails to capture substitution effects between properties and the absence of an outside option further limits the model's ability to account for scenarios where customers opt not to select any property.

Future work should incorporate detailed booking data to estimate utility parameters directly, include outside options to reflect customers opting out, and consider customer heterogeneity in price sensitivity. Relaxing simplifying assumptions and enhancing data quality can improve the model's accuracy, offering valuable insights for pricing optimization in the sharing economy.

References

Pouya Rezazadeh Kalehbasti, Liubov Nikolenko, and Hoormazd Rezaei. *Airbnb Price Prediction Using Machine Learning and Sentiment Analysis*, page 173–184. Springer International Publishing, 2021. ISBN 9783030840600. doi: 10.1007/978-3-030-84060-0_11. URL `http://dx.doi.org/10.1007/978-3-030-84060-0_11`.

Kenneth Train. *Discrete Choice Methods With Simulation*, volume 2009. 01 2009. ISBN 9780521766555. doi: 10.1017/CBO9780511805271.

Rajan Udwani. Submodular order functions and assortment optimization. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 34584–34614. PMLR, 23–29 Jul 2023. URL `https://proceedings.mlr.press/v202/udwani23a.html`.

Rajan Udwani. Operationalizing mnl choice model. Slides, IEOR 145 Fundamentals of Revenue Management, University of California, Berkeley, 2024. Delivered 8 Oct 2024.

```
! pip install gurobipy
```

```
Collecting gurobipy
  Downloading gurobipy-12.0.0-cp310-cp310-manylinux2014_x86_64.manylinux_2_17_x86_64.whl.metadata (15 kB)
Downloading gurobipy-12.0.0-cp310-cp310-manylinux2014_x86_64.manylinux_2_17_x86_64.whl (14.4 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 14.4/14.4 MB 21.3 MB/s eta 0:00:00
Installing collected packages: gurobipy
Successfully installed gurobipy-12.0.0
```

Import Library

```
import pandas as pd
import numpy as np
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import StandardScaler
import gurobipy as gp
from gurobipy import GRB
from google.colab import drive
import matplotlib.pyplot as plt
```

```
options = {
    "WLSACCESSID": "xxx",
    "WLSSECRET": "xxx",
    "LICENSEID": 2576443,
}
```

```
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

Load and Preview Data

```
data = pd.read_csv('/content/drive/MyDrive/Colab Notebooks/245 Project/listings.csv')
```

```
pd.set_option('display.max_columns', None)
data.head(2)
```

Method - Mixture of MNL

```
class AirbnbSubsetOptimizer:
    """
    Implements subset-based MNL pricing optimization for Airbnb listings,
    selecting optimal subsets of properties within each room size category
    """
    def __init__(
        self,
        base_wtb_per_person: float = 50,
        price_steps: int = 10,
        min_price_ratio: float = 0.2,
        total_demand: int = 100,
        max_properties_per_size: dict = None
    ):
        self.base_wtb = base_wtb_per_person
        self.price_steps = price_steps
```

```
        self.min_price_ratio = min_price_ratio
        self.total_demand = total_demand
        # Default max properties per room size if not specified
        self.max_properties_per_size = max_properties_per_size or {
            'small': 50,
            'medium': 30,
            'large': 20
        }
        # Customer type probabilities (α_j) for each room size
        self.customer_type_probs = {
            'small': 0.5,    # 50% probability of small room seekers
            'medium': 0.3,   # 30% probability of medium room seekers
            'large': 0.2     # 20% probability of large room seekers
        }
        self.utility_params = None

    def estimate_utility_parameters(self, df: pd.DataFrame) -> dict:
        """
        Estimate utility parameters from Airbnb data using linear regression.
        Uses actual review counts as target variable for booking preference.
        """

        # Clean and prepare features
        df_clean = df.copy()

        # Convert response rate to numeric
        df_clean['host_response_rate'] = pd.to_numeric(
            df_clean['host_response_rate'].str.rstrip('%').fillna('0'),
            errors='coerce'
        ) / 100

        # Convert boolean features to numeric
        df_clean['is_superhost'] = df_clean['host_is_superhost'].map(
            {'t': 1, 'f': 0}
        ).fillna(0)
        df_clean['identity_verified'] = df_clean['host_identity_verified'].map(
            {'t': 1, 'f': 0}
        ).fillna(0)

        # Normalize review scores
        df_clean['review_score'] = df_clean['review_scores_rating'].fillna(0) / 100

        # Convert price to numeric
        df_clean['price'] = pd.to_numeric(
            df_clean['price'].str.replace('$', '').str.replace(',', ''),
            errors='coerce'
        )

