sdmimd
Shri Dharmasthala Manjunatheshwara Institute for Management Development, Mysuru, India
6th International Marketing Conference on
Marketing in the AI Era - Marketing 5.0 - Reshaping Global Marketing on 17th January 2025

Case study: Optimal Pricing Strategy for a company selling Biscuit Products Using Customer Segmentation and Reinforcement Learning

Kiran Satyanarayana
kirannarayan82@gmail.com

Abstract

This paper presents an optimal pricing strategy for a fictional company biscuit products using customer segmentation and reinforcement learning. By leveraging customer segmentation to identify distinct consumer groups and employing a Q-learning agent, the proposed model dynamically adjusts prices to maximize revenue. This methodology is demonstrated through a case study involving three biscuit products, showcasing the effectiveness of the approach in real-world scenarios.

Introduction

Pricing strategy is a critical component of revenue management for consumer goods companies. In the competitive landscape of the biscuit market, companies need to optimize their pricing to cater to diverse customer segments while maximizing revenue. This paper proposes a dynamic pricing model that uses customer segmentation and reinforcement learning to achieve these goals.

Background

The consumer goods industry, and specifically the food sector, is highly competitive. Companies are constantly seeking innovative methods to stay ahead of the competition. One of the key challenges in this industry is setting optimal prices that maximize revenue without alienating price-sensitive customers. Traditional pricing strategies, while useful, often fall short in addressing the dynamic nature of consumer behaviour and market conditions. This paper explores the potential of combining customer segmentation with reinforcement learning to develop a more adaptive and effective pricing strategy.

Objectives

The primary objective of this research is to develop a dynamic pricing model that

Identifies distinct customer segments with varying price sensitivities.

Utilizes reinforcement learning to dynamically adjust prices based on real-time market conditions.

Maximizes revenue while maintaining customer satisfaction.

Literature Survey

Comprehensive Survey of Reinforcement Learning: From Algorithms to Practical Challenges:

This paper provides a detailed overview of reinforcement learning (RL) algorithms, from foundational tabular methods to advanced Deep Reinforcement Learning (DRL) techniques. It evaluates these algorithms based on scalability, sample efficiency, and suitability, and offers practical insights into addressing common challenges like convergence, stability, and the exploration-exploitation dilemma.

Impact of Price Sensitivity on Customer Satisfaction: An Empirical Study in Retail Sector:

ISBN code **978-93-83302-73-4**

This study examines how price sensitivity affects customer satisfaction in the retail sector[2]. Using tools like exploratory factor analysis and multiple regression analysis, it found a significant association between price sensitivity factors and customer satisfaction

Price Sensitivity Affected by Age Sensitivity:

This research explores how age influences price sensitivity [3.] It shows that different age groups have varying levels of price sensitivity, with younger consumers often being less price-sensitive compared to older consumers

Examining the Impact of Price Sensitivity on Customer Lifetime Value:

This study investigates how price sensitivity parameters affect customer lifetime value in the luxury business[4]. It identifies key dimensions of price sensitivity, including Quality Value, Time Value, Position Value, and Information Value

Scope for this paper

This paper builds a dynamic pricing mechanism based on customer segmentation in Reinforcement learning. This can be used across industries by companies which have diverse product to define optimal pricing for each of their segments and products based on the constraints they have. In this paper manufacturing costs and conversion rates have been taken as constraints, but the same constraints can be taken as different variables can be taken and this methodology can be implemented

Methodology

The proposed methodology involves two main components: customer segmentation and reinforcement learning.

Customer Segmentation

Customer segmentation is performed using K-means clustering, which groups customers based on income, age, and spending scores. This enables the identification of distinct consumer segments with varying preferences and price sensitivities.

Data Preparation

To perform customer segmentation, we first prepare the data. The data includes customer demographics, purchasing patterns, and preferences. For this study, we generate synthetic data to simulate real-world scenarios.

K-means Clustering

We use the K-means clustering algorithm to segment the customers based on their income, age, and spending scores.

The K-means algorithm partitions the data into three clusters, each representing a distinct customer segment. These segments can be characterized as follows:

Segment 1: High-income, health-conscious consumers who prefer premium, organic biscuits.

Segment 2: Middle-income, family-oriented consumers who look for value-for-money options.

Segment 3: Low-income, convenience-driven consumers who prefer affordable, everyday biscuits.

ISBN code **978-93-83302-73-4**

Reinforcement Learning Model

A Q-learning agent is developed to dynamically adjust prices based on the identified customer segments. The agent interacts with a simulated market environment to learn optimal pricing strategies through trial and error.

Market Environment

The market environment simulates the sales and costs associated with different pricing actions. The environment includes variables such as budget constraints and sales for each channel.

```
class MarketEnvironment:
def __init__(self, budget, channels):
self.budget = budget
self.channels = channels
self.sales = np.zeros(len(channels))
```

Defining the Market Environment

In the context of the MarketEnvironment class, "channels" refer to the different marketing or sales channels through which a product or service is promoted and sold. These channels have associated costs and conversion rates. For example, channels could include digital advertising, social media marketing, email campaigns, or any other medium through which the product reaches potential customers

```
channels = {
    'cost': [1, 2, 3],  # Cost of using each channel
    'conversion_rate': [10, 20, 30]  # Conversion rate of each channel
}
```

channels

__init__: Initializes the environment with a given budget and channels. The sales array is initialized to zeros, representing no sales at the start.

```
def reset(self):
        self.sales = np.zeros(len(self.channels))
        return np.zeros(len(self.channels))
```

Reset Method

Reset method resets the sales to zero and returns an array of zeros representing the initial state.