        # Create features matrix (excluding review count)
        X = df_clean[[
            'host_response_rate',
            'is_superhost',
            'identity_verified',
            'review_score',
            'price'
        ]].fillna(0)

        # Create target variable using actual review count
        y = df_clean['number_of_reviews'].fillna(0)

        # Standardize features and fit linear regression
        scaler = StandardScaler()
        X_scaled = scaler.fit_transform(X)
        model = LinearRegression()
        model.fit(X_scaled, y)

        # Get coefficients and normalize to appropriate scales
        # Note: Adjusting normalization factors to account for the change in scale
        coef = model.coef_

        # Calculate R-squared score
        r2_score = model.score(X_scaled, y)
        print(f"\nModel R-squared score: {r2_score:.3f}")

        # Print raw coefficients for inspection
        feature_names = X.columns
        print("\nRaw coefficients:")
```

```
        for feature, coef_value in zip(feature_names, coef):
            print(f"{feature}: {coef_value:.3f}")

        # Normalize coefficients to appropriate scales
        self.utility_params = {
            'host_response_rate': coef[0] * 0.1,
            'is_superhost': coef[1] * 0.1,
            'identity_verified': coef[2] * 0.1,
            'review_score': coef[3] * 0.1,
            'price_sensitivity': coef[4] / 1000  # Price is now at index 4
        }

        print("\nNormalized utility parameters:")
        for param, value in self.utility_params.items():
            print(f"{param}: {value:.5f}")

        return self.utility_params

    def prepare_optimization_data(self, df: pd.DataFrame) -> None:
        """Prepare data for optimization with room size categorization"""
        self.df = df
        self.n_properties = len(df)

        # Add room size categorization
        self.df['room_size'] = self.df['accommodates'].apply(
            lambda x: 'small' if x <= 3 else 'medium' if x <= 6 else 'large'
        )

        # Process features (same as before)
        self.features = pd.DataFrame()
        self.features['host_response_rate'] = pd.to_numeric(
            df['host_response_rate'].str.rstrip('%').fillna('0'),
            errors='coerce'
        ) / 100
        self.features['is_superhost'] = df['host_is_superhost'].map(
            {'t': 1, 'f': 0}
        ).fillna(0)
        self.features['identity_verified'] = df['host_identity_verified'].map(
            {'t': 1, 'f': 0}
        ).fillna(0)
        self.features['review_score'] = df['review_scores_rating'].fillna(0) / 100

        # Calculate base utilities
        self.base_utilities = np.zeros(self.n_properties)
        for feature in self.features.columns:
            self.base_utilities += (
                self.utility_params[feature] * self.features[feature].values
            )

        # Generate price points based on room size
        self.accommodates = df['accommodates'].fillna(1).astype(int)
        self.price_ceilings = self.base_wtb * self.accommodates * (1 + self.min_price_ratio)
        self.price_floors = self.price_ceilings * (1 - self.min_price_ratio)

        self.price_points = [
            np.linspace(floor, ceiling, self.price_steps)
            for floor, ceiling in zip(self.price_floors, self.price_ceilings)
        ]

        # Pre-calculate utilities and choice probabilities
        self.utilities = np.zeros((self.n_properties, self.price_steps))
        for i in range(self.n_properties):
            for j in range(self.price_steps):
                price = self.price_points[i][j]
                self.utilities[i, j] = (
                    self.base_utilities[i] +
                    self.utility_params['price_sensitivity'] * price
                )
        # Pre-calculate denominators for each room size category
        self.denominators = {}
        for size in ['small', 'medium', 'large']:
            size_mask = (self.df['room_size'] == size)
            size_indices = [i for i in range(self.n_properties) if size_mask.iloc[i]]
            exp_utilities = np.exp(self.utilities[size_indices, :])
            self.denominators[size] = exp_utilities.sum()

        # Calculate choice probabilities
```

```
        self.choice_probs = {}
        for size in ['small', 'medium', 'large']:
            size_mask = (self.df['room_size'] == size)
            size_indices = [i for i in range(self.n_properties) if size_mask.iloc[i]]
            exp_utilities = np.exp(self.utilities[size_indices, :])
            self.choice_probs[size] = exp_utilities / self.denominators[size]


    def optimize(self) -> pd.DataFrame:
        """
        Run the subset-based MNL pricing optimization using Gurobi
        """
        try:
            with gp.Env(params=options) as env, gp.Model(env=env) as model:
                # Binary variables for property selection and price points
                x = {}  # Property selection variable
                y = {}  # Price point selection variable