ISBN code **978-93-83302-73-4**

```
    def step(self, action):
        cost = np.dot(action, self.channels['cost'])
        if cost > self.budget:
            return self.reset(), -100, True

        self.sales += np.dot(action, self.channels['conversion_rate'])
        return np.array(self.sales / 1000), np.sum(self.sales), False
```

Step Method

The step function an action (array), calculates the cost, and updates sales.

cost: Computed using the dot product of action and the cost array.

If cost exceeds the budget, it resets and returns a penalty reward (-100) and a done signal (True).

If not, it updates the sales based on conversion rates and returns the new state (sales/1000), the reward (sum of sales), and the done signal (False).

Action and State Concepts

Action typically refers to the decision or set of decisions made by the agent at each step. Specifically, it represents how the agent allocates resources (such as budget) across the different marketing or sales channels to maximize sales or achieve a specific goal.

Actions are usually represented as arrays or vectors. For example, if have three channels, an action might be an array like [0.3, 0.5, 0.2], indicating how the budget is distributed across the three channels.

The environment takes this action array and uses it to calculate the total cost and the resulting sales. The step method in the MarketEnvironment class processes the action to determine these values.

Example

action = [0.3, 0.5, 0.2] # Allocating 30% of the budget to the first channel, 50% to the second, and 20% to the third

cost = np.dot(action, self.channels['cost'])

sales = np.dot(action, self.channels['conversion_rate'])

If there are three channels, possible actions might involve allocating different percentages of the budget across these channels. For example:

[1, 0, 0] - Allocate all budget to the first channel

[0, 1, 0] - Allocate all budget to the second channel

State size refers to the dimensionality of the state space, or the number of distinct elements that define the state of the environment at any given time. In rMarketEnvironment class, the state size is determined by the number of channels have.

channels = {'cost': [1, 2, 3], 'conversion_rate': [10, 20, 30]}

state_size = len(channels['cost']) # or len(channels['conversion_rate'])

ISBN code **978-93-83302-73-4**

Since each channel represents an element of the state (e.g., sales through that channel), if have three channels, the state size is 3.

Action Size

Action size refers to the dimensionality of the action space, or the number of distinct actions that the agent can take at each step. In r Q-learning agent, the action size is the number of possible actions the agent can choose from.

action_size = 3 # Assuming 3 possible actions corresponding to 3 channel

When creating the Q-learning agent, we need to initialize it with the appropriate state size and action size based on r environment:

env = MarketEnvironment(budget=1000, channels={'cost': [1, 2, 3], 'conversion_rate': [10, 20, 30]})

state_size = len(env.channels['cost']) # or len(env.channels['conversion_rate'])

action_size = 3 # Number of possible actions

State Size: Determines how the agent perceives the environment. Each state provides a snapshot of the environment that the agent uses to make decisions.

Action Size: Determines the range of actions the agent can take. The choice of action affects the next state and the reward received, thereby guiding the learning process.

Q learning : an overview

Initialize Q-Table: Create a table where each cell represents a state-action pair and initialize it with random values. This table will be updated as the agent learns.

Observe State: The agent looks at its current position or state in the environment.

Choose Action: The agent decides which action to take. It can explore new actions or exploit known actions that have high Q-values. This is often done using an ε-greedy strategy, where the agent sometimes chooses a random action (exploration) and other times chooses the best-known action (exploitation).

Perform Action and Receive Reward: The agent takes the action, moves to the next state, and receives a reward.

Update Q-Value: The agent updates the Q-value for the state-action pair using the formula:

α\alpha (learning rate): How much new information overrides old information.

γ\gamma (discount factor): Importance of future rewards.

rr: The reward received.

ss: Current state.

aa: Action taken.

s’s': Next state.

a’a': Next action.

Repeat: The agent repeats this process for many episodes (trials) until it learns the best way to maximize the reward. Q-learning helps an agent learn the best actions to take in different situations

ISBN code **978-93-83302-73-4**

by exploring, receiving rewards, and updating its knowledge to improve future decisions. It's like teaching a robot to navigate a maze and find the best path to the goal.

agent = QLearningAgent(state_size=state_size, action_size=action_size)

class QLearningAgent:

```
    def __init__(self, state_size, action_size, learning_rate=0.1, discount_factor=0.95):
        self.state_size = state_size
        self.action_size = action_size
        self.learning_rate = learning_rate
        self.discount_factor = discount_factor
        self.q_table = tf.Variable(tf.random.normal([state_size, action_size]))
```

Initializes the agent with state size, action size, learning rate, and discount factor. The Q-table is initialized with random values

Learning rate and Discount rate

Learning Rate **(α)**

The learning rate determines how much new information overrides old information. It controls how quickly the agent updates its knowledge.