                # Create variables for each property and price point
                for i in range(self.n_properties):
                    x[i] = model.addVar(vtype=GRB.BINARY, name=f'x_{i}')
                    for j in range(self.price_steps):
                        y[i, j] = model.addVar(vtype=GRB.BINARY, name=f'y_{i}_{j}')

                # Constraints for room size limits
                for size, max_count in self.max_properties_per_size.items():
                    size_mask = (self.df['room_size'] == size)
                    model.addConstr(
                        gp.quicksum(x[i] for i in range(self.n_properties)
                                  if size_mask.iloc[i]) <= max_count,
                        name=f'max_properties_{size}'
                    )

                # Constraints for price selection
                for i in range(self.n_properties):
                    # Can only select one price if property is selected
                    model.addConstr(
                        gp.quicksum(y[i, j] for j in range(self.price_steps)) == x[i],
                        name=f'price_selection_{i}'
                    )

                # Calculate choice probabilities for each room size using pre-calculated probs
                obj = 0
                for size, alpha in self.customer_type_probs.items():
                    size_mask = (self.df['room_size'] == size)
                    size_indices = [i for i in range(self.n_properties) if size_mask.iloc[i]]

                    # For each property in the size category
                    for idx, i in enumerate(size_indices):
                        for j in range(self.price_steps):
                            # Calculate revenue contribution with customer type probability
                            revenue = (
                                self.price_points[i][j] *
                                self.choice_probs[size][idx, j] *
                                alpha *
                                self.total_demand
                            )
                            obj += y[i, j] * revenue

                # Set objective
                model.setObjective(obj, GRB.MAXIMIZE)

                # Optimize
                model.optimize()

                # Extract results
                results = []
                for i in range(self.n_properties):
                    result = {
                        'property_id': self.df.index[i],
                        'room_size': self.df['room_size'].iloc[i],
                        'optimal_price': None,
                        'is_selected': x[i].X > 0.5
                    }

                    if result['is_selected']:
                        for j in range(self.price_steps):
```

```
                        if y[i, j].X > 0.5:
                            result['optimal_price'] = self.price_points[i][j]
                            break

                    results.append(result)

                results_df = pd.DataFrame(results)

                total_revenue = results_df['optimal_price'].sum()
                print(f"\nTotal Revenue: ${total_revenue:,.2f}")
                # Print summary statistics
                print("\nOptimization Results:")
                for size in ['small', 'medium', 'large']:
                    size_results = results_df[results_df['room_size'] == size]
                    selected = size_results['is_selected'].sum()
                    avg_price = size_results[size_results['is_selected']]['optimal_price'].mean()
                    print(f"\n{size.capitalize()} Properties:")
                    print(f"Selected: {selected}/{self.max_properties_per_size[size]}")
                    print(f"Average Price: ${avg_price:,.2f}")

                return results_df

        except gp.GurobiError as e:
            print(f"Optimization error: {e}")
            return None

def run_subset_analysis(
    df: pd.DataFrame,
    total_demand: int = 100,
    max_properties: dict = None
) -> pd.DataFrame:
    """
    Helper function to run the complete subset-based analysis pipeline
    """
    # Initialize optimizer with subset constraints
    optimizer = AirbnbSubsetOptimizer(
        base_wtb_per_person=50,
        price_steps=10,
        min_price_ratio=0.2,
        total_demand=total_demand,
        max_properties_per_size=max_properties
    )

    # Estimate parameters and prepare data
    optimizer.estimate_utility_parameters(df)
    optimizer.prepare_optimization_data(df)

    # Run optimization
    return optimizer.optimize()


# Example usage with custom property limits
max_properties = {
    'small': 50,  # Max 50 small properties
    'medium': 30, # Max 30 medium properties
    'large': 20   # Max 20 large properties
}

results = run_subset_analysis(
    df=data,
    total_demand=100,
    max_properties=max_properties
)
```

```
review_score: 2.19314
```

```
Coefficient statistics:
  Matrix range     [1e+00, 1e+00]
  Objective range  [3e-01, 5e+01]
  Bounds range     [1e+00, 1e+00]
  RHS range        [2e+01, 5e+01]
Found heuristic solution: objective -0.0000000
Presolve removed 428 rows and 4528 columns
Presolve time: 0.01s
Presolved: 1 rows, 158 columns, 158 nonzeros
Found heuristic solution: objective 975.5153280
Variable types: 0 continuous, 158 integer (120 binary)
Found heuristic solution: objective 1088.2461041

Root relaxation: objective 1.138156e+03, 1 iterations, 0.00 seconds (0.00 work units)

    Nodes    |    Current Node    |     Objective Bounds      |     Work
 Expl Unexpl |  Obj  Depth IntInf | Incumbent    BestBd   Gap | It/Node Time

*    0     0               0    1138.1558866 1138.15589  0.00%     -    0s

Explored 1 nodes (1 simplex iterations) in 0.06 seconds (0.00 work units)
Thread count was 2 (of 2 available processors)

Solution count 4: 1138.16 1088.25 975.515 -0

Optimal solution found (tolerance 1.00e-04)
Best objective 1.138155886564e+03, best bound 1.138155886564e+03, gap 0.0000%

Total Revenue: $21,492.00

Optimization Results:

Small Properties:
Selected: 50/50
Average Price: $138.00

Medium Properties:
Selected: 30/30
Average Price: $208.00

Large Properties:
Selected: 20/20
Average Price: $417.60
```

```
selected_properties = data.copy()
# Create a new column 'property_id' with index values
for i in range(len(selected_properties)):
  selected_properties.loc[selected_properties.index[i], 'property_id'] = selected_properties.index[i]

results_selected = results[results['is_selected'] == True]

# Merge DataFrames using only the 'on' argument
selected_details = pd.merge(
    selected_properties[['property_id', 'name', 'accommodates', 'neighbourhood_cleansed', 'price']],
    results_selected[['property_id', 'optimal_price']],
    on='property_id' # Removed right_index=True
)
selected_details['property_id'] = selected_details['property_id'].astype(int)
selected_details['optimal_price'] = selected_details['optimal_price'].apply(lambda x: '${:.2f}'.format(x))


selected_details
```

	property_id	name	accommodates	neighbourhood_cleansed	price	optimal_price
0	9	/Fire Place Bungalow\ 1917 SUNY Eagle 6Beds 2B...	10	FIFTEENTH WARD	$214.00	$480.00
1	15	/Miller Colonial\ 1946 SUNY Eagle Hill 5Bed 2B...	7	FIFTEENTH WARD	$243.00	$336.00
2	23	All The Comforts Of Home For You In Albany	9	FOURTEENTH WARD	$275.00	$432.00
3	25	Garden Apartment, on the Park, close to Capital.	4	SEVENTH WARD	$126.00	$208.00
4	26	The Metropolitan	2	NINTH WARD	$75.00	$120.00
...	...	...	...	...	...	...
95	356	Sunny 2nd Floor Room in Historic Manor Retreat	2	THIRTEENTH WARD	$65.00	$120.00
96	363	Walkable 1BR Apt wl Fire Pit, Near Empire Plaza!	4	SIXTH WARD	$92.00	$208.00
97	365	Modern home, King Bed, parking	4	FIFTEENTH WARD	$130.00	$208.00
98	389	Walk to convention center:Lark St:Capital	4	SIXTH WARD	$165.00	$208.00
99	393	Furnished 4 bed vintage home	7	THIRTEENTH WARD	NaN	$336.00

100 rows × 6 columns

Method 3 - Choose highest price in each size

```
class SimpleApproxOptimizer:
    """
    Implements simplified 1-m approximation by selecting highest revenue items
    for each room size based on demand proportion
    """
    def __init__(
        self,
        base_wtb_per_person: float = 50,
        price_steps: int = 10,
        min_price_ratio: float = 0.2,
        total_demand: int = 200
    ):
        self.base_wtb = base_wtb_per_person
        self.price_steps = price_steps
        self.min_price_ratio = min_price_ratio
        self.total_demand = total_demand
        self.customer_type_probs = {
            'small': 0.5,
            'medium': 0.3,
            'large': 0.2
        }
        self.utility_params = None

    def prepare_data(self, df: pd.DataFrame) -> None:
        """Prepare data and calculate potential revenues"""
        self.df = df.copy()

        # Add room size
        self.df['room_size'] = self.df['accommodates'].apply(
            lambda x: 'small' if x <= 3 else 'medium' if x <= 6 else 'large'
        )

        # Calculate potential revenue for each property
        for size, prob in self.customer_type_probs.items():
            mask = self.df['room_size'] == size
            if mask.any():
                # Calculate demand for this room size
                size_demand = int(self.total_demand * prob)
                self.df.loc[mask, 'type_demand'] = size_demand


        self.df['price'] = pd.to_numeric(
            self.df['price'].str.replace('$', '').str.replace(',', ''),
            errors='coerce'
        ).fillna(0)

    def optimize(self) -> pd.DataFrame:
        """
        Select highest revenue properties for each room size
        based on their demand proportion
```