Range: The learning rate is typically between 0 and 1.

High learning rate (close to 1): The agent learns rapidly, but may become unstable and forget previous knowledge too quickly.

Low learning rate (close to 0): The agent learns more slowly, but its updates are more stable.

-

Discount Rate (γ)

The discount rate, also known as the discount factor, determines the importance of future rewards. It helps balance immediate rewards against long-term benefits.

Range: The discount factor is typically between 0 and 1.

High discount factor (close to 1): Future rewards are considered nearly as important as immediate rewards. The agent is more farsighted.

Low discount factor (close to 0): Future rewards are considered less important than immediate rewards. The agent is more shortsighted.

Exploration and Exploitation

```
    def choose_action(self, state, epsilon=0.1):
        if np.random.rand() <= epsilon:
            return np.random.randint(self.action_size)
        else:
            return tf.argmax(self.q_table[state]).numpy()
```

ISBN code **978-93-83302-73-4**

Chooses an action using an ε-greedy policy.

With probability epsilon, it chooses a random action (exploration).

Otherwise, it selects the action with the highest Q-value (exploitation).

Learn: Updates the Q-values based on the received reward and next state.

Target Calculation: If the episode is done, the target is the reward. Otherwise, it is the reward plus the discounted maximum Q-value of the next state.

Loss Calculation: The loss is the squared difference between the target and the current Q-value.

Gradient Update: Uses TensorFlow's GradientTape to calculate gradients and updates the Q-table using the Adam optimizer.

```
    def learn(self, state, action, reward, next_state, done):
        target = reward if done else reward + self.discount_factor * tf.reduce_max(self.q_table[next_state]).numpy(
        with tf.GradientTape() as tape:
            loss = tf.square(target - self.q_table[state, action])
        gradients = tape.gradient(loss, [self.q_table])
        optimizer = tf.keras.optimizers.Adam(learning_rate=self.learning_rate)
```

Interaction loop

Define an interaction loop to define the agent , environment and learning

```
[ ]  env = MarketEnvironment(budget=1000, channels={'cost': [1, 2, 3], 'conversion_rate': [10, 20, 30]})
     agent = QLearningAgent(state_size=len(env.channels), action_size=3)
```

```
for episode in range(1000):
    state = env.reset()
    done = False
       while not done:
        action = agent.choose_action(state)
        next_state, reward, done = env.step(action)
        agent.learn(state, action, reward, next_state, done)
        state = next_state
```

Example of interaction loop

Initialization: The environment and agent are initialized.

Episodes Loop: Runs for a specified number of episodes.

State Reset: Resets the environment state.

Action Selection: The agent selects an action based on the current state.

Environment Step: The environment takes a step based on the action and returns the new state, reward, and done signal.

Learning: The agent updates its Q-values based on the new information.
This loop continues until the episode ends, updating the agent's knowledge and improving its strategy over time.

ISBN code **978-93-83302-73-4**

Sample output derived from Q learning and Segmentation

Segment	Cost	Conversion Rate	Pricing Strategy
0	[0.718, 0.590, 0.683]	[13.68, 12.91, 6.95]	[0.60, 1.49, 0.70]
1	[0.703, 0.718, 0.620]	[13.57, 14.86, 8.02]	[1.24, 0.47, 0.82]
2	[0.721, 0.614, 0.687]	[15.25, 15.76, 11.10]	[0.51, 0.99, 0.23]

Explanation:

Segment: The identifier for each customer segment.

Cost: The average cost values associated with each channel for this segment.

Conversion Rate: The average conversion rates associated with each channel for this segment.

Pricing Strategy: The pricing strategy (actions) derived from the Q-table for each segment.

This table presents the data in an easy-to-read format, making it clear how different segments have varying cost and conversion rates, and how the Q-learning agent has determined different pricing strategies for each segment.

Conclusion

This is a sample case study for implementing dynamic pricing using Reinforcement learning looking at cost and conversion rates as constraints. Pricing involves more parameters which can increase the constraints. This case study can be used as a model and replicated for other use case in other industries

References

1. Majid Ghasemi, Amir Hossein Mousavi, Dariush Ebrahimi, Comprehensive Survey of Reinforcement Learning: From Algorithms to Practical Challenges https://arxiv.org/abs/2411.18892?form=MG0AV3

2.Pankaj Gupta, Impact of Price sensitivity on Customer Satisfaction: An Empirical Study in Retail Sector,https://www.iosrjournals.org/iosr-jhss/papers/Vol19-issue5/Version-2/C019521721.pdf?form=MG0AV

3.Dr. Varalakshmi , Professor , Jain university – CMS Bengaluru et. all, Price sensitivity affected by age sensitivity, https://ijcrt.org/papers/IJCRT2305566.pdf?form=MG0AV3

4. Sarah Ahmed Awaad et all, Examining the impact of price sensitivity on customer lifetime value: empirical analysis, https://www.tandfonline.com/doi/pdf/10.1080/23311975.2024.2366441?form=MG0AV3