```
        """
        results = []
        total_revenue = 0

        for size, prob in self.customer_type_probs.items():
            # Get properties of this size
            size_props = self.df[self.df['room_size'] == size].copy()
            if size_props.empty:
                continue

            # Calculate demand for this room size
            size_demand = int(self.total_demand * prob)

            # Sort by potential revenue (using base price as proxy)
            size_props = size_props.sort_values('price', ascending=False)

            # Select top properties based on demand
            selected_props = size_props.head(size_demand)

            # Calculate revenue for selected properties
            for _, prop in selected_props.iterrows():
                result = {
                    'property_id': prop.name,
                    'room_size': size,
                    'optimal_price': prop['price'],
                    'is_selected': True,
                    'expected_demand': 1  # Each selected property gets one booking
                }
                revenue = prop['price'] * 1  # Price * single booking
                total_revenue += revenue
                results.append(result)

        results_df = pd.DataFrame(results)

        # Print summary
        print("\nSimple 1-m Approximation Results:")
        print(f"Total Expected Revenue: ${total_revenue:,.2f}")
        for size in ['small', 'medium', 'large']:
            size_results = results_df[results_df['room_size'] == size]
            if not size_results.empty:
                selected = len(size_results)
                avg_price = size_results['optimal_price'].mean()
                print(f"{size.capitalize()}: {selected} properties")
                print(f"Average Price: ${avg_price:,.2f}")

        return results_df

def compare_approaches(df: pd.DataFrame, total_demand: int = 200):
    """Compare subset optimization vs simple approximation"""
    # Run subset optimization
    print("Running subset optimization...")
    subset_opt = AirbnbSubsetOptimizer(
        total_demand=total_demand,
        max_properties_per_size={'small': 50, 'medium': 30, 'large': 20}
    )
    subset_opt.estimate_utility_parameters(df)
    subset_opt.prepare_optimization_data(df)
    subset_results = subset_opt.optimize()

    # Run simple approximation
    print("\nRunning simple approximation...")
    simple_opt = SimpleApproxOptimizer(total_demand=total_demand)
    simple_opt.prepare_data(df)
    approx_results = simple_opt.optimize()

    return subset_results, approx_results


# Run comparison
subset_results, simple_results = compare_approaches(
    df=data,
    total_demand=100
)
```

```
Coefficient statistics:
  Matrix range     [1e+00, 1e+00]
  Objective range  [3e-01, 5e+01]
  Bounds range     [1e+00, 1e+00]
  RHS range        [2e+01, 5e+01]
Found heuristic solution: objective -0.0000000
Presolve removed 428 rows and 4528 columns
Presolve time: 0.01s
Presolved: 1 rows, 158 columns, 158 nonzeros
Found heuristic solution: objective 975.5153280
Variable types: 0 continuous, 158 integer (120 binary)
Found heuristic solution: objective 1088.2461041

Root relaxation: objective 1.138156e+03, 1 iterations, 0.00 seconds (0.00 work units)

    Nodes    |    Current Node    |     Objective Bounds      |     Work
 Expl Unexpl |  Obj  Depth IntInf | Incumbent    BestBd   Gap | It/Node Time

*    0     0               0    1138.1558866 1138.15589  0.00%     -    0s

Explored 1 nodes (1 simplex iterations) in 0.14 seconds (0.00 work units)
Thread count was 2 (of 2 available processors)

Solution count 4: 1138.16 1088.25 975.515 -0

Optimal solution found (tolerance 1.00e-04)
Best objective 1.138155886564e+03, best bound 1.138155886564e+03, gap 0.0000%

Total Revenue: $21,492.00

Optimization Results:

Small Properties:
Selected: 50/50
Average Price: $138.00

Medium Properties:
Selected: 30/30
Average Price: $208.00

Large Properties:
Selected: 20/20
Average Price: $417.60

Running simple approximation...

Simple 1-m Approximation Results:
Total Expected Revenue: $22,794.00
Small: 50 properties
Average Price: $141.26
Medium: 30 properties
Average Price: $233.07
Large: 20 properties
Average Price: $436.95
```

Start coding or generate with AI.

Start coding or generate with AI.

Start coding or generate with AI.

Start coding or generate with AI.

Start coding or generate with AI.

Start coding or generate with AI.

Start coding or generate with AI.

Start coding or generate with AI.

Start coding or generate with AI.

Start coding or generate with AI.